PRESYNC: An Efficient Transaction Synchronization Protocol to Accelerate Block Propagation

Yixin Li ¹, Liang Liang ², Yunjian jia ², and Wanli Wen ²

¹Chongqing University
²Affiliation not available

October 31, 2023

Abstract

Block propagation is a critical step in the consensus process, which determines the fork rate and transaction throughput of public blockchain systems. To accelerate block propagation, existing block relay protocols reduce the block size using transaction hashes, which requires the receiver to reconstruct the block based on the transactions in its mempool. Hence, their performance is highly affected by the number of transactions missed by mempools, especially in the P2P network with frequent arrival and departure of nodes. In this paper, we introduce Presync, a transaction synchronization protocol that can reduce the difference of transactions between the block and the mempool with controllable bandwidth overhead. It allows mining pool servers to synchronize the transactions in candidate blocks before the propagation of a valid block. High-bandwidth mode conducts a full synchronization of the candidate block using short hashes, and the Merkle root is utilized to match the valid block. Low-bandwidth mode provides a lightweight synchronization by identifying the unsynchronized transactions, so that the missing transactions can be detected with a low redundancy. We study the performance of Presync through the stochastic modeling and experimental evaluations. The results illustrate that high and low-bandwidth modes can respectively reduce the end-to-end delay of compact block by 60% and 78% with bandwidth usages 25KB and 63KB, in a network with 5 active pool servers and 2/3 online probability of full nodes.
**Presync: An Efficient Transaction Synchronization Protocol to Accelerate Block Propagation**

Yixin Li, Liang Liang*, Yunjian Jia, Member, IEEE, and Wanli Wen, Member, IEEE

**Abstract**—Block propagation is a critical step in the consensus process, which determines the fork rate and transaction throughput of public blockchain systems. To accelerate block propagation, existing block relay protocols reduce the block size using transaction hashes, which requires the receiver to reconstruct the block based on the transactions in its mempool. Hence, their performance is highly affected by the number of transactions missed by mempools, especially in the P2P network with frequent arrival and departure of nodes. In this paper, we introduce Presync, a transaction synchronization protocol that can reduce the difference of transactions between the block and the mempool with controllable bandwidth overhead. It allows mining pool servers to synchronize the transactions in candidate blocks before the propagation of a valid block. High-bandwidth mode conducts a full synchronization of the candidate block using short hashes, and the Merkle root is utilized to match the valid block. Low-bandwidth mode provides a lightweight synchronization by identifying the unsynchronized transactions, so that the missing transactions can be detected with a low redundancy. We study the performance of Presync through the stochastic modeling and experimental evaluations. The results illustrate that high and low-bandwidth modes can respectively reduce the end-to-end delay of compact block by 60% and 78% with bandwidth usages 25KB and 63KB, in a network with 5 active pool servers and 2/3 online probability of full nodes.

**Index Terms**—Blockchain, propagation latency, transaction synchronization, stochastic model.

**I. INTRODUCTION**

Blockchain is a secure and distributed ledger that achieves transaction consistency across untrusted entities. It has attracted much attention from society, industry and academia as a promising technology to enable Web 3.0, a decentralized, permissionless, and trustless internet [1], [2]. Although public blockchains match the core ideas of Web 3.0, the transaction throughput of existing blockchains, 7 transaction per second (TPS) in Bitcoin [3] and 15 TPS in Ethereum [4], cannot satisfy the demand of world-wide decentralized applications.

The throughput limitation in the public blockchain relates to the performance-security dilemma [5], [6]. To reduce forks and maintain a high security level, block propagation delay should be much less than block interval, thus a blockchain system has strict limits on block size and block generation rate, which result in a low transaction throughput. An effective method to address this issue is accelerating the block propagation, so that more transactions can be carried by one block without increasing the propagation delay. Block propagation delay is a combination of round-trip time (RRT), block transmission time in the network, and block validation time for checking the header and transactions [7]. The standard block relay protocol of Bitcoin requires peers to transmit and validate the full block with all transactions, even though a large proportion of the transactions in the block already exist in their memory pools (or called mempools). This causes long block transmission time and block validation time, since a block can be larger than 1MB and contains thousands of transactions [8]. To compress a block, Bitcoin’s current block relay protocol, Compact [9], allows the transactions in the block to be transmitted by short hashes. Based on short hashes, the receiver can retrieve the transactions from its mempool and reconstruct the original block. Compared with the standard protocol, the performance of Compact is affected by the number of transactions missed by mempools, which is significant in a peer-to-peer (P2P) network with frequent arrival and departure of peers. Other block relay protocols, such as XThin [10], Xthinner [11], and Graphene [12], use Bloom filters to encode transactions for block compression. Similar to Compact, these protocols are also sensitive to the number of transactions missed by mempools, due to the tight coupling of block propagation with transaction synchronization.

In this paper, we propose a transaction synchronization protocol called Presync that can reduce the difference between the block and the mempool’s transactions to accelerate the block propagation. The key idea of Presync is to synchronize the transactions in candidate blocks before the propagation of a valid block. To increase bandwidth efficiency, transaction synchronization can only be initiated by the peers with high consensus resources (hashrate or stake), e.g., mining pool servers in Bitcoin [13] and validators in Ethereum [14]. Then, synchronization messages are constructed by the short hashes of transactions in candidate blocks. During the synchronization, any peers can request the missing transactions based on short hashes, so that most of the transactions can be transmitted and validated before the block propagation. The main contributions of this paper can be summarized as follows.

- We propose a transaction synchronization protocol to reconcile the mempools among peers based on candidate blocks, enabling a fast end-to-end delivery of the valid block. Low-bandwidth mode provides a lightweight synchronization by identifying the unsynchronized transactions in a candidate block, so that the missing transactions can be detected with a low redundancy. High-bandwidth mode allows peers to cache a condensed version of the candidate block, and the Merkle root is utilized to match the header of a valid block efficiently.

- We present a stochastic model to analyze the steady number of unsynchronized transactions, and the relation
between the block sender and the receiver’s mempools. By studying the impact of low and high-bandwidth modes on the size of a compact block and the number of missing transactions, we derive the key performance metrics in Presync, including the end-to-end delay, bandwidth usage, transaction throughput, and forking probability.

- We evaluate the performance of Presync through the numerical calculation and experimental simulation. The simulation results validate the stochastic model and show that low-bandwidth mode can reduce the end-to-end delay of compact block by 28%–74% with bandwidth usage 8–45KB, and high-bandwidth mode can reduce the delay by 36%–90% with bandwidth usage 13–126KB, when the number of active pool servers increases from 1 to 10.

