“Stake” project
PFE - ASR

POLAARK
&
paulglx

I. Introduction **3**
II. Analysis of current System **4**
A. CometBFT: Framework overview 4
B. Current algorithm 5
1. Complexity 6
Best Case 6
Worst Case 6
2. Paper Application of current algorithm 7
III. Our approach : Stake-based solution **9**
A. Main idea 9
B. Benefits of this approach 9
C. Pseudo code 10
IV. Implementation **11**
A. The message and the transaction object 11
B. Receiving Transactions 14
C. The Broadcast 16
V. Performance Analysis for the stake based approach **17**
A. Test environment 17
B. Data collection 17
C. Test parameters 18
D. Results 18
VI. Another alternative? Stake-Threshold Based Transaction Propagation **20**
A. Idea 20
B. Implementation overview 21
C. Results 23
VII. Conclusion **26**

I. Introduction

Blockchain technology, initially popularized by Bitcoin, has evolved into a transformative force across industries. At its core, blockchain is a decentralized and immutable ledger that records transactions securely and transparently. By distributing transaction data across a network of computers (nodes), blockchain eliminates the need for intermediaries, enhancing trust and reducing costs. Beyond cryptocurrencies, blockchain finds diverse applications in sectors such as finance, supply chain management, healthcare, and voting systems. Its ability to facilitate peer-to-peer transactions, verify ownership, and ensure data integrity has led to innovations in smart contracts, decentralized finance (DeFi), and secure digital identities. Blockchain's decentralized nature and cryptographic security make it a powerful tool for enhancing transparency, efficiency, and accountability in various domains, promising continued advancements in how businesses and societies manage and secure digital assets and information.

In blockchain systems, efficiently propagating transactions across nodes in a distributed network is critical to ensuring consistency and achieving fast consensus. CometBFT, a consensus framework built on Tendermint, relies on a gossip-based mechanism to broadcast transactions within its "*mempool*" to all connected nodes. This mechanism ensures that all nodes are aware of pending transactions by continuously sending transaction data to peers in a decentralized manner.

While this approach is robust and ensures reliability, it can lead to significant bandwidth usage. Transactions are broadcast repeatedly to multiple peers, often resulting in redundant data transmission. This can become particularly problematic in large-scale networks or under high transaction loads, where bandwidth usage directly impacts network performance and scalability.

To address this challenge, our solution introduces a novel approach: instead of universally gossiping transactions to every node, transactions are selectively broadcasted until a predetermined threshold is reached. This threshold is dynamically determined based on the stake held by validators rather than a fixed number of peers. Each participating peer signs the transaction once the threshold is met, ensuring consensus on transaction validity across the network.

By implementing this selective broadcasting strategy, we aim to optimize bandwidth consumption effectively. This approach minimizes redundant data transmissions by limiting the dissemination of transactions to a subset of nodes until consensus criteria, tied to validator stake participation, are fulfilled. This innovation not only enhances the efficiency of transaction propagation in CometBFT but also contributes to reducing network congestion and operational costs associated with high-bandwidth usage.

Furthermore, this project underscores the broader implications for blockchain scalability and resource efficiency. By refining gossip-based mechanisms with stake-based thresholds, we enhance the reliability and performance of blockchain networks, supporting their sustainable growth and adoption across various applications.

II. Analysis of current System

- **Bandwidth consumption**
- ***Performance metrics, bottlenecks (si possible) comparer données utiles et données transférées au total***

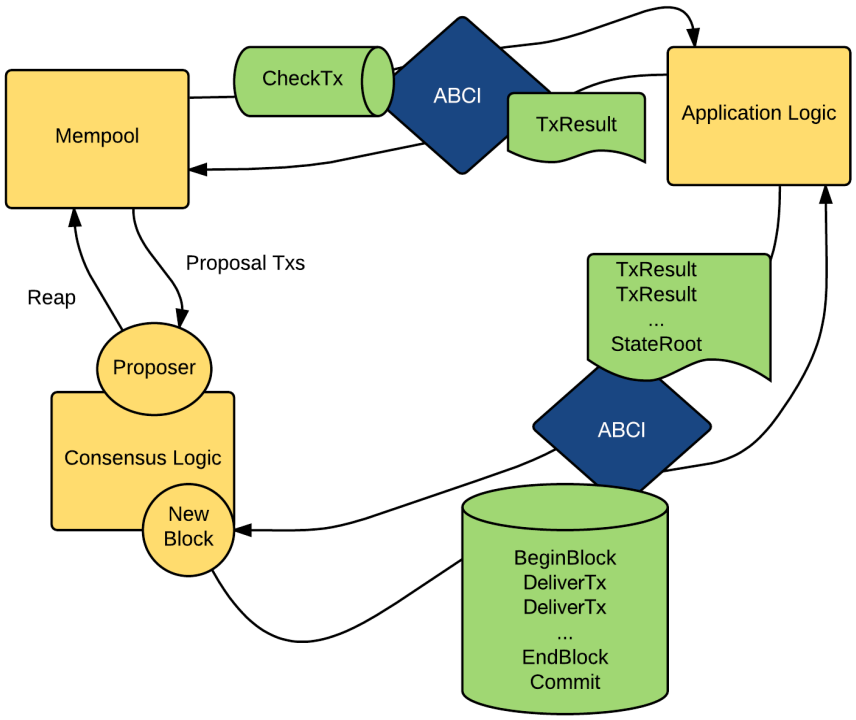
A. CometBFT: Framework overview

CometBFT is a Byzantine Fault Tolerant (BFT) middleware designed to securely replicate state machines across multiple nodes, ensuring that all non-faulty machines process an identical transaction log and compute the same state. This robustness is maintained even if up to one-third of the nodes exhibit arbitrary or malicious behavior. The framework consists of two main components: a blockchain consensus engine and the Application Blockchain Interface (ABCI). The consensus engine, rooted in the Tendermint algorithm, ensures that transactions are consistently ordered across all nodes. The ABCI enables developers to implement application logic in any programming language, promoting the replication of deterministic state machines. This flexibility makes CometBFT suitable for a diverse array of distributed applications, including cryptocurrencies and e-voting platforms.

The consensus mechanism in CometBFT operates through a series of rounds, each comprising three primary steps: Propose, Prevote, and Precommit. In the Propose step, a designated proposer suggests a new block for addition to the blockchain. During the Prevote step, validators broadcast their votes on the proposed block. If a block receives more than two-thirds of the votes in favor during the Prevote step, the process advances to the Precommit step, where validators confirm their commitment to the block. Once the block secures the necessary pre-commit votes, it is finalized and added to the blockchain. This structured approach ensures that all non-faulty nodes reach consensus on the same sequence of blocks, maintaining system integrity even in the presence of malicious actors.

