DEXON:
A Highly Scalable, Decentralized DAG-Based Consensus Algorithm

Tai-Yuan Chen, Wei-Ning Huang, Po-Chun Kuo, Hao Chung and Tzu-Wei Chao

DEXON Foundation,
Taiwan

{*popo,w,pk,haochung,n*}*@dexon.org*

November 19, 2018, v2.0

Abstract

A blockchain system is a replicated state machine that must be fault tolerant. When designing a blockchain system, there is usually a trade-off between decentralization, scalability and security. In this paper, we propose a novel blockchain system, DEXON, which achieves high scalability while remaining decentralized and robust in the real-world environment.

We have two main contributions. First, we present a highly scalable sharding framework for blockchain. This framework takes an arbitrary number of single chains and transforms them into the *blocklattice* data structure, enabling *high scalability* and *low transaction confirmation latency* with asymptotically optimal communication overhead. Second, we propose a single-chain protocol based on our novel verifiable random function and a new Byzantine agreement that achieves high decentralization and low latency.

Keywords: Blockchain, Blocklattice, Consensus, Byzantine Agreement, Byzantine Fault Tolerance, Replicated State Machine, Total Ordering

Contents

1 Introduction

Blockchain systems are being challenged to demonstrate rigorous robustness and high performance in real-world situations. Many applications demand low transaction confirmation latency and high transaction throughput. However, most blockchain systems do not satisfy these criteria. For example, the confirmation latency of Ethereum is about 5 to 10 minutes and the throughput is limited to about 30 transactions per second. By contrast, some blockchain systems achieve high performance but sacrifice the robustness of the systems. For example, EOS is operated with only 21 supernodes, and is vulnerable to DDoS attacks.

We propose a novel blockchain framework, *DEXON*, which achieves high performance and remains robust on the real-world Internet. To this end, we design a data structure, called *blocklattice*, which allows numerous single chains to grow concurrently. Then, we propose a novel method to integrate these single chains into a globally-ordered chain without additional communication.

The blocklattice structure is a directed acyclic graph (DAG) that consists of many single chains. The blocks in the different chains acknowledge (ack) each other and collectively form this DAG structure. We use the total ordering algorithm to achieve consensus on the blocklattice so that all users are guaranteed to have the same view of the ordering of all the blocks. Because the single chains can grow concurrently, the throughput of DEXON can be easily scaled up.

We also propose our single-chain protocol, which is based on Algorand [GHM$^+$17] but with some minor improvements. The Algorand consensus protocol is a breakthrough that allows millions of nodes to join the protocol; each node has fair opportunity to propose and validate blocks. For this protocol, we present a new verifiable random function and Byzantine agreement achieving low confirmation time even for a large population of nodes. We emphasize that our blocklattice structure with its total ordering algorithm is a generic framework that applies to any kind of single-chain protocol. Thus, if the throughput of the underlying single-chain protocol increases by a factor of 2, the total throughput of the whole system also increases by a factor of 2.

1.1 Main Features of DEXON

DEXON has the following advantages:

High Scalability

Most blockchain solutions are not able to scale their throughput even with increased resources. Our blockchain system can adjust the number of chains dynamically while preserving the same latency. The throughput of our system is only bound by the available network bandwidth and computational power. In addition, the scaling methodology of the DEXON consensus is generic. That is, the blocklattice structure with its total ordering algorithm can be applied to any kind of single-chain consensus protocol.

Low Latency

Latency is the amount of time from block proposal to confirmation. Therefore, latency is one of the paramount properties for any blockchain. We propose a fast Byzantine agreement that is expected to terminate in 6λ time, where λ is the upper bound of the network's gossip period. The DEXON consensus algorithm achieves second-level latency instead of traditional minute-level latency as exhibited by blockchains such as Ethereum or Bitcoin. A second-level latency blockchain opens a new era that provides numerous variations in service that cannot be delivered by traditional blockchains.

High Decentralization

The DEXON consensus is based on "*proof-of-participation (PoP)*"; that is, every node has equal chance to propose a block. We adopt a *verifiable random function (VRF)* to decide who can issue a block; this serves to minimize the communication cost so that a large population can join the protocol.

Transaction Ordering Fairness

In traditional blockchain systems, a single proposer can determine the transaction ordering; this makes traditional blockchain systems vulnerable to front-run attacks. By contrast, in the DEXON consensus algorithm, no single block proposer can determine the consensus timestamp of a proposed block.

Unpredictable Randomness

Randomness is often a desired functionality in various smart contracts such as decentralized applications (DApps), particularly gaming DApps. Normally, a blockchain system cannot generate unpredictable randomness on-chain, so DApp developers must rely on some trusted third party for random input, such as the service provided by Oracalize. However, the DEXON consensus generates on-chain unpredictable randomness on the fly as it achieves consensus. Once a block has been confirmed by the DEXON Byzantine agreement, a committee of nodes generates a threshold signature, which is an unpredictable value. Thus, the hash of the signature serves as the unique, unpredictable, and unbiased randomness of the block.

Explicit Finality

In blockchain systems based on proof-of-work, such as Bitcoin or Ethereum, transaction confirmation is probabilistic. Only after users have waited for a long sequence of block confirmations can a transaction be considered as *probabilistically finalized*; the system is thus vulnerable to double spending attacks. For use cases such as payment networks, an explicit finality with probability 1 is necessary. In the DEXON consensus, every transaction is confirmed to be finalized with probability 1 and is secured by DEXON's Byzantine agreement algorithm, which has been proven correct with rigorous mathematical logic.

Low Communication Overhead

A two-phase-commit consensus algorithm has an $\mathcal{O}(n^2)$ communication complexity in any n-node setting and is thus highly costly to scale up. The DEXON consensus adopts VRF to reduce the amount of nodes joining the protocol from n parties to $\mathcal{O}(\log n)$. Thus, the communication cost is reduced from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$. In this situation, the number of nodes can be scaled to millions while maintaining only hundreds of nodes that must communicate with each other.

Energy Efficiency

DEXON Consensus has asymptotically optimal computation overhead, making it highly energy efficient.

Low Transaction Fees

The transaction fees of a typical blockchain increase when the blockchain network is congested. The DEXON consensus is highly scalable and has low communication overhead, which enables it to maintain the lowest possible transaction fees in large-scale deployments.

When designing a blockchain system, a trade-off usually exists between decentralization, performance, and safety, which we call the *trilemma* problem in blockchains. For example, EOS and Hashgraph achieve high performance, but they are operated by few supernodes and are vulnerable to DDoS attack. By contrast, Algorand is decentralized and has robust safety, but its throughput is limited.

In DEXON, we balance the requirements of the trilemma. DEXON has scalable transaction throughput and low confirmation latency. However, DEXON remains highly decentralized and robust in practical deployment environments.

1.2 Related Work

Proof-of-Work Bitcoin [Nak08] is the first blockchain protocol whose consensus delivers Nakamoto consensus: This means that the Bitcoin system solves a mathematical puzzle as a proof to generate next block, and once a block has been followed by six continuous blocks, the block is confirmed. This mechanism causes Bitcoin to have a latency of approximately one hour. Even more problematically, proof-of-work-based consensus consumes exorbitant quantities of energy.

Proof-of-Stake Numerous proof-of-stake consensus systems [GHM+17, HMW, BHM18, DGKR18] have been proposed in recent years. In these schemes, the nodes with adequate stakes have the right to propose their blocks. The probability that a node can propose a block is proportional to the stakes the node owns.

DAG-based consensus Phantom, SPECTRE, IOTA, Conflux, and Mechcash are DAG-based consensus systems, all of which are, at their core, variants of the Nakamoto consensus. This leads two disadvantages: first, these systems demonstrate low performance (throughput is low and latency is long); second, the finality is probabilistic, allowing some attacks (such as selfish-mining) to exist. To conclude, constructing a DAG-based consensus with the Nakamoto consensus limits performance and safety.

Algorand is a breakthrough proposed by Gilad et al. [GHM+17], that reduces the communication complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \ln n)$ and thus supports large population of nodes (e.g. 500K nodes). They use a verifiable random function (VRF) to protect nodes from DDoS attack, and the VRF is also a lottery that decides which node has the right to propose a block or to vote for each round of their Byzantine agreement protocol. The consensus of Algorand is based on Byzantine agreement among samples from the whole set of nodes. Thus, the probability of the correctness of whole system is based on hypergeometric distribution. This is the reason why Algorand can only tolerate less than one third of total number of nodes to be malicious while Algorand achieves high decentralized.

Dfinity [HMW] is a permissioned blockchain and is designed for large population of nodes (around 10K nodes). Dfinity contains a randomness beacon which generates new randomness by a VRF with information from new confirmed block. They use the output of a VRF to select a leader and electors for a round. By hypergeometric distribution, Dfinity only samples hundreds of nodes to notarize a block instead of using all nodes, and the correctness holds with high probability.

The consensus of Hashgraph [BHM18] adopts Byzantine agreement on a graph and their round-based structure costs a latency of $\mathcal{O}(\ln n)$ for each round of Byzantine Agreement, which means its confirmation time increases with the number of nodes. This limits the decentralized level of Hashgraph.

Roadmap In Section 2, we introduce the cryptographic primitives and the system model used in this paper. In Section 3, we formally introduce our Byzantine agreement protocol and describe how to build up a single chain by our Byzantine agreement protocol. In Section 4, we introduce how DEXON reaches a consensus on a blocklattice by using its total ordering algorithm. Finally, we discuss the sharding scheme and the method to adjust system parameters in Section 5.

2 Preliminaries and Model

2.1 Cryptographic Primitive

In this section, we introduce the cryptographic primitives used in this paper, including our hash function, digital signature, threshold signature, and verifiable random function.

Hash Function and Digital Signature In this paper, we model the hash function $Hash$ as a random oracle. That is, the hash function acts as a truly random function which can only be accessed by "querying" an oracle. We also assume that an unforgeable digital signature Sig is available. We define $Sig_{sk}(m)$ to be the signature of the message m signed by the secret key sk.