The rest of the paper is organized as follows. Section II describes the consensus process in Bitcoin network, and reviews standard block relay and compact block relay. Section III introduces the workflows, message constructions, and message caching in Presync. Section IV presents a stochastic model to analyze the performance of low and high-bandwidth modes. Then, the analytical model is validated by simulation experiments in Section V. Section VI summarizes the related works on block propagation acceleration, and Section VII concludes the paper.

II. PRELIMINARIES

A. Blockchain Network and Consensus Process

We consider the Bitcoin network consisting of lightweight nodes, full nodes, miners, and pool servers [15]. Lightweight nodes are resource-constrained devices such as smartphones, which are allowed to issue transactions without storing the full blockchain ledger. Full nodes are the peers maintaining a full copy of the blockchain ledger, responsible for the validation and propagation of new transactions and blocks. Miners are the mining machines engaged in hash operations to find a valid block that meets a given difficulty target. Due to the high difficulty, miners participate in mining pools and share the block reward as a regular income. Pool servers are deployed by the operators of mining pools, which use pool-mining protocols [16], [17] to coordinate the activities of miners and distribute the block reward. Pool servers are connected to dozens of full nodes and maintains an up-to-date mempool to generate a candidate block as a mining task.

The consensus process of a transaction can be summarized as follows: 1) Transaction generation: The lightweight node creates and signs a new transaction, and transmits it to a full node. 2) Transaction propagation: The full node validates the transaction and forwards it to the randomly connected full nodes, known as flooding or gossiping [18], [19]. 3) Block mining: The pool server receives the transaction and includes it in a candidate block for pooled mining. 4) Block propagation: The miner finds a valid block as one confirmation of the transaction, then the block is propagated to all full nodes using gossiping and block relay protocols. 5) Block accumulation: The new block waits for a given number of subsequent blocks, as a sufficient proof that the transaction cannot be reversed.

The consensus process of a transaction can be summarized as follows: 1) Transaction generation: The lightweight node creates and signs a new transaction, and transmits it to a full node. 2) Transaction propagation: The full node validates the transaction and forwards it to the randomly connected full nodes, known as flooding or gossiping [18], [19]. 3) Block mining: The pool server receives the transaction and includes it in a candidate block for pooled mining. 4) Block propagation: The miner finds a valid block as one confirmation of the transaction, then the block is propagated to all full nodes using gossiping and block relay protocols. 5) Block accumulation: The new block waits for a given number of subsequent blocks, as a sufficient proof that the transaction cannot be reversed.
Prefilled transactions are the transactions in the block but not in the mempool of node A. Short transaction IDs are the 6-byte hashes of the overlapping transactions between the block and the mempool. Based on the short IDs, node B can retrieve the required transactions from its mempool. If there are any missing transactions, a `getblocktxn`-`blocktxn` roundtrip is needed for recovery, shown in Fig. 1(b). Then, node B can reconstruct the block based on `cmpctblock` and `blocktxn`. In Compact, the sizes of `cmpctblock`, `getblocktxn`, and `blocktxn` depend on the number of missing transactions, which are caused by frequent arrival and departure of nodes in the P2P network, known as network churn [21], [22].

III. PRESYNC OVERVIEW

The primary goal of Presync is to reduce the difference between the block and the mempool’s transactions. We achieve this goal by synchronizing the transactions in candidate blocks.

A. Low-Bandwidth Mode

Low-bandwidth mode provides a lightweight synchronization by addressing the redundancy of short IDs. In low-bandwidth mode, the pool server only synchronizes the transactions that have not been synchronized by any pool servers at previous block heights\(^1\). As shown in Fig. 2(a), the synchronization message in low-bandwidth mode, `synclow`, consists of a header II and the short IDs of unsynchronized transactions. Synchronized and unsynchronized transactions (simply called synced and unsynced txns) are distinguished by the marks in the mempool, detailed in section D. The header II has a low-difficulty hash for avoiding spamming and redundant synchronization. Specifically, header I, header II, and header III are defined as follows: 1) A candidate block is generated by the pool server, and the hash of block header has a zero difficulty, which is defined as a header I. 2) A `synclow` message is constructed by the pool server, and the hash of block header meets a low difficulty target, which is defined as a header II. 3) A valid block is found by the pool server, and the hash of block header meets the high difficulty target, which is defined as a header III. Since the difficulty (depending on the number of leading zeros in a hash) for header II is much lower than that for header III, it is expected that most of the synchronization from pool servers can be finished before the block propagation.

The `synclow` is propagated from a pool server to full nodes for synchronizing the transactions in their mempools. As shown in Fig. 1(c), node B enables the low-bandwidth mode by setting the boolean to 0 in a `synccon`. In response to the `synccon`, node A sends a `syncaddr` containing the IP addresses that have established synchronization connections, as a reference for node B. To forward a `synclow`, node A sends an `inv` that contains two fields [23]: a type field for identifying the synchronization message so that its low-difficulty hash will be accepted by node B, and a hash field for transferring the header II’s hash. Upon receiving the `synclow`, node B first validates the header’s hash, then synchronizes its mempool with the short IDs in the `synclow`. The missing transactions are requested and recovered by the `getblocktxn` and the `blocktxn`, respectively.

After the synchronization, the number of the overlapping transactions between the block and node A’s mempool increases, thus a larger proportion of transactions in the `cmpctblock` will be transmitted by short IDs, resulting in a lower latency. When the `cmpctblock` is received by node B, more transactions can be retrieved from its mempool, which shortens the time for transaction recovery and validation. With enough synchronization from pool servers, node B only needs to validate the block header and coinbase transaction.

B. High-Bandwidth Mode

High-bandwidth mode is designed for fast block reconstruction, with higher redundancy of short IDs and memory overhead. In high-bandwidth mode, the pool server synchronizes all the transactions in a candidate block, which allows a complete validation of transactions before the block propagation. As shown in Fig. 2(b), the `synchigh` consists of a header II, a coinbase transaction, and short transaction IDs. When a `synchigh` is received, the full node first validates the header, then requests the missing transactions using a `getblocktxn`. Once the node has all the transactions in the `synchigh`, the transaction fees in the coinbase transaction can be validated based on the inputs and the output of transactions [18]. After that, the `synchigh` is cached by the full node for matching the valid block.

After the synchronization, the delivery of a valid block is illustrated in Fig. 1(d). Node A directly sends the header III to the IP addresses that have enabled the high-bandwidth mode. Since the hash of header III is different from that of header II, node B retrieves the transactions in the valid block from its cache based on the Merkle root. The Merkle root is calculated by the pair-wise hashes in a binary tree, which summarizes

\(^1\)The block height identifies the position of a block in the blockchain ledger.
C. Data Structures in Presync

Presync introduces five new messages: *syncon*, *syncaddr*, *syncoff*, *synclow*, and *synchigh*.