Complementing this consensus mechanism is the mempool, a data structure within each node that temporarily stores transactions awaiting inclusion in a block. When a new transaction is received, the mempool first verifies its validity using the ABCI CheckTx method. Valid transactions are then stored in an ordered in-memory pool, ready to be proposed in subsequent consensus rounds. CometBFT supports different types of mempools, such as the flood mempool, which utilizes a concurrent linked list structure to efficiently

manage transactions. This design ensures that only valid transactions are propagated across the network, enhancing the overall efficiency and reliability of the system.



System overview from CometBFT documentation[1]

B. Current algorithm

To address the inefficiencies of the current gossip mechanism in CometBFT's mempool, we started by thoroughly analyzing the algorithm on paper. We recreated the existing transaction propagation process across various network scenarios, simulating different node topologies, transaction loads, and peer behaviors. This exercise helped us identify key inefficiencies, such as excessive redundancy in transaction broadcasts and uneven utilization of network resources.

In this scenario, we analyze a 6-node cluster where node 1 acts as the initiator of a transaction. The goal is to understand how the current gossip algorithm handles transaction propagation under both best-case and worst-case conditions.

[1] https://docs.cometbft.com/v0.37/imgs/abci.png

Algorithm: Gossip Transaction in Mempool

Inputs:

- tx: the transaction to be processed
- sender: the peer who sent the transaction
- mempool: the local mempool containing cached transactions
- neighbors: the list of connected peers (neighbors)

Steps:

1. Receive
   1.1 Validate the transaction `tx`
   - If the transaction is invalid, terminate the process.
   1.2 Check if `tx` exists in the mempool cache:
   - If not in cache:
   - Add `tx` to the mempool cache.
   - Else:
   - Terminate the process (transaction already processed).
   1.3 Add the sender's ID to the transaction's metadata (to track who sent the transaction).

2. Broadcast
   2.1 For each peer `p` in `neighbors`:
   - If `p` is the sender or the local node itself, skip.
   - Otherwise, send `tx` to peer `p`.

1. Complexity

Best Case

Time Complexity: O(n), where n is the number of neighbors.

- Each node sends the transaction to all its neighbors once.
- There are no redundant broadcasts, as all nodes receive the transaction only once from their closest neighbor.

Worst Case

Assumptions:

a. Each peer may receive the same transaction from all its neighbors multiple times.

b. Each peer sends the transaction to n−1 neighbors (excluding the sender and itself).
c. Each peer performs n−2 redundant checks against its cache.

Broadcast Complexity:

d. Each peer sends the transaction to n−1 neighbors, and every neighbor receives it.
e. Total sends across the network: $O(n\times(n-1))=O(n^2)$

Cache Check Complexity:

f. Each peer checks the transaction against the cache n−1 times for each neighbor.
g. Total cache checks: $O(n^2)$.

Overall Time Complexity: $O(n^2)$.

2. Paper Application of current algorithm

In the best-case scenario, transactions propagate optimally due the topology of the network (network is a line) and so we have:

1. Node 1 broadcasts the transaction, and node 2 is the first to receive it.
2. Node 2 begins broadcasting but not to 1 as it is the sender but to 3 as it is the only remaining neighbor.
3. This pattern repeats in subsequent steps, where nodes continue broadcasting as soon as they receive the transaction, resulting in a synchronized spread across the network.

This best-case scenario can theoretically occur when:

- Broadcasts are initiated immediately upon receiving the transaction.
- The network latency between peers is minimal and consistent.
- Transactions are propagated in such a way that each node broadcasts before redundant copies arrive from other peers.

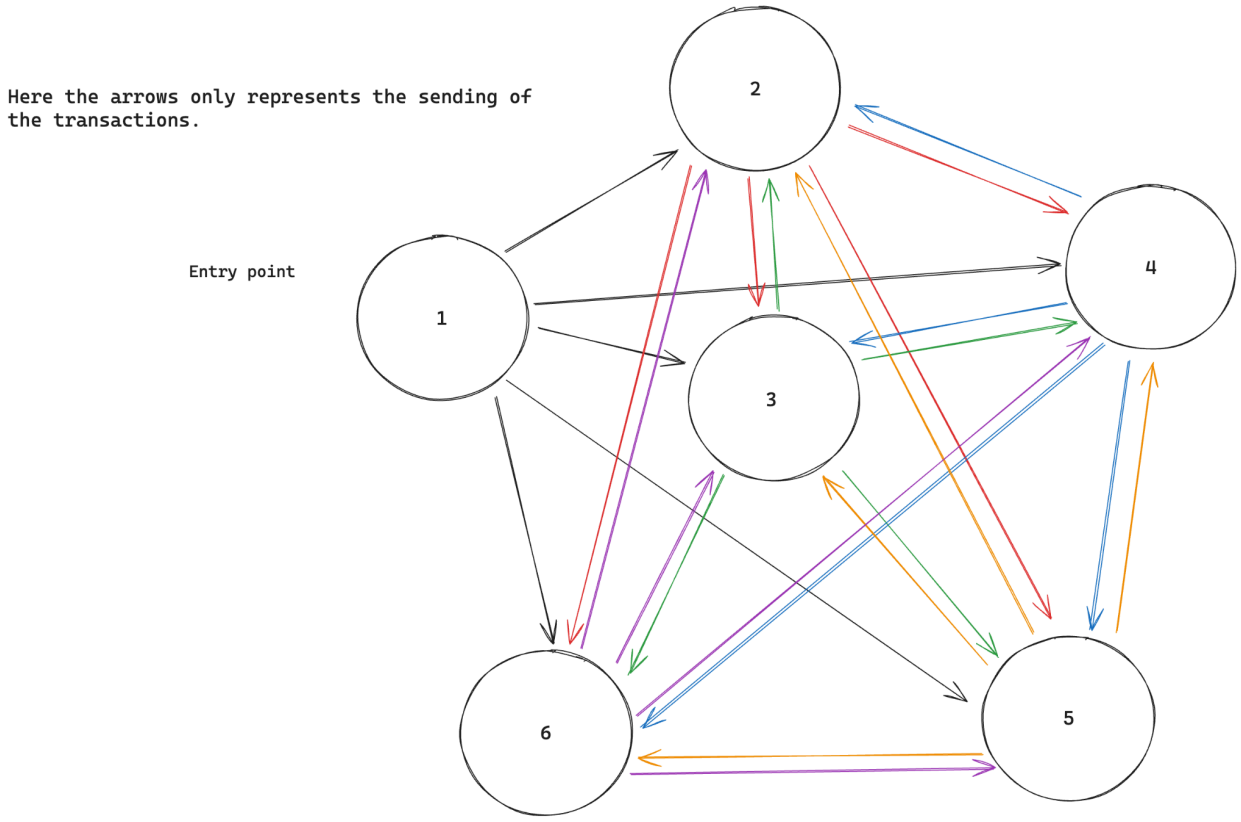
In this scenario, all nodes process the transaction efficiently with minimal redundancy in broadcasts. Unfortunately this network topology is not really present in real life.

