DEXON Consensus Algorithm

Infinitely Scalable, Low-Latency Byzantine Fault Tolerant Blocklattice

DEXON Foundation

Aug 9 2018, v1.0

Abstract

A blockchain system is a replicated state machine that must be fault tolerant. Fault tolerance is achieved through a consensus algorithm to ensure *integrity* and *consistency* of a state machine among non-faulty nodes when *Byzantine* failures may exist. In other words, non-faulty nodes must reach Byzantine agreement [BT85] on the total ordering [MMS99] of all transactions. However, existing blockchain consensus algorithms such as [Nak08, But14] suffer from issues such as limited scalability and high transaction confirmation latency, making deployment of real-world mass-adopted applications onto blockchain systems unfeasible. In this paper, we present the state-of-the-art DEXON consensus algorithm. The proposed DEXON consensus algorithm adopts a novel *blocklattice* data structure, enabling it to achieve *infinite scalability* and *low transaction confirmation latency* with asymptotically optimal communication overhead.

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

Contents

1 Introduction

Blockchain systems are being challenged to meet rigorous real-world application requirements, such as low transaction confirmation latency and high transaction throughput. However, existing blockchain systems are unable to satisfy these criteria. To overcome these limitations, new consensus models have been designed and newer ones, such as [PSF17, BHM18, HMW], are emerging.

The goal of the DEXON consensus algorithm is to make real-world mass-adopted decentralized applications (DApps) feasible by serving as a *low-latency* and *infinitely scalable* transaction-processing blockchain infrastructure. The DEXON consensus algorithm is *leaderless* and *symmetric*. Therefore, it achieves *high availability* because it has no single point of failure. Moreover, the transaction throughput (transactions per second, TPS) scales linearly with the total number of operating nodes in the network. The DEXON consensus algorithm reaches Byzantine agreement on the total ordering of all transactions among non-Byzantine nodes even when up to one-third of the nodes in the network are Byzantine. This is considered as the optimal resilience to *Byzantine fault tolerance* [Bra87, CL99]. The DEXON consensus algorithm ensures the following two crucial properties of *distributed ledger technology* (DLT):

- *Correctness*: consistency among all non-Byzantine nodes is achieved.
- *Liveness*: deadlock or livelock does not happen in the system.

The high-level concept of DEXON is that an arbitrary number of nodes maintain their own blockchains, and each block in a node's blockchain, proposed by that node individually, must follow other blocks as a means of acknowledging that those blocks are correct. Then, the node broadcasts the block to every node in the DEXON system. Consider the blocks as vertices and the relations of blocks following them as edges in a graph. This forms a lattice-like structure, which we call *blocklattice*. The blocklattice is the most evolutionary and fundamental structure of the DEXON consensus. Each node operates its own blockchain as a set of vertices in the blocklattice, and ack actions function as the edges in the blocklattice. After proposing or receiving a block, each node executes the total ordering algorithm and the timestamp algorithm, individually, to obtain a globally-ordered chain that contains all the valid blocks that have been proposed from all the nodes in the DEXON system. Thus, each node maintains globally-ordered data, called the *DEXON compaction chain*, so that it can be further collected into one block through the Merkle tree technique. All the procedures are done by each node except broadcasting the block; thus, the DEXON algorithm achieves low latency and infinite scalability.

DEXON tolerates a situation in which up to one-third of all nodes exhibit Byzantine behaviour and offers the following advantages:

Infinite Scalability
Most blockchain solutions are not able to scale even with increased resources, and block production remains slow. The DEXON consensus is an asynchronous BFT algorithm and the system throughput is only bounded by the available network bandwidth. DEXON's non-blocking consensus algorithm can easily help the network scale.

Low Latency
Latency is the amount of time from block proposal to confirmation. Therefore, latency is one of the paramount properties for any blockchain. 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 can not be delivered by traditional blockchains.

Optimal Communication Overhead
A two-phase-commit consensus algorithm has an $\mathcal{O}(N^2)$ communication complexity in any N-node setting and is costly. Instead, the communication overhead of the DEXON consensus is bounded by the frequency of acking f, which remains at constant as the network transaction throughput increases. Additionally, the DEXON consensus can be coupled with randomized sampling; in that situation, the number of nodes can be scaled to millions while maintaining constant communication costs.

Explicit Finality
In Proof-of-Work blockchain systems 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 secured by the BFT algorithm.

Low Transaction Fees
The transaction fees of a typical blockchain increases when the blockchain network is congested. The DEXON consensus is infinitely scalable and has optimal communication overhead, which enables it to maintain the lowest possible transaction fees in large-scale system deployments.

Fairness
In traditional blockchain systems, a single miner can determine the transaction ordering and the randomness in smart contracts, making them vulnerable to front-run attacks and biased-randomness attacks. By contrast, in the DEXON consensus algorithm, no single block miner can determine the consensus timestamp. Instead, the transaction ordering is determined by the majority of nodes in the network, making front-run attacks impossible, and thus, DEXON is fair. DEXON also provides cryptographically, provably secure random oracle on the blocklattice data structure for unbiased randomness in smart contract executions.

Energy Efficiency
The DEXON consensus can be easily generalized to delegated proof of stake (DPoS) in practice. DEXON DPoS has asymptotically optimal computation overhead, making it highly energy efficient, and thus increasing the feasibility of running high-capacity decentralized applications efficiently.

Now, we formally state the problem which this paper is focused:

> *Each node proposes new blocks and receives blocks from other nodes; the goal is that each node can maintain a globally-ordered sequence of blocks consisting of all valid blocks proposed by nodes.*

We ideally solve the problem because all the algorithms proposed in this paper are symmetric and operated individually in one node (except the reliable broadcast, in which the block informs other nodes). Thus, we optimize the latency and throughput in the scenario.
Roadmap. The remainder of this paper is organized as follows. In section 2, we introduce the basic terms and notations used in this paper; we describe the system model in section 3. Section 4 and section 5.4 present our reliable broadcast protocol and our timestamp algorithm, respectively. Section 6 presents how the DEXON consensus works. Section 7 presents simulation results to describe the properties of DEXON. Section 8 presents a discussion relative to those of other blockchains; section 9 concludes this paper.

2 Basic Terminology and Notations

In this section, we describe the notation listed in Table 1. $\mathcal{N}$ is the set of all nodes in the system, whose size can be arbitrary in the DEXON consensus algorithm, and $\mathcal{F}$ is the set of Byzantine nodes in the system, whose maximum capacity is limited by $f_{max} = \lfloor(|\mathcal{N}| - 1)/3\rfloor$. We define $\mathcal{B}^p = \{b | b \text{ is proposed by } p\}$, that is, the set of all blocks in node p's blockchain. $\mathcal{M}$ denotes the set of all valid blocks and $\mathcal{P}$ denotes 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. $\mathcal{C}$ is the set of candidate blocks that only acked 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 1.

We use $T_p[q]$ to represent the last updated timestamp of node q in node p's local view. We use $T(b)$ to represent the timestamp array in block b, that is, $T(b)[q]$ is the ack timestamp of q in the block b when a node proposes b.

Next, we present the definition of specifications of the reliable broadcast (with respect to our total ordering algorithm) as the following:

1. *Correctness*: all non-Byzantine nodes will eventually generate the same blocklattice (with respect to the ordered chain).

2. *Weak 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]. Moreover, the DEXON reliable broadcast is non-blocking and infinitely scalable; all blocks are finalized instantly. Furthermore, *weak liveness* is only defined for a fully asynchronous network. All non-Byzantine nodes achieve eventual consistency on the total

Notation	Description
$\mathcal{N}$	set of all nodes in the system
$\mathcal{F}$	set of Byzantine nodes in the system
$\mathcal{B}^p$	set of all blocks proposed by p
$\mathcal{M}$	set of all valid blocks
$\mathcal{P}$	set of pending blocks
$\mathcal{C}$	set of candidate blocks
$\mathcal{A}$	set of preceding candidate blocks
$b_{q,i}$	the block with height i from node q
$T_p[q]$	last-updated timestamp of node q in node p's local view
$T(b)[q]$	timestamp of node q in block b
f_{max}	$\lfloor(\lvert\mathcal{N}\rvert - 1)/3\rfloor$, the maximum number of Byzantine nodes that can be tolerated
$T_{transmit}$	network transmission delay between non-Byzantine nodes, which its value follows Gaussian distribution $\mathcal{G}(t_t, s_t)$
$T_{transmit\_max}$	$t_t + 6s_t$, as an upper bound of network transmission delay
$T_{propose}$	block proposing time, which its value follows Gaussian distribution $\mathcal{G}(t_p, s_p)$

Table 1: Notation

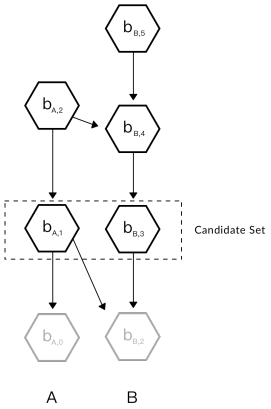

Figure 1: A simple example illustrates the notation: $\mathcal{N} = \{A, B\}$, $\mathcal{F} = \{\}$, $\mathcal{M} = \{b_{A,0}, b_{A,1}, b_{A,2}, b_{B,2}, b_{B,3}, b_{B,4}, b_{B,5}\}$, $\mathcal{P} = \{b_{A,1}, b_{A,2}, b_{B,3}, b_{B,4}, b_{B,5}\}$, $\mathcal{C} = \{b_{A,1}, b_{B,3}\}$. Also, $b_{A,2}$ acks $b_{A,1}$ and $b_{B,4}$, and both $b_{A,2}$ and $b_{B,5}$ indirectly ack $b_{B,3}$.

ordering of delivered blocks. The system eventually progresses (outputs newly delivered total ordering blocks). However, *strong liveness* is not achievable in a fully asynchronous network, which is defined by: if the information dissemination latency between all non-Byzantine nodes are bounded by a time bound T_1, there exists a time bound T_2, after which all blocks proposed by non-Byzantine nodes are finalized and a total ordering is delivered. The overall system is guaranteed not to deadlock or livelock for more than T_2.