1) The *syncon* is used to enable low or high-bandwidth mode, and its structure is shown in Table I(a). The first field in this message advertises the protocol version of the node for compatibility checking. The second field contains a boolean to identify the requested bandwidth mode, where “0” represents the low bandwidth and “1” represents the high bandwidth. The node receiving a *syncon* will first examine the protocol version. If the version is compatible, the node extracts the IP address from the packet header, and marks this address as an outbound connection in its local IP address list.

2) The *syncaddr* is used to acknowledge the synchronization connection and share the known IP addresses. As shown in Table I(b), the first field identifies the bandwidth mode for the receiver to determine an inbound connection. The second and third fields introduce the IP addresses that already have synchronization connections. Based on the IP addresses, a newly connected node can established more connections using the *syncon*.

3) The *syncoff* is used to close a synchronization connection, shown in Table I(c). Both of the fields contain a boolean “0” or “1”. In the first field, “0” represents the low bandwidth and “1” represents the high bandwidth. In the second field, “0” represents the outbound and “1” represents the inbound. The *syncoff* will be used in the following cases:

- When the outbound bandwidth is overloaded, the node first uses a *syncoff*(1,1) to close the low-bandwidth inbound connections of its neighbors. If there is no high-bandwidth mode, the node uses a *syncoff*(0,1) to close the low-bandwidth inbound connections.
- When the inbound bandwidth is overloaded, the node uses a *syncoff*(1,0) or *syncoff*(0,0) to close the outbound connections of its neighbors.
- When the local cache is overloaded, the node uses a *syncoff*(1,0) to close the high-bandwidth outbound connections of its neighbors, and clears the *synchigh* that are received from these connections.

4) The *synclow* is used to synchronize the mempools of the nodes in low-bandwidth mode. The structure of the header in a *synclow* is the same as a conventional block header [23], which consists of the version, previous block hash, Merkle root, timestamp, difficulty target, and nonce. Among them, the difficulty target in a *synclow* is lower than that in a valid block. Presync limits the number of *synclows* or *synchighs* by adjusting the difficulty target; for example, if the difficulty in the *synclow* is 10 times lower than that in the valid block, the message interval will be 1 minute. A pool server cannot propagate the *synclow* and the *synchigh* with a same header hash simultaneously, since they produce the same *inv* which will be answered by a full node only once. Before forwarding a *synclow*, the full node needs to retrieve all the short IDs in the *synclow* and recover the transactions missed by its mempool. In this way, each hop can request and validate any transactions using the short IDs in the *synclow*, ensuring that only the short IDs of valid transactions can be propagated across the network.

5) The *synchigh* is used to synchronize the mempools of the nodes in high-bandwidth mode. The major difference between the *synchigh* and the *synclow* is the coinbase transaction. The input of coinbase transaction contains the block height, and the output contains the Bitcoin address of a pool server. These messages are essential for a full node to avoid redundant caching, so that the *synchigh* of a pool server will be cached only once at each height. On the other hand, all the other transactions in a candidate block will be synchronized by the *synchigh*, as the necessary information to validate the coinbase transaction and the Merkle root. The full node validates the total amount of transaction fees in the coinbase transaction by calculating the inputs and outputs of all transactions. Note that the order of short IDs in a *synchigh* must be in line with the transactions in a candidate block. According to the

<table>
<thead>
<tr>
<th>TABLE I</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DATA STRUCTURES IN PRESYNC</strong></td>
</tr>
<tr>
<td>(a) The structure of <em>syncon</em></td>
</tr>
<tr>
<td><strong>Size</strong></td>
</tr>
<tr>
<td>4 bytes</td>
</tr>
<tr>
<td>1 bytes</td>
</tr>
<tr>
<td>(b) The structure of <em>syncaddr</em></td>
</tr>
<tr>
<td><strong>Size</strong></td>
</tr>
<tr>
<td>1 bytes</td>
</tr>
<tr>
<td>1+ bytes</td>
</tr>
<tr>
<td>Variable</td>
</tr>
<tr>
<td>(c) The structure of <em>syncoff</em></td>
</tr>
<tr>
<td><strong>Size</strong></td>
</tr>
<tr>
<td>1 bytes</td>
</tr>
<tr>
<td>1 bytes</td>
</tr>
<tr>
<td>(d) The structure of <em>synclow</em></td>
</tr>
<tr>
<td><strong>Size</strong></td>
</tr>
<tr>
<td>80 bytes</td>
</tr>
<tr>
<td>1+ bytes</td>
</tr>
<tr>
<td>Variable</td>
</tr>
<tr>
<td>(e) The structure of <em>synchigh</em></td>
</tr>
<tr>
<td><strong>Size</strong></td>
</tr>
<tr>
<td>80 bytes</td>
</tr>
<tr>
<td>300 bytes</td>
</tr>
<tr>
<td>1+ bytes</td>
</tr>
<tr>
<td>Variable</td>
</tr>
</tbody>
</table>
order of transactions, the full node can construct a Merkle tree using transaction hashes and validate the Merkle root. By the time a block header is received by the full node, the required transactions will be matched based on the Merkle root, which is a double-SHA256 hash with strong collision resistance [24].

D. Marks and Caching in Presync

There are three challenges in Presync: how to distinguish between the low and high-bandwidth modes at each peer; how to distinguish between the synchronized and unsynchronized transactions in the mempool; how to cache the synchigh with low redundancy.

1) The first challenge is addressed by adding marks in the IP address list. Specifically, there are four types of marks (0,0), (0,1), (1,0), (1,1) representing the low-bandwidth outbound connection, the low-bandwidth inbound connection, the high-bandwidth outbound connection, and the high-bandwidth inbound connection, respectively. The mark will be added or removed as follows:

- When a synccon is received, the node identifies the bandwidth mode in the synccon, then extracts the IP address from the packet header and marks this address as an outbound connection using (0,0) or (1,0).
- When a syncaddr is received, the node identifies the bandwidth mode in the first field and marks the IP address as an inbound connection using (0,1) or (1,1).
- When a syncoff is received, the node removes the mark according to the boolean of the syncoff, e.g., the syncoff(0,0) removes the mark (0,0).
- When a syncoff is sent by a node, the removed mark is contrary to the second boolean of the syncoff, e.g., the syncoff(0,0) removes the mark (0,1).

2) The second challenge is addressed by adding marks in the mempool, where (0) represents a unsynchronized transaction and (1) represents a synchronized transaction. The transactions received by the mempool during the initial transaction propagation are marked as (0) by default. When a synclow or synchigh is constructed or received, the node marks all the selected transactions as (1) in its mempool. The transaction and its mark will be removed from the mempool only when this transaction has been confirmed by a valid block.

