UCLouvain

École polytechnique de Louvain

DAVE: Deterministic, Adaptive, Verifiable and Efficient transaction placement for sharded UTXO blockchains

Authors: **Maxime Clément, Alexandre Moueddene**
Supervisors: **Etienne Rivière, Pierre Schaus**
Readers: **Michał Król, Emanuel Onica**
Academic year 2020–2021
Master [120] in Computer Science and Engineering

Acknowledgements

As a team, we would like to express our gratitude to several people. We thank Prof. Etienne Rivière for sharing his knowledge and guiding us in promising directions when we needed it. We are also thankful, for his help regarding the writing of this thesis and most certainly for meeting with us on a weekly basis. We thank Prof. Pierre Schaus for taking the time to meet with us during the year and giving helpful advice. We are also thankful for his help during the writing of this thesis. We thank Dr. Michał Król for taking the time to attend weekly meetings with us, for all his valuable help, as well as providing direct contact with people involved in projects we used and for reviewing this thesis.

We also want to thank our supervisors as well as Emanuel Onica for taking the time to read our work and be part of our jury. In addition, we want to thank our friend Thomas who has helped us with several aspects of the dissertation and Dr. Sergi Rene who helped us run Byzcuit.

Finally, we would like to thank the Brussels Innovation fund Innoviris for their financial support to our experimental evaluation in the context of the FairBCaaS project.

Now individually, Alexandre would like to thank his family and especially his parents for the support throughout the year as well as throughout all his academic years since kindergarten. Maxime would also like to thank his family and parents, as well as his girlfriend Rosalie for all the support and motivation they gave him.

Abstract

Sharding is a well-known method that allows UTXO blockchains to scale up. This technique introduces two types of transactions, the intra-shard and inter-shard transactions. Their respective temporal cost is very different as the inter-shard transactions are a lot more time-consuming. The main problem of existing sharded blockchains is that they tend to shard transactions randomly, which creates a lot of inter-shard transactions. In addition to that, if an attacker managed to hack the system, he could place his transactions where it suits him best without anyone being able to detect it since the placement strategy is random.
We developed DAVE, a transaction placement strategy that manages to minimize the overall number of inter-shard transactions while making the placement strategy deterministic and computed at the miner side. When introduced in existing randomly sharded blockchains, this new way to take placement decisions improves both their throughput and latency. DAVE even enables the execution of a verification mechanism allowing anyone to check that a transaction has been placed where it was supposed to.

Contents

Chapter 1

Introduction

Cryptocurrencies are more popular than they have ever been. Certainly, it is a promising technology but it does not come without drawbacks. In this thesis, we use Bitcoin as an example to illustrate what kind of challenges we have to face.
Therefore we present first the background needed to understand how the Bitcoin protocol works, this goes from the blockchain principle to the design of the Bitcoin protocol itself. This protocol has very poor scalability which means that increasing the size of its network will not lead to better performances in terms of throughput and latency. Indeed, all miners must validate and replay transactions, and that it is intrinsically unscalable. Blockchain technologies is a fast-moving sector as we find more and more blockchain-based applications in our everyday lives. This goes far beyond the scope of cryptocurrencies and the scalability issues of current protocols have to be addressed. For example, the case of high-frequency trading would require blockchains with extremely low confirmation time to satisfy current's standards. High throughput is also a requirement for future blockchain applications, for instance in smart cities that are big data generators.

Fortunately, different approaches aiming to improve its scalability exist and we present the most interesting ones in a dedicated state-of-the-art section. One of the best solutions found implies the introduction of parallelism in the network, this family of blockchains is what we call the sharded blockchains. The main idea behind sharding is that subsets of the network's nodes, also called shards, validate transactions to build several blockchains in parallel instead of one.
In models such as Bitcoin, a transaction is always related to one or more previous transactions. In order to validate a transaction t, the shards have to communicate to find the(se) previous transaction(s) to which t is related. A transaction for which the related previous transactions are in different (resp. the same) shard(s) is called an inter-shard (resp. intra-shard) transaction. Inter-shard transactions imply a lot more communication in the network and are therefore a bottleneck for its overall performances.