Threshold Signature Let n be the number of parties. A (n,t)-*threshold signature* scheme allows arbitrary t parties to sign a message and any party with the public key can verify the threshold signature. A threshold signature scheme consists of five probabilistic polynomial-time algorithms:

- KeyGen(1^Λ): a distributed *key generation* algorithm takes as input a security parameter Λ, and outputs a public key PK, a set of verification keys $\{VK_i\}_{i=1}^n$, and the secret key SK_i for each party i.
- ShareSign(SK_i, m): a *share signing* algorithm takes as input a message m and a secret key SK_i. It outputs a signature share $SSign_i(m)$.
- Verify-ShareSign(m, VK_i, σ_i): a *share verification* algorithm takes as input a message m, the verification key VK_i and a signature share σ_i on m from the party i. It outputs 1 if σ_i is valid and outputs 0 otherwise.
- Combine$(m, PK, \{VK_i\}_{i\in T}, \{\sigma_i\}_{i\in T})$: a *share combining* algorithm takes as input a message m, the public key PK, the set of verification key $\{VK_i\}$ and t verified signature shares $\{\sigma_i\}_{i\in T}$ on the message m where T is a subset of $\{1, 2, \cdots, n\}$ such that $|T| = t$. It outputs a threshold signature $TSign(m)$.
- Verify-TSign(m, PK, Σ): a *signature verification* algorithm takes as input a message m, the public key PK, and a threshold signature Σ. It outputs 1 if Σ is valid and outputs 0 otherwise.

It is required that except with negligible probability, Verify-ShareSign$(m, VK_i, \text{ShareSign}(SK_i, m)) = 1$ holds for any party i. It is also required that except with negligible probability, Verify-TSign$(m, PK, \Sigma) = 1$ holds, where Σ is the threshold signature of t valid signature shares $\{\sigma_i\}_{i\in T}$ for any subset $T \subset \{1, 2, \cdots, n\}$.

For the security definition, one can refer to [LJY14].

Verifiable Random Function Let (PK, SK) be a pair of a public key and a private key. *Verifiable random function (VRF)* is a kind of pseudo-random function such that only a user that has SK can compute the function whereas anyone can verify the validity of the function evaluation by PK and public information. Typically, a VRF consists of the following algorithms:

- KeyGen(1^Λ): a probabilistic polynomial time algorithm takes as input a security parameter Λ, and generates a public key and private key pair (PK, SK).
- VRF(SK, x): a deterministic algorithm takes as input the private key and an initial value x, and outputs verifiable random value y.
- Prove(SK, x): a deterministic algorithm takes as input the private key and an initial value x, and output the proof of correctness w.
- Verify(x, y, w): a deterministic algorithm verifies the validity of the VRF using the proof w.

For the correctness and security of VRF, we refer to [MRV99, DY05].

2.2 Terminology and System Model

Terminology In DEXON, an *user* is uniquely identified by its public key and secret key pair. All users can transfer and receive the stakes from other users. They can also verify the correctness of the blockchain. Our blockchain is maintained by a special set of users S_{node} where the members of S_{node} are called *nodes.* We define two special sets of nodes: the *CRS set* S_{CRS} and the *notary set* S_{notary} (a node can be in two sets at the same time). The nodes in S_{CRS} and S_{notary} are crucial for our single chain algorithm, which will be explained in Section 3.

A node can pack a batch of transactions into a *block.* In our single chain algorithm, we say a node *proposes* a block if the block is a candidate that can be selected by our Byzantine agreement protocol. We say a node *issues* a block if the block is selected and becomes a block in the single chain.

System Model We assume the adversary can corrupt the nodes *adaptively.* That is, the adversary can choose which nodes are corrupted during the protocol. The corrupted nodes are called *Byzantine* and the nodes that are not corrupted are called *correct.* A Byzantine node can deviate from the protocol arbitrarily; it can engage in problematic malfunctions such as sending conflicting messages, violating algorithm criteria, delaying the messages between other nodes, and so on. We also assume the adversary has full control of the network. The adversary can learn all the messages delivered on the network and determine the delay and the order of the delivered messages.

If the adversary does not delay any messages between the correct nodes, we assume the network is *weakly-synchronous.* That is, there exists a known *time bound* λ for the messages between any two correct nodes. We say the network is *partitioned* if the messages between the correct nodes are delayed such that the delivering time exceeds λ.

3 Single Chain

In this section, we introduce how a set of nodes build an agreed single chain. In short, the members in S_{CRS} generate a public randomness. Then, for each height of the single chain, the members in S_{notary} propose their blocks. Then, they try to reach an agreement regarding who is the leader for that height and all nodes adopt the leader's proposal for the next block. As long as all the correct nodes can agree on who is the leader for each height, they can build up an agreed blockchain.

In Section 3.1, we introduce two useful primitives for Byzantine agreement protocol: *common reference string* and *verifiable random function.* In Section 3.2, we introduce our Byzantine agreement protocol based on leader election. Finally, we formally describe how to build a single chain in Section 3.3.

3.1 Common Reference String and Verifiable Random Function

In this section, we introduce two useful primitives for building a single chain: common reference string (CRS) and verifiable random function (VRF).

Common Reference String An *epoch* consists of a specific number of blocks. The CRS in our setting is actually a public randomness generated by a deterministic algorithm for each epoch; no user in the system can predict the CRS of any future epoch. In our setting, the CRS of epoch i is updated by $R_i = Hash(TSig(R_{i-1}))$, where $TSig(\cdot)$ is a threshold signature function whose input is some set of share-signatures produced by the nodes in S_{CRS} of epoch $i-1$.

This method is similar to DFinity [HMW], but their system categorizes the group when users join the system. We emphasize that the users in their system have a high incentive to be malicious because each group has non-negligible probability to have the right to propose a block and compute the randomness. However, our system selects CRS nodes to compute the randomness according to results from a previous epoch.

Verifiable Random Function The verifiable random function introduced by Micali, Rabin and Vadhan [MRV99] is a type of pseudorandom function by which anyone can verify the validity of the function evaluation from public information. Several practical VRFs have been proposed [DY05, GHKW17] and most of them are based on bilinear functions. Algorand [GHM+17] and Ouroboros Praos [DGKR18] have demonstrated that VRF is a powerful primitive for achieving cryptographic sortition in blockchains. Our system works with PKI; each user can access all other users' public keys (preregistered in previous blocks) and user j computes the verifiable random function as:

$$\left|R_i - Hash\left(Sig_{sk_j}(x)\right)\right|, \tag{1}$$

where R_i is CRS of epoch i and x is public information. Thus, $Hash(Sig_{sk_j}(x))$ is verifiable with $user_j$'s public key and unpredictable.

Our VRF has three benefits compared to the VRF in Algorand. First, our design is fairer. The verifiable random function used in Algorand is $Hash(Sig_{sk_j}(Q_{i-1}, x))$, where Q_{i-1} is the randomness from the previous block. Whether an adversary can choose to propose a block depends on the randomness of that adversary's block. If the randomness is beneficial for Byzantine nodes (e.g. higher probability of proposing the next block), then the adversary proposes the block. Thus, the overall advantage of the adversary increases up to $(1/3)/(1 - 1/3) = 1/2$. The main problem is that the proposer decides the block and the randomness at the same time. Therefore, we separate the permission of proposing a block and generating the randomness in order to avoid such bias attacks. Second, our design is more flexible to compute because each user can compute part of the VRF for any status at any time, say, $Hash\left(Sig_{sk_j}(x)\right)$. At the beginning of each epoch, any user can get the CRS of the epoch and compute the probability of proposing a block in the epoch. However, Algorand's VRF requires Q_{i-1} to compute Q_i. Third, our design has better space consumption. In Algorand, each block must store the randomness for VRF, but our design uses the same randomness for many blocks in one epoch. Thus, the space complexity is reduced by a constant.

A further optimization is only to update CRS when new users join. Then, the space complexity is reduced to a constant value that is independent of the number of blocks.

3.2 Byzantine Agreement

The *Byzantine general problem* was introduced by Lamport, Shostak, and Pease [LSP82]; it allows a set of nodes to agree on a single bit $b \in \{0, 1\}$ where some of the nodes may be malicious. Recently, Chen et al. [CGMV18] proposed a Byzantine agreement protocol based on leader election, which achieves fast agreement in a synchronous network, upholds safety in an asynchronous network, and recovers from any partition rapidly.

Now, we introduce our Byzantine agreement protocol, which is based on Algorand's protocol but includes some minor improvements. In DEXON, each single chain is maintained by a notary set S_{notary}. Thus, only the members in S_{notary} are involved in the Byzantine agreement protocol. Let n be the number of nodes in S_{notary} and t be the number of Byzantine nodes in S_{notary}.

Comparing to Algorand, our leader election procedure is independent of the round index, so the nodes are not required to propose their values at each round. Consequently, except for the first round, the running time of each round reduces to 2λ. Also, if no partition is present, our protocol terminates in t rounds in the worst case and can be expected to terminate in $\frac{7}{4}$ rounds.

Let $t_{\max} = \lfloor (n-1)/3 \rfloor$, R_i denote the CRS at epoch i and V denote the set of values that can be decided. We also define two special values $\perp$ and SKIP that are not in V. For each node $q \in S_{\text{notary}}$, q has four internal variables: r_q records the index of the round at which q is working, $\mathsf{lockvalue}_q$ records the candidate value that q supports, $\mathsf{lockround}_q$ records the index of the round from which $\mathsf{lockvalue}_q$ comes and clock_q is q's local clock. Let sk_q and pk_q denote the secret key and public key of q, respectively. We define status to be the public predictable information of the block (e.g. shard ID, chain ID, block height). For the leader election, each node $q \in S_{\text{notary}}$ computes the signature $\sigma_q = Sig_{sk_q}(\mathsf{status})$ with its secret key sk_q. We define three kinds of messages:

1. the *initial message* of the node q: $(\text{“}\mathsf{init}\text{”}, v_q, q, \sigma_q)$

2. the *pre-commit message* of the value v from the node q at the round r: $(\text{“}\mathsf{pre\text{-}com}\text{”}, v, q, r)$

3. the *commit message* of the value v from the node q at the round r: $(\text{“}\mathsf{com}\text{”}, v, q, r)$

With these notations, we introduce the leader election algorithm which will be a subroutine of our Byzantine agreement protocol. Let M_q denote the set of initial messages that the node q receives from other nodes. The node q verifies the signatures in M_q and sets U_q to be the set of nodes whose signatures are valid. Then, q computes

$$\ell_q = \underset{j \in U_q}{\operatorname{argmin}} \left| R_i - Hash\left(\sigma_j\right) \right| .$$

We say ℓ_q is the *leader* of q.