3) The synchigh is cached by full nodes to match the block header efficiently. At each block height, the synchigh of each pool server will be cached by a full node at most once to avoid redundancy. As mentioned before, a redundant synchigh is distinguished based on the block height and Bitcoin address in the coinbase transaction. Specifically, when a synchigh is received, the node compares the block height of synchigh with the height of its blockchain ledger: if the former is less than or equal to the latter, the new synchigh is ignored by the node; if the former is greater than the latter by one, the new synchigh replaces the cached synchigh that has the same Bitcoin address; if the former is greater than the latter by two or more, it means that there are missing blocks, so the node downloads these blocks and compares the block heights again. Whenever a new block is received, the node inspects the height of synchigh in the local cache. The synchigh that falls behind the blockchain ledger by six block heights is considered as a stale message and is removed from the cache.

IV. STOCHASTIC MODEL FOR PRESYNC

In the section, we develop a stochastic model to capture the impact of two bandwidth modes on the number of transactions in mempools, as a theoretical basis for performance analysis.

A. Definitions and Assumptions

We consider a blockchain network with $n$ mining pools and over ten thousand full nodes. During the consensus process, a transaction can be received by full nodes in three phases: 1) Transaction propagation; the new transaction is propagated to the network. 2) Synchronization; the synclow or synchigh is propagated to synchronize the transaction. 3) Block propagation; the block is propagated to confirm the transaction. Due to the online-offline dynamics of network churn, the mempool of a full node may miss a few transactions, which is defined as an incomplete mempool. On the other hand, since mining pools are engaged in the time-sensitive hashing competition, a rational operator will keep the pool server online all the time to maximize block rewards. Accordingly, almost all transactions can be received by a pool server in the first phase, thus the mempool of a pool server is defined as a complete mempool.

Let $M$ be a set consisting of all the transactions in a complete mempool, and $B$ be a set consisting of all the transactions in a block, satisfying $B \subseteq M$. During the block propagation, two random full nodes are selected as the sender and the receiver of the block, and the transactions in their mempools are denoted by set $S$ and set $R$, respectively. As shown in Fig. 3, the sender constructs a compact block based on the relationship between $B$ and $S$. The prefilled transactions are the transactions in $B$ but not in $S$, denoted by $B_{12}$. The short IDs are extracted from the intersectional transactions between $B$ and $S$, denoted by $B_{34}$. From the perspective of the receiver, $B$ will be divided into four subsets, satisfying $B_1 \cup B_2 = B_{12}$ and $B_3 \cup B_4 = B_{34}$. Among them, $B_1$ is the set of transactions that has not been validated by the receiver. $B_2$ and $B_4$ are the sets of transactions that already available for block reconstruction. $B_3$ is the set of missing transactions that will be recovered by the getblocktxn-blocktxn roundtrip.

Let $m$ be the number of transactions in the set $M$, and $b$ be the number of transactions in the set $B$. The number of
transactions in $B_1$, $B_2$, $B_3$, $B_4$ are denoted by $b_1$, $b_2$, $b_3$, $b_4$ respectively, satisfying $b_1 + b_2 + b_3 + b_4 = b$. Suppose that transaction propagations are independent Bernoulli trials, where a transaction is received by a full node with probability $p$, and a transaction is missed with probability $q = 1 - p$. Considering the network churn and transaction propagation failure, we have $p = p_{on}(1 - p_f)$, where $p_{on}$ is the online probability of a full node, and $p_f$ is the probability of a propagation failure that has been analyzed in our previous work [25]. For $b$ Bernoulli trials, $b_1$ is the expected number of transactions missed by both $S$ and $R$; $b_2$ is the expected number of transactions missed by $S$ but received by $R$; $b_3$ is the expected number of transactions in $S$ but not in $R$; $b_4$ is the expected number of transactions in both $S$ and $R$. Based on the expectation of binomial random variables, we can obtain

$$b_1 = bq^2, \ b_2 = bgp, \ b_3 = bpq, \ b_4 = bp^2,$$
$$b_{12} = b_1 + b_2 = bq, \ b_{34} = b_3 + b_4 = bp.$$  \hspace{1cm} (1)

### B. Low-Bandwidth Mode Analysis

Consider that mining pools are coordinated by $n$ pool servers, $k (k \geq 1)$ of which have enabled the low-bandwidth mode, called active pool servers. Suppose that the mempool of a pool server consists of $\Delta$ unsync txns and $m - \Delta$ sync txns in steady state, shown in Fig. 4. At block height $h$, the pool server constructs a candidate block by selecting $b$ txns from a set of $m$ txns in the mempool. On the average, the candidate block contains $\Delta b/m$ unsync txns and $(m - \Delta)b/m$ sync txns, so there remain $\Delta (m - b)/m$ unsync txns and $(m - \Delta)(m - b)/m$ sync txns in the mempool. Among the remaining txns, $(m - \Delta)(m - b)/m$ sync txns stay unchanged until the next block construction, while $\Delta (m - b)/m$ unsync txns can be synchronized by the other pool servers during the mining process. At height $h + 1$, a valid block has been found by one of the pool servers. Let $E$ be the event that the valid block belongs to a pool server in low-bandwidth mode, and $\bar{E}$ be the complementary event.

- If the event $E$ occurs, the remaining unsync txns has been randomly selected by $k - 1$ servers for synchronization. An unsync txn becomes a sync txn only when it has been selected by at least one server. Therefore, among $\Delta (m - b)/m$ txns, there are $\Delta (m - b)/m[1 - ((m - b)/m)^{k-1}]$ txns that have been synchronized by the servers and $\Delta ((m - b)/m)^k$ txns that have not been synchronized.
- If the event $\bar{E}$ occurs, the remaining unsync txns has been randomly selected by $k$ servers. In this case, there are $\Delta (m - b)/m[1 - ((m - b)/m)^k]$ txns that have been synchronized and $\Delta ((m - b)/m)^{k+1}$ txns that have not been synchronized.