The problem is that most of the sharded blockchains place transactions randomly between shards, introducing this way a lot of inter-shard transactions. A solution would then be to design a transaction placement method that minimizes the number of inter-shard transactions. Yet, the only sharded blockchain we heard of which does not place the transactions randomly, unfortunately, has several weaknesses as well. The decision is

computed at the user side and the transaction placement cannot be verified afterwards.

In this master thesis, we present DAVE, a Deterministic, Adaptative, Verifiable and Efficient transaction placement for sharded blockchains. It was designed to address all the problems encountered with existing sharded blockchains. We explain in detail how this solution works and evaluate it. The solution can be divided into two parts, first the efficient sharding algorithm which is DAVE and then how it could be introduced in a real network to obtain a working system.

Chapter 2

Background and problem statement

This chapter presents a quick explanation of the blockchain principle before continuing with a well-detailed description of the UTXO model. Then, a state-of-the-art section mentions interesting works done to improve the weaknesses of this model. Finally, a problem statement will summarize the remaining problems that still need to be solved.

2.1 The Bitcoin blockchain

Described first by Satoshi Nakamoto in 2008 [1] and then implemented in 2009, Bitcoin is a cryptocurrency secured by cryptography and based on the blockchain principle. The blockchain principle is not new, the main idea was first described back in 1991 by Stuart Habert and W. Scott Stornetta [2]. They tried to find a mathematically sound and computationally practical solution to the document time-stamping problem. This problem is about the difficulty to attribute a date to a document in such a way that we can be sure the date is valid and cannot be modified after the document was written. The main idea evoked was to use hash functions as security and cooperation in a distributed network of users to certify the validity of a document time-stamping. These are the basis of blockchains, cryptography to ensure security and a distributed network of nodes to build the blockchain.

Let's explain with an example the blockchain principle and more particularly the public blockchain principle[1], as it is the model used for Bitcoin. For this, we will introduce Alice, Bob and Charlotte. These three people will help us now but also later in this work to discover and understand how the Bitcoin system actually works. Fig. 2.1[2], illustrates what happens if these three people want to make transactions that will be added to the Bitcoin blockchain. We can see that the transactions are put together in a data structure called a block and that this block will then be added to a chain of already existing blocks that we call the blockchain. The word ledger is also often used for blockchain. Thanks to cryptography, all information stored in the blockchain are linked. Indeed, a block header is computed based on its content but also based on the previous block. Therefore, if anyone wanted to remove something from the blockchain, we would immediately see it because the continuity of the chain would be broken. From this, it is obvious that any

[1] Private blockchains exist as well but we will not discuss them in this thesis.

[2] Some images were taken from the websites espace-emeraude.com [3] and fotomelia.com [4] to make this illustration.

block added to the blockchain contains information on all the preceding blocks. The website JavaTpoint summarizes the blockchain principle by claiming that "a blockchain is a constantly growing ledger which keeps a permanent record of all the transactions that have taken place in a secure, chronological, and immutable way" [5]. These are the main properties to keep in mind when talking about blockchains. Also, if Alice, Bob and Charlotte wanted to find information about the transactions they made in the past, they can simply look at the content of the blockchain at any time. We say that it is publicly available.

