DEXON Consensus Algorithm

Infinitely Scalable, Low-Latency Byzantine Fault Tolerant Blocklattice

DEXON Foundation

July 6 2018, Draft 5

Abstract

A blockchain system is a replicated state machine (RSM) that needs to be fault-tolerant. The 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 need to reach Byzantine Agreement [BT85] on the total ordering [MMS99] of all transactions. However, existing blockchain consensus algorithms like [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. DEXON consensus algorithm adopts novel *blocklattice* data structure, enabling it to achieve *infinite scalability* and *low transaction confirmation latency* with asymptotically optimal communication overhead.

Keywords: Blockchain, 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 are emerging as [PSF17, BHM18, HMW].

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* since there is no single point of failure. Moreover, the transaction throughput (TPS, Transactions Per Second) 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 with up to one third of Byzantine nodes in the network. This is considered as the optimal resilience to *Byzantine fault tolerance* [Bra87, CL99]. The DEXON consensus algorithm ensures the following two important properties of *distributed ledger technology* (DLT):

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

The high-level concept of DEXON is, there are arbitrary number of nodes maintaining its own blockchain, and each block in the blockchain, proposed by the node individually, has to follow other blocks as a mean of acknowledging those blocks are correct. Then, the node broadcasts the block to every node in the DEXON system. Considering the blocks as vertices and the relation of blocks following it as edges in a graph, it forms a lattice-like structure, which we are calling, *blocklattice*. Blocklattice is the most evolutional and fundamental structure of DEXON consensus. Each validator operates its own blockchain as the vertices in the blocklattice, and the ack action is the edge. After proposing or receiving a block, each node executes the total ordering algorithm and the timestamp algorithm, individually, to get a globally-ordered chain which 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 *DEXON compaction chain*, so that it can be further collected into one block through the Merkle tree technique. Note that, all the procedures are done by each node except broadcasting the block, thus, it achieves low-latency and infinite scalability.

Tolerating up to ⅓ nodes with Byzantine behaviour, DEXON has the following advantages:

Infinite Scalability
Most of blockchain solutions are not able to scale even with the increased resources, and block production remains slow. 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.

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

Explicit Finality
In a PoW blockchain system like Bitcoin or Ethereum, its transaction confirmation is probabilistic and only by waiting for a long sequence of block confirmations can a transaction be considered as *probabilistically finalized*, and the system is thus vulnerable to double spending attacks. For use cases like payment networks, an explicit finality with probability 1 is necessary. In the DEXON consensus, every transaction is confirmed to be finalized with probability 1, secured by the BFT algorithm.

Low Transaction Fees
The transaction fees become higher when the blockchain network is congested. The DEXON consensus is infinitely scalable and has optimal communication overhead, making it able 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. While in DEXON consensus algorithm, there is no single block miner that 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, fair. Furthermore, DEXON also provides cryptographically, provably secure random oracle on the blocklattice data structure for unbiased randomness in smart contract executions.

Energy Efficiency
DEXON consensus can be easily generalized to DPoS in practice. DEXON DPoS has asymptotically optimal computation overhead, making it most energy efficient, and thus making high-capacity decentralized applications more feasible to run efficiently.

Now, we formally state the problem we focused on this paper:

> *Each node will propose new blocks and receive blocks from other nodes, the goal is that each node can maintain a globally-ordered sequence of blocks consists of all valid blocks proposed by nodes.*

In this paper, we ideally solve the problem since all the algorithms we proposed are symmetric and operated individually in one node (except the reliable broadcast the block informs to 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 terminologies and notations used in this paper and then we describe the system model in section 3. Section 4 and section 5 present our reliable broadcast protocol and our timestamp algorithm, respectively. Section 6 presents how the DEXON consensus works, and then we conclude the properties of DEXON in comparison to other blockchains.

2 Basic Terminologies and Notations

In this section, we describe the notation as shown 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 which 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 is the "source message" in 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 call A indirectly acks B. For example, if A acks B and B acks C, A indirectly acks C. An example is provided below for better understanding of the state, as shown in figure 1.

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 node can be tolerance
$T_{transmit}$	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

We use $T_p[q]$ to represent the last updated timestamp of node q in node p's local view, and 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 one node proposed b.

Next, we give the definition of specifications of the reliable broadcast (resp. total ordering algorithm) as the following:

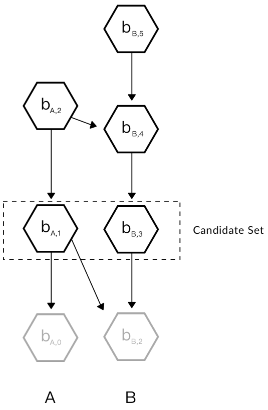

Figure 1: a simple example illustrates the notation: $\mathcal{N} = \{A, B\}$, $\mathcal{F} = \{\}$, $\mathcal{M} = \{b_{A,0}, b_{A,1}, b_{A,2}, b_{B,2}, b_{B,3}, b_{B,4}, b_{B,5}\}$, $\mathcal{P} = \{b_{A,1}, b_{A,2}, b_{B,3}, b_{B,4}, b_{B,5}\}$, $\mathcal{C} = \{b_{A,1}, b_{B,3}\}$. Also, $b_{A,2}$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

1. *Correctness*: all non-Byzantine nodes will eventually generate the same blocklattice (resp. 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]. Also, the DEXON reliable broadcast is non-blocking and infinitely-scalable; all blocks will be finalized instantly. Furthermore, the *weak liveness* is only defined for a fully asynchronous network. All non-Byzantine nodes achieve eventual consistency on the total ordering of delivered blocks. The system eventually progresses (output new delivered total ordering blocks). On the other hand, the *strong liveness* is not achievable in 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 will be finalized and total ordering is delivered. And the overall system is guaranteed not to deadlock or livelock for more than T_2.