3 System Model

We assume the network is weakly-synchronous[DSU04], meaning that a known bound for messages between any two non-Byzantine nodes exists. We also assume that a collision-free hash function H is available and a secure public-key cryptosystem with a secure digital signature algorithm, $SIGN$, exists.

We assume all operating nodes in the set $\mathcal{N}$ process in the network, and each node can be identified by a registered unique ID and a registered public key cryptosystem's public key. Among all nodes, there exist $|\mathcal{F}| \leq f_{max}$ nodes that may be faulty, including both crash faults and Byzantine faults. A crashed node does not participate in the DEXON consensus algorithm after it has crashed, or its block could require an arbitrary time to disseminate. A Byzantine node can deviate from the consensus algorithm arbitrarily, by such behaviors as sending conflicting messages, violating algorithm criteria, and delaying the messages between other nodes. Each node can propose a batch of transactions wrapped in a *block*.

Because of the usage of the collision-free hash function and the secure digital signature algorithm, the block content and the identity of the block proposer are unforgeable and unmalleable.

4 Mechanisms for Reliable Broadcast

The DEXON reliable broadcast property guarantees that the DEXON *blocklattice* data structure is eventually consistent for each node; therefore, every non-Byzantine node can output a consensus timestamp for each block on the blocklattice.

4.1 Blocklattice Data Structure

The DEXON consensus algorithm runs on a data structure different from those of traditional blockchains. In DEXON, each node has its own blockchain, and each blockchain cross-references the blockchains of other nodes. We call this novel data structure *blocklattice*. A block has the following data fields:

`block_proposer_id`	block proposer's ID
`transactions`	array of transactions
`acks`	array of acks
`timestamps`	array of timestamps of all nodes
`block_hash`	hash of everything insides the block except signature
`prev_block_hash`	hash of the block proposer's previous block
`signature`	digital signature of the `block_hash`, signed by the block proposer
`block_height`	height of the block, starting at 0
`notary_ack`	notary ack for compaction chain notarization

The first proposed block of a node is called a *genesis block*. The genesis block's `block_height` is 0, and its `prev_block_hash` is an empty string. Any further blocks proposed by a node, must *follow* the previous proposed block by including the `prev_block_hash` field and monotonically increase the `block_height`, as illustrated in Figure 2.

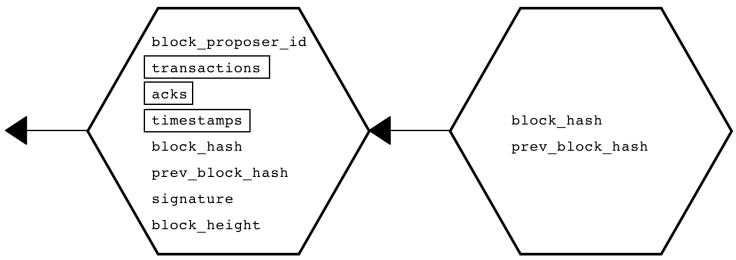

Figure 2: Two Continuous Blocks Proposed by Node p

When a node proposes a new block, it *acks* the latest blocks proposed by other nodes 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 acks field contains the acked block's information. An *ack* 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 all blockchains from the nodes together with the acks between blocks. An example of a DEXON blocklattice data structure is illustrated in Figure 3.

4.2 DEXON Reliable Broadcast Algorithm

The acking algorithm is notably simple and efficient. Every node continuously acks the latest blocks from other nodes. Once a block has been acked by $2f_{max} + 1$ nodes, it is considered to be *strongly acked*.

To store the state of a block, we create a local integer $b.acked\_count$ to store the number of times a block b has been acked and a local boolean $b.strongly\_acked$ to store whether b is strongly acked. As an example,

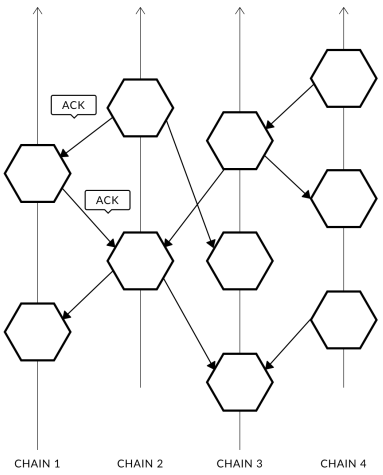

Figure 3: Blocklattice

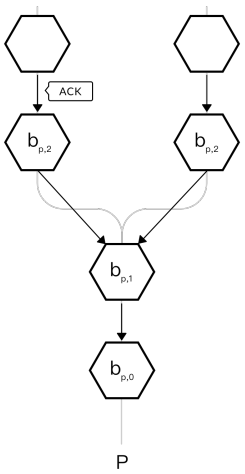

Figure 4: Fork in blocklattice

in Figure 10, the block $b_{E,0}$ is strongly acked. This value is calculated locally and is not broadcasted to the network with a block. Suppose a block b is a block proposed by node p, we increment $b.acked\_count$ when:

1. b is acked directly.
2. b is acked indirectly and the directly acked block is also proposed by p.

The ack count calculation procedure is in Algorithm 1.

Procedure *Update_Ack_Count for node p*
  Input : block b proposed by node q
  Output: blocks that are strongly acked
  foreach *block* $\bar{b}$ *proposed by node* $r \in \mathcal{N}$ *acked by* b **do**
    Increase $\bar{b}.acked\_count$.
    foreach *block* $b' \in \mathcal{B}^r$ *that is not acked by* $q \wedge b'.block\_height < \bar{b}.block\_height$ **do**
      Increase $b'.acked\_count$.
  foreach *block* b' *where* $b'.strongly\_acked = false$ **do**
    if $b'.acked\_count > 2f_{max}$ **then**
      $b'.strongly\_acked =$ true
      Output b'.

Algorithm 1: Update Ack Count

All valid blocks will eventually be acked directly or indirectly; every node acks the latest block of each other node's blockchain, and each block will eventually become a part of some node's canonical chain. Thus, a node is not required to ack every block proposed by other nodes.

If a block $b_{p,i}$ acks another block $b_{q,j}$, it must satisfy the following conditions:

1. **All blocks directly acked by $b_{q,j}$ must be received by p.**
2. **All historical blocks must be received:** p must have received all blocks $b_{q,k}$ for $0 \leq k < j$. Notably, p is **not** required to receive any other historical blocks from any other blockchain except q's blockchain. Purple blocks in Figure 5 indicate that the blocks must be received by p.
3. **Acking on both sides of a fork is not allowed:** p will violate the rule if $b_{p,i}$ acks on both sides of a fork. A fork is illustrated in Figure 4, and acking on both sides of a fork is illustrated in Figure 6. Even with different blocks, p must not ack on both sides of a fork, as illustrated in Figure 7.
4. **Acking on the same or older blocks is not allowed:** if the latest block acked by p on q is $b_{q,k}$, then $j > k$ must hold. Two violations are illustrated in Figure 8 and Figure 9.

Once a node p decides to use its block $b_{p,i}$ to ack a block $b_{q,j}$, p puts the information of $b_{q,j}$ into an ack data structure and stores that ack data structure in the `acks` array of $b_{p,i}$. Another critical item of information that must be calculated is the `timestamps` array in $b_{p,i}$. A `timestamps` array [Fid88] is an $|\mathcal{N}|$-dimensional array that represents a local view of all other nodes' clocks. When a node p proposes a block $b_{p,i}$ that acks on another block $b_{q,j}$, we update the timestamps by using Algorithm 2.