When a pool server receives the valid block, the txns in the block will be removed from its mempool, thus the unsync txns consist of $b$ new transactions and $\Delta ((m - b)/m)^{x+1}$ ($x = k - 1$ or $k$) txns that are not synchronized by pool servers during the mining process. Based on the stability of unsync txns, we have

$$\Delta = b + P(E)\Delta \left(\frac{m - b}{m}\right)^k + P(\bar{E})\Delta \left(\frac{m - b}{m}\right)^{k+1}$$
$$= b + \left[\frac{k}{n}\left(\frac{m - b}{m}\right)^k + \frac{n - k}{n}\left(\frac{m - b}{m}\right)^{k+1}\right]\Delta \hspace{1cm} (2)$$

After the synchronization of the low-bandwidth mode, the number of txns in the set $S$ and set $R$ increases, thus $b_{12}, b_{34}, b_1, b_2, b_3, b_4$ change to $b_{12}', b_{34}', b_1', b_2', b_3', b_4'$. To determine these values, we consider the $\Delta b/m$ unsync txns and $(m - \Delta)b/m$ sync txns in a valid block respectively. We first consider $\Delta b/m$ unsync txns:

- If the event $E$ occurs, each unsync txn has been synchronized $I + 2$ times, namely $1$ time from the txn owner, $1$ time from the block owner, and $I$ times from $k - 1$ servers. Specifically, the block owner is a pool server in low-bandwidth mode, so its $sync_{low}$ must contain all the unsync txns in the valid block. In addition, there are $k - 1$ servers that randomly select $b$ txns from a set of $m$ txns for synchronization, thus each txn in the mempool has been selected $I$ times with probability

$$P\{I = i\} = \left(\frac{k - 1}{i}\right) \left(\frac{b}{m}\right)^i \left(\frac{m - b}{m}\right)^{k - 1 - i}, \hspace{1cm} (3)$$

where $i = 0, 1, \cdots, k - 1$.

- If the event $\bar{E}$ occurs, each unsync txn has been synchronized $J + 1$ times, namely $1$ time from the txn owner, and $J$ times from $k$ servers with probability

$$P\{J = j\} = \left(\frac{k}{j}\right) \left(\frac{b}{m}\right)^j \left(\frac{m - b}{m}\right)^{k - j}, \hspace{1cm} (4)$$

where $j = 0, 1, \cdots, k$.

In summary, for $\Delta b/m$ unsync txns in the valid block, the probability that a txn is not in the set $S$ is

$$p_u(q) = P(E) \sum_{i=0}^{k-1} P\{I = i\} q^{i+2} + P(\bar{E}) \sum_{j=0}^{k} P\{J = j\} q^{j+i+1}$$
$$= \frac{k}{n}q^2 f_1(q, k - 1) + \frac{n - k}{n} q f_1(q, k), \hspace{1cm} (5)$$

where $f_1(q, k)$ is the probability generating function of the binomial distribution with parameters $q$ and $k$. The probability that a block is a valid block is $p_b = 1 - p_u(q)$.
since a txn is not in \( S \) only when the full node has missed all \( i + 2 \) or \( j + 1 \) messages. Function \( f_1(x, y) \) is defined as

\[
f_1(x, y) = \left( \frac{b}{m} x + \frac{m - b}{m} \right)^y.
\]

For \((m - \Delta)b/m\) sync txns, each txn must be synchronized by pool servers at least one time before the current block height, thus \( I \geq 1 \) and \( J \geq 1 \) are necessary conditions. Using equations (3) and (4), we have

\[
P\{I = i \mid I \geq 1\} = \frac{\left( \frac{k-1}{i} \right) \left( \frac{b}{m} \right)^i \left( \frac{m-b}{m} \right)^{k-i}}{1 - \left( \frac{m-b}{m} \right)^k},
\]

\[
P\{J = j \mid J \geq 1\} = \frac{\left( \frac{j}{b} \right) \left( \frac{m-b}{m} \right)^{j-1}}{1 - \left( \frac{m-b}{m} \right)^k},
\]

where \( i = 1, \ldots, k - 1 \) and \( j = 1, \ldots, k \). Compared to unsync txns, sync txns are synchronized by pool servers before the current block height. If \( E \) occurred, a sync txn has been synchronized \( I + 1 \) times, namely 1 time from the txn owner, and \( I (I \geq 1) \) times from \( k - 1 \) servers. If \( E \) occurred, a sync txn has been synchronized \( J + 1 \) times, namely 1 time from the txn owner, and \( J (J \geq 1) \) times from \( k \) servers. In summary, for \((m - \Delta)b/m\) sync txns, the probability that a txn is not in the set \( S \) is

\[
p_s(q) = P(E) \sum_{i=1}^{k-1} P\{I = i \mid I \geq 1\} q^{i+1} + P(\bar{E}) \sum_{j=1}^{k} P\{J = j \mid J \geq 1\} q^{j+1}.
\]

Recalling that \( b_i \) is the expected number of transactions missed by both \( S \) and \( \bar{R} \). Hence, using \( q^2 \) to replace \( q \) in (10) yields

\[
b_i = \Delta \frac{b}{m} p_u(q) + (m - \Delta) \frac{b}{m} p_s(q^2).
\]

Then, based on the relationship of subsets, we have

\[
b_2 = b_{12} - b_1, \quad b_3 = b_2, \quad b_4 = b_{34} - b_3.
\]

C. High-Bandwidth Mode Analysis

Suppose that \( k \) servers have switched to the high-bandwidth mode. After the synchronization of the high-bandwidth mode, the number of txns in subsets are \( b_{12}^h, b_{34}^h, b_1^h, b_2^h, b_3^h \). In high-bandwidth mode, a pool server will use short IDs to synchronize all the txns in a candidate block, thus a sync txn may be repeatedly synchronized until it is included in a valid block. Specifically, a new txn will be synchronized in three periods: (i) A new txn is propagated on the network for initial synchronization. (ii) An unsync txn is waiting in the mempool until it is marked as “sync”. In this period, the txn is synchronized \( I \) times if \( E \) occurs, or \( J \) times if \( \bar{E} \) occurs, where \( I \geq 1, J \geq 1 \). (iii) A sync txn is waiting in the mempool until it is included in a valid block. Let \( L \) denote the number of valid blocks found by pool servers during this period, and \( L \) is a geometric random variable with probability mass function

\[
P\{L = l\} = \left( \frac{m-b}{m} \right)^{l-1} \left( \frac{b}{m} \right), \quad l = 1, 2, \ldots
\]

which follows that the first \( l - 1 \) blocks do not contain the sync txn and the \( l \)th block contain the sync txn. During the mining process of the \( l - 1 \) blocks, the txn is synchronized \( I \) or \( J \) times at each block height, where \( I \geq 0, J \geq 0 \). During the mining process of the \( l \)th block, the txn is synchronized \( I + 1 \) or \( J \) times, where \( I \geq 0, J \geq 0 \).