3 System Model

We assume the network is fully asynchronous[DSU04], meaning, that the messages between any two nodes may have arbitrary delay, but the messages will eventually be delivered. Moreover, there are no bounds on the differences between the values given by any two clocks from different nodes. We also assume a collision-free hash function H and a secure public-key cryptosystem with a secure digital signature algorithm, $SIGN$, exists.

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

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

4 Mechanisms for Reliable Broadcast

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

4.1 Blocklattice Data Structure

DEXON consensus algorithm runs on a data structure different from traditional blockchains. In DEXON, each node has its own blockchain, cross-referencing others' chains. We call this novel data structure *blocklattice*. A block has the following data field:

`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 the `transactions`, `acks` and `timestamps`
`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

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 shown in figure 2.

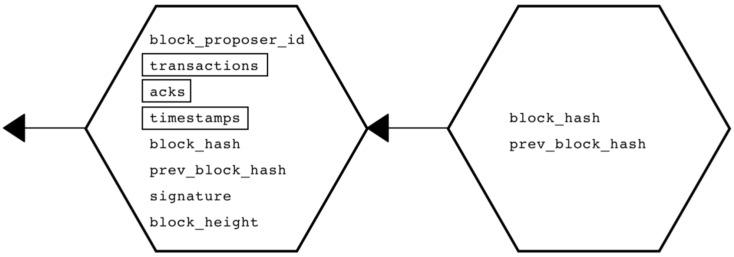

Figure 2: Two Continuous Blocks Proposed by Node p

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 many blockchains from the nodes together with the acks between blocks. An example of DEXON blocklattice data structure is shown in figure 3.

4.2 DEXON Reliable Broadcast Algorithm

The acking algorithm is very 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 as *strongly acked*.

To store the state of a block, we create a local variable *b.acked_count* to store the number of times a block b is acked and *b.strongly_acked* to store if b is strongly acked, as an example in figure 7: the block $b_{E,0}$ is strongly acked. This value is calculated locally and will not be broadcasted to the network with a block. Suppose a block b is a block proposed by node p, we increase *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.

Since all valid blocks will eventually be acked directly or indirectly, by acking to the latest block of each blockchain of other nodes, all blocks will eventually become a part of a node's canonical chain. Thus, a node does not need to ack all blocks proposed by other nodes.

If a block $b_{p,i}$ acks on 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.

```
Procedure Update_Ack_Count for node $p$
    Input  : block $b$
    Output: blocks that are strongly acked
    foreach ack $\alpha$ in $b.acks$ do
        Find acked block $b_{acked}$ by information in $\alpha$.
        Add $b_{acked}.acked\_count$.
        foreach block $b'$ proposed by node $q \in \mathcal{N}$ do
            if $b'$ is indirectly acked by $b$ and $b' \in \mathcal{B}^q$ then
                Add $b'.acked\_count$.
    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

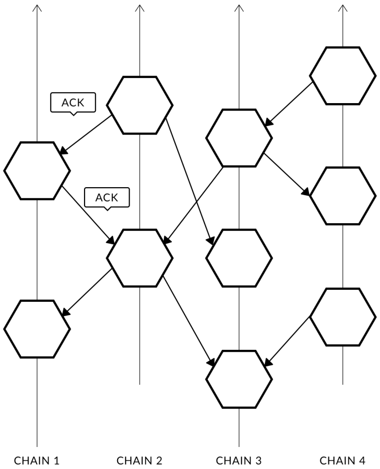

Figure 3: Blocklattice

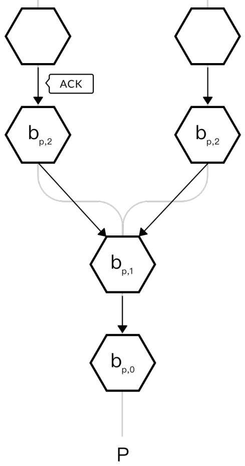

Figure 4: Fork in Blocklattice

2. All historical blocks must be received: p must have received all blocks $b_{q,k}$ for $0 \leq k \leq j$. Note that p does **not** have to receive any other historical blocks from other blockchain except q's blockchain.

3. Acking on both sides of a fork is not allowed: p will violate the rule if it acks on both sides of a fork, including acking by its parents and children. A fork is shown in figure 4, and acking on fork is shown in figure 5.

4. Acking on older blocks is not allowed: if the last time p acked q on $b_{q,k}$, then $j > k$ must hold. A violation is shown in figure 6.

Once a node decides to ack another block b, it will put the information of that block in the ack data structure and store it in the `acks` array. Another important information that must be calculated is the `timestamps` array in a block. A `timestamps` array [Fid88] is a $|\mathcal{N}|$-dimensional array that represents a local view of all other node's clock. When a node p propose a block $b_{p,i}$ that acks on another block $b_{q,j}$, we update the timestamps using algorithm 2.