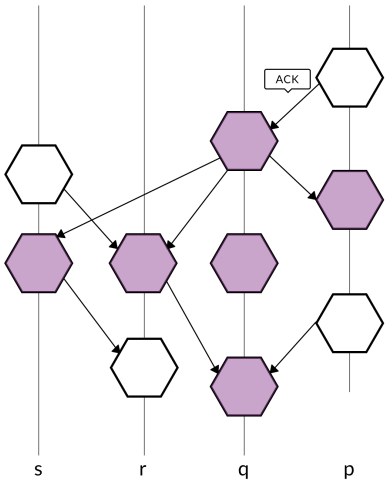

Figure 5: Conditions 1 and 2: Blocks that a node must receive for acking a block

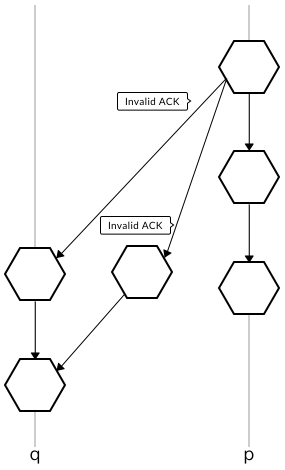

Figure 6: Condition 3: Invalid acking fork with a block

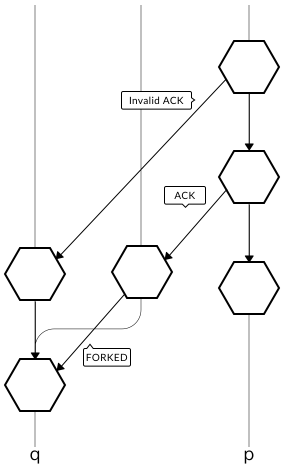

Figure 7: Condition 3: Invalid acking fork with a child block

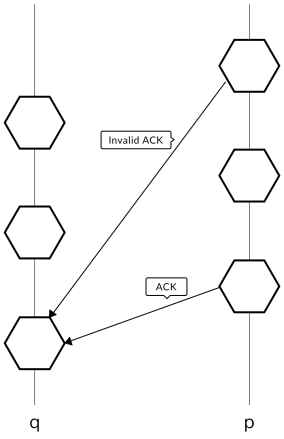

Figure 8: Condition 4: Acking twice on the same block

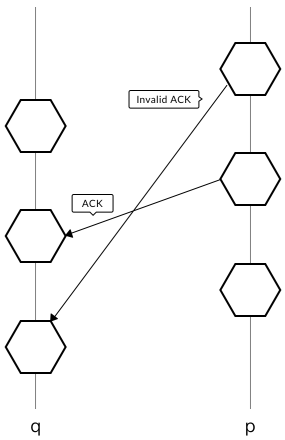

Figure 9: Condition 4: Acking an older block

Procedure *Update_Timestamps for node p*
  Input : block b received from node q
  // parameter: ϵ is a small, predetermined positive constant
  // p updates a local view timestamp of q according to $T(b)[q]$
  $T_p[q] = T(b)[q] + \epsilon$
  // p updates local view timestamps of other nodes according to $T(b)$
  foreach *node* $r \in \mathcal{N}$ **do**
    if $T(b)[r] > T_p[r]$ **then**
      $T_p[r] = T(b)[r]$

Algorithm 2: Update Timestamps

Procedure *DEXON_Reliable_Broadcast for node p*
  // parameter: $T_{delay}, T_{restrict}$
  // receiving block
  when receiving a block b from node q
    // sanity check ensures that the block format is correct and it is not from a fork
    // sanity check also ensures that the *timestamps* array is legitimate following our *Update_Timestamps* algorithm
    if *b passes sanity check* **then**
      Broadcast b.
      if *all blocks in* $\mathcal{B}^q$ *is received and not in* $W \wedge$ *all blocks directly acked by b are received and not in* $W \wedge q$ *is not restricted* **then**
        // Ack candidate set A stores blocks that p wants to ack.
        Put b into A.
        Update_Ack_Count(b)
        foreach *node* $r \in S$ **do**
          Update $\Delta_{q,r}$ by block b
          if $|\{ i : |\{ j : \Delta_{i,r}(j) > T_{delay} \}| > 2f_{max} \}| > 2f_{max}$ **then**
            Force r to fork its blockchain.
      else
        // Waiting set W stores blocks that will be acked in the future.
        Put b into W.
    else
      Drop b.
    if $\exists b' \in W$ *from node* $q \in \mathcal{N}$ *s.t. all blocks in* $\mathcal{B}^q$ *are received and not in* $W \wedge$ *all blocks directly acked by* b' *are received and not in* $W \wedge q$ *is not restricted* **then**
      Put b' into A set.
      Update_Ack_Count(b')
      Remove b' from W.
  end;
  // block proposing is done periodically
  when p wants to propose a block b
    foreach *block* b' *in* A **do**
      Write ack information of b' into the *acks* array of b.
      Update_Timestamps(b')
    Set $T_p[p]$ to the local clock of p.
    Put the *transactions* array, *acks* array, and *timestamps* array into b.
    Broadcast b.
    Update_Ack_Count(b)
    Clear A set.
  end;
  // nack and restrict
  when $|\{ i : |\{ j : \Delta_{i,r}(j) > T_{delay} \}| > 2f_{max} \}| > f_{max} \vee |\{ k : \Delta_{p,r}(k) > T_{delay} \}| > 2f_{max}$
    // suspected set S stores the nodes that are suspected to be faulty.
    Put r into S set.
    Restrict r for time $T_{restrict}$.
  end;

Algorithm 3: DEXON Reliable Broadcast

The complete DEXON reliable broadcast algorithm is listed as Algorithm 3. Information about T_{delay}, $T_{restrict}$, *nack* and *forcible forking* is presented in section 5.2.

Unlike the two-phase commit in PBFT[CL99], DEXON only requires one phase for main chain selection. The reason is that we can construct reliable communication channels between non-Byzantine nodes. To achieve this, nodes are allowed to broadcast and to pull information of blocks from other nodes. With the help of hash functions and digital signatures, the system constructs fast and robust reliable communication channels between non-Byzantine nodes. It can be proven that any node cannot fork through one phase of acking with reliable channels.

The DEXON reliable broadcast algorithm achieves non-blocking block proposing among all nodes, because each node can propose blocks at any time. The latency of an acked block is considered asymptotically optimal. Because the expected time for a block to be disseminated to $\frac{2}{3}|\mathcal{N}|$ nodes is $T_{transmit}$, the expected latency for a block to be strongly acked is $2T_{transmit}$. By the definition of $T_{transmit}$, our reliable broadcast is only bounded by the overall network conditions and not by some predefined parameters.

The novel technique employed by the DEXON reliable broadcast algorithm is to transform the traditional blocking PBFT algorithm to a non-blocking version on the DEXON blocklattice. The DEXON reliable broadcast algorithm does not have a leader responsible for proposing blocks; every node in the network can all propose blocks, and all blocks can jointly achieve finalization on the fly. This scales the throughput of PBFT by N times if N nodes exist in the network, and thus, achieves infinite scalability.

The DEXON reliable broadcast algorithm guarantees that all non-Byzantine nodes will determine a consistent blocklattice of strongly acked blocks. Once a block has been strongly acked, its total ordering among all blocks and *consensus timestamp* are determined by the *DEXON timestamp algorithm.*

4.2.1 Correctness

We prove the correctness of the DEXON reliable broadcast algorithm by two different theorems, *agreement* and *integrity.* In section 4.2.1 and section 4.2.2, *a node p outputs block b* means b is output by the DEXON reliable broadcast algorithm running on p.

Theorem 4.1. *(Agreement of the DEXON Reliable Broadcast) If a non-Byzantine node proposes a block b, all non-Byzantine nodes eventually output b.*

Proof. If a non-Byzantine node proposes a block b, all non-Byzantine nodes will eventually receive and ack it, directly or indirectly. Because at least $2f_{max} + 1$ non-Byzantine nodes exist and their acks will eventually be received by all non-Byzantine nodes, b will be output by every non-Byzantine node. □

Lemma 4.2. *If a Byzantine node proposes block b and b' at the same block height h, and a non-Byzantine node p outputs b, all non-Byzantine nodes eventually output b.*

Proof. If a Byzantine node only proposes one valid block at h, it will be no different from a non-Byzantine node, so we consider the case that it proposes two different blocks at the same height (fork). This proof is done by contradiction. Suppose a Byzantine node proposes two blocks b and b' at height h and a non-Byzantine node p outputs b but a non-Byzantine node q outputs b', then $b \neq b'$. By the DEXON reliable broadcast algorithm, this indicates we have at least $2f_{max} + 1$ acks on b and $2f_{max} + 1$ acks on b', the total of which is at least total of $4f_{max} + 2$ acks at the same block height of a node's blockchain. But non-Byzantine nodes can not ack on two different blocks of the same height (fork), so if all Byzantine nodes ack on both b and b', at most we can have $4f_{max} + 1$ acks, which is still smaller than $4f_{max} + 2$. Now we obtain a contradiction, so $b = b'$, and this completes the proof. □

Theorem 4.3. *(Integrity of the DEXON Reliable Broadcast) For any block b, every non-Byzantine node outputs b at most once, and the output set of blocks form a valid blocklattice. A valid blocklattice is a blocklattice that follows the four acking conditions we defined in section 4.2.*

Proof. For the first part of the argument, by our algorithm, once b is strongly acked and output, its *strongly_acked* is set to *true*. We will only output a block if its *strongly_acked* is equal to $false$ and has more than $2f_{max} + 1$ acks; thus, each block will only be output at most once. For the second part of the argument, in most of the cases, if b violates the acking rules, it will not pass the sanity check and thus will not receive any ack from non-Byzantine nodes. Because there are at most f_{max} Byzantine nodes, b will not be output by any non-Byzantine nodes. The only case in which b violates the acking rules and passes sanity check is that b is from a fork, because a non-Byzantine node can only receive one branch of a fork. By Lemma 4.2, we know that either one branch of a fork will be output by all non-Byzantine nodes or both the branches will not be output. In either cases, the output blocklattice is valid. This completes the proof. □