Our Byzantine agreement protocol (Algorithm 1) is a round-based protocol. Initially, for all correct nodes $q \in S_{\text{notary}}$, q initializes its internal variables by $r_q = 1$, $\mathsf{lockvalue}_q = \perp$, $\mathsf{lockround}_q = 0$ and $\mathsf{clock}_q = 0$ and also chooses its initial value $v_q \in V$.

Our protocol has four steps in each round. At Step 1, all the nodes gossip their own initial value v_q in the format $(\text{“}\mathsf{init}\text{”}, v_q, q, \sigma_q)$.

When $\mathsf{clock}_q = 2\lambda$, q enters Step 2. If $\mathsf{lockvalue}_q = \perp$, q verifies the initial messages it receives and computes the set U_q of nodes whose signatures are valid. If $U_q \neq \emptyset$, q identifies its leader ℓ_q and pre-commits ℓ_q's value; otherwise, q pre-commits $\perp$. If $\mathsf{lockvalue}_q \neq \perp$, node q pre-commits $\mathsf{lockvalue}_q$. We say node q pre-commits on a value v if node q gossips the message $(\text{“}\mathsf{pre\text{-}com}\text{”}, v, q, r)$ where r is the round number that node q is working at.

When $\mathsf{clock}_q = 4\lambda$, node q enters Step 3. If node q has seen $2t_{\max} + 1$ pre-commit messages of the same value $v \in V \cup \{\perp\}$ at round r_q, node q updates $\mathsf{lockvalue}_q = v$ and $\mathsf{lockround}_q = r_q$ and commits v. Otherwise, node q commits SKIP. Note that node q must commit some value at Step 3. We say node q commits on a value v if node q gossips the message ("com", v, q, r) where r is the round number that q is working at. After node q gossips the commit message, q enters Step 4, at which q waits for the forward conditions.

Suppose a node q is working at round r_q. The node q updates its internal variables as soon as one of the following conditions holds:

1. If node q has seen $2t_{\max} + 1$ pre-commit messages of the same value $v \in V \cup \{\perp\}$ at the same round r such that $r_q \geq r > \mathsf{lockround}_q$, q sets $\mathsf{lockvalue}_q = v$ and $\mathsf{lockround}_q = r$.

2. (forward condition) If node q has seen $2t_{\max} + 1$ pre-commit messages of the same value $v \in V \cup \{\perp\}$ at the same round r such that $r > r_q$, q sets $\mathsf{clock}_q = 2\lambda$, $\mathsf{lockvalue}_q = v$, $\mathsf{lockround}_q = r$ and starts the round r from Step 2.

3. (forward condition) If the node q has seen $2t_{\max} + 1$ commit messages of any value at the same round r such that $r \geq r_q$, q sets $\mathsf{clock}_q = 2\lambda$ and starts the round $r + 1$ from Step 2.

We say that node q achieves the *forward condition*, if the condition 2 or the condition 3 holds. Node q goes into the next round immediately if it achieves the forward condition even if it does not achieve the forward condition at Step 4.

Node q *decides* on a value v as soon as node q has seen $2t_{\max} + 1$ commit messages of the same value $v \in V \cup \{\perp\}$ at the same round r.

The protocol for a node q is summarized as Algorithm 1.

```
Procedure DEXON_Byzantine_Agreement for node q
    Input  : an initial value $v_q \in V$ from node $q$ and the public key $\{pk_q\}$ from all nodes
    Output: an agreed value $v_{fin} \in V \cup \{\perp\}$ from some node
    Initialize $r_q = 1$, $\mathsf{lockvalue}_q = \perp$, $\mathsf{lockround}_q = 0$ and $\mathsf{clock}_q = 0$
    Step 1: when $\mathsf{clock}_q = 0$,
        gossip("init", $v_q, q, \sigma_q$)
    Step 2: when $\mathsf{clock}_q = 2\lambda$,
        if $\mathsf{lockvalue}_q = \perp$ and $U_q \neq \emptyset$ then
            node $q$ identifies its leader $\ell_q$ at $q$'s current view
            gossip("pre-com", $v_{\ell_q}, q, r_q$)
        else if $\mathsf{lockvalue}_q = \perp$ and $U_q = \emptyset$ then
            gossip("pre-com", $\perp, q, r_q$)
        else
            gossip("pre-com", $\mathsf{lockvalue}_q, q, r_q$)
    Step 3: when $\mathsf{clock}_q = 4\lambda$,
        if node $q$ has seen $2t_{max} + 1$ pre-commit messages of the same value $v \in V \cup \{\perp\}$ at round $r_q$ then
            $\mathsf{lockvalue}_q = v$
            $\mathsf{lockround}_q = r_q$
            gossip("com", $v, q, r_q$)
        else
            gossip("com", SKIP, $q, r_q$)
    Step 4: when $\mathsf{clock}_q \in (4\lambda, \infty)$
        wait until the forward condition is achieved
```

Algorithm 1: Synchronized Byzantine Agreement Algorithm

3.2.1 Agreement

We first show that our protocol will reach agreement; that is, all the correct nodes will decide on the same value.

Lemma 3.1. *Assume $t \leq t_{max}$. Suppose a node p receives $2t_{max} + 1$ commit messages of v_p and another node q receives $2t_{max} + 1$ commit messages of v_q. If both these $2t_{max} + 1$ commit messages all come from the round r, then $v_p = v_q$.*

Proof. We prove this lemma by contradiction. Suppose $v_p \neq v_q$. Because as many as t Byzantine nodes exist, there exists at least one correct node that both commits on v_p and v_q by the pigeonhole principle. However, correct nodes can only commit on one value at one round, which leads to a contradiction. □

Theorem 3.2 (Agreement). *Assume $t \leq t_{max}$. Regardless of partition, if a correct node p decides on some value v_p and a correct node q decides on some value v_q, then $v_p = v_q$. That is, the correct nodes will never decide on different values.*

Proof. Because p decides on v_p and q decides on v_q, p and q must see $2t_{\max}+1$ commit messages of v_p and $2t_{\max}+1$ commit messages of v_q, respectively. Suppose both these $2t_{\max}+1$ commit messages come from the same round r. By Lemma 3.1, we have $v_p = v_q$.

Suppose the $2t_{\max}+1$ commit messages that p receives come from the round r_p and the $2t_{\max}+1$ commit messages that q receives come from the round r_q. Without loss of generality, we assume $r_p < r_q$. Because there are up to $t_{\max}$ Byzantine nodes, there must be at least $t_{\max}+1$ correct nodes commit on v_p so that p can receive $2t_{\max}+1$ commit messages of v_p. For all rounds $r > r_p$, these $t_{\max}+1$ correct nodes will always pre-commit on v_p until they see $2t_{\max}+1$ pre-commit messages of $v' \neq v_p$ at Step 3. However, only $2t_{\max}$ nodes remain, so these $t_{\max}+1$ correct nodes will never pre-commit any $v' \neq v_p$ for all $r > r_p$. Thus, for all $r > r_p$, if some value v has $2t_{\max}+1$ pre-commit messages, then $v = v_p$.

Because q receives $2t_{\max}+1$ commit messages of v_q, there must exist at least $t_{\max}+1$ correct nodes that commit on v_q at the round r_q. These $t_{\max}+1$ correct nodes commit on v_q only if they have seen $2t_{\max}+1$ pre-commit messages of v_q at round r_q. Therefore, $v_q = v_p$. □

3.2.2 Termination

We now analyze when the algorithm terminates if no partition exists or if the system recovers from a previous partition.

Proposition 3.3 (Termination without partition). *Assume $t \le t_{max}$. If all the correct nodes start at the r-th round within time λ and no partition exists, all the correct nodes will decide on some values in $t+1$ rounds.*

Proof.
Case 1: Some correct node has decided. If a correct node p has decided on value v_p, p must have seen $2t_{\max}+1$ commit messages of v_p. Because p propagates these $2t_{\max}+1$ commit messages, all the correct nodes will hold this information after time λ and decide on v_p in one round.
Case 2: Some correct node has seen $2t_{max}+1$ pre-commit messages. Suppose no node has decided but there exists a correct node p that has seen $2t_{\max}+1$ pre-commit messages of a value v_p. Because p propagates these $2t_{\max}+1$ pre-commit messages, all the correct nodes will hold this information after time λ. Thus, all the correct nodes will commit v_p so that they can reach an agreement on v_p.
Case 3: No correct node has seen $2t_{max}+1$ pre-commit messages. Because no correct node has ever seen $2t_{\max}+1$ pre-commit messages, $\mathsf{lockvalue}_q = \perp$ for all correct node q. Thus, they will identify their own leader by their local view and by the leader's pre-commit value. Because all correct nodes start at the r-th round within time λ, they can receive all the initial values from other correct nodes before identifying the leaders. Thus, there exists some correct nodes that pre-commit different values relative to each other only if a Byzantine node proposes different initial values to different nodes[1]. However, the correct nodes will propagate the initial value so all correct nodes will have the same set of initial values after time λ. Thus, to prevent the correct nodes from agreeing on the same leader, Byzantine nodes must propose different initial values to different nodes at every round. However, a node can only propose an initial value once or it will be caught. Thus, the best strategy of Byzantine nodes is that different Byzantine nodes propose their initial values at different rounds so t Byzantine nodes can only interfere during t rounds. Thus, all the correct nodes will decide on some values in $t+1$ rounds with certainty. □

Proposition 3.4. *Assume $t \le t_{max}$. Suppose all correct nodes start at r-th round within time λ and no partition exists. Then, it is expected that all correct nodes will decide on some values in $\frac{7}{4}$ rounds.*

Proof. From the proof of Proposition 3.3, we know that if some correct node has decided on a value v or has seen $2t_{\max}+1$ pre-commit messages of a value v, then all the correct nodes will decide on v in one round.

In a network without partition, the best strategy for the Byzantine nodes has been described in Case 3 in the proof of Proposition 3.3. However, to interfere with k rounds successfully, the Byzantine nodes must win the leadership [2] in the following k rounds. The probability of such an event is

$$\prod_{i=0}^{k-1}\left(\frac{t-i}{n-i}\right) \le \left(\frac{t}{n}\right)^k .$$

Thus, in expectation, the number of interfered rounds can be upper-bound by

$$\sum_{i=1}^{t}\left(\frac{t}{n}\right)^i \cdot i \le \frac{\frac{t}{n}}{(1-\frac{t}{n})^2}.$$

[1] Note that not proposing any initial value is considered to be equivalent to proposing $\perp$.