```
Procedure Update_Timestamps for node $p$
    Input  : block $b$ received from node $q$
    // parameter: $\epsilon$ is a small, predetermined positive constant
    // $p$ updates local view timestamp of $q$ according to $T(b)[q]$
    $T_p[q] = T(b)[q] + \epsilon$
    // $p$ updates local view timestamp of other nodes according to $T(b)$
    for $i$ from 0 to $|\mathcal{N}|$ - 1 do
        if $T(b)[i] > T_p[i]$ then
            $T_p[i] = T(b)[i]$
    end
```

Algorithm 2: Update Timestamps

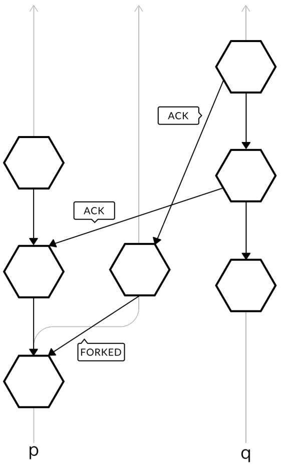

Figure 5: Acking in Blockchain Forks

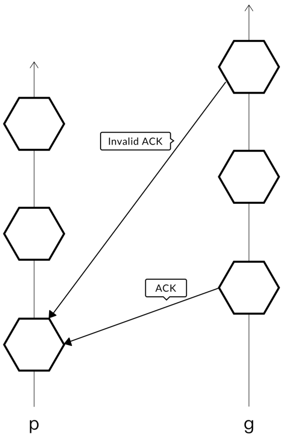

Figure 6: Invalid Acking the same block

The complete DEXON reliable broadcast algorithm is shown in algorithm 3. Information about T_{delay}, $T_{restrict}$, *nack* and *force forking* will be described in section 5.2.

Different from the two-phase commit in PBFT[CL99], we only need one phase for main chain selection. The reason is that we can construct reliable communication channel between non-Byzantine nodes. To achieve this, not only pushing (broadcasting) but pulling information of blocks from other nodes are allowed. With the help of hash function and digital signature, this constructs fast and robust reliable communication channels between non-Byzantine nodes. It can be proven that any node can not fork through one phase acking with reliable channels.

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

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

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

4.2.1 Correctness

We proof the correctness by two different theorems, *agreement* and *integrity.* Note that in section 4.2.1, *a node p outputs block b* means b is output by DEXON reliable broadcast algorithm running on p.

Theorem 4.1. *(Agreement of DEXON Reliable Broadcast) If a non-Byzantine node proposes 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 on it, directly or indirectly. Since there are at least $2f_{max}+1$ non-Byzantine nodes, and these 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 difference from non-Byzantine node, so we consider the case that it proposing 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', and $b \neq b'$. By DEXON reliable broadcast algorithm, this infers we have at least $2f_{max}+1$ acks on b and $2f_{max}+1$ acks on b', which is at least total of $4f_{max}+2$ acks on 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 get a contradiction, so $b = b'$, and this completes the proof. □

```
**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 block format is correct and it is not from a fork
        // sanity check also ensures 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 *acks* array of $b$.
            *Update_Timestamps*$(b)$
        Set $T_p[p]$ to local clock of $p$.
        Put *transactions* array, *acks* array and *timestamps* array into $b$.
        Broadcast $b$.
        *Update_Ack_Count*$(b)$
        Clear $A$ set.
    **end**;
    // nack and restrict
    **when** $|\{\, i : |\{\, j : \Delta_{i,r}(j) > T_{delay} \,\}| > 2f_{max} \,\}| > f_{max}$
        // 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

Theorem 4.3. *(Integrity of DEXON Reliable Broadcast) For any block b, every non-Byzantine node outputs b at most once, and the output set of block forms a valid blocklattice. A valid blocklattice is a blocklattice that follows every acking rules 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 it has more than $2f_{max}+1$ acks, so each block will only be output at most once. For the second part of the argument, most of the cases if b violates the acking rules, it will not pass the sanity check thus will not receive any ack from non-Byzantine nodes. Since there are at most f_{max} Byzantine nodes, b will not be output by any non-Byzantine nodes. The only case b violates the acking rules and passes sanity check is that b is from a fork, since a non-Byzantine node can only receive one branch of a fork. By 4.2, we know that either one branch of a fork will be output by all non-Byzantine nodes, or both of the branches will not be output. Whichever the case is, the output blocklattice is valid. This completes the proof.