Theorem 4.4. *(Correctness of the DEXON Reliable Broadcast) Each non-Byzantine node will output the same and valid blocklattice (a set of blocks) to the DEXON total ordering algorithm.*

Proof. By Theorem 4.1, we know a block proposed by a non-Byzantine node will eventually be output; thus, a blocklattice output by every non-Byzantine node will eventually be the same. By Theorem 4.3, we know a blocklattice output by every non-Byzantine node will be valid. This completes the proof. □

4.2.2 Liveness

We now argue that any block proposed by a non-Byzantine node will be output by the DEXON reliable broadcast within $2t_t + t_p + 6\sqrt{2s_t^2 + s_p^2}$ with high probability under the assumptions of our network model. Suppose a node p proposes a block b at time t_0. b is expected to reach every non-Byzantine node at $t_0 + T_{transmit}$. By our reliable broadcast algorithm, a non-Byzantine node will ack any legitimate block it received. Therefore, it is expected that before time $t_0 + T_{transmit} + T_{propose}$, all non-Byzantine nodes will propose blocks that ack b, and those blocks will reach other non-Byzantine nodes at $t_0 + 2T_{transmit} + T_{propose}$. At this time, at least $|\mathcal{N} \setminus \mathcal{F}|$ blocks ack b. Because $|\mathcal{N} \setminus \mathcal{F}| \geq 2f_{max} + 1$, b is strongly acked and output by the algorithm, and the probability b is not output within $2t_t + t_p + 6\sqrt{2s_t^2 + s_p^2}$ is lower than 10^{-8} by the property of Gaussian distribution.

5 Mechanisms for Total Ordering

In this section, we present our main techniques to compact the proposed blocklattice into the compaction chain and to generate the timestamp for each block in the compaction chain. For the compaction, we introduce a basic total ordering algorithm in section 5.1 and present the nack mechanisms that defend against Byzantine nodes in section 5.2. We also present a novel method to reduce the influence of Byzantine nodes in section 5.3. In section 5.4, we present a timestamping algorithm that ensures the causality of blocks in the blocklattice.

5.1 DEXON Total Ordering Algorithm

First, we introduce the DEXON total ordering algorithm, which ensuring 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 sets $\Phi = 2f_{max} + 1$ to guarantee the liveness of the total ordering algorithm. Section 4 introduced the framework, in which each node proposes its blocks and broadcasts its blocks. In this scenario, a problem arises: each node receives all block information asynchronously, thus, the order of blocks from the blocklattice can not 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*, $ANS_p(b) = \{\, j : \exists b_{j,k} \in \mathcal{P} \text{ s.t. } b_{j,k} \text{ directly or indirectly acks } b \,\} \cup \{b\}$, is the set of nodes that proposed blocks ack block b in node p's 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. $\#AHV_p = |\,\{\, j : AHV_p(b)[j] \neq \bot \text{ and } AHV_p(b)[j] \neq \infty \,\}\,|$ 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

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

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, this means that such blocks have a relatively high priority to be output by the algorithm. Thus, regardless of what blocks are received afterwards, the blocks in $\mathcal{C} \setminus \mathcal{A}$ can not 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 input, 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 criterion consist two parts: the *internal stability* and the *external stability*. Informally, internal stability one 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 prioritizes 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 *criterion* to output for node p is 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) $\exists b \in \mathcal{A}$ s.t. $\#AHV_p(b) > \Phi$ and
    $\forall b \in \mathcal{A}$, $|ANS_p(b)| \geq n - \Phi$

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

Next, we formalize two simplifying *criterion* by following theorems.

Theorem 5.1. *(Simplifying Criterion for normal delivery) External Stability (a) implies Internal Stability, that is, if $|ANS_p| = n$, then, $\forall b \in \mathcal{C} \setminus \mathcal{A}, \exists b' \in \mathcal{A}$ s.t. $Grade_p(b', b) = 1$*

Proof. First, we prove $\mathcal{A}_p \neq \phi$ by contradiction. Assume $\mathcal{A}_p = \phi$, by the definition of a preceding set, $\forall b \in \mathcal{C}_p, \exists b' \in \mathcal{C}_p$ s.t. $Precede(b', b) = 1$. However, by the definition of $Precede$ 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 5.2. *(Simplifying Criterion for early delivery) Internal Stability implies External Stability (b), that is, if $\forall b \in \mathcal{C} \setminus \mathcal{A}, \exists b' \in \mathcal{A}$ s.t. $Grade_p(b', b) = 1$, then $\exists b \in \mathcal{A}$ s.t. $\#AHV_p(b) > \Phi$ and $\forall b \in \mathcal{A}$, $|ANS_p(b)| \geq n - \Phi$*

Proof. It is easy to prove $\mathcal{A} \neq \phi$. 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 proposed 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 $\mathcal{C}_p \setminus \mathcal{A}_p \neq \phi$

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 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 direct or indirect 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 $criterion_p$ holds,
  output $\mathcal{A}_p$ in the lexicographical order of the hash value of each block
  //update $\mathcal{A}_p$ and $\mathcal{C}_p$
  $\mathcal{C}_p = \mathcal{C}_p \setminus \mathcal{A}$
  $\mathcal{A}_p \leftarrow \phi$
  foreach $b \in \mathcal{P}_p$ *only acks the blocks 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 4: DEXON Total Ordering Algorithm

The DEXON total ordering algorithm is listed as Algorithm 4. There are two events must be considered: one is a block received and the other one 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 has nothing to do. When the second event occurs, this means the potential functions of the preceding candidate 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.

5.1.1 Correctness

Lemma 5.3. *(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 $r \in \mathcal{N}$ s.t. $AHV_p(b)[r] \neq \perp$ 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. WLOG, let $AHV_p(b)[r] = k$, $AHV_q(b)[r] = k'$ 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 $i < k$, $AHV_p(b)[r] < k$, 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, contradiction. Case 2: $AHV_q(b)[r] = k'$, the argument is similar to case 1, thus, by the causality of blocks in the blocklattice, $k' = k$, contradiction. □

Lemma 5.4. *(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 the both of local views of nodes p and q are partial DAGs, they will be the same eventually. By Lemma 5.3, 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, $Grade_p(b_1, b_2) = Grade_q(b_1, b_2)$. □

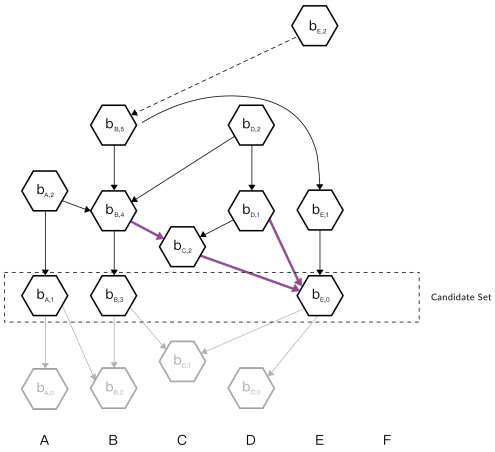

Figure 10: Example of the DEXON total ordering algorithm: if this is in node C's local view, $\mathcal{S} = \{b_{A,1}, b_{B,3}, b_{E,0}\}$, $\mathcal{A} = \{b_{A,1}, b_{B,3}, b_{E,0}\}$, $AHV_C(b_{A,1}) = (1, \infty, \infty, \infty, \infty, \perp)$, $AHV_C(b_{B,3}) = (\infty, 3, \infty, \infty, \infty, \infty, \perp)$, $AHV_C(b_{E,0}) = (\infty, \infty, 2, 1, 0, \perp)$, $\#AHV_C(b_{A,1}) = 1$, $\#AHV_C(b_{B,3}) = 1$, $\#AHV_C(b_{E,0}) = 3$, $ANS_C(b_{A,1}) = \{A\}$, $ANS_C(b_{B,3}) = \{A, B, D\}$, $ANS_C(b_{E,0}) = \{A, B, C, D, E\}$

Lemma 5.5. *(Arrival of the Preceding set) The first time the criterion holds for node p, q, then $\mathcal{A}_p \subset \mathcal{C}_q$ and $\mathcal{A}_q \subset \mathcal{C}_p$ hold.*

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 $|ANS_p| = n$, then $\mathcal{A}_q \subset \mathcal{C}_q \subset \mathcal{C}_p$ and $|ANS_q| = n$, then $\mathcal{A}_p \subset \mathcal{C}_p \subset \mathcal{C}_q$. It holds in this case.