In the worst-case scenario, the inefficiencies of the gossip algorithm become evident. Consider the following:

1. Each node receives the transaction multiple times, once from every neighbor except the sender.
2. Each node must check whether the transaction is already in its cache for every received copy, resulting in **n-1 cache checks** for a node with n neighbors.
3. Each node broadcasts the transaction to all neighbors except the sender and itself, resulting in **n-2 redundant broadcasts** per node.

This results in:

- Significant overhead in cache lookups, with redundant processing at every step.
- Excessive broadcasts, especially as the network scales with more nodes or higher connectivity.



III. Our approach : Stake-based solution

A. Main idea

We propose an approach that involves validators signing transactions upon receipt, appending their public key signatures, and verifying existing signatures before propagating transactions. Once a transaction accumulates signatures representing a cumulative stake above a predefined threshold, it is proposed to the consensus process, reducing unnecessary network traffic.

Upon receiving a transaction, a validator performs initial validation checks, such as verifying the transaction's integrity and authenticity. If valid, the validator signs the transaction and appends its public key signature. The validator also verifies existing signatures on the transaction, ensuring each corresponds to a recognized validator and accurately reflects their stake. This process creates a cumulative signature map, representing the total stake endorsing the transaction.

As the transaction propagates through the network, validators continue to sign and verify it, updating the cumulative stake. Once the accumulated stake exceeds a predefined threshold, indicating sufficient endorsement, validators cease further propagation of that transaction. The transaction is then proposed to the consensus mechanism, where validators who have not yet received it can fetch it as needed.

B. Benefits of this approach

Validator signatures on each transaction serve as cryptographic proofs of transaction authenticity. By signing transactions, validators confirm their validity, ensuring that only legitimate transactions are processed. This mechanism helps prevent fraudulent activities, such as double-spending, thereby maintaining the integrity of the blockchain.

Furthermore, requiring validators to sign transactions fosters accountability. Each signature is traceable to a specific validator, creating a transparent record of participation. This traceability discourages malicious behavior, as validators can be held responsible for their actions, thereby enhancing trust within the network. With this approach we can also identify each validator that signs the transaction, using the validator map we can easily access its stake and so its power on transaction validation.

Moreover by halting the propagation of transactions once they have garnered enough validator signatures, the network reduces redundant data transmission. This approach conserves bandwidth (which was our initial starting point) and processing power, optimizing the use of network resources and improving overall performance.

C. Pseudo code

1. Receive Tx:
   a. Validate Tx.
   b. If Tx not in cache:
      - Add Tx to mempool cache.
      - Record sender information.
   c. If Tx in cache:
      - Terminate (already processed).

2. Track Validations:
   a. Update cumulative voting power for Tx.
   b. If cumulative voting power < threshold:
      - Gossip full Tx to peers.
   c. If cumulative voting power ≥ threshold:
      - Stop gossiping full Tx.
   d. Peers that are late (that doesn't have the validated transaction) fetch it from the consensus

To ensure that a transaction is validated by trusted peers, we aggregate the signatures from each peer that has validated the transaction. Before broadcasting the transaction to other peers, we verify the aggregated signature to confirm that it meets the required threshold of validations. This approach ensures the transaction's authenticity and reduces unnecessary validations by downstream peers. We are adding each signature in a map where we have the public key as an id and the signature as a value.

This is what the message (transaction) sent and received to peer looks like:

```
txMessage := &protomem.Txs{
    Txs: []*protomem.Transaction{
        {
            TransactionBytes: entry.Tx(),
            Signatures:       signaturesMap,
        },
    },
}
```

It has been updated using the protobuf code. One thing valuable to note is that this implementation makes the code not retrocompatible. Indeed we change we transit betweens peers so all peers would have to have the same version of the code. The previous message types won't be working on this implementation as well as the current message and the current message structure won't be red in the previous implementation.

We send the relevant information of the transaction as an array of bytes and we send the signatures in the map.

IV. Implementation

The implementation can be found in this repo : https://github.com/POLAARK/cometbft

To implement this new logic we had to change various parts of the code. In this part we are going to go through the main changes and the logic related.

A. The message and the transaction object

The Entry interface is used in the code to handle transactions, both for receiving and sending them. It serves as the interface for mempoolTx, which represents the deserialized transaction object when accepted into the mempool. We have extended this interface to include all necessary logic related to signatures.

```
type Entry interface {
   // Tx returns the transaction stored in the entry.
   Tx() types.Tx

     // Height returns the height of the latest block at the moment the entry was
created.
   Height() int64

   // GasWanted returns the amount of gas required by the transaction.
   GasWanted() int64

   // IsSender returns whether we received the transaction from the given peer ID.
   IsSender(peerID nodekey.ID) bool

   ValidateSignatures(validators *types.ValidatorSet) (int64, error)

   // GetSignatures retrieves the current map of public keys to their signatures.
   GetSignatures() map[string][]byte

   // SetSignatures replaces the current map of signatures with the given map.
   SetSignatures(signatures map[string][]byte)

   // AddSignature adds a new signature to the transaction for the given public key.
   AddSignature(pubKey crypto.PubKey, signature []byte)


   SetThresholdReached(reached bool)

   GetThresholdReached() bool
}
```

The mempoolTx struct implements the Entry interface. Below is its definition:

```
/// mempoolTx is an entry in the mempool.
type mempoolTx struct {
   height    int64    // height that this tx had been validated in
   gasWanted int64    // amount of gas this tx states it will require
   tx        types.Tx // validated by the application
   lane      LaneID
```

```
    seq       int64
    timestamp time.Time // time when entry was created

    // signatures of peers who've sent us this tx (as a map for quick lookups).
    // Map keys are string representations of PubKey.
    signatures map[string][]byte
    signatureMutex sync.Mutex // Protects access to the signatures map

    // ids of peers who've sent us this tx (as a map for quick lookups).
    // senders: PeerID -> struct{}
    senders sync.Map

    // Is the threshold reached on this transaction
    isTresholdReached bool
}
```

Since multiple goroutines access the signatures map concurrently, we use a mutex (signatureMutex) to prevent race conditions. We use a thresholdReached to mark the transaction not to be broadcast. Below, we present two key methods that illustrate our approach, highlighting some of the challenges we encountered and overcame.