□

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

Proof. By theorem 4.1, we know a block proposed by 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 DEXON reliable broadcast within $2t_t + t_p + 6\sqrt{2s_t^2 + s_p^2}$ with high probability under the assumption of our network model. Suppose a node p propose 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 on any legitimate block it received. So 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, we have collect at least $|\mathcal{N} \setminus \mathcal{F}|$ blocks ack on b. Since $|\mathcal{N} \setminus \mathcal{F}| \geq 2f_{max}+1$, b will be 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 blocklattice into the compaction chain and to generate the timestamp for each block in the compaction chain.

5.1 DEXON Total Ordering Algorithm

First, we introduce DEXON total ordering algorithm, ensuring the blocklattice data structure can be compacted into the compaction chain. The DEXON total ordering algorithm is described with a parameter Φ, it can be seen as the threshold of the output criteria. DEXON sets $\Phi = 2f_{max} + 1$ in order to guarantee the liveness of the total ordering algorithm. In the previous section, we introduced the framework of DEXON: each node proposes the block and broadcasts the block. In this scenario, a problem arises: each node receives the block information asynchronously , thus, the order of block from the blocklattice can not be decided directly by the receiving time. So, we introduce the DEXON total ordering algorithm which is a symmetric algorithm and outputs the total order for each valid block.

DEXON Total Ordering algorithm is based on [DKM93], which is a weak total order algorithm [DSU04]. The main idea of DEXON Total Ordering algorithm is to dynamically maintain a directed acyclic graph (DAG) from the received blocks. More precisely, the vertex corresponds to the block and the edge corresponds to the ack relation between blocks. Intuitively, once a block in the graph gets 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 Block Set*, $ABS_p(b) = \{\, j : \exists b_{j,k} \in \mathcal{P}$ s.t. $b_{j,k}$ directly or indirectly acks $b \,\}$, is the set of blocks that ack block b in node p's view. Also, the global Acking Block Set ABS_p for node p is defined by $\bigcup_{b \in \mathcal{C}_p} ABS_p(b)$.

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

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

3. $\#AHV_p = |\,\{\, j : AHV_p(b)[j] \neq \perp \text{ and } AHV_p(b)[j] \neq \infty \,\}\,|$ 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

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

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 all the probable blocks received by p in the future. And the $Grade_p(b_1, b_2)$ function outputs the three possible relations between block b_1 and b_2 in the future: first one is $Grade_p(b_1, b_2) = 1$ means block b_1 always precedes block b_2 regardless of arbitrary following input. Second one 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 are 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 such blocks have higher priority to be output by the algorithm. So, no matter what blocks are received afterwards, the blocks in $\mathcal{C} \setminus \mathcal{A}$ can not precede the blocks in $\mathcal{A}$.

Now, we present 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 produces blocks when specific criteria are satisfied. Note that, no matter the order of the input, the algorithm is executed by each node individually outputs blocks in the same order. The only requirement to the input is the set (incl. acking information) of input of each node is the same eventually.

The criteria consist two parts: the *internal stability* and the *external stability*. Informally, the former one ensures each block in the preceding set has the highest priority to be output than other blocks in the candidate set, and the latter ensures that each block in the preceding set always has higher priority to be output no matter what blocks are received. Let n be the number of total nodes, the *criteria* to output for node p:

- 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) $|ABS_p| = n$, or
  - (b) $\exists b \in \mathcal{A}$ s.t. $\#AHV_p(b) > \Phi$ and
    $\forall b \in \mathcal{A}$ s.t. $|ABS_p(b)| \geq n - \Phi$

We call the output *normal deliver* if it satisfied Internal Stability and Internal Stability (a), and *early deliver* if it satisfied Internal Stability and Internal Stability (b).

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

Theorem 5.1. *(Simplify Criterion for normal deliver) External Stability (a) implies Internal Stability, i.e., if $|ABS_p| = n$, then $\forall b \in \mathcal{C} \setminus \mathcal{A}, \exists b' \in \mathcal{A}$ s.t. $Grade_p(b', b) = 1$*

Proof. First, we show $\mathcal{A}_p \neq \phi$ by contradiction. Assume $\mathcal{A}_p = \phi$, by the definition of preceding set, $\forall b \in \mathcal{C}_p, \exists b' \in \mathcal{C}_p$ s.t. $Preced(b', b) = 1$. But 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 $\mathcal{A}_p = \mathcal{C}_p$, this is correct statement for the theorem. □

Theorem 5.2. *(Simplify Criterion for early deliver) Internal Stability implies External Stability (b), i.e., if $\forall b \in \mathcal{C} \setminus \mathcal{A}, \exists b' \in \mathcal{A}$ s.t. $Grade_p(b', b) = 1$ and $\mathcal{C} \setminus \mathcal{A} \neq \phi$ then*
$\exists b \in \mathcal{A}$ s.t. $\#AHV_p(b) > \Phi$ and $\forall b \in \mathcal{A}$ s.t. $|ABS_p(b)| \geq n - \Phi$,

Proof. $\mathcal{C} \setminus \mathcal{A} \neq \phi$ directly implies $\mathcal{A} \neq \phi$. 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. $|ABS_p(b)| < n - \Phi$, 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 (worst) case $|ABS_p| > \Phi$. When the blocks proposed by all node are 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. $|ABS_p(b)| \geq n - \Phi$. □