Case 2: WLOG, let $|ANS_q| = n$ 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 a preceding set, $\exists b' \in \mathcal{A}_p$, s.t. $Precede_p(b', b) = 1$. However, becuase $b \in \mathcal{A}_q, \forall b'' \in \mathcal{A}_q, Precede_q(b'', b) = 0$. By the Lemma 5.3, $b' \in \mathcal{C}_q$ is also present in the preceding set, and $Precede_q(b', b) = 1$, 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 proposed by r are received; this means $b \in \mathcal{C}_q$. Furthermore, $\mathcal{A}_q \subset \mathcal{C}_p$ holds as well. □

Theorem 5.6. *(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 proof this by contradiction. Assume $\mathcal{A}_p \neq \mathcal{A}_q$, WLOG, there exists a block $b \in \mathcal{A}_p \setminus \mathcal{A}_q$. By Lemma 5.5, 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 $ANS_q(b') > \phi$. 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 5.4, $Grade_p(b', b) \neq 1$, contradiction because of $b \in \mathcal{A}_p$ in our assumption. Case 2: (early delivery) we have $|ANS_p(b)| \geq n - \phi$ 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 proposed 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 5.3, 5.4, and 5.5 and the case of $i = 1$. This completes the proof. □

5.1.2 Liveness

Lemma 5.7. *Let b_n be the block proposed by non-Byzantine nodes and b_f be the block including nack proposed by the Byzantine node. Then, any non-Byzantine node receives the strongly acked block b_n from any non-Byzantine node in any time interval Δ_n or b_f from any non-Byzantine node in any time interval $\Delta_f \leq 3 \cdot \Delta_n$ with high probability.*

Proof. For non-Byzantine nodes, they follow the rule that each node proposes block in the specific time interval $\Delta_n = t_p + 6s_t$ with high probability. For Byzantine nodes, if they do not propose blocks, they will be nacked by other nodes after Δ_n. The worst case is that the number of votes to nack is not enough due to network latency; in the worst case, other nodes will ack the block proposing by the node after $T_{restrict}$. Thus, we set

$\Delta_f \leq 3 \cdot \Delta_n$ to ensure every block proposed by Byzantine nodes will be acked or that Byzantine node will be nacked by other nodes. □

Lemma 5.8. *Let node p be a node and Δ be a specific bound, 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 $T + \Delta$.*
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 $T + \Delta$.*

Proof. For the first statement, suppose b is not output and $|ANS_p(b)| \neq |\mathcal{N}_p \setminus \mathcal{F}_p|$. Because we assume that each non-Byzantine node receives a block within Δ after it is proposed and all non-Byzantine nodes will ack it or ack the following blocks proposed 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 propose a block during time T and $T + \Delta$, otherwise it will be nacked. Thus, $|ANS_p| = |\mathcal{N}_p|$ holds, and the criterion must hold by Theorem 5.1. □

Lemma 5.9. *If $\mathcal{C}_p \neq \phi$ and the criterion holds, then $\mathcal{A}_p \neq \phi$ for any node p.*

Proof. Because $\mathcal{C}_p \neq \phi$ 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, $\mathcal{A}_p \neq \phi$ holds. □

Theorem 5.10. *(Liveness for total ordering) Let node p be a node and Δ be an specific bound. For any time interval $\Delta_o \geq 3\Delta_n + 2\Delta$ 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 $T + 2\Delta$: by Lemma 5.8, the criterion will hold before $T + 2\Delta$. By Lemma 5.9, the output set is not empty. Combined with Lemma 5.7, the theorem is proved.

□

Theorem 5.11. *(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 proposed blocks in Γ. Because b is not output, no block in Γ will be 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 proposed 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 $B \in \Gamma'$ s.t. they are acked by some blocks whose heights are higher than those of the blocks in Γ for some node in Λ. Thus, we have $\#AHV_p(b') < f_{max}, \forall b' \in B$. 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. □

5.1.3 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 the $AHV'_p(b)$ 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 block proposed 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 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, this must to update all $\mathcal{O}(n^2)$ elements in the matrix 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 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 there exists an element in the preceding set that precedes the block corresponding to the row. Therefore, the complexity of all aforementioned cases is bounded by $\mathcal{O}(n^2)$.

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

5.2 Implicit Nack

Fischer-Lynch-Patterson (FLP) impossibility proves that no deterministic consensus exists in an asynchronous network with even only one faulty node. Our basic DEXON consensus algorithm is deterministic in an asynchronous network, thus, it has no liveness property. The problem is that if some nodes do not propose blocks, the total ordering algorithm will not return an output. To address this problem, we designed our protocol to be weak synchronous; this design circumvents FLP, that is, there exists a time bound between non-Byzantine nodes. This is a reasonable setting in practice because the proposing time and transmission time can be bounded.

We use an *implicit voting* mechanism to avoid the halting case as follows. We define the inspecting time vector $\Delta_{p,d}$ to be the differential between the current vector time and the last updating time from node d in node p's local view. Let S be the set of the nodes that are suspected to be faulty. When a node proposes a block whose inspecting time vector $\Delta_{p,d}$ satisfies the following property:

$$|\{\, i : |\{\, j : \Delta_{i,d}(j) > T_{delay} \,\}| > 2f_{max} \,\}| > f_{max} \vee |\{\, k : \Delta_{p,d}(k) > T_{delay} \,\}| > 2f_{max}$$

the algorithm appends node d into S. From a high level perspective, this means a node's local view surveys at least $f_{max} + 1$ nodes, and the node calculates that at least $2f_{max} + 1$ are proposing blocks, but the node does not calculate that d proposed any block more than T_{delay} after the last time d proposed a block. Each node does not ack blocks proposed by $q \in S$ for a specific time interval $T_{restrict}$. During that interval, if a node receives $2f_{max} + 1$ blocks whose inspecting time vectors satisfy the property for node q in p's view, p will establish a nack block for node q. Specifically, p nacks q if

$$|\{\, i : |\{\, j : \Delta_{i,d}(j) > T_{delay} \,\}| > 2f_{max} \,\}| > 2f_{max}$$

After q has been nacked, p forks q's original blockchain and generates a nacked block on the forked blockchain on its own blocklattice. A nack block only contains the following information: `node_id`, `prev_block_hash` and `block_height`. The insight is that each node acts on behalf of the delayed node q to propose a block (the nack block), which is output to the total ordering algorithm to maintain liveness.

The nack is implicit bacause it is computed locally with the original information on the blocklattice. This is why we call the mechanism "implicit voting." Just as a node use "ack" to vote the validation of other blocks, a node can use implicit "nacks" to signal that some other node seems inactive. We designed implicit nack to be similar to ack, so the $2f_{max} + 1$ nacking blocks are valid for the implicit voting without being strongly acked because there must exist at least $f_{max} + 1$ nacks from non-Byzantine nodes. Furthermore, a node is forbidden to ack on a node it nacked during $T_{restrict}$, just like a node cannot ack on a fork. The only difference is that when a node p nacks node q, it does not update its timestamp vector. This means that $T_p[q]$ is updated only if q proposes a block. Because nack follows the conditions and properties of ack, the correctness and liveness can directly follow the proof of acking in a reliable broadcast. Because there exists a time bound between non-Byzantine nodes, each non-Byzantine node must force node q to fork after a specific time interval. After the forced forking holds, each node will only ack blocks proposed by q on the forked blockchain after the nack block, which means q can only propose blocks on the blockchain forked by other nodes. In addition, the empty block is recorded in the globally-ordered chain; this global record that the block violated the protocol. If the number of records in evidence exceeds a bound that we set, the node will be eliminated and its stake will be confiscated by other nodes.

Next, we set the two time conditions, T_{delay} and $T_{restrict}$. Let the transmitting time $T_{transmit}$ be described by a Gaussian distribution $\mathcal{G}(t_t, s_t)$ and the proposing time $T_{propose}$ be described by a Gaussian distribution $\mathcal{G}(t_p, s_p)$. One node p considers another node q inactive (did not propose blocks) if it sees at least $f_{max} + 1$ nodes did not receive any block from q after $T_{delay} = T_{propose} + T_{transmit}$, and the value of T_{delay} can be set to

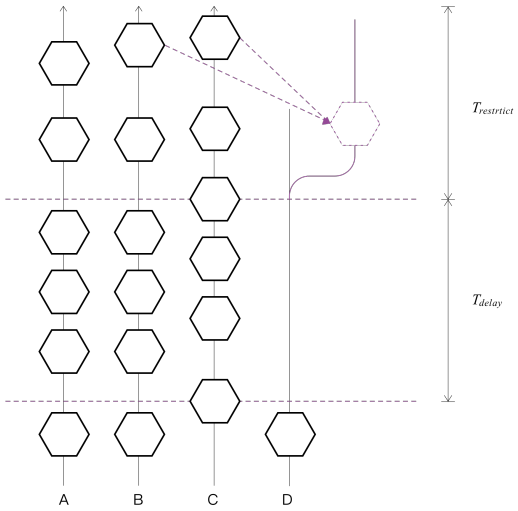

Figure 11: Implicit voting

six times the standard deviation so that the probability that an active node is active but is considered as faulty is less than 10^{-8}. On the other hand, $T_{restrict} = T_{propose} + T_{transmit}$ and the value of T_{delay} can be set higher than six times the standard deviation of $(t_p + t_t)$ so that the system can collect nacking blocks from $2f_{max} + 1$ nodes.

If a node is nacked, it is banned for a period of time T_{ban}, which is counted by the number of time total ordering output after the system has output the nack block created by other nodes. T_{ban} is a predefined parameter, which can grow exponentially with the number of times a node is nacked. For example, if node p nacks node q, p can only ack or nack q again after p's total ordering algorithm has produced output T_{ban} times. During the period that q is banned, the number of $|\mathcal{N}|$ shrinks to $|\mathcal{N}| - 1$ so that total ordering algorithm can continuously output without blocks proposed from q. This nacking system works as a failure detector regarding the membership required for the total ordering algorithm; therefore, this nacking mechanism maintains the liveness of the DEXON consensus algorithm.