AddSignature: The following method securely adds a signature to the transaction and updates the signature count.

```
// AddSignature safely adds a signature to the map and increments the signature count.
func (memTx *mempoolTx) AddSignature(pubKey crypto.PubKey, signature []byte) {

    memTx.signatureMutex.Lock()
    defer memTx.signatureMutex.Unlock()

    if memTx.signatures == nil {
        memTx.signatures = make(map[string][]byte)
    }

    memTx.signatures[pubKey.Address().String()] = signature
    atomic.AddInt32(&memTx.signatureCount, 1)
}
```

Two important aspects of this implementation are:

- Mutex Usage: The mutex ensures that modifications to signatures are synchronized, preventing race conditions, which were a common issue before implementing this safeguard.
- Public Key Conversion: The public key is received as a crypto.PubKey object. Since Go maps require string keys, we first extract the raw byte representation of the key (pubKey.Bytes()) and then convert it to a string. This guarantees that we retain all key data while ensuring correct map key functionality.

ValidateSignatures: The following method verifies that each signature correctly corresponds to its respective public key. If any signature is invalid, an error is returned.

```
// ValidateSignatures checks if each signature matches its corresponding public key.
// If any signature is invalid, it returns an error.
func (memTx *mempoolTx) ValidateSignatures(validators *types.ValidatorSet) (int64,
error) {
    var accumulatedVotingPower int64
    var invalidSignatures []string
```

```
    memTx.signatureMutex.Lock()
    defer memTx.signatureMutex.Unlock()

    for pubKeyStr, signature := range memTx.signatures {
        // Convert the stored string back to a PubKey.
        pubKeyBytes, err := hex.DecodeString(strings.ToLower(pubKeyStr))
        if err != nil {
            //
        }
        pubKeyAdrr := bytes.HexBytes(pubKeyBytes)
        if err != nil {
            invalidSignatures = append(invalidSignatures, pubKeyStr)
            continue
        }

        // Look up the validator by the public key's address.
        _, validator := validators.GetByAddress(pubKeyAdrr)
        pubKey := validator.PubKey

        if validator == nil {
            // Using info-level log; ensure the output is consistent.
            print("MEMPOOLTX INFO: Invalid signature from unknown validator")
            invalidSignatures = append(invalidSignatures, pubKeyStr)
            continue
        }

        // Verify the signature against the transaction.
        if !pubKey.VerifySignature(memTx.Tx().Hash(), signature) {
            invalidSignatures = append(invalidSignatures, pubKeyStr)
            continue
        }

        accumulatedVotingPower += validator.VotingPower
    }

    if len(invalidSignatures) > 0 {
         return accumulatedVotingPower, fmt.Errorf("invalid signatures found for public
keys: %v", invalidSignatures)
    }

    return accumulatedVotingPower, nil
}
```

The *ValidateSignatures* function aims to verify the validity of digital signatures associated with a transaction stored in the *mempool* and to accumulate the voting power of validators whose signatures are valid.

One interesting aspect of this function is the use of *mutexes* to control access to the transaction's signatures, preventing concurrent modifications during the validation process. Given that multiple goroutines may access this data simultaneously, we have encountered several issues with race conditions, making this synchronization mechanism crucial.

For each *signature* in the signature map, the corresponding key—stored as a hexadecimal string representing the public key—is first converted into a byte array. This conversion is

essential to reconstruct the public key in a usable format, allowing the function to search for the corresponding validator within the provided validator set.

Once the public key is reconstructed (after converting it to *HexBytes*), the function attempts to find the validator associated with this address in the validator set. If no validator is found for the given key, the signature is considered invalid, and its identifier is added to the list of invalid signatures. Otherwise, the recovered *public key* is used to verify the signature against the transaction hash, which serves as a unique fingerprint for the transaction data. If verification fails, the signature is also marked as invalid.

The most important aspect of this function is the following: for each successfully validated signature, the corresponding validator's *voting power* is added to a cumulative total. This total represents the sum of the voting power of validators who have approved the transaction. In the absence of errors, the function returns the *accumulated total*, which helps determine whether the transaction has gained sufficient support (in terms of voting power) to be proposed to the *consensus* layer.

The overall goal of this logic is to ensure that only transactions with an adequate level of endorsement from validators—measured by their voting power—are considered valid for potential inclusion in the consensus process.

B. Receiving Transactions

On the receiving side, we had to modify the message deserialization process. The transaction reception mechanism operates within the *Receive* method of the reactor, which continuously listens for incoming messages.

To maintain backward compatibility, the following line remains unchanged:

```
protoTxs := msg.GetTxs()
```

This ensures that the bytes received in the message are correctly deserialized into a protobuf object.

After deserializing the transactions, we update newly introduced metrics and then call *TryAddTx* in the same way as before, using the transaction bytes extracted from the message:

```
tx := types.Tx(protoTransaction.TransactionBytes)
```

This approach guarantees that our changes do not break compatibility with existing logic. Then we just handle adding the signatures to the transaction. Indeed, Once a transaction has been added to the peer's mempool (via *TryAddTx*) and transformed into a *mempoolTx* object, it is initialized with an empty signature map. However, we need to populate it with the signatures that were included in the received message. To do so we pass the signatures (extracted from the message to the TryAddTx method):

```
_, err := memR.TryAddTx(tx, e.Src, protoTransaction.Signatures)
```

Within *TryAddTx*, the mempool's *CheckTx* method is invoked first and it brings us in the implementation of the mempool. We pass in the signatures map and use it at different places.
One of those place is this one:

```
   if added := mem.addToCache(tx); !added {
       mem.metrics.AlreadyReceivedTxs.Add(1)

       // Check if the transaction is still in the mempool.
       if elem, exists := mem.txsMap[txKey]; exists {
           memTx := elem.Value.(*mempoolTx)

           if err := mem.AddAndValidateSignatures(txKey, signatures); err != nil {
                mem.logger.Error("Signature validation failed", "txKey", txKey, "err",
err)
               return nil, err
           }
           mem.logger.Info("Updated signatures for existing transaction",
               "txHash", tx.Hash(), "totalSignatures", len(memTx.signatures))

       } else {
           mem.logger.Warn("Tx in cache but missing from mempool",
               "tx", log.NewLazyHash(tx))
       }

       // Record a new sender for a tx we've already seen.
       // Note it's possible a tx is still in the cache but no longer in the mempool
       // (eg. after committing a block, txs are removed from mempool but not cache),
       // so we only record the sender for txs still in the mempool.
       if err := mem.addSender(txKey, sender); err != nil {
            mem.logger.Error("Could not add sender to tx", "tx", log.NewLazyHash(tx),
"sender", sender, "err", err)
       }

       return nil, ErrTxInCache
   }
   signatures, err := mem.signIfNeeded(tx.Hash(), signatures)
   if  err != nil {
       mem.logger.Error("Failed to sign transaction", "txKey", txKey, "err", err)
   }
   mem.logger.Info("SIGNED TRANSACTION", "signature", signatures)
   txVotingPower, err := mem.validateSignatures(tx.Hash(), signatures);
   if err != nil {
       mem.logger.Error("Signature validation failed", "txKey", txKey, "err", err)
       return nil, err
   }
```

The goal of this is to allow transactions to collect validator signatures while maintaining their integrity and preventing unnecessary duplicates.