Therefore, the *criteria* are simplified into the conditions as following:

- Normal Deliver: $|ABS_p| = n$
- Early Deliver: $\forall b \in \mathcal{C}_p \setminus \mathcal{A}_p, \exists b' \in \mathcal{A}_p$ 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 $r \in ABS_p$ do
                $AHV_p(b_{q,i})[r] = \infty$
        if $q \notin ABS_p$ then
            foreach  $b \in \mathcal{C}_p$ do
                if $b$ is direct or indirect acked by $b_{q,i}$, then
                    $AHV_p(b)[q] = i$,
                    $ABS_p(b) = ABS_p(b) \cup \{q\}$
                else
                    $AHV_p(b)[q] = \infty$
    end;
    when $criterion_p$ holds,
        output $\mathcal{A}_p$ in lexicographical order of 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 $AHV_q(b), ABS_q(b)$
        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 shown in algorithm 4. There are two events: one is a block received and the other one is, the criteria are satisfied. When a block $b_{q,i}$ is received, the algorithm updates the 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 the $AHV_p(b_{q,i})$ according to ABS_p. Otherwise, $b_{q,i}$ is not a candidate. If node q has never been in node p's view, each candidate block updates their 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, ABS_p for all candidate blocks would not change and there is nothing to be done in the algorithm. When second event happened, this means the potential functions of the preceding candidate are good enough so that 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 criteria 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, $AHV_p(b)[r] \neq \bot$ and $AHV_q(b)[r] \neq \bot$, then $AHV_p(b)[r] = AHV_q(b)[r]$.*

Proof. Let $r \in \mathcal{N}$ s.t. $AHV_p(b)[r] \neq \bot$ and $AHV_q(b)[r] \neq \bot$. Consider if both functions $AHV_p(b)[r] = AHV_p(b)[r] = \infty$, then this lemma is correct. W.L.O.G., let $AHV_p(b)[r] = k$, $AHV_q(b)[r] = k'$ or ∞, for some

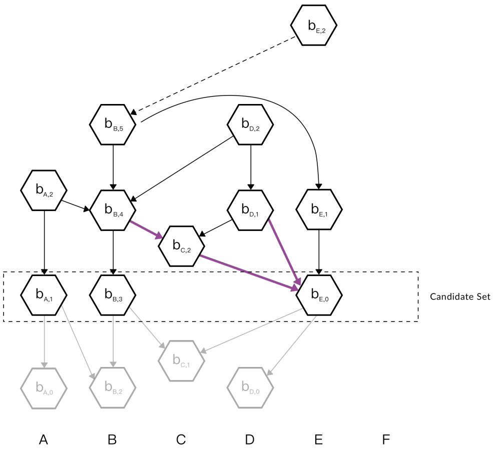

Figure 7: an example of Dexon Total Ordering algorithm: if this is in node C's local view, $\mathcal{S} = \{b_{A,1}, b_{B,3}, b_{E,0}\}$, $\mathcal{A} = \{b_{A,1}, b_{B,3}, b_{E,0}\}$, $AHV_C(b_{A,1}) = (1, \infty, \infty, \infty, \infty, \perp)$, $AHV_C(b_{B,3}) = (\infty, 3, \infty, \infty, \infty, \infty, \perp)$, $AHV_C(b_{E,0}) = (\infty, \infty, 2, 1, 0, \perp)$, $\#AHV_C(b_{A,1}) = 1$, $\#AHV_C(b_{B,3}) = 1$, $\#AHV_C(b_{E,0}) = 3$, $ABS_C(b_{A,1}) = \{A\}$, $ABS_C(b_{B,3}) = \{A, B, D\}$, $ABS_C(b_{E,0}) = \{A, B, C, D, E\}$

integer $k' \neq k$, since 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, 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, $k' = k$, contradiction. □

Lemma 5.4. *(Consistency of Grade) The first time the criteria 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. Since the both of local view of nodes p, q are partial DAG, they will be the same eventually. By lemma 5.3, the function $AHV(b)$ of node p and q are consistency 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 5.5. *(arrival of preceding set) The first time the criteria 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, both of node p, q are normal deliver; second, one is normal deliver and the other is early deliver; third, both of them are early deliver. Case 1: we have $|ABS_p| = n$, then $\mathcal{A}_q \subset \mathcal{C}_q \subset \mathcal{C}_p$ and $|ABS_q| = n$, then $\mathcal{A}_p \subset \mathcal{C}_p \subset \mathcal{C}_q$. It holds in this case.