The sync txn in a valid block has gone through three periods, and the probability that a sync txn is not in the set \( S \) is given by equation (14). On the other hand, the unsync txn in a valid block only go through period (i) and period (iii) with \( l = 1 \). Therefore, the probability that an unsync txn is not in \( S \) is the same as the equation (5) in low-bandwidth mode. From equations (5) and (14), we obtain

\[
b_{12}^h = \Delta \frac{b}{m} p_u(q) + (m - \Delta) \frac{b}{m} p_s(q^2),
\]

\[
b_{34}^h = \Delta \frac{b}{m} [1 - p_u(q)] + (m - \Delta) \frac{b}{m} [1 - p_s(q^2)].
\]
Then, the number of txns in subsets are given by
\[
\begin{align*}
    b^h_1 &= \Delta \frac{b}{m} p_u(q^2) + (m - \Delta) \frac{b}{m} p_s(q^2), \\
    b^h_2 &= b^h_{12} - b^h_1, \\
    b^h_3 &= b^h_2 + b^h_4 = b^h_{34} - b^h_3.
\end{align*}
\]
(16)

D. Performance Metrics

End-to-end delay $t_e$ is defined as the time for a single-hop block delivery between two full nodes, consisting of roundtrips, transmission, and validation as follows:
\[
t_e = \begin{cases} 
    1.5 t_r + \text{cmpctblock}/\theta + t_h + b_1 t_v, & \text{if } b_3 = 0, \\
    2.5 t_r + (\text{cmpctblock} + \text{getblocktxn} + \text{blocktxn})/\theta \\
    + t_h + (b_1 + b_3) t_v, & \text{otherwise},
\end{cases}
\]
(17)
where $t_r$ is the round-trip time; $\theta$ is the TCP throughput; $t_h$ and $t_v$ are the validation time for a header and a transaction respectively; $\text{cmpctblock}$, $\text{getblocktxn}$, $\text{blocktxn}$ represent the corresponding message sizes, which are given by
\[
\begin{align*}
    \text{cmpctblock} &= \text{header} + \text{nonce} + b_{12} \cdot \text{prefilledtxn} + b_{34} \cdot \text{shortid}, \\
    \text{getblocktxn} &= \text{blockhash} + b_3 \cdot \text{index}, \\
    \text{blocktxn} &= \text{blockhash} + b_3 \cdot \text{txn},
\end{align*}
\]
(18)
referring to the data structures in compact block relay [9]. Then, the end-to-end delay of the low or high-bandwidth mode can be obtained by substituting $b^l_{12}$, $b^l_{34}$, $b^l_1$, $b^l_3$ or $b^h_{12}$, $b^h_{34}$, $b^h_1$, $b^h_3$ into equations (17) and (18).

Bandwidth usage $W$ is defined as the bandwidth consumption of a full node at each block height, for transaction propagation, synchronization, and block propagation. At each block height, there are $b$ new transactions, $b_T$ of which can be received by a full node, so the bandwidth usage on transactions is given by
\[
W_t = b_T (c \cdot \text{inv} + \text{getdata} + \text{txn}),
\]
(19)
where $c$ is the number of connections of a full node. The bandwidth usage on the synchronization of low-bandwidth mode and high-bandwidth mode are given by
\[
\begin{align*}
    W^l_s &= k (c \cdot \text{inv} + \text{getdata} + \text{synclow} + 2 \text{blockhash}) \\
    &\quad + (b_{12} - b^l_{12}) (\text{index} + \text{txn}), \\
    W^h_s &= k (c \cdot \text{inv} + \text{getdata} + \text{synchigh} + 2 \text{blockhash}) \\
    &\quad + (b_{12} - b^h_{12}) (\text{index} + \text{txn}),
\end{align*}
\]
(20)
where
\[
\begin{align*}
    \text{synclow} &= \text{header} + \Delta h / m \cdot \text{shortid}, \\
    \text{synchigh} &= \text{header} + \text{coinbase} + b \cdot \text{shortid}.
\end{align*}
\]
(21)
Note that $b_{12} - b^l_{12}$ and $b_{12} - b^h_{12}$ are the number of transactions requested by a full node during the synchronization, which do not affect the overall transaction load in the network. The extra bandwidth usage incurred by Presync is only $k (c \cdot \text{inv} + \text{getdata} + \text{synclow} or \text{synchigh} + 2 \text{blockhash})$. After the synchronization, the bandwidth usage on the block can be calculated by
\[
W_b = c \cdot \text{inv} + \text{getdata} + \text{cmpctblock} + \text{getblocktxn} + \text{blocktxn},
\]
(22)

Forking probability $p_k$ is the probability that at least two blocks are found by mining pools at the same block height, due to the block propagation delay. To determine this probability, we divide block propagation process into multiple rounds $r$ ($1 \leq r \leq n$), where a round ends when the number of pool servers that receive the new block increases by 1. The time interval from round $r$ to round $r+1$ is denoted by $t_r$, which has a positive relation with the end-to-end delay $t_e$ [25]. Let $\lambda$ be the block generation rate of all mining pools, and $T_k(r)$ be the time to find a fork in round $r$. $T_k(r)$ is an exponential random variable with expectation $E[T_k(r)] = 1/(n - r \cdot \lambda)$. Using the complementary event that the fork does not occur in all rounds, forking probability can be expressed as
\[
p_k = 1 - \prod_{r=1}^{n-1} P\{T_k(r) > t_r\} = 1 - \exp \left( - \sum_{r=1}^{n-1} \frac{n - r}{n} \lambda t_r \right).
\]
(23)
Meanwhile, the expected number of forks at each block height, called fork rate $\gamma$, is given by
\[
\gamma = \sum_{r=1}^{n-1} \frac{n - r}{n} \lambda t_r.
\]
(24)
$p_k$ and $\gamma$ take the minimum values when a new block can be received by all pool servers in one hop, which yield
\[
p_{k_{\text{min}}} = 1 - \exp \left( - \frac{n - 1}{n} \lambda t_e \right), \quad \gamma_{\text{min}} = \frac{n - 1}{n} \lambda t_e.
\]
(25)

Transaction throughput $\vartheta$ is the number of transactions that can be confirmed by blockchain per second, measured by transaction per second (TPS). Transaction throughput depends on block generation rate $\lambda$, the number of transactions in a block $b$, and fork rate $\gamma$. Using equation (24), we obtain
\[
\vartheta = \frac{\lambda b}{1 + \gamma} = \frac{\gamma b}{(1 + \gamma) \sum_{r=1}^{n-1} \frac{n - r}{n} t_r}.
\]
(26)
Now we consider the relationship between transaction throughput and fault tolerance. Fault tolerance is a security threshold that decrives how much hashing power is required to conduct a double-sending attack. According to [26], fault tolerance can be expressed as $f = 1/(2 + \gamma)$ and satisfies $f = 1/2$ when fork rate $\gamma = 0$, known as an ideal 50% security threshold. Substituting $\gamma = 1 - f$ into equation (26), we have
\[
\vartheta = \frac{b(1 - 2f)}{(1 - f) \sum_{r=1}^{n-1} \frac{n - r}{n} t_r},
\]
(27)
and its maximum value is given by
\[
\vartheta_{\text{max}} = \frac{bn(1 - 2f)}{t_e(n - 1)(1 - f)}.
\]
(28)

V. Evaluation Results and Discussions

In this section, we evaluate the performance of Presync by setting up a simulation experiment in Matlab, and then compare the simulation results with the analytical results.
A. Parameter Settings and Initialization

We consider a blockchain network with 10 mining pools that are coordinated by pool servers. The number of pool servers in low or high-bandwidth mode ranges from 0 to 10, especially \( k = 0 \) indicates the baseline performance of Compact protocol. Besides the parameter settings in Table II, we set the message sizes based on Bitcoin protocol documentation [23] and Bitcoin visuals [28]: header = 80 bytes, nonce = 8 bytes, shortsid = 6 bytes, blockhash = 32 bytes, index = 1 byte, inv = getdata = 37 bytes, coinbase = 300 bytes, txn = 560 bytes, prefilledtxn = index + txn = 561 bytes.