When a transaction is received, the system first attempts to add it to the *mempool cache*.

If it is already present, the code checks whether the transaction is still in the *mempool*. If it remains there, its *signatures* can be updated by calling *AddAndValidateSignatures*, merging them with the signatures that have just been received from another peer.

If it is not already present (that means the transaction is new) the *signIfNeeded* function is called to attempt signing the transaction locally, indeed on reception of the transaction if the current peer did not sign the transaction it has to try and sign it. Once the signatures are collected, they must be validated using validateSignatures, which verifies their authenticity and accumulates the corresponding voting power. If validation fails, an error is raised, and the transaction is considered invalid.

The entire logic is designed to ensure that only properly signed transactions, backed by sufficient voting power, remain valid within the mempool. The system also dynamically updates signatures, preventing the need to reinject an entire transaction each time a validator adds their approval.

In the following we add the signatures to the newly entry in the mempool. We update the transaction if the total stake of the peer that validate it is greater than the threshold we have chosen. This is done in *AddAndValidateSignatures* if the transaction is already present, or directly after creation of the transaction in the mempool with the function:

```
func (memTx *mempoolTx) SetThresholdReached(reached bool)  {
   memTx.isTresholdReached = reached
}
```

C. The Broadcast

On broadcast we have added two things.
First we check if the stake have reached the threshold set:

```
        if entry.GetThresholdReached() {
            memR.Logger.Info("Do not send, threshold reached")
            continue
        }
```

If it is reached we go to the next transaction to broadcast.

The second thing to do is the set the newly transaction object to be sent:

```
            txMessage := &protomem.Txs{
                Txs: []*protomem.Transaction{
                    {
                        TransactionBytes: entry.Tx(),
                        Signatures:       entry.GetSignatures(),
                    },
                },
            }
```

V. Performance Analysis for the stake based approach

A. Test environment

In order to evaluate the impact of our solution regarding bandwidth, it is essential to prepare an adapted runtime environment.

The key metrics we need to monitor say a lot about several aspects of the system : the total execution time of operations, the quantity of bytes transmitted between nodes, the number of transactions sent and received for each node and the size of the signatures sent/received. To do this we had to understand how the metric system worked and implement new metrics as well as call the right variable at the right time.

B. Data collection

When running the given end-to-end test suite, a network of nodes is created (using Docker), and each node spawns a Prometheus instance. A lot of different metrics are gathered, only short lived (to the duration of the node's lifetime).

New metrics were created in order to get relevant measurements regarding the changes to the mempool.

For mempool transaction sizes:

- **cometbft_mempool_tx_size_bytes_sent**
- **cometbft_mempool_tx_size_bytes_received**

For transaction amounts:

- **cometbft_mempool_tx_size_transactions_sent**
- **cometbft_mempool_tx_size_transactions_received**

For data related to signatures :

- **cometbft_mempool_tx_size_signatures_sent_size**
- **cometbft_mempool_tx_size_signatures_received_size**

We also created a sh script to export data to CSV files in the "monitoring/exported_data" directory.

C. Test parameters

To benchmark our solution we implemented a flexible and re-usable python script name test_process.py that can be found at the root of the project that allows us to quickly test configuration. It creates testnet network configuration files on the flow and automatically builds and destroys neworks.

In all the tests we have made, we have chosen to use the following configuration:
5 transactions per second during 40 seconds. That results in each experience having a maximum of 200 transactions committed in the chain.

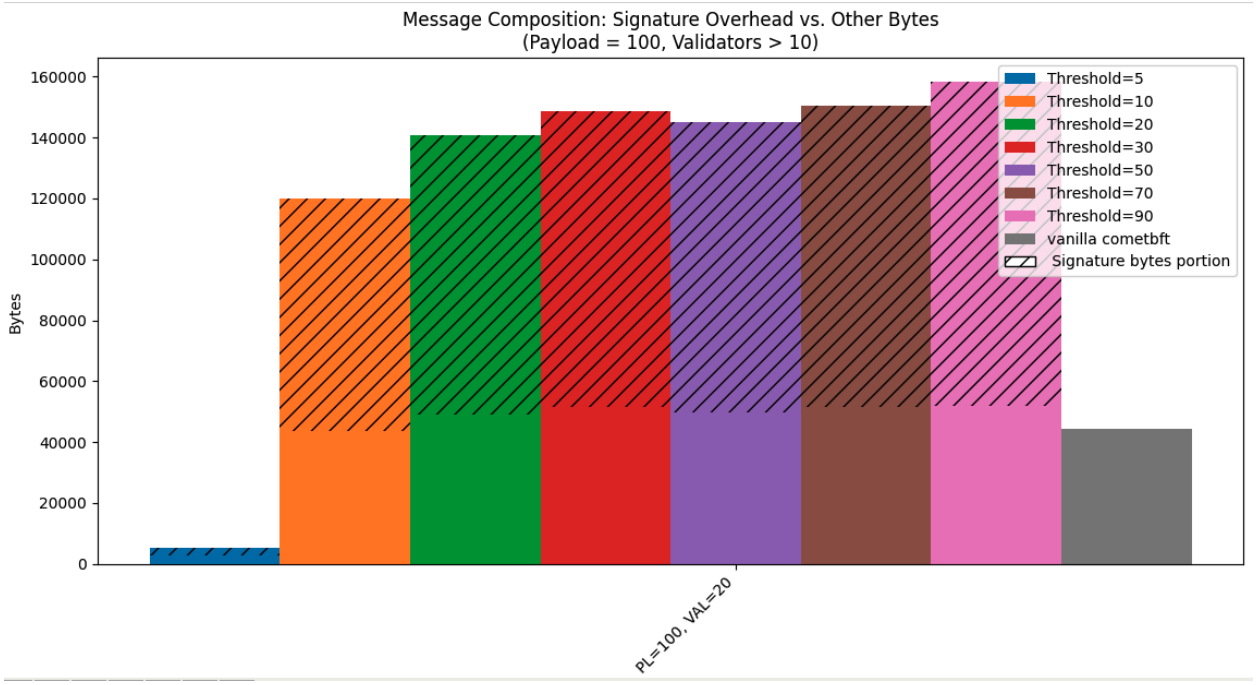
The tests have been ran while varying all these parameters to better measure their impact:

- Payload size : 100, 250, 500 bytes
- Number of validators : 4, 8, 12, 16, 20 nodes
- Threshold values: 10, 25, 40, 50, 75, 90 percent (+ vanilla CometBFT)

A little and fun statistics that makes 90 different test configurations that run each for 40 secs (a bit more to stabilize the network) so that is more than one hour just to visualize those metrics.