Case 2: W.L.O.G., let $|ABS_q| = n$ and early deliver happened in node p. First, $\mathcal{A}_p \subset \mathcal{C}_p \subset \mathcal{C}_q$ since $|ABS_q| = n$. Second, we show $\mathcal{A}_q \subset \mathcal{A}_p$ by contradiction. Assume there exists a block $b \in \mathcal{A}_q \setminus \mathcal{A}_p$. By definition of preceding set, $\exists b' \in \mathcal{A}_p$, s.t. $Precede_p(b', b) = 1$. On the other hand, since $b \in \mathcal{A}_q, \forall b'' \in \mathcal{A}_q, Precede_q(b'', b) = 0$. By the lemma 5.3, $b' \in \mathcal{C}_q$ and is also in preceding set. Thus, we have $Precede_q(b', b) = 1$, 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 $\#AHV(b') > \phi$. Since $|ABS_q(b')| > \phi$ and $|ABS_p(b)| \geq n - \phi$, there must exist a node $r \in ABS_p(b) \cap ABS_q(b')$ by pigeonhole principle. Thus, b and b' are received if the block proposed by r is received, this means $b \in \mathcal{C}_q$ On the other hand, $\mathcal{A}_q \subset \mathcal{C}_p$ 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 proof this theorem. For $i = 1$ (the first time the criteria holds), for node p, q, we show that $\mathcal{A}_p^i = \mathcal{A}_q^i$: we proof this by contradiction. Assume $\mathcal{A}_p \neq \mathcal{A}_q$, W.L.O.G., 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 $ABS_q(b') > \phi$. Now, consider two case: either normal deliver or early deliver happened on node p. Case 1: (normal deliver) $|ABS_p| = n$ 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 deliver) we have $|ABS_p(b)| \geq n - \phi$ since $b \in \mathcal{A}_p$. By pigeonhole principle, there exists a node $r \in ABS_p(b) \cap ABS_q(b')$. When the block proposed by node r in both local view of node p and q is received, it shows $b \in \mathcal{C}_q$, 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 consistency in every nodes' local view. Therefore, it follow the same statement of lemma 5.3, 5.4, 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 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 follows the rule that each node proposes block in specific time interval $\Delta_n = t_p + 6s_t$ with high probability. For Byzantine nodes, if they don't propose block, they will be nacked by other nodes after Δ_n. But the worst case is the number of vote to nack is not enough due to network latency, other nodes will ack the block proposing by the node after $T_{restrict}$. Thus, we take $\Delta_f \leq 3 \cdot \Delta_n$ to ensure every block proposed by Byzantine node will be acked or that node will be nacked by other nodes. □

Lemma 5.8. *Let node p be a node, Δ be a specific bound, the following statements are correct:*

1. *if b is received by p at time T, then, either b has been output or $|ABS_p(b)| = |\mathcal{N}_p \setminus \mathcal{F}_p|$ holds at time $T + \Delta$.*
2. *if $b \in \mathcal{P}$ and $|ABS_p(b)| \geq |\mathcal{N}_p \setminus \mathcal{F}_p|$ 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 $|ABS_p(b)| \neq |\mathcal{N}_p \setminus \mathcal{F}_p|$. Since we assume all non-Byzantine node receives block within Δ after it is proposed and all non-Byzantine node will ack it or ack the following blocks proposed by that node. Thus, $|ABS_p(b)| \geq |\mathcal{N}_p \setminus \mathcal{F}_p|$. For the second statement, we have $|ABS_p| \geq |ABS_p(b)| \geq |\mathcal{N}_p \setminus \mathcal{F}_p|$ and the Byzantine must propose blocks during time T and $T + \Delta$, otherwise it will be nacked. Thus, $|ABS_p| = |\mathcal{N}_p|$ holds, and the criterion must hold by **??**. □

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

Proof. Since $\mathcal{C}_p \neq \phi$ and the criterion holds, the 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, Δ be an specific bound, for any time interval $\Delta_o \geq 3\Delta_n + 2\Delta$ the criterion holds and output a non-empty set in the interval Δ_o.*

Proof. We first show that if a block b is received by p at time T, the criterion holds and output a non-empty set before $T + 2\Delta$: by lemma 5.8, the criterion will hold before $T + 2\Delta$. And 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 valid block, total ordering will output it eventually.*

Proof. Assume there exists a valid block b in p's local view s.t. it is not output by DEXON total ordering algorithm. Let the block which acks b be the set Γ and the set Λ be the set of node which proposed blocks in Γ. Since b is not output, no block in Γ will be candidate block in any time. Thus, only the elements in set Γ' or Γ_s could be output, where Γ' is the set of block proposed by the node in $\mathcal{N} \setminus \Lambda$ and Γ_s is the set of block whose height is lower than the block in Γ for each node in Λ. Consider if the algorithm outputs many times, we can find a set of blocks $B \in \Gamma'$ s.t. they are acked by some blocks whose height is higher than the blocks in Γ for some node in Λ. Thus, we have $\#AHV_p(b') < f_{max}, \forall b' \in B$. Now, consider two case: first case is $\#AHV(b) > \phi$, then $b \in \mathcal{A}$. Second case is $\#AHV_p(b) \leq \phi$, but $|ABS_p| > 2f_{max}$ and $\#AHV_p(b') < f_{max} < \phi$. Thus, either b or some blocks in Γ_s will be in $\mathcal{A}$. Since Γ_s is finite set, block b will eventual output by the algorithm. □

5.2 Implicit Nack

FLP impossibility shows there's no deterministic consensus exists in asynchronous network with even only one faulty node. Our DEXON consensus algorithm is deterministic in asynchronous, thus, it has no liveness property. The problem is if some nodes don't propose blocks, the total ordering algorithm won't output. To fix this problem, we change the protocol into a semi-asynchronous mode to circumvent FLP, that is, there exists a time bound between non-Byzantine nodes. This is a reasonable setting in practice since the proposing time and transmission time can be bounded.