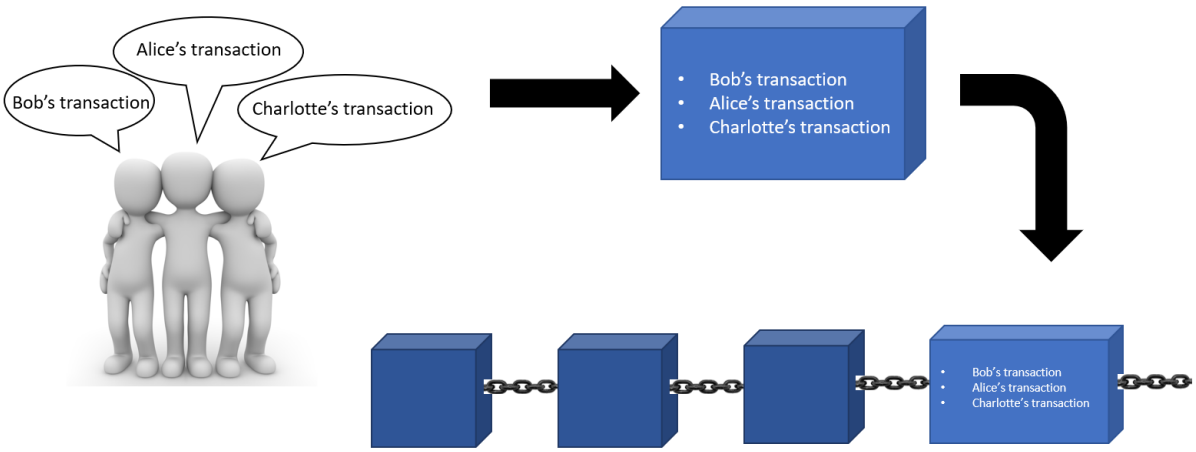


Figure 2.1: The blockchain principle

Like most cryptocurrencies, Bitcoin relies on a decentralized network, meaning that there is no need for a trusted third party to execute transactions. The nodes of a peer-to-peer network (also called P2P network) timestamp transactions by hashing them into an ongoing chain of hash-based proof-of-work[3]. By peer-to-peer network, we mean a network exchange model where each entity is both client and server at the same time, in opposition to the client-server model. Using a P2P model has several properties as the ones mentioned by the website Indeed [6]. These are easy file sharing, a reduction of the costs, adaptability, reliability, efficiency and high performances. Even if a P2P model should improve the performances of the network, performance is often considered a weakness for Bitcoin. We will talk about the Bitcoin performances more in details later as it is what we are interested in.

Often, people are confused between the terms Bitcoin and blockchain. As explained, the blockchain is an abstract principle from which several blockchain technologies were developed while Bitcoin is a cryptocurrency based on the blockchain principle. We already have described the blockchain principle so we will now focus on the Bitcoin model itself more in details in the following chapter.

[3]These concepts are explained in the next section.

2.2 UTXO model

UTXO is short for Unspent Transaction Output and is a model used by many cryptocurrencies such as Bitcoin, Litecoin and others. It keeps track of coins' ownership. Here we will analyze the UTXO model in the context of the Bitcoin protocol but most of what is said below is still valid in other protocols based on UTXO.

2.2.1 Keys and Addresses

Coins can change of owner during an action called **transaction** which is made of a series of inputs sending coins to several outputs. The number of inputs and outputs does not have to be the same, it is up to the user to decide where he wants to send coins.
The model's name was taken from the objects it uses, the UTXOs, which represent the money a user received as an output in a transaction and that is still spendable. An UTXO becomes spent when it serves as input in a transaction. This means that a transaction uses UTXOs, spends them and creates new ones. In the simplified representation of a transaction below, O1 and O2 are UTXOs since they are the outputs of the transaction, and thus receive funds. While I1, I2 and I3 are not UTXOs anymore. Indeed, they are the inputs of this transaction and their coins are being spent.

$$I1 \ + \ I2 \ + I3 \ \longrightarrow \ O1 \ + \ O2$$

More precisely, UTXOs are outputs associated with specific **addresses** that contain a certain amount of coins. This amount can be a fraction of coins, it does not necessarily need to be an integer. But what is an address? How do users get one?
The creation of an address starts with what is called a **private key** as in Fig. 2.2, which is simply a number made of 256 bits. A user can easily generate this number randomly with the only requisite being that no one already created it before. The probability of two people creating the same private key is extremely small since the total number of private keys is approximately 10^{77}(since $2^{256} = 1.16 * 10^{77}$). Once a private key is created, it has to be kept in a safe place because anyone in possession of it will be able to use the funds associated with it. Also, the event of a user forgetting this number would lead to the harsh but sadly very real possibility of never being able to access the coins. That is why there are so many Bitcoins out there whose owners can no longer access them.