D. Results

For this part we are going through the main result that we find with the stake based approach idea:



This is very interesting, but at the same time we could have thought about it (we didn't). We have to keep in mind that our scenario is a complete network—this is what the basic testnet configuration from CometBFT proposes. As you can see, it seems that the implementation outperforms the vanilla implementation only in the case of a threshold of 5 percent. This can be easily explained: indeed, 5 percent of 20 amounts to 1 validator (knowing that, in our scenario, all validators have the same stake).

Let's simulate the scenario:

- a client sends a transaction;
- a node receives it, signs it, and sends it to its peers (everyone, because we are in a complete network).
- Upon reception, the signature is validated and the node sees that the threshold has been reached (1 validator needed), so it does not forward the transaction again.
- Everyone has received the transaction and everyone is satisfied with it. This is the most efficient scenario because there are no redundant sends—there were only 19 sends!

On the other hand, for any other threshold value, we observe that the payload size remains nearly the same as in the vanilla implementation. However, what causes the total bandwidth usage to double is the additional overhead from the signatures. To illustrate this, let's simulate a scenario with a threshold value of 10 percent. The exact percentage doesn't matter, as the result follows the same pattern.

- A client sends a transaction, and a node receives it, signs it, and then broadcasts it to all its peers (since we are in a fully connected network).
- Upon reception, each node validates the signature and checks whether the threshold has been reached. In this case, two signatures are required to meet the threshold, but each node sees that only one signature is present.
- Therefore, each node forwards the transaction to all its peers except for the sender (to avoid sending it back).
- Since there are 20 nodes in the network, each node sends the transaction to 18 others (excluding itself and the original sender).
- This forwarding process occurs simultaneously across all nodes.
- As a result, each node receives the same transaction from 18 different sources at the same time.
- Since each received transaction carries a signature, the total number of signatures instantly reaches 20 (one from itself, one from the original sender, and the 18 received from peers).

In this scenario, the initial transaction propagation consists of 19 sends—one from the initiator to all others. Then, each of the 19 receiving nodes broadcasts the transaction to 18 others, leading to an additional 18 × 19 transmissions.

More generally, for any threshold that requires more than one validator, the total number of transmissions follows the pattern:

$$(n-1)+(n-1)\times(n-2)$$

which results in $O(n^2)$ complexity. This is precisely the standard case of our gossip-based dissemination algorithm, where messages are redundantly forwarded throughout the network. Consequently, while the threshold mechanism optimizes propagation in specific cases (such as when only one signature is required), for higher thresholds, it ultimately behaves like a traditional gossip protocol, leading to significant bandwidth consumption.

An interesting optimization arises when the stake is not uniformly distributed among validators. In such a scenario, if the transaction is initiated by the node with the highest stake, we could achieve the best-case propagation efficiency. If this node's stake alone exceeds the required threshold, the transaction would be immediately considered valid without requiring further propagation.

This would significantly reduce network overhead, as the transaction would not need to be redundantly forwarded by other nodes. Instead of the O(n2)O(n^2)O(n2) complexity observed in the standard gossip case, we would effectively have O(1)O(1)O(1) propagation in the ideal scenario. This highlights a potential improvement: leveraging stake distribution to optimize transaction dissemination, particularly in networks where a few validators hold a large portion of the total stake.

VI. Another alternative? Stake-Threshold Based Transaction Propagation

A. Idea

The idea that we had and that we implemented for a part centers around using the stake of validators as a determining factor in the selection of peers for transaction propagation. The idea is that the initiator selects a subset of validators whose combined stake exceeds a

predefined threshold. Knowing this he supposed that the transactions will be seen by an important part of the network and a reliable one because of their combined high stake.
The idea to make sure that the system still works even if the transaction has not been proposed to consensus after the send to the high stake validator is that on failure, the gossiping mechanism gets back to flooding, i.e. sending the transactions to all the peers not regarding their cumulative stake.

B. Implementation overview

The current implementation can be found on the branch:

feat/propagation_counting_peers

It remains experimental and is not functional for real-world use cases. Our initial approach didn't take into account stake directly, but for the purpose of testing, we used a uniform stake across all nodes (i.e., each node had the same stake). Therefore, using a stake threshold in this case is equivalent to selecting a number of nodes based on the total number of nodes, which is the approach we used. The selected peers are then tasked with broadcasting the transaction.
Since peers join in random order, we simply begin broadcasting to the first peers that arrive until we've broadcasted to enough peers to meet the threshold. The changes made in this part of the work are fairly basic, so we won't go into further details here. If needed the main implementations are on mempoolTx.go in the mempool and in broadcastTransaction on reactor.go.

Another part of our work involved adapting this approach to be more applicable to real-world systems. This was implemented on the branch:

feat/stake_threshold_based_transaction_idea

which aimed to build on the previous logic but incorporate stake in relation to peers. However, we ran into limitations with CometBFT, as it does not provide a direct link between the network layer (the peers used to broadcast transactions) and the application layer (the validators responsible for transaction validation).
To address this, we proposed a new layer above the Switch mechanism, which we called the GossipSelector layer. This layer is responsible for selecting peers to gossip transactions to, based on the stake threshold and available peers. It is updated whenever a new peer connects to the reactor.
Here's a brief overview of the GossipElector implementation:

```
type GossipElector struct {
   peers        *PeerSet // Ensure peers are accessible
   logger log.Logger
}

// NewGossipElector initializes a GossipElector with a validator set and peer set.
func NewGossipElector(vs *types.ValidatorSet, ps *PeerSet) *GossipElector {
   return &GossipElector{
       validatorSet: vs,
       peers:        ps,
       logger: log.NewLogger(os.Stdout),
   }
}
```

```
// SelectPeersForPropagation selects peers for gossiping transactions based on stake.
// SelectPeersForPropagation selects peers for gossiping transactions based on stake.
func (g *GossipElector) SelectPeersForPropagation() []Peer {
   // Load threshold percentage from environment variable…

   // Retrieve total voting power from validator set…

   thresholdAbsolute := int64(totalVotingPower * int64(thresholdPerc) / 100)

   // Build a list pairing each peer with its stake.
   var peerStakes []peerWithStake

   // Collect eligible peers based on validator set.
   for _, v := range g.validatorSet.Validators {
       id := nodekey.PubKeyToID(v.PubKey)

       // Retrieve peer from the peer set.
       peer := g.peers.lookup[id]
       g.logger.Info("Peer matched !", "peer", peer)
       if peer != nil {
           peerStakes = append(peerStakes, peerWithStake{
               peer:  peer.peer,
               stake: v.VotingPower,
           })
       }
   }


   // Shuffle peers randomly.
   rand.Shuffle(len(peerStakes), func(i, j int) {
       peerStakes[i], peerStakes[j] = peerStakes[j], peerStakes[i]
   })

   // Select peers until cumulative stake exceeds threshold.
   var selected []Peer
   var cumulative int64
   for _, ps := range peerStakes {
       if cumulative >= thresholdAbsolute { // Corrected comparison
           break
       }
       selected = append(selected, ps.peer)
       cumulative += ps.stake
   }

   return selected
}
```