[2] Notice that the leader is the node j with the lowest value $|R_i - Hash\,(\sigma_j)|$.

Because $n \geq 3t_{\max} + 1 \geq 3t + 1$, the expected number of interfered rounds is less than $\frac{3}{4}$. After an interfered round, all the correct nodes will decide in the next round, so it is to be expected that they will all terminate in $\frac{7}{4}$ rounds. □

Proposition 3.5 (Termination when the partition is resolved)**.** *Assume $t \leq t_{max}$. If the partition is resolved, all the correct nodes will decide on some values in $t+2$ round. It is to be expected that all correct nodes will decide on some values in $\frac{11}{4}$ rounds.*

Proof. If there exists a node p that has decided on a value v_p, p must have seen $2t_{\max} + 1$ commit messages of v_p. All the correct nodes will receive these $2t_{\max} + 1$ commit messages of v_p within time λ after the partition is resolved and decide on v_p.

Suppose no node has decided and p is the node working on the latest round r_p. To enter the round r_p, p must achieve the forward condition at round $r_p - 1$. Because the partition is resolved, all correct nodes will also achieve the forward condition within time λ after the partition is resolved and also enter the round r_p. Later on, if some node q achieves the forward condition and enters the round $r_p + 1$, other correct nodes will also achieve the forward condition within time λ. Thus, all correct nodes start at the round $r_p + 1$ with time difference $< \lambda$ and Proposition 3.3 guarantees that they will decide on some values within the following $t+1$ rounds. Similarly, it is to be expected that all correct nodes will decide on correct values in $\frac{11}{4}$ rounds. □

3.2.3 Communication Complexity

We now analyze the communication complexity of a single node for a single round. Because a correct node will help to propagate the messages, all correct nodes will gossip $\mathcal{O}(n)$ messages in a single round. Thus, the communication complexity for all nodes is $\mathcal{O}(n^2)$ in a single round.

As discussed in Section 3.2.2, if no partition exists or the system recovers from a partition, our Byzantine agreement protocol terminates in $t+1$ rounds in the worst case and is expected to terminate in $\frac{11}{4}$ rounds. We assume $n \geq 3t+1$, so the protocol terminates in $\mathcal{O}(n)$ rounds in the worst case and is expected to terminate in $\mathcal{O}(1)$ rounds. Therefore, the total communication complexity of the protocol is $\mathcal{O}(n^3)$ in the worst case and $\mathcal{O}(n^2)$ in the expected case.

3.3 Consensus on a Single Chain

In this section, we describe how to build up a single chain by Byzantine agreement protocol. Our single chain protocol has four parts: system setup, registering public key, proposing block, notarizing blocks and updating epoch.

Overview The members in S_{CRS} are responsible for updating the CRS. The members in S_{notary} have right to propose a block and are responsible for running the Byzantine agreement protocol to decide whose block can be chosen for the single chain.

When the members in S_{notary} reach an agreement of a block, they sign their respective signature shares of the agreement. The block proposer of the next block has to verify these signature shares and combine them into a threshold signature Σ_h. The threshold signature Σ_h should be stored in the next block. If someone tampers the content of a block, the hash of the block changes and become inconsistent with the threshold signature in the next block. Because it is infeasible to forge the threshold signature, the integrity of the blocks is guaranteed.

The members in S_{CRS} and S_{notary} are re-elected for each epoch. Each epoch i corresponds to a randomness R_i of CRS. When S_{CRS} is re-elected, the members in S_{CRS} will update the CRS for the next epoch.

Registering Public Key Any user with enough deposit can register its own public key in the chain. The users who want to enter or leave S_{node} can announce the request. The request will be recorded in the blocks. The node is called *valid* if it owns enough deposit[3] and has registered a valid public key. Let $view^i$ denote the snapshot of the whole system at the first block of the epoch i. Let S^i_{node} denote the set of all valid nodes in $view^i$. The set S^{i+1}_{node} can be updated by S^i_{node} and the blocks in the epoch i.

Proposing Blocks Only the nodes in S_{notary} can propose new blocks. To propose a block, a node q computes the hash v_q of the block and takes v_q as the initial value for the Byzantine agreement protocol. Note that node q can send the hash of the block to join the BA scheme as early as possible, but has to propose the complete block immediately so that other nodes can determine the block and commit it.

A node is allowed not to propose any block. Because the leader election is determined by the public predictable information status, R_i, and the secret keys, the node q can compute the value $\left|R_i - Hash\left(Sig_{sk_q}(\mathsf{status})\right)\right|$

[3] To avoid nothing-at-stake problem, the node who wants to register its public key needs to lock its deposit. In practice, this can be done by smart contracts.

locally. If the value is too large, q is unlikely to be chosen as the leader so it may give up the chance to propose the block.

Practically, we can set a threshold δ of proposing a block. That is, the node q will propose a block if and only if $\left|R_i - Hash\left(Sig_{sk_q}(\mathsf{status})\right)\right| \leq \delta$. Suppose n is the number of nodes in S_{notary}. Then, the probability that at least one node proposes a block is $1-(1-\delta)^n$. Choosing $\delta = 20/n$, the probability is larger than $1-10^{-8}$. If no one proposes a block, all the correct nodes will pre-commit $\perp$ and they will soon decide on $\perp$. Thus, the protocol will not halt. All the correct nodes can agree on an empty block and start the protocol for the next block soon.

Notarizing Blocks The nodes in S_{notary} are responsible for running the Byzantine agreement protocol (Algorithm 1) to decide the next block of the chain. As shown in Section 3.2, as long as there is no partition, the members in S_{notary} can reach an agreement even if they do not start within time λ.

To notarize a block, we choose a $(n, t_{\max}+1)$-threshold signature scheme. When a node $q \in S_{\text{notary}}$ decides on some value v for the block with height h, node q signs a signature share σ_q by its secret key SK_q[4] and announces σ_q. The block proposers of the next block (of height $h+1$) have to verify at least $t_{\max}+1$ signature shares for the block with height h and combine the signature shares into a threshold signature Σ_h. The threshold signature Σ_h should be packed into the next block. We say a block with height h is *notarized* if its threshold signature Σ_h is packed in the next block and the next block is issued.

Assume the number of Byzantine nodes is up to $t_{\max}$. Then, anyone can make sure the block with the hash v is notarized by S_{notary} if more than $t_{\max}+1$ nodes in S_{notary} have signed on v. To verify the fact, we only need to check whether the block hash v matches the threshold signature Σ_h stored in the next block and verify whether Σ_h is valid or not.

Updating Epoch The members in S_{CRS} and S_{notary} are re-elected for each epoch. Let S^i_{CRS} and S^i_{notary} denote the CRS set and the notary set of the epoch i.

The set S^i_{CRS}, the set S^i_{notary}, and the randomness R_i are decided before the epoch i starts. When the epoch i starts, the members in S^i_{CRS} generate the randomness R_{i+1} by

$$R_{i+1} = Hash\left(TSig(R_i)\right),$$

where $TSig(R_i)$ is the threshold signature signed by the nodes in S^i_{CRS}. After R_{i+1} is decided, the set S^{i+1}_{CRS} and the set S^{i+1}_{notary} can be elected by Fisher-Yate shuffle with the randomness R_{i+1} from the set S^i_{node}. Finally, the members in S^{i+1}_{CRS} and S^{i+1}_{notary} run the key generation algorithm of the threshold signature scheme, respectively.

The timeline of building a single chain is summarized in the Figure 1.

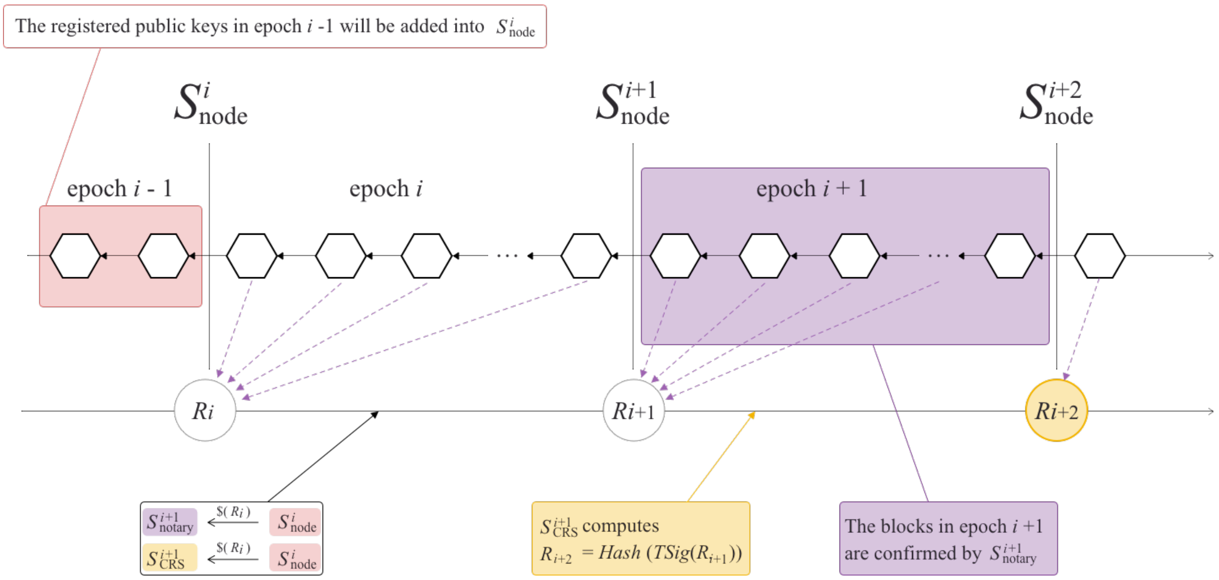
The registered public keys in epoch i -1 will be added into S^i_{node}
S^i_{node}
S^{i+1}_{node}
S^{i+2}_{node}
epoch i - 1
epoch i
epoch i + 1
Ri
Ri+1
Ri+2
S^{i+1}_{notary}
$(R_i)
S^i_{node}
S^{i+1}_{CRS}
$(R_i)
S^i_{node}
S^{i+1}_{CRS} computes
R_{i+2} = $Hash$ $(TSig(R_{i+1}))$
The blocks in epoch i +1
are confirmed by S^{i+1}_{notary}

Figure 1: The consensus of a single chain

[4] Note that the key pair (PK_q, SK_q) for the threshold signature is different from the key pair (pk_q, sk_q) that the node q registers on the block.