5.2.1 Correctness

Suppose the last block proposed by node q is $b_{q,j}$, a block $b_{p,i}$ proposed by non-Byzantine node p forced forks q's blockchain and nacks on a block $b_{q,k}$, and after a period of time, q proposes $b_{q,l}$ on its original blockchain. Recall the four acking conditions defined in section 4.2. We now prove that implicit nack shares the four properties and conditions with ack:

1. **All blocks directly acked by $b_{q,k}$ must be received by p.** Because a nacked block will not ack any block, this condition is satisfied.

2. **All historical blocks must be received:** p must have received all blocks $b_{q,h}$ for $0 \leq h < k$. By the definition of nack, if p wants to nack $b_{q,k}$, it must have acked $b_{q,j}$, and by the definition of ack, we know the historical blocks of $b_{q,j}$ are received.

3. **Acking on both sides of a fork is not allowed:** p will violate the rule if $b_{p,i}$ acks on both sides of a fork. By the definition of nack, once p nacks on $b_{q,k}$, it cannot ack on $b_{q,l}$ within $T_{restrict}$. After $T_{restrict}$, p will ack on $b_{q,l}$ and update its timestamp vector, which means it will not nack on $b_{q,k}$; thus, this property is satisfied.

4. **Acking on the same or older blocks is not allowed:** if, during the most recent ack, p acked q on $b_{q,h}$, then $k > h$ must hold. By the definition of nack, p can not nack on any block older than the block it acked most recently. It is possible that node p nacks on the same block proposed by q several times, but only one nack will be counted once so the condition still holds.

Because acking and implicit nack share the same properties, a proof of their correctness directly follows the proof of a reliable broadcast. That is, the DEXON reliable broadcast with ack and implicit nack will output the same and valid blocklattice on every non-Byzantine node eventually.

5.2.2 Liveness

We can prove that if a node does not propose any block after a certain time bound, then, that node will be nacked. Suppose the last time q proposed that a block b is at t_0. b will be received by all non-Byzantine nodes at $t_0 + T_{transmit}$. We can expect at $t_0 + 2T_{transmit} + T_{propose}$, each non-Byzantine nodes will propose a block to ack b. At $t_0 + 2T_{transmit} + T_{propose} + T_{delay}$, all non-Byzantine nodes will propose blocks that nack q, and at $t_0 + 2T_{transmit} + T_{propose} + T_{delay} + T_{restrict}$, each non-Byzantine node will receive at least $2f_{max} + 1$ nack, and q will be nacked and forced to fork. We conclude that the probability q is not nacked after $t_0 + 4t_t + 3t_p + 6\sqrt{4s_t^2 + 3_p^2}$ is less than 10^{-8}.

5.3 κ-level Total Ordering

In this section, we present the κ-level Total Ordering, which is designed to reduce the influence of Byzantine nodes. This design is intended to change the potential functions AHV so that the early delivery condition is satisfied frequently. Because all nodes proposes blocks quite concurrently, the block that has voting rights to AHV (the minimum height in the pending set) in the original total ordering algorithm usually does not ack any other candidate block. Therefore, the rate of early delivery is remarkably very low, that is, less than 3% in our experiment. Our strategy is to budge the voting rights to $(\kappa+1)^{th}$ minimum height block for each node in the pending set from the minimum height block, that is, we replace the potential functions as follows:

1. $AHV_p(b)[q] = \begin{cases} \bot, & \text{if } q \notin ANS_p \\ \bot, & \text{if } b_{q,k} \text{ acks } b, \text{ where } k < \kappa + min\{i | b_{q,i} \in \mathcal{P}\} \\ k, & \text{if } b_{q,k} \text{ acks } b, \text{ where } k = \kappa + min\{i | b_{q,i} \in \mathcal{P}\} \\ \infty, & \text{otherwise.} \end{cases}$
2. $ANS_p(b) = \{\ j : \exists b_{j,k} \in \mathcal{P} \text{ s.t. } b_{j,k} \text{ directly or indirectly acks } b, \text{where } k \geq \kappa + min\{i \mid b_{j,i} \in \mathcal{P}\}\ \}$

This original version is the case of $\kappa = 0$. The present discussion generalize from that case for other values of κ. The experimental effects our modification are listed in Table 4. The κ total ordering achieved success; almost all outputs satisfied the condition of early delivery. This proves that our design significantly improves the early delivery rate relative to other design.

	$\kappa = 0$	$\kappa = 1$	$\kappa = 2$
normal condition	0%	47.3%	100%
large network latency	0 %	0%	0%
large proposing interval	99.9%	100%	100%
large proposing interval & network latency	2.6%	36.1%	83.0%

Table 4: Early delivery rate on 100 blocks per node, the percentages of each delivery type

The complexity analyses follows the analysis in section 5.1.3 and retains the same time and space complexity because our modification does not affect the implementation method.

5.3.1 Correctness and Liveness

The proof of correctness directly follows the original proof because the modification does not change any property in the original proof. On the other hand, the liveness of our design is not directly guaranteed by the original proof, but a slightly modified version of the proof works with some small modifications as follows:

1. $\Gamma_s = \{\ b_{q,j} : j < j' + \kappa, \text{where } b_{q,j'} \in \Gamma \text{ and } q \in \Lambda\ \}$, thus, Γ_s is still a finite set.
2. The potential functions AHV and ANS of block b dare not be worse, because $b_{q,j'>j}$ indirectly acks b if $b_{q,j}$ acks b.

With these two properties, Theorem 5.11 holds for the κ-level total ordering algorithm.

5.4 DEXON Consensus Timestamp Algorithm

In this section, we present the DEXON timestamp algorithm, which ensures that the timestamp is decided from consensus so that the Byzantine nodes cannot bias it. The DEXON timestamp algorithm first greedily chooses a chain so that each block acks its previous block (and the pseudo code is listed in Algorithm 5). The chain does not include all blocks of the chain output by total order algorithm. An example is presented in Figure 12.

Procedure *Main_Chain_Selection*
  Input : ordered chain $< b_0, b_1, ... >$
  Output: ordered chain $< b'_0, b'_1, ... >$ with b'_i acked by b'_{i+1}
  output b_0
  $i = 0, j = 1$
  while $j \leq$*number of elements in input chain* **do**
    if b_i *acked by* b_j **then**
      output b_j
      $i = j$
    $j = j + 1$
  end

Algorithm 5: Main Chain Selection

Once the chain has been constructed, the timestamp problem in the chain becomes easier. The problem can be formally formulated: give a sequence of vectorized timestamps $< t_0, t_1, ... >$, where $t_i = (t_{i,0}, t_{i,1}, ...t_{i,n-1})$, for all i, there exists a j s.t. $t_{i,j} < t_{i+1,j}$ and for all j s.t. $t_{i,j} \leq t_{i+1,j}$, the goal is to obtain a function f s.t. for all i, $f(t_i) \leq f(t_{i+1})$. A trivial method to achieve the requirement is to set $f(t_i) = \sum_j t_{i,j}$, but this can be biased by Byzantine nodes.

Let $\mathcal{V}_s$ be the set of the nodes proposing the following blocks after block s in $chain_{TS}$ and let $|\mathcal{V}_s| = \lfloor n/3 \rfloor + 1$ describe a case that can be guaranteed to contain at least one non-Byzantine node. Define $\mathcal{U}_s = \{Median(T_q(b)) : b$ is the first block proposed by node $q \in \mathcal{V}_s\}$ after block s. We use the median of the timestamp of each block to avoid bias from Byzantine nodes because the maximum number of Byzantine nodes is slightly smaller than $n/3$. We use the maximum of $\mathcal{U}$ to prevent Byzantine nodes from acking the old blocks maliciously so that their blocks have newer timestamps. For blocks not in the picked chain, we use interpolation to compute the timestamp of the block; the pseudocode is listed in Algorithm 7. The median of the ack time vector means that the median of all elements in the vector resists biased attacks because the maximum ratio of the adversary is 1/3.

5.4.1 Correctness

The correctness of integrity in the DEXON timestamp algorithm can be categorized into two cases: the blocks are in the $Chain_{TS}$ or they are not. If the blocks are in the $Chain_{TS}$, the timestamp is decided after the next blocks have been proposed by $f_{max} + 1$ different nodes which exist because of the liveness of the total ordering algorithm. Thus, the median timestamp vector increases because for each element in the ack time vector that always increases and the maximum value of the set $\mathcal{U}$ also increases. In the case of blocks not in the $Chain_{TS}$, the timestamp is interpolated from the timestamp of the closest previous and following blocks in the $Chain_{TS}$; thus the sequence of the timestamp keeps increasing. This concludes the proof regarding the total ordering property of the DEXON timestamp algorithm.

5.4.2 Liveness

For the weak liveness of validity, the algorithm outputs only if there are blocks are received from $2f_{max} + 1$ distinct nodes. Because we set $\Phi = 2f_{max} + 1$, the $2f_{max} + 1$ non-Byzantine nodes will ack the blocks from each other, this means the blocks proposed by non-Byzantine nodes will be either directly or indirectly acked by other non-Byzantine nodes. Then, these blocks will be output by the algorithm.

The weak liveness of agreement can be easily verified because the generation of $Chain_{TS}$ is deterministic.