This GossipElector object is created after the initialization of the reactor, which already contains the validator set. The peers are then added to the *PeerSet* via the method *AddPeerToSet(peer Peer)* every time a peer connects to the reactor.

However, we encountered a key issue with the following code:

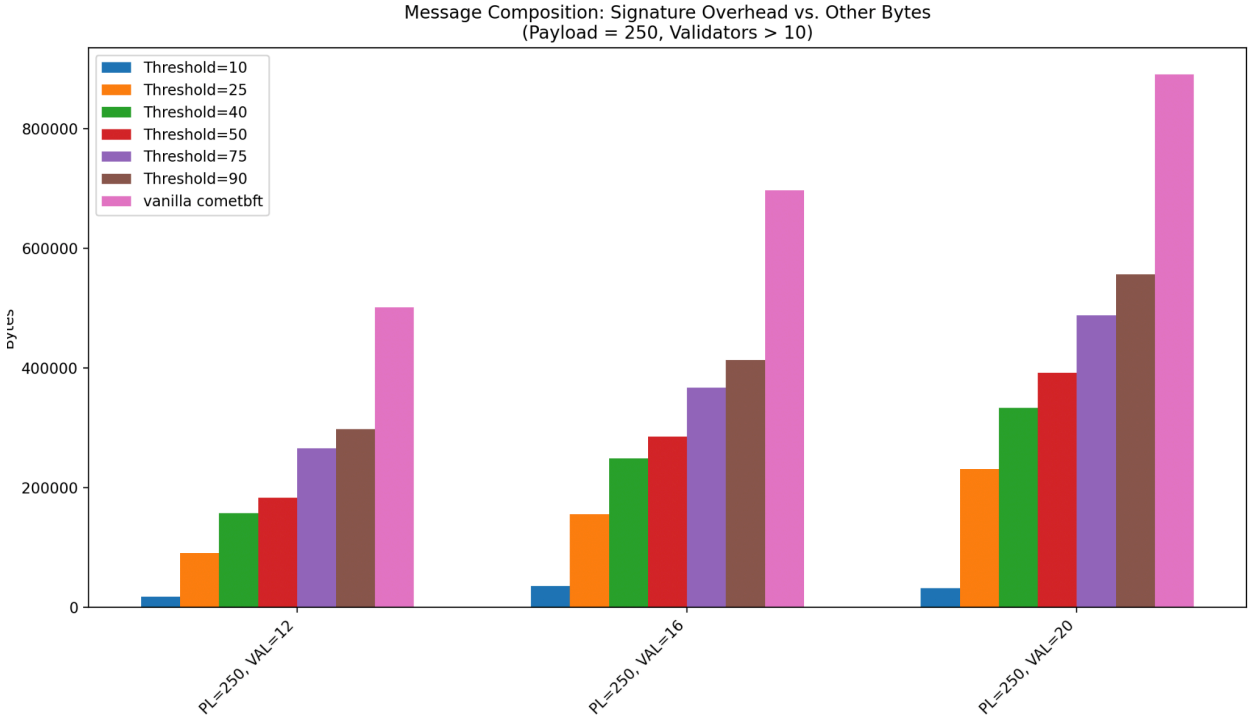
```
       id := nodekey.PubKeyToID(v.PubKey)

       // Retrieve peer from the peer set.
       peer := g.peers.lookup[id]
```

While the idea seemed promising, we quickly ran into a mismatch: the network layer uses a different public key for peers than the application layer uses for validators. As a result, the public key of a validator doesn't match the peer ID, making it impossible to link the validators with their corresponding peers.

In conclusion, although the concept showed potential, we faced significant implementation challenges, particularly around linking the two objects—the network layer and the application layer—properly. Given the time constraints and the complexity of CometBFT's architecture, we were unable to resolve these issues to fully integrate the stake-weighted propagation with the network's actual peer connections.

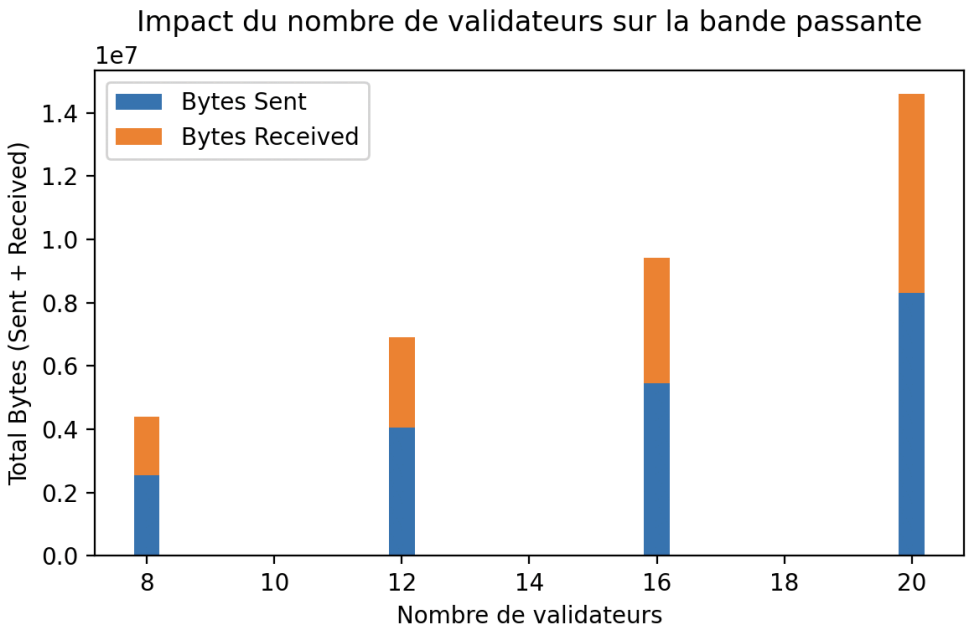
C. Results

We have obtained some interesting results and visualizations for this approach, based on the functional implementation in the *feat/stake_threshold_based_transaction_idea branch.* We used our benchmark framework (*test_process.py*) to produce those results.