4 Blocklattice

In DEXON, the blocklattice arises from numerous single chains and the ack information in the blocks of those chains. We introduce the total ordering algorithm on the blocklattice to produce a globally-ordered chain, called the *compaction chain*.

In this section, we first present our main algorithm to compact the proposed blocklattice into the compaction chain and then to generate the timestamp for each block in the compaction chain. For the compaction, we introduce a basic total ordering algorithm in Section 4.2. In Section 4.3, we present our timestamping algorithm that ensures the unbiased consensus timestamp of blocks in the blocklattice, which is close to real-world time. We present the consensus on a blocklattice in Section 4.4.

4.1 Notation

Let $\mathcal{N}$ be the set of chains and $\mathcal{F}$ be the set of chains which contains the blocks from Byzantine nodes in the DAG, whose maximum capacity is limited by $f_{max} = \lfloor (|\mathcal{N}| - 1)/3 \rfloor$. Let $\mathcal{P}$ denote the set of pending blocks, which means the block is received by a node and has not been output into the compaction chain.

When a block A follows another block B, we call this relation A *acks* B. Let $\mathcal{C}$ be the set of candidate blocks that only ack the blocks that are already in the compaction chain. Informally, $\mathcal{A}$ denotes the set of preceding candidate blocks that have higher priority than other candidate blocks, which corresponds to the "source message" in the TOTO protocol [DKM93].

We use the subscript $(\cdot)_p$ to denote a set or function in p's local view, for example, $\mathcal{C}_p$ is the candidate set in p's local view. We use $b_{q,i}$ to denote the i-th block from node q. We define *indirect ack* by the transitive law of ack: if a block A acks another block B through the transitive law of ack, then we say A indirectly acks B. For example, if A acks B and B acks C, A indirectly acks C. An example is provided in Figure 2.

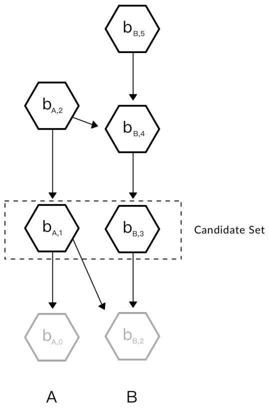

Figure 2: A simple example illustrates the notation: $\mathcal{N} = \{A, B\}$, $\mathcal{C} = \{b_{A,1}, b_{B,3}\}$. Also, $b_{A,2}$ acks $b_{A,1}$ and $b_{B,4}$. Both $b_{A,2}$ and $b_{B,5}$ indirectly ack $b_{B,3}$.

Next, we present the definition of specifications of our total ordering algorithm as the following:

1. *Correctness*: all non-Byzantine nodes will eventually generate the same ordered chain.
2. *Liveness*: the system will not halt with up to f_{max} Byzantine nodes, and every block will eventually be finalized.

For a formal definition, we follow the specifications in [DSU04].

4.2 DEXON Total Ordering Algorithm

First, we introduce the DEXON total ordering algorithm, which ensures that the blocklattice data structure can be compacted into the compaction chain. The DEXON total ordering algorithm is described with a parameter $|\mathcal{N}| \geq \Phi > |\mathcal{N}|/2$; it can be seen as the threshold of the output criterion. We set $\Phi = 2f_{max} + 1$ to guarantee the liveness of the total ordering algorithm. In this scenario, a problem arises: each node receives all block information asynchronously, thus, the order of blocks from the blocklattice cannot be decided directly by the receiving times. Therefore, we introduce the DEXON total ordering algorithm; it is a symmetric algorithm and it outputs the total order for each valid block.

The DEXON total ordering algorithm is based on [DKM93], which is a weak total order algorithm [DSU04]. The main idea of the DEXON total ordering algorithm is to dynamically maintain a directed acyclic graph

(DAG) from the received blocks. More precisely, each vertex corresponds to some block and each edge corresponds to some ack relation between blocks. Intuitively, once a block in the graph obtains enough acks from other blocks, the algorithm outputs the block.

Before we state the algorithm, we will first introduce some functions and properties for the algorithm. We use the following three functions as potential functions that evaluate the quality of each candidate block in order to decide the output order.

1. *Acking Node Set*, $ANS_p(b) = \{\, j : \exists b_{j,k} \in \mathcal{P}$ s.t. $b_{j,k}$ directly or indirectly acks $b\,\} \cup \{p\}$, is the set of nodes that issued blocks ack block b in node p's local view. Moreover, the global Acking Node Set ANS_p for node p is defined by $\bigcup_{b \in \mathcal{C}_p} ANS_p(b)$.

2. $AHV_p(b)$ is the *Acking block-Height Vector* of block b in node p's view, defined as:

$$AHV_p(b)[q] = \begin{cases} \bot, & \text{if } q \notin ANS_p \\ k, & \text{if } b_{q,k} \text{ acks } b, \text{ where } k = \min\{i : b_{q,i} \in \mathcal{P}\} \\ \infty, & \text{otherwise.} \end{cases}$$

3. $\#AHV_p = |\,\{\, j : AHV_p(b)[j] \neq \bot$ and $AHV_p(b)[j] \neq \infty \,\}\,|$ is the number of integer elements in $AHV_p(b)$.

Next, we define two functions,

$$Precede_p(b_1, b_2) = \begin{cases} 1, & \text{if } |\{j : AHV_p(b_1)[j] < AHV_p(b_2)[j]\}| > \Phi \\ -1, & \text{if } |\{j : AHV_p(b_1)[j] > AHV_p(b_2)[j]\}| > \Phi \\ 0, & \text{otherwise,} \end{cases}$$

and

$$Grade_p(b_1, b_2) = \begin{cases} 1, & \text{if } Precede_{p+}(b_1, b_2) = 1 \\ 0, & \text{if } |\{j : AHV_{p+}(b_1)[j] < AHV_{p+}(b_2)[j]\}| < \Phi - (n - |ANS_p|) \\ \bot, & \text{otherwise,} \end{cases}$$

where $(\cdot)_{p+}$ is the local view of p in the future. That is, the pending set becomes $\mathcal{P}_p \cup \mathcal{S}_p$, where $\mathcal{S}_p$ consists of all the probable blocks received by p in the future. The $Grade_p(b_1, b_2)$ function outputs the three possible relations between block b_1 and b_2 in the future: the first possibility is $Grade_p(b_1, b_2) = 1$, which means block b_1 always precedes block b_2 regardless of arbitrary following input. The second possibility is $Grade_p(b_1, b_2) = 0$ which means block b_1 cannot precede block b_2 regardless of arbitrary following input. The last possibility comprises all other relations.

We say a candidate block b is *preceding* if $\forall\, b' \in \mathcal{C}_p$, $Grade_p(b', b) = 0$, and the *preceding set* $\mathcal{A}$ is the set of all preceding blocks, which means that such blocks have a relatively high priority to be output by the algorithm. Thus, regardless of which blocks are received afterwards, the blocks in $\mathcal{C} \setminus \mathcal{A}$ cannot precede the blocks in $\mathcal{A}$.

Now, we present the DEXON total ordering algorithm, which is a symmetric algorithm. The DEXON total ordering algorithm is an event-driven online algorithm. The input is one block per iteration and the algorithm produces blocks when specific criteria are satisfied. Regardless of the order of the inputs, the algorithm is executed by each node individually, and the algorithm outputs blocks in the same order. The only requirement regarding input is that the set (including acking information) of input for each node is the same eventually. The criteria consist of two parts: the *internal stability* and the *external stability.* Informally, internal stability ensures each block in the preceding set has the highest priority to be output compared with all other blocks in the candidate set, and external stability ensures that each block in the preceding set always has higher priority to be output irrespective of the type of blocks received. Let n be the number of total nodes. The *criteria* to output the preceding set $\mathcal{A}_p$ for node p is defined as follows:

- Internal Stability: $\forall b \in \mathcal{C}_p \setminus \mathcal{A}_p, \exists b' \in \mathcal{A}_p$ s.t. $Grade_p(b', b) = 1$
- External Stability:
  - (a) $|ANS_p| = n$, or
  - (b) $\exists b \in \mathcal{A}_p$ s.t. $\#AHV_p(b) > \Phi$ and
    $\forall b \in \mathcal{A}_p$, $|ANS_p(b)| \geq n - \Phi$

We call the output *normal delivery* if it satisfies internal stability and external stability (a), and *early delivery* if it satisfies internal stability and external stability (b).

Next, we simplify two *criteria* by the following theorems.

Theorem 4.1 (Simplifying Criterion for normal delivery)**.** *External Stability (a) implies Internal Stability. That is, if* $|ANS_p| = n$*, then,* $\forall b \in \mathcal{C}_p \setminus \mathcal{A}_p, \exists b' \in \mathcal{A}_p$ *s.t.* $Grade_p(b', b) = 1$*.*

Proof. First, we prove $\mathcal{A}_p \neq \emptyset$ by contradiction. Assume $\mathcal{A}_p = \emptyset$. By the definition of a preceding set, $\forall b \in \mathcal{C}_p, \exists b' \in \mathcal{C}_p$ s.t. $Precede_p(b', b) = 1$. However, by the definition of $Precede_p$ function, $b' \in \mathcal{A}_p$, contradiction.
Second, assume $\exists b' \in \mathcal{C}_p \setminus \mathcal{A}_p, \forall b \in \mathcal{A}_p$ s.t. $Precede(b, b') \neq 1$. Case 1: $\forall b'' \in \mathcal{C}_p \setminus \mathcal{A}_p$, $Precede(b'', b') \neq 1$, then $b' \in \mathcal{A}_p$, contradiction. Case 2: $\exists b'' \in \mathcal{C}_p \setminus \mathcal{A}_p$, $Precede(b'', b') = 1$, then $b'' \in \mathcal{A}_p$, contradiction.
Third, if $\mathcal{A}_p = \mathcal{C}_p$, it is easy to check the correctness for the theorem. □

Theorem 4.2 (Simplifying Criterion for early delivery)**.** *If both internal stability and* $\mathcal{C}_p \setminus \mathcal{A}_p \neq \emptyset$ *hold, then, external stability (b) also holds.*