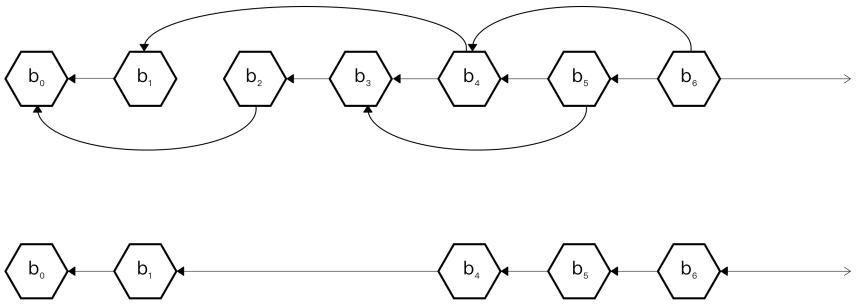

Figure 12: an example of TS_Chain_Generation: the input is $< b_0, b_1, b_2, b_3, b_4, b_5, b_6 >$, and the output is $< b_0, b_1, b_4, b_5, b_6 >$.

Procedure *TS_Chain_Generation*
  Input : ordered chain $Chain_{TS} =< b'_0, b'_1, ... >$ with b'_i acked by b'_{i+1}
  Output: $Chain_{TS}$ $\{b'_i\}$ with timestamp information
  foreach b'_i *in* $Chain_{TS}$ **do**
    $b'_i.timestamp = Median(T(b'_i))$
    output b'_i
Procedure *Compute_Timestamp*
  Input : ordered chain $Chain_{order} =< b_0, b_1, ... >$ and ordered chain $Chain_{TS} =< b'_0, b'_1, ... >$ with b'_i acked by b'_{i+1}
  Output: CompactionChain $\{b_i\}$
  foreach b_i *in* $Chain_{order}$ **do**
    if $b_i \in Chain_{TS}$ **then**
      Let b' be the block corresponding to b_i
      $b_i.timestamp = b'.timestamp$
      output b_i
    else
      interpolate the timestamp of b_i from the closest blocks

Algorithm 7: Compute Timestamp Algorithm

6 DEXON Consensus

In this section, we combine our techniques, introduced previously, and present the DEXON consensus. The main idea is that each node maintains both its own local view and the DEXON compaction chain consists of timestamped blocks.

6.1 DEXON Consensus Algorithm

Each node can propose a block at any time, and then broadcast the information to other nodes. This forms a blocklattice framework from each node's local view. The local view stays consistent through the DEXON reliable broadcast to defend the Byzantine fault. The DEXON consensus algorithm first determines the total ordering of all blocks. After total ordering has been determined, one can compact a DEXON blocklattice structure into the DEXON compaction chain. On the other hand, each node generates a DEXON compaction chain that is guaranteed by the DEXON total ordering algorithm and computes each block by the DEXON timestamp algorithm (which resists bias from numerous adversaries that can comprise up to one-third of the number of total nodes). The consensus algorithm is presented as follows:

Procedure *DEXON for each node p*
  Input : Event of node p proposes a block or receives a block
  Output: DEXON compaction chain and each block has a timestamp
  $Chain_{order} = DEXON\_Total\_Ordering(ReliableBroadcast(b))$
  $Chain_{TS} = TS\_Chain\_Generation(Main\_Chain\_Selection(Chain_{order}))$
  $DEXONCompactionChain = Compute\_Timestamp(Chain_{order}, Chain_{TS})$

Algorithm 8: DEXON Consensus

6.2 Compaction Chain Notarization

Each node should *notarize* blocks on the compaction chain to create notarized block by including a `notary_ack` in each proposed block. A `notary_ack` has the following fields:

`notary_block_hash`	hash of the notary block on compaction chain
`notary_block_height`	height of the notary block on compaction chain, starting at 0
`notary_block_timestamp`	timestamp of the notary block on compaction chain

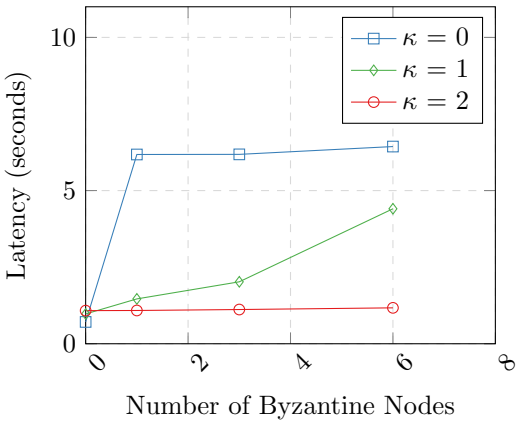

Figure 13: Experiment with Byzantine nodes, $|\mathcal{N}| = 19$. Proposing time is 0.2 seconds and network latency is 0.25 seconds

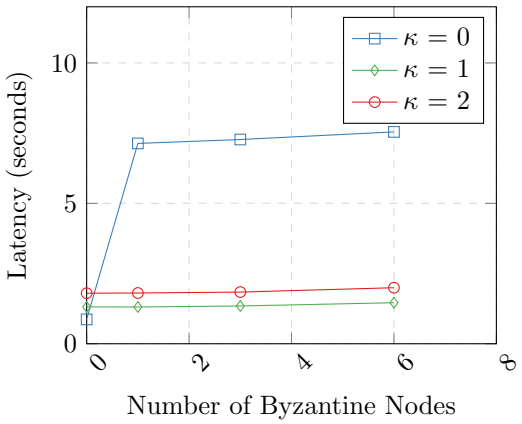

Figure 14: Experiment with Byzantine nodes, $|\mathcal{N}| = 19$. Proposing time is 0.5 seconds and network latency is 0.25 seconds

The notarization mechanism is similar to acking. A node can only notarize a block if it verifies and agrees on all of its history on compaction chain. Once a block in the compaction chain received more than $f_{max} + 1$ `notary_ack`, it is considered *notarized*. Notarization is useful. A light node can perform SPV (simple payment verification) given a notarized block b with $f_{max} + 1$ `notary_ack` as a proof. Because at least one non-Byzantine node notarize b and all of its history on compaction chain, all non-Byzantine nodes are guaranteed to eventually agree on b and its history; thus, the transactions and the timestamps before b can be confirmed.

The former notarized blocks can be periodically pruned and compacted in order to reduce required storage space. The state changes of the system can then be processed, stored, and verified by aggregating the states into a Merkle tree, which guarantees Byzantine agreement on the final state by notarizing the Merkle tree root hash.

7 Simulation Results

We implemented the DEXON consensus algorithm in the GO programming language on an Intel Core-i7 2.9GHz (7th generation) CPU with 16GB 2133 MHz DDR3 RAM. Figure 13 and Figure 14 indicate the influence of Byzantine nodes. When $\kappa = 0$, even one Byzantine node can affect the latency of every node because normal delivery operates most of the time. When $\kappa = 1$ or 2, the figure proves that latency remains close to constant, which means Byzantine nodes cannot affect the latency of other nodes.

The network latency and proposing time are the two main parameters that affect the latency of blocks; the results are illustrated in Figure 16 and Figure 15. We estimate the relation to be $latency = 2T_{transmit} + T_{proposing} + T_{total\_ordering}$ because the total ordering time is quite low, that is, < 0.1 seconds when $|\mathcal{N}| \leq 30$.

The throughput is proportional to the number of nodes in the system and inversely proportional to the proposing time interval of each node because total ordering is non-blocking. The results of our simulation are illustrated in Figure 17 and Figure 18. The simulation experiments confirmed our theoretical estimates.

Figure 19 proves the influence of fail-stop nodes. The x-axis is time and the y-axis is the average number of blocks confirmed by each node, that is, the total number of blocks output by the total ordering algorithm divided by the number of correct nodes. We set six fail-stop nodes among 19 nodes and the fail-stop nodes stopped at the 15th second. On average, the system could be expected to output two blocks per node per second. When the fail-stop happened, all cases were output by the NACK mechanism or with the early delivery condition. The $\kappa = 0$ case had an early delivery rate of almost zero; thus, it only produced output by the NACK mechanism. On the other hand, the $\kappa = 1$ or 2 cases had high early delivery rates and maintained their output rates.