Then, we use "implicit voting" mechanism to avoid the halting case as following. 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 $\Delta_{p,d}$ satisfied following property:

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

the algorithm appends node d into S. From a high level aspect, this means in a node's local view, it sees at least $f_{max}+1$ nodes thinks at least $2f_{max}+1$ are proposing blocks, but they didn't think d proposed any block more than T_{delay} after the last time d proposed a block. Each node would not ack blocks proposed by $q \in S$ for a specific time interval $T_{restrict}$. During the interval, if a node receives $2f_{max}+1$ blocks whose inspecting time vector satisfies the property for node q in p's view, p will establish a nack block for node q. In precisely, p nacks q if

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

After q is nacked, p will fork q's original blockchain and generate 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 "helps" the delayed node q to propose a block (the nack block), which will be output to total ordering algorithm to maintain liveness.

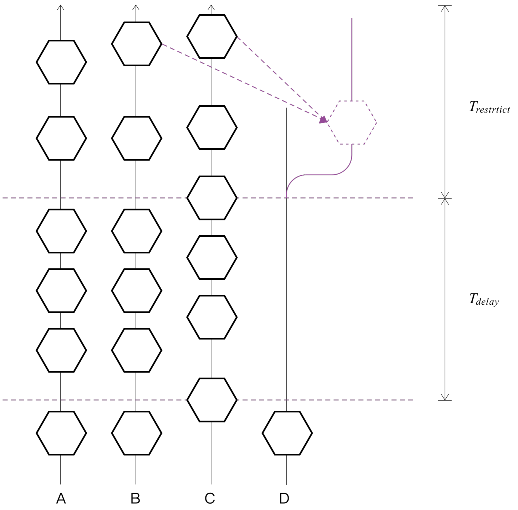

Figure 8: illustration of implicit voting and force forking

The nack is implicit since it is computed locally with the original information on blocklattice. This is why we call the mechanism "implicit voting". Like we use "ack" to vote the validation of other blocks, we use implicit "nacks" to vote if a node seems inactive. Note that we design implicit nack 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. Also, a node is forbidden to ack on a node it nacked during $T_{restrict}$, just like a node can not ack on a fork. The only difference is that when a node p nacks node q, it does not update its timestamp vector. That means $T_p[q]$ is updated only if q proposes a block. Since nack follows the conditions and properties of ack, the correctness and liveness can directly follow the proof of acking in reliable broadcast. Since there exists a time bound between non-Byzantine nodes, each non-Byzantine node will force node q to fork after a specific time interval. After the force forking holds, each node will only ack the block proposed by q (indirectly or directly) acks the empty block, which means q can only propose block on the blockchain forked by other nodes. Also, the empty block will be recorded in the globally-ordered chain, this is the evidence that the block violates the protocol. If the number of the 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 Gaussian distribution $\mathcal{G}(t_t, s_t)$ and the proposing time $T_{propose}$ be Gaussian distribution $\mathcal{G}(t_p, s_p)$. One node p considers another node q inactive (didn't propose blocks) if it sees at least $f_{max}+1$ nodes didn't receive any block from q after $T_{delay} = T_{propose} + T_{transmit}$, and it can be set as 6 times of standard deviation so that the probability of the node is active but is considered as faulty is less than 10^{-8}. On the other hand, $T_{restrict} = T_{propose} + T_{transmit}$ and it can be set as 6 times of standard deviation higher $(t_p + t_t)$ so that it 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 it output the nack block created by other nodes. T_{ban} is a predefined parameter, which can grow exponentially with the number of time 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 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 for membership needed in total ordering algorithm, keeping the liveness of DEXON consensus algorithm.