Proof. It is easy to prove $\mathcal{A} \neq \emptyset$. Let $b' \in \mathcal{A}$ be the block that precedes a block, then $\#AHV_p(b') > \Phi$ holds. Next, assume $\exists b \in \mathcal{A}$ s.t. $|ANS_p(b)| < n - \Phi$. Then, $\#AHV_p(b) < n - \Phi$ holds. $|\{j | AHV_p(b')[j] < AHV_p(b)[j]\}| > \Phi - (n - \Phi) = 2\Phi - n$ and for the case $|ANS_p| > \Phi$. When the blocks issued by all nodes have been received, $|\{j | AHV_p(b')[j] < AHV_p(b)[j]\}|$ can reach more than $(2\Phi - n) + (n - \Phi) = \Phi$. This means b is preceded by b'. Thus, $b \notin \mathcal{A}$, contradiction. Therefore, $\forall b \in \mathcal{A}$ s.t. $|ANS_p(b)| \geq n - \Phi$. □

Hence, the *criteria* are simplified into the following conditions:

- Normal Delivery: $|ANS_p| = n$
- Early Delivery: $\forall b \in \mathcal{C}_p \setminus \mathcal{A}_p, \exists b' \in \mathcal{A}_p$ s.t. $Grade_p(b', b) = 1$ and $\mathcal{C}_p \setminus \mathcal{A}_p \neq \emptyset$

```
Procedure DEXON_Total_Ordering for node p
    Input  : a block $b_{q,i}$ from node q and its acking information per iteration
    Output: ordered block series
    when receiving a block $b_{q,i}$,
        if $b_{q,i}$ only acks the blocks that have been output, then
            $\mathcal{C}_p = \mathcal{C}_p \cup \{b_{q,i}\}$
            foreach $r \in ANS_p$ do
                $AHV_p(b_{q,i})[r] = \infty$
        if $q \notin ANS_p$, then
            foreach  $b \in \mathcal{C}_p$ do
                if b is directly or indirectky acked by $b_{q,i}$, then
                    $AHV_p(b)[q] = i$,
                    $ANS_p(b) = ANS_p(b) \cup \{q\}$
                else
                    $AHV_p(b)[q] = \infty$
    end;
    when $criteria_p$ holds,
        output $\mathcal{A}_p$ in the lexicographical order of the hash value of each block
        //update $\mathcal{A}_p$ and $\mathcal{C}_p$
        $\mathcal{C}_p = \mathcal{C}_p \setminus \mathcal{A}$
        $\mathcal{A}_p \leftarrow \emptyset$
        foreach $b \in \mathcal{P}_p$ only acks the blocks that have been output do
            $\mathcal{C}_p = \mathcal{C}_p \cup \{b\}$
        foreach $b \in \mathcal{C}_p$ do
            compute $AHV_q(b), ANS_q(b)$
        foreach $b \in \mathcal{C}_p$ do
            if b is preceding, then
                $\mathcal{A}_p = \mathcal{A}_p \cup \{b\}$
    end;
```

Algorithm 2: DEXON Total Ordering Algorithm

The DEXON total ordering algorithm is listed as Algorithm 2. There are two events that must be considered: one is a block received and the other is that some criteria are satisfied. When a block $b_{q,i}$ is received, the algorithm updates its potential function of candidate blocks according to the ack information from $b_{q,i}$. If $b_{q,i}$ only acks the blocks that have been output, it is a candidate block. Then, the algorithm updates $AHV_p(b_{q,i})$ according to ANS_p. Otherwise, $b_{q,i}$ is not a candidate. If node q has never been in node p's view, each candidate block updates its potential function. If node q has been in node p's view, there must exist a $j < i$ s.t. $b_{q,j} \in \mathcal{P}$. Thus, both potential functions AHV_p, ANS_p for all candidate blocks would not change and the algorithm would do nothing. When the second event occurs, this means that the potential functions of the preceding candidate blocks are adequate and the preceding candidate can be output. After the algorithm outputs, it continues to collect the next candidate blocks and to update their potential functions.

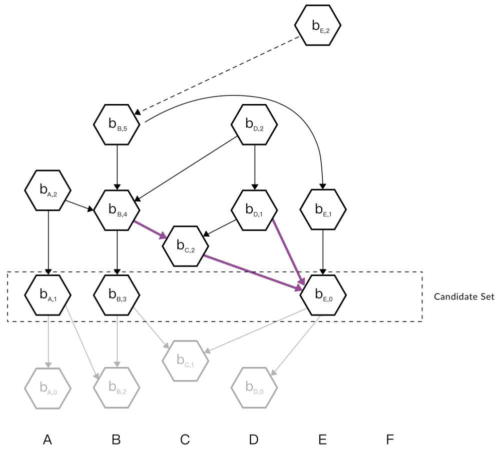

Figure 3: Example of the DEXON total ordering algorithm: if this is in node C's local view, $\mathcal{C} = \{b_{A,1}, b_{B,3}, b_{E,0}\}$, $\mathcal{A} = \{b_{A,1}, b_{B,3}, b_{E,0}\}$, $AHV_C(b_{A,1}) = (1, \infty, \infty, \infty, \infty, \perp)$, $AHV_C(b_{B,3}) = (\infty, 3, \infty, \infty, \infty, \perp)$, $AHV_C(b_{E,0}) = (\infty, \infty, 2, 1, 0, \perp)$, $\#AHV_C(b_{A,1}) = 1$, $\#AHV_C(b_{B,3}) = 1$, $\#AHV_C(b_{E,0}) = 3$, $ANS_C(b_{A,1}) = \{A\}$, $ANS_C(b_{B,3}) = \{A, B, D\}$, $ANS_C(b_{E,0}) = \{A, B, C, D, E\}$

4.2.1 Correctness and Liveness

Lemma 4.3 (Consistency of AHV)**.** *The first time the criterion holds for node p, q, $AHV_p(b)$ and $AHV_q(b)$ are consistent for any block $b \in \mathcal{C}_p \cap \mathcal{C}_q$. That is, for each node r, if $AHV_p(b)[r] \neq \perp$ and $AHV_q(b)[r] \neq \perp$, then $AHV_p(b)[r] = AHV_q(b)[r]$.*

Proof. Let $r \in \mathcal{N}$ satisfy $AHV_p(b)[r] \neq \perp$ and $AHV_q(b)[r] \neq \perp$. Consider that if both functions $AHV_p(b)[r] = AHV_p(b)[r] = \infty$ hold, then this lemma is correct. Without loss of generality, assume $AHV_p(b)[r] = k$, $AHV_q(b)[r] = k'$ or ∞, for some integer $k' \neq k$, because node $r \in \mathcal{N}_q$. Case 1: $AHV_q(b)[r] = \infty$, let i be the minimum number s.t. $b_{r,i} \in \mathcal{P}_q$. If $i < k$, $AHV_p(b)[r] < k$, which leads to a contradiction. If $i > k$, $b_{r,k}$ must be in $\mathcal{P}_q$. By the causality of blocks in the blocklattice, i must be equivalent to k, which leads to a contradiction. Case 2: $AHV_q(b)[r] = k'$, the argument is similar to case 1, thus, by the causality of blocks in the blocklattice, $k' = k$, which leads to a contradiction. □

Lemma 4.4 (Consistency of Grade)**.** *The first time the criterion holds for node p, q, $Grade_p(b_1, b_2)$ and $Grade_q(b_1, b_2)$ are consistent for any two blocks $b_1, b_2 \in \mathcal{C}_p \cap \mathcal{C}_q$. That is, $Grade_p(b_1, b_2) = Grade_q(b_1, b_2)$.*

Proof. Because both of the local views of nodes p and q are partial DAGs, they will be the same eventually. By Lemma 4.3, the output of function $AHV(b)$ of node p and q are consistent for every block $b \in \mathcal{C}_p \cap \mathcal{C}_q$. Thus, $Grade_p(b_1, b_2) = Grade_q(b_1, b_2)$. □

Lemma 4.5 (Arrival of the Preceding set)**.** *When the first time the criterion holds for node p, q, then $\mathcal{A}_p \subset \mathcal{C}_q$ and $\mathcal{A}_q \subset \mathcal{C}_p$ holds.*

Proof. We have three cases: first, node p, q are normal delivery; second, one node is normal delivery and the other is early delivery; third, both are early delivery. Case 1: we have $|ANS_p| = n$, so $\mathcal{A}_q \subset \mathcal{C}_q \subset \mathcal{C}_p$ and $|ANS_q| = n$. Consequently, $\mathcal{A}_p \subset \mathcal{C}_p \subset \mathcal{C}_q$. The lemma holds in this case.

Case 2: Without loss of generality, we assume $|ANS_q| = n$ and early deliver happened in node p. First, $\mathcal{A}_p \subset \mathcal{C}_p \subset \mathcal{C}_q$ because $|ANS_q| = n$. Second, we prove $\mathcal{A}_q \subset \mathcal{A}_p$ by contradiction. Assume there exists a block $b \in \mathcal{A}_q \setminus \mathcal{A}_p$. By the definition of the preceding set, $\exists b' \in \mathcal{A}_p$, s.t. $Precede_p(b', b) = 1$. However, because $b \in \mathcal{A}_q, \forall b'' \in \mathcal{A}_q, Precede_q(b'', b) = 0$. By the Lemma 4.3, $b' \in \mathcal{C}_q$ is also present in the preceding set, and $Precede_q(b', b) = 1$, which leads to a contradiction. This means $\mathcal{A}_q \subset \mathcal{A}_p \subset \mathcal{C}_p$.