8 Discussion

8.1 Comparison with other blockchains

Current mainstream blockchains such as Bitcoin and Ethereum comply with the longest chain rule (LCR). Both Bitcoin and Ethereum are difficult to scale; each requires a consensus that processes transactions linearly, block by block. Hence, numerous emerging blockchains are attempting to solve that scalability limitation. The following are two vital properties we must guaranteed when designing a DLT:

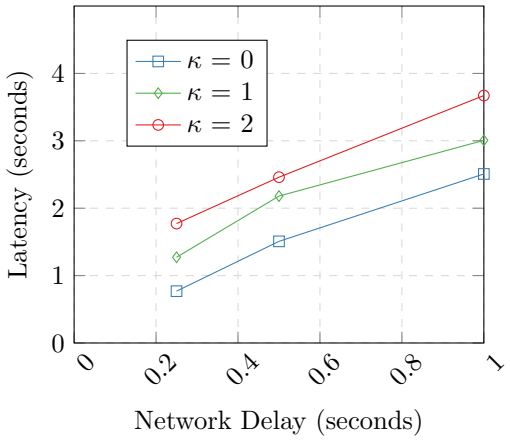

Figure 15: Experiment on network latency with Byzantine nodes, $|\mathcal{N}| = 19$. Proposing time is 0.5 seconds

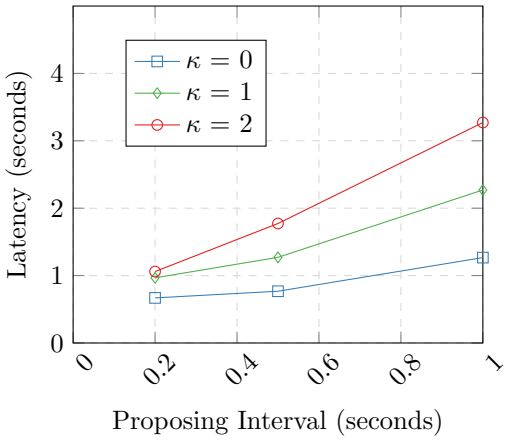

Figure 16: Experiment on proposing latency with Byzantine nodes, $|\mathcal{N}| = 19$. Network latency is 0.25 seconds

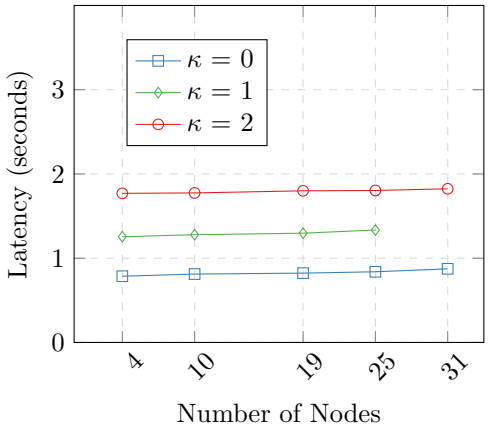

Figure 17: Experiment on network latency without Byzantine nodes. Proposing time is 0.5 seconds and network latency is 0.25 seconds

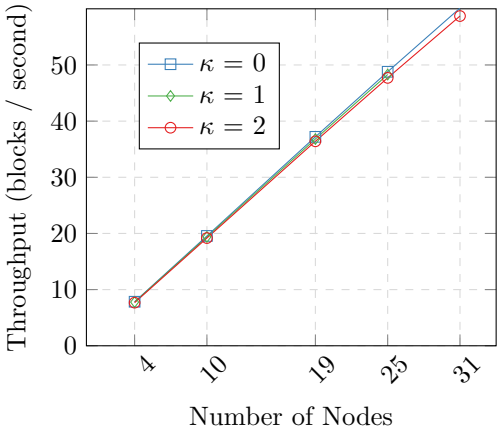

Figure 18: Experiment on throughput without Byzantine nodes. Proposing time is 0.5 seconds and network latency is 0.25 seconds

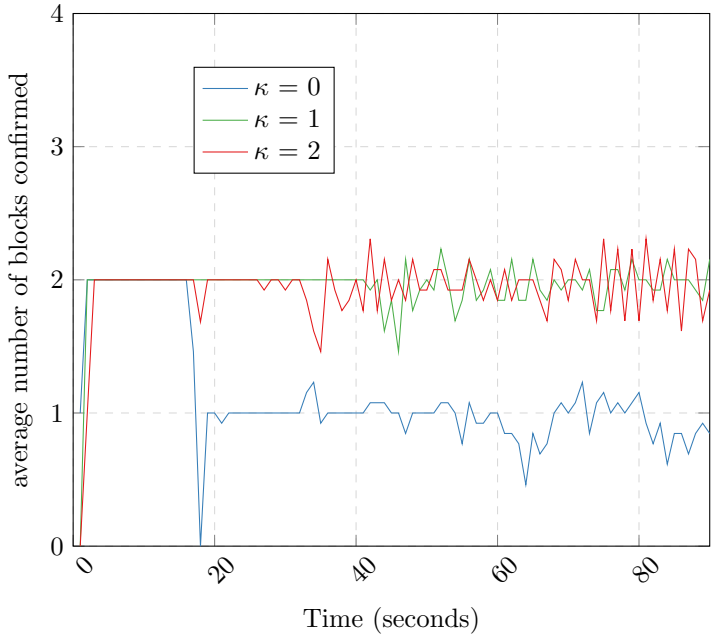

Figure 19: Experiment on fail-stop with NACK with Byzantine nodes, $|\mathcal{N}| = 19$. Proposing time is 0.5 seconds and network latency is 0.25 seconds. Six fail-stop nodes fail and stop at the 15th second.

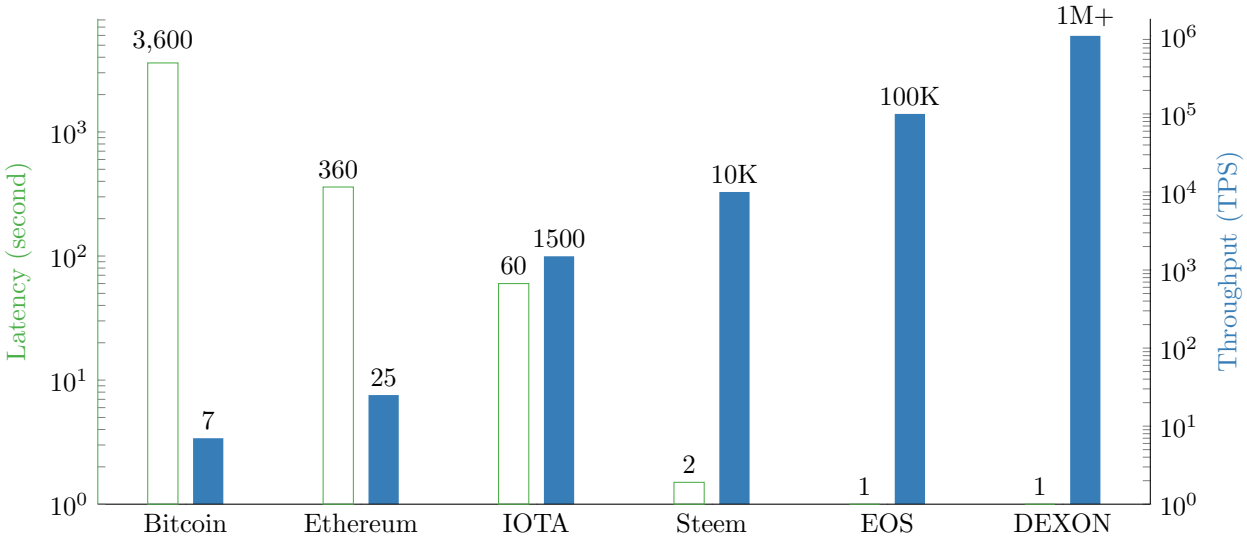

Figure 20: Comparison with other blockchains

1. Safety: guaranteed if all nodes produce the same output and the outputs produced by the nodes are valid according to the rules of the protocol. This is also referred to as consistency of the shared state.
2. Liveness: guaranteed if all non-faulty (honest) nodes participating in consensus eventually produce a value.

The famous trilemma in a blockchain consensus system is as follows:

1. Security: defined by how many attackers a system can tolerate and recover from the failure.
2. Scalability: defined by how many transactions a system can process.
3. Decentralization: defined by how many computational resources each participant must control for the system to run.

Some blockchain systems claim they have solved the scalability problem; IOTA, EOS and Hedera are well-known examples. However, each one still compromises some properties of safety, liveness, and fault tolerance or the trilemma. IOTA proposes the tangle to increase parallelism but it lacks a token economy model and is not highly secure. EOS combines DPoS and PBFT with the tradeoff of centralization. Hedera is secure and scalable, but its liveness is not guaranteed.

A comparison of latency and throughput for different blockchains is illustrated in Figure 20, which proves that the DEXON consensus has the lowest latency and highest throughput among the surveyed blockchains. However, the throughput is eventually bounded by network bandwidth, and it can be configured easily.

8.2 Communication Complexity

Overall, the communication complexity of a two-phase-commit algorithm such as PBFT is $O(k * N^2)$, where k is the number of blocks to be confirmed. On the other hand, the communication complexity of DPoS DEXON aBFT is $O(f * N^2)$, where f is the acking frequency. This means that the overall complexity is not related to the number of blocks to be confirmed. The actual communication overhead becomes negligible when k, which is the number of blocks to be confirmed, is large. With a randomization technique called cryptographic sortition, DEXON's overall communication overhead can be reduced to $O(f * N * log(N))$. The cryptographic sortition is constructed on a verifiable random function (VRF) [HMW18]. The VRF not only drives the consensus; it is also the foundation for network scaling and decentralization.

9 Conclusion

The DEXON blocklattice is presented and is based on novel techniques to compact all the blockchains generated by individual nodes into a globally-ordered blockchain. It achieves fairness, low communication overhead, and extremely low latency. With infinite scalability of transaction throughput and guaranteed transaction finality, real-world DApps that support billions of users can finally be developed.

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.

[Bra87] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, October 1985.

[But14] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. 2014.

[CL99] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[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.

[Fid88] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. pages 55–66, 1988.

[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.

[HMW18] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548*, 2018.

[MMS99] Louise E. Moser and P. M. Melliar-Smith. Byzantine-resistant total ordering algorithms. *Information and Computation*, 150:75–111, 1999.

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

[PSF17] Serguei Popov, Olivia Saa, and Paulo Finardi. Equilibria in the tangle. *CoRR*, abs/1712.05385, 2017.