In the experiment, we simulate the mempools of the pool server, the block sender, and the receiver by creating arrays \( M, S \), and \( R \) in Matlab. Each element in the array represents the state of a transaction, where 0 represents missing state, 1 represents unsync state, and 2 represents sync state. In the synchronization phase, a candidate block \( B \) includes the first \( b \) elements in a random permutation “randperm(m)”. The synclow contains all the elements in \( B \) with value 1, while the synchigh contains all the elements with value 1 or 2. If full nodes receive an inv for synchronization, \( S \) and \( R \) are updated based on synclow or synchigh. The synchronization process is repeated \( k \) times, and all the elements with value 1 change to 2 after the synchronization, which are recorded in \( M \) to avoid redundant synchronization.

In the block propagation phase, a random permutation is conducted to generate a new block \( B \). Each block propagation generates a new group of \( b_{12}, b_{34}, b_1, b_3 \), which determine the sizes of cmpctblock, getblocktxn, blocktxn. \( b_{12} \) is the number of elements that satisfy \( B(i) \geq 1 \) and \( S(i) = 0 \); \( b_1 \) is the number of elements that satisfy \( B(i) \geq 1, S(i) = 0, \) and \( R(i) = 0; b_{34} \) is the number of elements that satisfy \( B(i) \geq 1 \) and \( S(i) \geq 1; b_3 \) is the number of elements that satisfy \( B(i) \geq 1, S(i) \geq 1, \) and \( R(i) = 0. \) After the block propagation, all the nonzero elements in \( B \) should be reset in \( M, S, \) and \( R \), i.e., the elements in \( M \) are set to 1, and the elements in \( S \) and \( R \) are set to 0 or 1 based on Binomial random number “binornd(1, \( p \))”. Then, the experiment returns to the synchronization phase.

B. End-to-End Delay and Bandwidth Usage

Fig. 5(a) and (b) show the end-to-end delay from block height 0 to 50, where “Compact” represents the baseline performance of Compact protocol, and “Presync low” and “Presync high” represent the performance of low and high-bandwidth modes respectively. The end-to-end delay of Compact fluctuates around 3363ms, since the sizes of cmpctblock, getblocktxn, and blocktxn are different at each block height. The message sizes depend on the number of transactions in the sender and receiver’s mempools. During transaction propagations, each transaction enters a mempool with probability \( p \), similar to Bernoulli trials. With thousands of trials, the number of transactions in the sender’s mempool is close to that in the receiver’s mempool, thus the delay remains in a variation range of \( -4\% \sim 4\% \) of the average value.

In low-bandwidth mode with \( k = 1 \), the average end-to-end delay is 2441ms, with a variation range about \(-70\% \sim 40\% \).
C. Analytical Results and Comparisons

In the numerical calculation, we first determine \( b_1, b_2, b_3, b_4 \) before and after the synchronization, using equations (1), (11), (12), (16). Then, the analytical results can be obtained by substituting \( b_1, b_2, b_3, b_4 \) into the expressions of performance.
metrics. In Fig. 7, we compare the performance between the low and high-bandwidth modes, when the number of active servers $k$ increases from 0 to 10. Meanwhile, the online probability of a full node $p_{on}$ is set to $2/3$ or 1 for comparing the cases with and without the network churn. The simulation results take the average values of 50 block heights.

The end-to-end delay is shown in Fig. 7(a). When $k = 0$ and $p_{on} = 2/3$, the end-to-end delay is 3346ms, where the total round-trip time is 500ms; the transmission time of a compact block is 1738ms; the transmission time of missing transactions is 1028ms, and the validation time is 80ms. Among them, the round-trip time can be reduced to $1.5t_r$ if there is no missing transactions in the receiver’s view, or $0.5t_r$ if the header match succeeds. When $k = 0$, $b_1 + b_2$ accounts for 40% of the transactions in a block, shown in Fig. 8(a), thus the transmission time of a compact block is the dominant factor in the end-to-end delay. After $k$ changes to 1.5, 10, $b_1$, $b_2$, $b_3$ decrease gradually, thus the end-to-end delay is reduced by 28%, 60%, 74% in low-bandwidth mode, and 36%, 78%, 90% in high-bandwidth mode. The end-to-end delay will reach a low bound when $b_1$, $b_2$, $b_3$ approach 0. The lower bound in low-bandwidth mode is $1.5t_r$, and the lower bound in high-bandwidth mode is $0.5t_r$ due to the match of block header.

When $p_{on} = 1$, the end-to-end delay has an overall decrease and approaches the lower bound faster than that when $p_{on} = 2/3$. This is because most of the short IDs will be redundant when there are less transactions missed by a mempool, see $p_{on} = 1$ in Fig. 7(b). To reduce the redundancy of short IDs, $k$ needs to be constrained in the network with a high online probability of full node. Compared to $p_{on} = 1$, a full node is more likely to miss the synclow and synchigh when $p_{on} = 2/3$, leading to the fluctuation of the average values in simulation experiments.

In the synchronization phase, the bandwidth usage of a full node is shown in Fig. 7(c). In this phase, most of the bandwidth usage comes from the transaction load. For example, when $p_{on} = 2/3$ and $k = 5$, the bandwidth usage in low-bandwidth mode is 355KB, consisting of 330KB transaction load and 25KB extra bandwidth usage; the bandwidth usage in high-bandwidth mode is 453KB, consisting of 390KB transaction load and 63KB extra bandwidth usage. When $p_{on} = 1$ and $k = 5$, the extra bandwidth usages remain unchanged, while the transaction loads drop to 102KB in low-bandwidth mode and 105KB in high-bandwidth mode. This means that the number of transactions recovered by low-bandwidth mode is similar to that recovered by high-bandwidth mode, thus most of the short IDs in synchighs are redundant when $p_{on} = 1$.