Case 3: $\forall b \in \mathcal{A}_p$, let $b' \in \mathcal{A}_q$ be the block with $\#AHV_q(b') > \Phi$. Because $|ANS_q(b')| > \Phi$ and $|ANS_p(b)| \geq n - \Phi$, there must exist a node $r \in ANS_p(b) \cap ANS_q(b')$ by the pigeonhole principle. Thus, b and b' are received if the blocks issued by r are received; this means $b \in \mathcal{C}_q$. Furthermore, $\mathcal{A}_q \subset \mathcal{C}_p$ holds as well. □

Theorem 4.6 (General Correctness for total ordering)**.** *Let $\mathcal{A}_p^i, \mathcal{A}_q^i$ be the i-th set output by node p, q, then $\mathcal{A}_p^i = \mathcal{A}_q^i$ holds.*

Proof. We use inductive assumption to prove this theorem. For $i = 1$ (the first time the criterion holds), for node p, q, we prove that $\mathcal{A}_p^i = \mathcal{A}_q^i$: we prove this by contradiction. Assume $\mathcal{A}_p \neq \mathcal{A}_q$. Without loss of generality, there exists a block $b \in \mathcal{A}_p \setminus \mathcal{A}_q$. By Lemma 4.5, we have $\mathcal{A}_p \subset \mathcal{C}_q$ which implies $b \in \mathcal{C}_q \setminus \mathcal{A}_q$. Let $b' \in \mathcal{A}_q \subset \mathcal{C}_q$ be the block s.t. $Grade_q(b', b) = 1$. This implies $ANS_q(b') > \emptyset$. Consider two cases: either normal delivery or early delivery happened on node p. Case 1: (normal delivery) $|ANS_p| = n$ means $b' \in \mathcal{C}_p$. By Lemma 4.4, $Grade_p(b', b) \neq 1$, contradiction because of $b \in \mathcal{A}_p$ in our assumption. Case 2: (early delivery) We have $|ANS_p(b)| \geq n - \emptyset$ because $b \in \mathcal{A}_p$. By the pigeonhole principle, there exists some node $r \in ANS_p(b) \cap ANS_q(b')$. When the blocks issued by the node r in both local views of nodes p and q are received, it proves $b \in \mathcal{C}_q$, but $b \notin \mathcal{A}_q$, contradiction. Thus, the $i = 1$ case holds. For $i + 1$ case, once the output set is the same, the sub-DAG is eventually consistent in every node's local view. Therefore, it follows the statement of Lemmas 4.3, 4.4, and 4.5 and the case of $i = 1$. This completes the proof. □

Lemma 4.7. *Let p be a correct node and Δ_{BA} be the upperbound of processing time of BA, then the following statements are correct:*

1. *If b is received by p at time T, then either b has been output or $|ANS_p(b)| = |\mathcal{N}_p \setminus \mathcal{F}_p|$ holds at time $T + \Delta_{BA} + \lambda$.*
2. *If $b \in \mathcal{P}$ and $|ANS_p(b)| \geq |\mathcal{N}_p \setminus \mathcal{F}_p|$ holds at time T, then the criterion holds during time T and $T + \Delta_{BA} + \lambda$.*

Proof. For the first statement, suppose b is not output and $|ANS_p(b)| \neq |\mathcal{N}_p \setminus \mathcal{F}_p|$. We assume that each correct node receives a block within $\Delta_{BA} + \lambda$ after it is issued and all correct nodes will ack it or ack the following blocks issued by that node. Thus, $|ANS_p(b)| \geq |\mathcal{N}_p \setminus \mathcal{F}_p|$. For the second statement, we have $|ANS_p| \geq |ANS_p(b)| \geq |\mathcal{N}_p \setminus \mathcal{F}_p|$ and any Byzantine node must issue a block during time T and $T + \Delta_{BA} + \lambda$. Thus, $|ANS_p| = |\mathcal{N}_p|$ holds, and the criterion must hold by Theorem 4.1. □

Lemma 4.8. *If $\mathcal{C}_p \neq \emptyset$ and the criterion holds, then $\mathcal{A}_p \neq \emptyset$ for any node p.*

Proof. Because $\mathcal{C}_p \neq \emptyset$ and the criterion holds, internal stability implies there exists at least one element in $\mathcal{A}_p$ that precedes the element in $\mathcal{C}_p$. Thus, $\mathcal{A}_p \neq \emptyset$ holds. □

Theorem 4.9 (Liveness for total ordering). *Let p be a correct node and $\Delta_{BA} + \lambda$ be the upperbound of processing time of BA. For any time interval $\Delta_o \geq 2\Delta_{BA} + \lambda$ the criterion holds and the set of output is non-empty in the interval Δ_o.*

Proof. We first prove that if a block b is received by p at time T, the criterion holds and the set of output is non-empty before $T + 2\Delta_{BA} + \lambda$: by Lemma 4.7, the criterion will hold before $T + 2\Delta_{BA} + \lambda$. By Lemma 4.8, the output set is not empty. Because each chain has liveness, the theorem is proved.

□

Theorem 4.10 (Validity of liveness for total ordering). *For each input of a valid block b, the total ordering algorithm will output b eventually.*

Proof. Assume there exists a valid block b in p's local view s.t. it is not output by the DEXON total ordering algorithm. Let the set of blocks that acks b be the set Γ and let the set Λ be the set of nodes that issued blocks in Γ. Because b is not output, no block in Γ will be a candidate block at any time. Thus, only the elements in set Γ' or Γ_s could be output, where Γ' is the set of blocks issued by the node in $\mathcal{N} \setminus \Lambda$ and Γ_s is the set of blocks whose heights are lower than those of the blocks in Γ for each node in Λ. That is, $\Gamma_s = \{\, b_{q,j} : j < j', \text{where } b_{q,j'} \in \Gamma \text{ and } q \in \Lambda \,\}$. If the algorithm produces output many times, we can discover a set of blocks $B \in \Gamma'$ that are acked by some blocks whose heights are higher than those of the blocks in Γ for some node in Λ. Thus, we have $\#AHV_p(b') < f_{max}$, for all $b' \in B$. Now, consider two cases: the first case is $\#AHV(b) > \Phi$, then $b \in \mathcal{A}$. The second case is $\#AHV_p(b) \leq \Phi$, but $|ANS_p| > 2f_{max}$ and $\#AHV_p(b') < f_{max} < \Phi$. Thus, either b or some blocks in Γ_s will be in $\mathcal{A}$. Because Γ_s is a finite set, the algorithm will eventually output block b. □

4.2.2 Complexity

In this section, we analyze the complexity of our total ordering algorithm. First, we describe the data structure we used: we store both AHV' and ANS vectors for each block in the pending set, and $AHV'_p(b)$ is defined as

$$\begin{cases} \perp, & \text{if } q \notin ANS_p, \\ k, & \text{where } k \text{ is the minimum number s.t. } b_{q,k} \text{ acks } b, \\ \infty, & \text{otherwise.} \end{cases}$$

We also store a global vector $GAHV_p$, which is the minimum height of blocks issued by each node in p's local view. Thus, the $AHV_p(b)[j]$ can be computed from $GAHV_p[j]$ and $AHV'_p(b)[j]$ in $\mathcal{O}(1)$.

Second, we start to analyze the algorithm: when a block is received, if it is a candidate block, the algorithm updates the AHV vectors for the all blocks in the candidate set. This costs $\mathcal{O}(n)$ time. Because no block acks it, only the criterion $|ANS_p| = n$ must be determined. Otherwise, some block indirectly or directly acks candidate blocks in the DAG_p; thus, both AHV' and ANS of all the blocks acked by the received block must be updated. This costs $\mathcal{O}(n^2)$ time because the total number of blocks in DAG_p is $\mathcal{O}(n^2)$ and the computation of updating for each block is $\mathcal{O}(1)$.

Third, to maintain the preceding set, a direct method to implement the algorithm is to compute all candidate blocks every time; this method is exactly the same as the pseudo code, but it costs $\mathcal{O}(n^3)$ because $|\mathcal{A}| \times |\mathcal{C} \setminus \mathcal{A}|$ blocks exist and for each iteration, the method requires $\mathcal{O}(n)$ to compute $|\{j|AHV_p(b)[j] < AHV_p(b')[j]\}|$. We use a lazy-computation to reduce the complexity because we can store the current value and update it lazily. Specifically, we use a matrix of $(|\mathcal{A}| + |\mathcal{C} \setminus \mathcal{A}|)$ rows and $|\mathcal{C} \setminus \mathcal{A}|$ columns, consisting of elements of the form (i_1, i_2), each of which is $|\{j|AHV_p(b_{i_1})[j] < AHV_p(b_{i_2})[j]\}|$ for $b_{i_1} \in \mathcal{A}$ if $i_1 < |\mathcal{A}|$, $b_{i_1} \in \mathcal{C} \setminus \mathcal{A}$ if $i_1 \geq |\mathcal{A}|$, and $b_{i_2} \in \mathcal{C} \setminus \mathcal{A}$. Thus, every time a block is received, the matrix updates the current relations of AHV functions between candidate blocks. The preceding set and $criterion_p$ can both be determined by the matrix within $\mathcal{O}(n^2)$ time. Several operations can be conducted on the matrix. One operation is updating the matrix when a block is received, all the elements in the matrix are updated and each update costs a constant operation time. If the elements in a row all have value less than $\Phi - n + |ABS_p|$, this means that the block corresponding to the row should be in the preceding set. Thus, the algorithm adds the block into the preceding set and updates the matrix. If there exists an elements in a row whose value is larger than Φ, it means that there exists an element in the preceding set that precedes the block corresponding to the row. Therefore, the complexity of all aforementioned cases is bound by $\mathcal{O}(n^2)$.

Therefore, both time and space complexity of the total ordering algorithm is bound by $\mathcal{O}(n^2)$. Note that, no communication exists between any two nodes in this total ordering algorithm.

4.3 Timestamping Algorithm

In this section, we present the timestamp algorithm, which ensures that the timestamp is decided from consensus so that the Byzantine nodes cannot bias it. To compute the consensus timestamp for block b, the timestamping algorithm first constructs a vector, whose elements are the newest time of blocks from each chain before block b. Then, the median of the vector is the consensus timestamping of block b. The algorithm is shown in Algorithm 3, where $b.Chain\_ID$ means b is generated from the $chain\_ID$ and $b.block\_timestamp$ is the block timestamp of b.

```
Procedure Compute_Timestamp
    Input  : ordered chain $Chain_{order} =< b_0, b_1, ... >$
    Output: CompactionChain $\{b_i\}$ with consensus timestamp for each block
    Let $n$ be the number of chains and $V$ is $n$-dimensional vector
    foreach ID of the chain do
        $V[ID]$ = time of genesis block.
    $i = 0$
    while $i \leq$ height of ordered chain do
        $V[Chain_{ordered}[i].Chain\_ID] = Chain_{ordered}[i].block\_timestamp$
        $Chain_{ordered}[i].consensus\_timestamp = median(V)$
        $i + +$
    end
```

Algorithm 3: Compute Timestamp Algorithm

4.4 Consensus on the Blocklattice

Overview The DEXON consensus is the consensus of a blocklattice that consists of acking information and numerous single chains as presented in the design in Section 3.3. With the total ordering algorithm, the blocklattice is processed to a compaction chain by each node individually. The timestamping algorithm computes the consensus timestamp for each block in the compaction chain. The parameter in the system contains the number of chains, Φ, threshold κ for total ordering, and the network latency bound λ. We first explain how to add acking information into block information, and then present the mechanism of the consensus of blocklattice including adapting the total ordering and notarizing the blocks so that user can verify the validity of the blocks in the compaction chain. In the end of this section, we introduce the load balancer that allows each chain to gather the transactions that are not contained in the blocks of other chains.