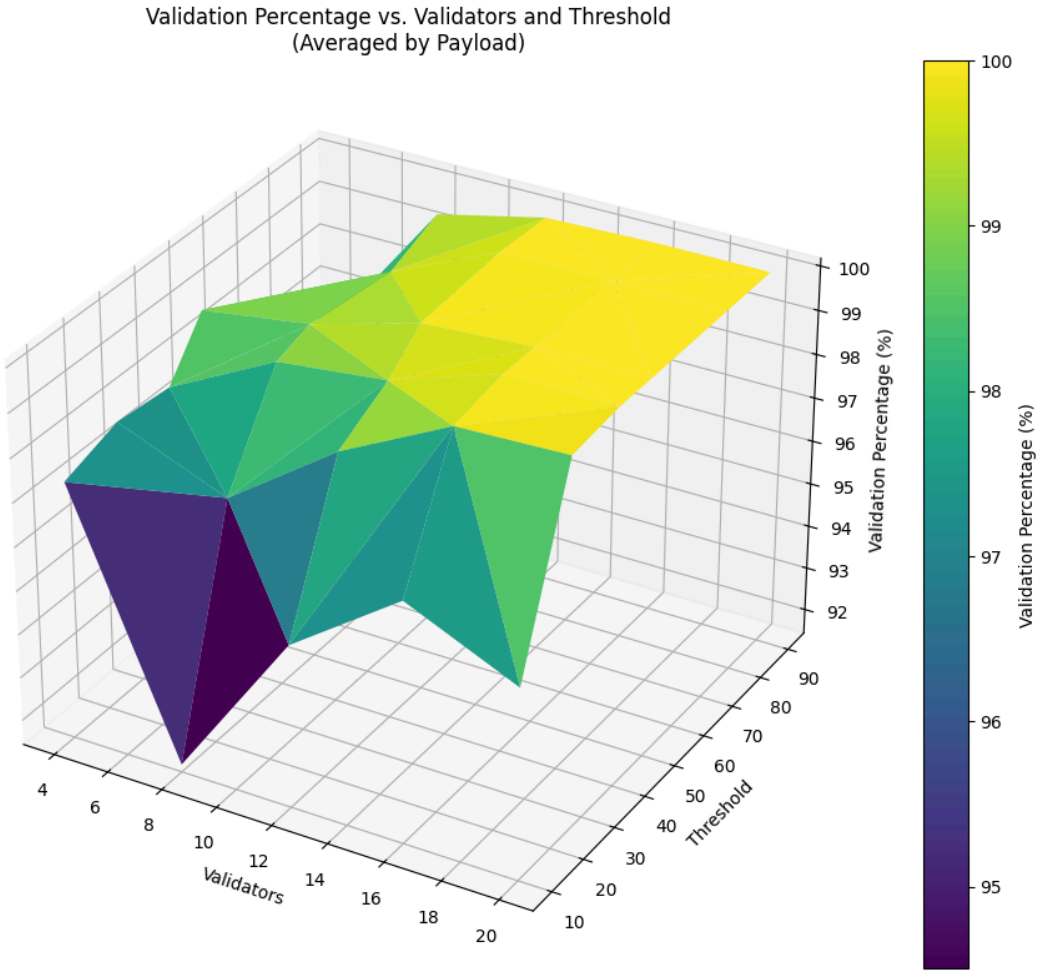


The diagram below illustrates that, under a stake-based approach with highly optimistic parameters and a fully connected network, the total bandwidth consumption of the system increases as the threshold increases. This makes sense as the more the threshold increases the more we send the transaction to various peers.

To generate this diagram, we measured the total bytes exchanged across the network over five runs, each processing a load of 200 transactions. As expected, the total bytes exchanged scales linearly with the number of validators.



As we could expect, the number of validators linearly varies with the total bytes exchanged.



The *validation percentage* represents the proportion of transactions successfully committed to the blockchain—meaning they passed through the mempool, were pushed to consensus, and ultimately validated.

The previous graph shows that, under our implementation, the validation percentage generally remains high (95-100%) across realistic configurations, particularly when the number of validators is large and the threshold is above 50%. This suggests that the approach is robust (in a complete network). The lowest validation percentages (around 94-95%) appear in scenarios with fewer validators (4-8) and low thresholds (10-30%).

Lower thresholds (10-30%) exhibit greater variability in validation performance. As the threshold increases, the validation percentage stabilizes and remains consistently high. There seems to be an optimal range—around a 50-60% threshold—where performance remains stable while maintaining efficiency.

A higher number of validators generally leads to better validation percentages. Additionally, as the number of validators increases, the impact of threshold variation becomes less significant. In real-world scenarios, where validator counts are significantly higher, these effects are likely to be even more pronounced.

VII. Conclusion

The work we carried out focused on optimizing transaction propagation by using validator stake as a criterion for selecting the peers to which transactions should be sent. The underlying idea was to reduce bandwidth consumption by having transactions validated only by a subset of validators whose total stake exceeds a certain threshold. However, this approach proved to be more complex than expected. In our first implementation, which used a uniform stake among nodes to simplify testing, the idea was to set a threshold based on the stake. But as the threshold increased and required more than one validator, bandwidth consumption also increased, exhibiting behavior equivalent to traditional gossip—but worse due to the added signature size.

The architecture of CometBFT also posed a major challenge. We were not familiar with the Go programming language and all its concepts (goroutines, channels…). Moreover, the codebase is vast and intricate, requiring significant effort in digging through and reading the code to understand the objects we were interacting with. At the same time, this complexity was appealing—we successfully implemented new mechanisms within an industrial-grade system, which is quite rewarding.

Although the idea of optimized propagation by pre-selecting validators before gossiping based on stake, which we explored towards the end of our project, is promising, it is not yet ready for production use due to these technical difficulties. However, we identified several ways to improve this approach. For example, it would be possible to strengthen the link between the network layer and the application layer or to adopt a hybrid approach where stake-based selection is complemented by a gossip-based diffusion mechanism when necessary. A push-pull mechanism could also be considered once the initial propagation round is complete.

To conclude, while our results are not groundbreaking, this final-year project introduces real ideas and potential improvements for reducing CometBFT's bandwidth consumption. We demonstrated the effectiveness of stake-based propagation in a specific configuration. We provided a testing framework that enables detailed, fast, and flexible benchmarking for any metric. And finally, we learned a great deal from this project—which, in the end, might be the most important part.