Fig. 7(d) illustrates that the bandwidth usage on block decreases with $k$, and the difference between the low and high-bandwidth modes first increases then decreases with $k$. The reason is that the bandwidth usage in the block propagation phase mainly depends on $b_1 + b_2 + b_3$, or equivalently $1 - b_4$. As shown in Fig. 8, the differences between $b_4$ in high-bandwidth mode and $b_4$ in low-bandwidth mode are 0, 5%, 10%, 5% when $k = 0, 1, 5, 10$, which follow the behavior of first increasing and then decreasing with $k$. When $b_4$ approaches 100%, a full node can retrieve almost all transactions from its mempool using short IDs, thus the bandwidth usage on block will be minimized.

Fig. 7(e) shows the sum of the bandwidth usage in synchronization and block propagation phases. For $p_{on} = 2/3$, the bandwidth usage of high-bandwidth mode is lower than that of low-bandwidth mode when $k \in [1, 6]$, then the behavior is opposite when $k \in [7, 10]$. The reason is that the short IDs have a low redundancy when $k \in [1, 6]$, which effectively reduce the transactions missed by mempools. When $k \in [7, 10]$, the redundancy of the short IDs in high-bandwidth mode increases faster than that in low-bandwidth mode, resulting in the waste of bandwidth resource. For $p_{on} = 1$, there is a small number of transactions missed by mempools, leading to a high redundancy of short IDs. From the perspective of bandwidth usage, low-bandwidth mode is a better choice in the network with a high online probability of full node.

Fig. 7(f) shows that the steady value of synchronized transactions in the mempool increases with $k$, and its upper bound is affected by $b$ and $m$. For example, when $b = 3000$ and $m = 6000$, the upper bound of synchronized transactions is 3000, namely 1/2 of the transactions in the mempool; when $b = 1200$ and $m = 6000$, the upper bound is 4800, namely 4/5 of the transactions in the mempool, since a smaller $b$ leads to a longer waiting time in the mempool.

**D. Forking Probability and Transaction Throughput**

Fig. 9(a) shows that forking probability ranges from 0 to 1, when block generation rate $\lambda \in [10^{-3}, 10^3]$. Compared to the baseline performance, the low-bandwidth mode can reduce forking probability by 17% to 52% and the high-bandwidth mode can reduce forking probability by 22% to 71%, when $k$ increases from 1 to 10. The high-bandwidth mode achieves a better performance, but it has higher bandwidth usage, memory overhead, and complexity. Meanwhile, the high-bandwidth mode is not suitable for the network with a high online probability of full node, due to the high redundancy of short IDs and low performance gain.

Fig. 9(b) shows a negative correlation between transaction throughput and fault tolerance. In blockchain, transaction throughput can be improved by adjusting block generation rate or block size, but it incurs more forks and compromises fault tolerance. Presync reduces forking probability by shortening the time for roundtrips, transmission, and validation during the block propagation, which achieves throughput improvement without compromising fault tolerance. For a given fault tolerance, low-bandwidth mode can achieve 4× throughput gain.
at most, and high-bandwidth mode can achieve 10× gain at most, in a network with \( k = 10 \) and \( p_{on} = 2/3 \).

VI. RELATED WORK

A. Block Relay Protocol

Xtreme Thinblocks [10] (XThin) is a block relay protocol deployed in Bitcoin Unlimited clients. In XThin, the receiver creates a Bloom filter encoding all the transaction hashes in its mempool, which is sent with the getdata. Then, the sender transmits the 8-byte hashes of all transactions in the block, and the transactions that do not match the filter. In this way, the block size and extra roundtrips can be reduced significantly. Xthinner [11] is a variant of Xthin that uses a state machine to encode the 3-byte transaction hashes in a block. Compared to Xthin, Xthinner has a smaller block size, but with higher coding overhead and more hash collisions. Graphene [12] conducts a further block compression through the combination of a Bloom filter and an Invertible Bloom Lookup Table (IBLT). In Graphene, the receiver’s getdata includes a count of transactions in its mempool. The sender responds with a Bloom filter and an IBLT, which can be used to construct the block. The construction is efficient when the receiver’s mempool contains all transactions in the block; otherwise, the extra roundtrip and IBLT are needed to recover the missing transactions. Dino [31] is a protocol that transmits block construction rule instead of short hashes or a Bloom filter. The sender maintains a list of transaction hashes that have been relayed before, which is used to predict the existing transactions of the receiver. Based on the prediction, the sender can specify the interval, deletion, and reordering of transactions for block construction.

In summary, these protocols follow a principle: the transactions already exist in the receiver’s mempool can be compressed into short IDs, a Bloom filter, or a construction rule, while the missing transactions are directly sent to the receiver. Hence, the performance of these protocols are affected by the number of transactions missed by mempools, especially in the P2P network with frequent arrival and departure of nodes. Presync addresses this challenge by synchronizing the transactions in candidate blocks, so that the missing transactions can be detected and transmitted in the block interval.

B. Topology Optimization

The topology optimization of the P2P network is another way to speed up message propagation. Kadcast-NG [32] is a structured P2P overlay network for blockchain, which utilizes a binary tree-based topology to realize an efficient message broadcast with tunable overhead. NKN [33] is a structured P2P network for web 3.0, and its topology is constantly updated based on Chord distributed hash table. Swift [34] optimized the structured P2P network and broadcast mechanism based on unsupervised learning and greedy algorithm. Compared to Kadcast-NG and NKN, Swift can achieve faster convergence and lower broadcast latency with the same bandwidth overhead. Fring [35] is a P2P overlay network with multiple zones, where the full nodes are clustered based on the geographical distance to reduce the round-trip time. [36] proposed a recommendation-based neighbor selection algorithm that can dynamically update the outbound connections of a miner based on the propagation ability.

Structured P2P networks enable efficient broadcast through topology optimization, and are expected to work in parallel with block relay protocols for minimizing fork rate. On the other hand, according to [37], the challenges in structured P2P networks are anonymity enhancement, topology hiding, and denial of service resistance, since many attacks rely on the knowledge of network topology.

VII. CONCLUSIONS

In this paper, we introduced Presync, a transaction synchronization protocol to reduce the difference between the block and the mempool’s transactions with controllable bandwidth overhead. High-bandwidth mode allowed full nodes to cache the synchronization messages and match the header of a valid block, enabling a fast block reconstruction. Low-bandwidth mode provided a lightweight synchronization using the short IDs of unsynchronized transactions, so that the missing transactions are detected with a low redundancy. We presented a stochastic model for performance analysis, and validated this model through experimental evaluations. Compared to low-bandwidth mode, high-bandwidth mode can reconstruct the valid block with 0.5RRT, but it uses extra bandwidth resources to synchronize the coinbase transaction and the transactions that have been synchronized at previous block heights. Meanwhile, the redundancy of short IDs in high-bandwidth mode will be much higher than that in low-bandwidth mode, in the network with a high online probability of full node.

REFERENCES