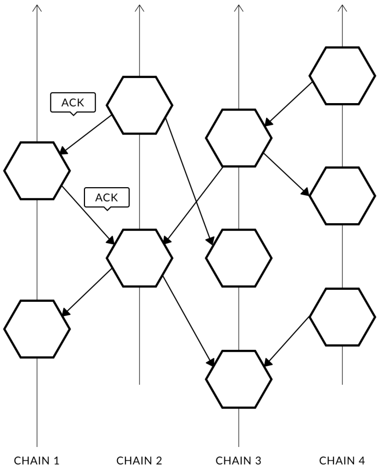

Figure 4: Blocklattice

Block Information To form a blocklattice structure, each block must add additional *acking* information. When a user proposes a new block, the proposed block *acks* the latest notarized blocks of other chains it has received. This means it sees the blocks from other nodes and supports them. We say that a block acks another block if its ack field contains the acked block's information. An *ack field* has the following data structure:

`block_proposer_id`	block proposer's ID
`acked_block_hash`	hash of the acked block
`block_height`	height of the block, starting at 0

The DEXON blocklattice is formed by numerous single chains with acks between blocks. An example of a DEXON blocklattice data structure is illustrated in Figure 4.

Single Chain The protocol is described as Section 3. Note that these single chains can share the same CRS, so the system is only required to have one CRS set S_{CRS}.

Compacting into Total-Ordered Chain Each single chain generates its blocks individually and these generated blocks form a blocklattice. Then, each node executes the total ordering algorithm with the blocklattice as input, and the output of the total ordering algorithm is the compaction chain, which is compacted from the blocklattice. Then, the system applies the timestamping algorithm to compute the consensus time for each block in the compaction chain.

Notarizing the Blocks in the Compaction Chain Once a block is in the compaction chain, the height of the block in the compaction chain, the consensus timestamp, and the threshold signature are confirmed. These three items of information are recorded in one of the next blocks of the compaction chain. Thus, the validity of the block in the compaction chain can be verified by the following blocks which contains the notarizing information.

Load Balancer In DEXON, many single chains grow concurrently. This leads to a problem: some transactions (especially those with high transaction fees) may be packed into several single chains simultaneously. Note that this problem does not do harm to the correctness of DEXON. Suppose a transaction `tx` is notarized in two different single chains. Because all nodes can agree on the same compaction chain after the total ordering algorithm, the transaction in the block with a smaller block height in the compaction chain will be executed and the other will not be executed due to contradiction. However, this problem wastes the space utility of blocks.

To solve this problem, we introduce a constraint of packing transactions, which is called *load balancer*. Suppose the number of single chains is N. Let $S_{\text{notary},j}$ be the notary set of the j-th single chain. The node can pack a transaction `tx` into its proposed block only if

$$Hash(\texttt{tx}) \bmod N = j.$$

With load balancer, each transaction is only allowed to be packed into one single chain and the problem is resolved.

4.5 Resistance Against Byzantine Nodes

We briefly discuss how robust DEXON is against different ratio of Byzantine nodes. As a general framework for combining many single chains, DEXON can resist against $\lfloor(|S_{\text{node}}| - 1)/3\rfloor$ Byzantine nodes as long as the correctness and the liveness of all individual single chain holds. That is, given N single chains with agreement, DEXON can integrate these single chains into a globally-ordered chain with unbiased timestamp.

As for the single chain algorithm, in Section 3.2, we show that the agreement and the termination of our Byzantine agreement protocol holds if $t \le \lfloor(|S_{\text{notary}}| - 1)/3\rfloor$, where t is the number of Byzantine nodes in S_{notary}.

Thus, if we adopt our Byzantine agreement protocol as DEXON's underlining single chain algorithm, we need to guarantee that the number of Byzantine nodes in each notary set is smaller than $\lfloor(|S_{\text{notary}}| - 1)/3\rfloor$. Let $S_{\text{notary},i}$ be the notary set of i-th single chain and t_i is the number of Byzantine nodes in $S_{\text{notary},i}$. Because the notary set of each single chain is chosen independently, the probability that $t_i \le \lfloor(|S_{\text{notary},i}| - 1)/3\rfloor$ for all i will follow hypergeometric distribution. Consequently, there is a trade-off between the size of notary set, the ratio of Byzantine nodes and the probability of failure. The following table illustrates that the lowest size of S_{notary} with different ratio of Byzantine nodes and different probability of failure given $|S_{\text{node}}| = 10\text{k}$ and $|S_{\text{node}}| = 100\text{k}$.

	ratio of Byzantine nodes in S_{node}			
failure	1/4	1/5	1/4	1/5
probability	$S_{\text{node}} = 10\text{k}$		$S_{\text{node}} = 100\text{k}$	
2^{-40}	1237	481	1402	489
2^{-60}	1789	724	2165	774
2^{-80}	2272	952	2900	1054

Table 1: The lowest sizes of S_{notary} such that less than 1/3 members in S_{notary} are Byzantine nodes for $|S_{\text{node}}| =$ 10k and 100k under the ratio of Byzantine nodes 1/4 and 1/5 with different probability are illustrated. For example, if $|S_{\text{node}}| = 10\text{k}$ and the number of Byzantine nodes is $10000 \times 1/4 = 2500$, the lowest size of S_{notary} is 1237 such that less than 1/3 members in S_{notary} are Byzantine nodes with probability of at least $1 - 2^{-40}$.

5 Extension of DEXON

5.1 Total Ordering as Sharding Framework

As mentioned before, compacting a blocklattice into a compaction chain by the total ordering algorithm is a generic method for any blockchain with single-chain structure. A real-time total ordering can support up to 90 nodes in our experiment on a laptop (this experiment does not involve parallelization). A further way to create more sharing chains is to shard many total orderings, and then, to sort the many compacted outputs by each total ordering algorithm according to the unbiased timestamp.

5.2 Configuration Change

One of the main features of our consensus is the support of configuration changes while the consensus is operating. For example, the parameters of the algorithm or the number of chains may be adjusted in order to reach higher efficiency or higher throughput. To change configuration, all nodes must initiate at the same state; otherwise, the system will be inconsistent. Fortunately, the output of total ordering is a sequence of sets and each node outputs the same set through either normal delivery or early delivery at the same height of the compaction chain. Moreover, each block in the compaction chain is marked by a consensus timestamp, which is close to real-world time. Thus, a way to change configuration is to change the system at the same height or at a specific consensus time (e.g. the first block after the specific consensus time).

Adding/Deleting Chain As previously mentioned, the consensus algorithm can change configuration by the state of total ordering. In this section, we demonstrate how to achieve configuration changes by the consensus in each single chain.

Given a specific time T, the block b_i after T is the first block after a configuration change in chain i. Let $prev(b)$ be the parent block of b; let the blocks before the configuration change be in round r_1; and let the blocks after the configuration change be in round r_2. That is, all blocks before $prev(b_i)$ are in r_1 and all blocks after b_i are in r_2. Now, we add a rule for acking: any block in round r_2 can only ack the last block in round r_1 or blocks in round r_2.

We use total ordering algorithm to compact the blocks in r_1 until the system has received all $\{b_i\}_{i\in\{1,\ldots,n\}}$. If some blocks in r_1 are not output by total ordering, we use a deterministic topological sort to output them (e.g. output the candidate set with lexicographical order each time). Then, we start a new total ordering algorithm with input blocks of round r_2. This is correct because of the liveness of each single chain. However, the confirmation time may delay due to the quality of synchronization. The worst case of confirmation time increases as $\lambda + T_{BA} + T_{total\_ordering}$, but configuration changes rarely occur.

For the consensus timestamp, the causality may be demolished when adding or deleting chains because the median of the latest time of each chain may be early. However, the number of the blocks whose block time is earlier is very small. Thus, an easy way to fix this is to add a condition $b_i.timestamp = max(b_{i-1}.timestamp, b_i.timestamp)$ that ensures the consensus timestamp to continuously increase.

References

[BHM18] Leemon Baird, Mance Harmon, and Paul Madsen. Hedera: A governing council & public hashgraph network. Whitepaper, May 2018. https://s3.amazonaws.com/hedera-hashgraph/hh-whitepaper-v1.1-180518.pdf.

[CGMV18] Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. Algorand agreement: Super fast and partition resilient byzantine agreement. *IACR Cryptology ePrint Archive*, 2018:377, 2018.

[DGKR18] Bernardo David, Peter Gazi, Aggelos Kiayias, and Alexander Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, pages 66–98, 2018.

[DKM93] Danny Dolev, Shlomo Kramer, and Dalia Malki. Early delivery totally ordered multicast in asynchronous environments. In *Digest of Papers: FTCS-23, The Twenty-Third Annual International Symposium on Fault-Tolerant Computing, Toulouse, France, June 22-24, 1993*, pages 544–553, 1993.

[DSU04] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[DY05] Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In *Public Key Cryptography - PKC 2005, 8th International Workshop on Theory and Practice in Public Key Cryptography, Les Diablerets, Switzerland, January 23-26, 2005, Proceedings*, pages 416–431, 2005.

[GHKW17] Rishab Goyal, Susan Hohenberger, Venkata Koppula, and Brent Waters. A generic approach to constructing and proving verifiable random functions. In *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part II*, pages 537–566, 2017.

[GHM$^+$17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 51–68, New York, NY, USA, 2017. ACM.

[HMW] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview seriesconsensus system. Whitepaper. https://dfinity.org/pdf-viewer/pdfs/viewer?file=../library/dfinity-consensus.pdf.

[LJY14] Benoît Libert, Marc Joye, and Moti Yung. Born and raised distributively. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing - PODC '14*. ACM Press, 2014.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, jul 1982.

[MRV99] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *Proceedings of the 40th Annual Symposium on the Foundations of Computer Science*, pages 120–130, New York, NY, October 1999. IEEE.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.