5.3 DEXON Consensus Timestamp Algorithm

In this section, we present DEXON timestamp algorithm, ensuring that the timestamp is decided from consensus so that the Byzantine node cannot bias it. DEXON timestamp algorithm first greedily chooses a chain so that each block acks its previous block (and the pseudo code is shown in algorithm 5). Note that the chain does not include all the blocks of the chain output by total order algorithm. An example is given in figure 9.

```
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 is constructed, the timestamp problem in the chain becomes easier. We formally formulate the problem: give a sequence of vectorized timestamps $< t_0, t_1, ... >$, where $t_i = (t_{i,0}, t_{i,1}, ...t_{i,n-1})$, and 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 find a function f s.t. for all i, $f(t_i) \leq f(t_{i+1})$. A trivial way to achieve the requirement is letting $f(t_i) = \sum_j t_{i,j}$, 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 $|\mathcal{V}_s| = \lfloor n/3 \rfloor + 1$ is the case that can be guaranteed to contain at least one non-Byzantine node. $\mathcal{U}_s = \{Median(T_q(b)) : b$ 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 Byzantine nodes' bias since the maximum number of Byzantine nodes is slightly smaller than $n/3$. We use the maximum of $\mathcal{U}$ to avoid Byzantine nodes ack the old blocks maliciously so that their blocks have newer timestamp. For blocks not in the picked chain, we use interpolation to compute the timestamp of the block, as shown in algorithm 7. The median of the ack time vector means the median of all the elements in the vector resists biased attacks since the maximum ratio of the adversary is 1/3.

5.3.1 Correctness

For the correctness of integrity in DEXON timestamp algorithm, it can be categorized into two cases: the blocks in the $Chain_{TS}$ and otherwise. In the case of blocks in the $Chain_{TS}$, the timestamp is decided after the next blocks proposed by $f_{max} + 1$ different nodes which exist because of the liveness of total ordering algorithm. And the median of timestamp of vector increases since for each element in the ack time vector always increases so does the maximum value of the set $\mathcal{U}$. In the case of blocks not in the $Chain_{TS}$, the timestamp is interpolated from the timestamp of the closest previous and after blocks in the $Chain_{TS}$, thus the sequence of the timestamp keeps increasing. This concludes the total ordering property of DEXON timestamp algorithm.

5.3.2 Liveness

For the weak liveness of validity, the algorithm outputs only if there are blocks received from distinct $2f_{max} + 1$ nodes. Since 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.

For the weak liveness of agreement, it is easy to check since the $Chain_{TS}$ generation is deterministic.

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 its own local view and the DEXON compaction chain within timestamped blocks.

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

Algorithm 7: Compute Timestamp Algorithm

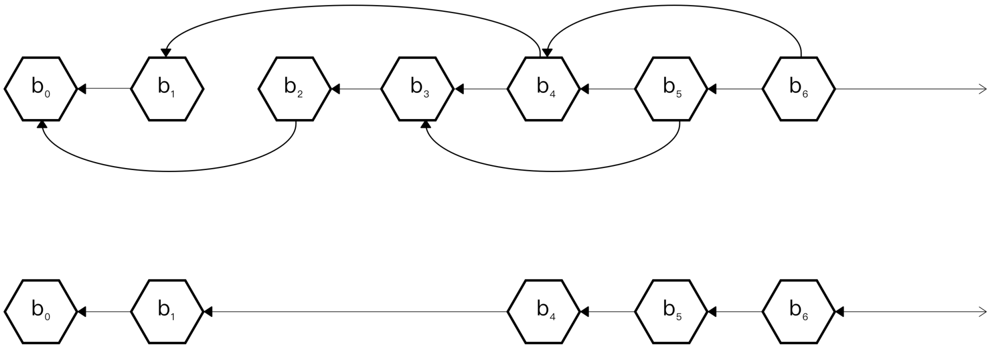

Figure 9: an example of TS_Chain_Generation: the input is $< b_0, b_1, b_2, b_3, b_4, b_5, b_6 >$, and the output is $< b_0, b_1, b_4, b_5, b_6 >$.

6.1 Blocklattice Compaction

The old blocks can be periodically pruned and compacted to reduce the storage space usage and a signed-state can be generated by nodes with over ⅓ signatures in the blocks referencing to a block on the DEXON compaction chain. 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 of the state machine by including the Merkle tree root hash in the blocks proposed.

Each node can propose a block in any time, then broadcast the information to other nodes. This forms a blocklattice framework from each node's local view. The local view stays consistent through reliable broadcast in order to defend the Byzantine fault. DEXON Consensus algorithm first determines the total ordering of all blocks. After the total ordering is determined, one can compact a DEXON blocklattice structure into the DEXON compaction chain. On the other hand, each node generates the DEXON compaction chain which is guaranteed by DEXON total ordering algorithm and compute each block by DEXON Timestamp algorithm which resists bias from adversaries of a number of up to one third the number of total nodes. The consensus algorithm is shown below:

```
Procedure DEXON for each node p
    Input  : Event of node $p$ proposes a block, or receiving a block
    Output: (ordered) DEXON compaction chain and each node has 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

7 Discussion

7.1 Comparing to other blockchains

Current mainstream blockchains comply with the longest chain rule (a.k.a LCR), like Bitcoin and Ethereum, both of which are difficult to scale because the consensus processes transactions linearly, block by block. Hence, many blockchains emerge that target to solve the scalability limitation. The following are two important properties we need to guaranteed when designing a DLT:

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.

And the famous trilemma in blockchain consensus system:

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 needs to control for the system to run.

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

7.2 Communication Complexity

Overall, the communication complexity of a two-phase-commit algorithm like 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. That is, the overall complexity is not related to the number of blocks to be confirmed. The actual communication overhead becomes negligible when k, the number of blocks to be confirmed is large. Furthermore, applying a randomization technique called cryptographic sortition, DEXON's overall communication overhead reduces to $O(f*N*log(N))$. The cryptographic sortition is built on verifiable random function (VRF) [HMW18]. The VRF not only drives the consensus, it will also be the foundation for network scaling and decentralization.

8 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 made possible.

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.