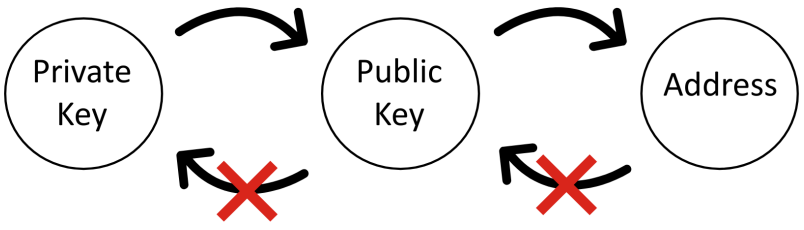


Figure 2.2: Scheme of the creation of addresses.

From this private key, a one-way cryptographic function called ECDSA[4] generates what is called a **public key**. This step will lead us from the left-most circle of Fig. 2.2 to the middle one. The crossed-out arrows illustrate the one-way property of the cryptographic functions used. As its name indicates, the public key is meant to be visible by others. It is only thanks to the combination of those two keys that the user will be able to form transactions. But in real life, people go one step further and transform the public key into an **address** using a cryptographic hash function. Note that with only one public key, we can create as many addresses as needed. Those addresses hold the user's coins, as explained previously.

To give a quick idea of how a Bitcoin transaction can be seen intuitively, we will make a metaphor between the private keys and the real keys used to close doors. So we will also talk about locking a Bitcoin instead of sending it to the address of someone else and about unlocking a Bitcoin instead of spending it.
We consider as an example the situation where Alice wants to send Bitcoins. In fact, Alice can only spend her own Bitcoins but she can give them to anyone. Alice could unlock every UTXO she possesses because only she knows the corresponding key to unlock it. However, she could lock it to anyone's address because to lock we do not need the private key associated with the given address. If Alice wants to send her Bitcoins to Charlotte, she will lock them on one (or more) of Charlotte's addresses so that only Charlotte could unlock it later on by using her private key.
Then Charlotte could simply unlock the Bitcoins to immediately lock them on another of her own addresses. She would do that because by doing so, Alice would not know anymore where the Bitcoins are without looking at the Bitcoin blockchain for the corresponding transaction.

As stated before, transactions are made of one or more addresses of input and one or more addresses of output. This means that the money of the addresses of input is transferred to the addresses of output. Someone might wonder where does the private key plays a role in this. In a transaction, each input is signed by the owner's private key related to the address (in this process the private key is never revealed). This way, we are sure that the real owner of the address is aware of the transaction and accepts it, otherwise, the money could not be transferred.

Why using addresses instead of public keys then? Because it serves as additional protection against the decryption of someone's private key. Thanks to this, a user might never reveal his public key either because he can always use his addresses in transactions. Then if someone managed to reverse the hash function (which is highly unlikely), the attacker would still have to reverse the ECDSA algorithm in order to find the private key from the public one. This should be sufficient protection against attackers. Another reason addresses are used instead of public keys in transactions is that addresses are represented with 160 bits while a public key needs 256. Hence, a transaction with addresses requires less space than one with public keys.

[4]ECDSA stands for Elliptic Curve Digital Signature Algorithm.

In practice, people will unlikely always have enough funds in a single address to send the desired amount to the output address. Fortunately, they can combine multiple addresses such that their sum is bigger or equal to the amount they want to send. Thus, if the sum is bigger than the expected amount to be sent, the excess of money will be placed in a sender's owned output address also called **change address**. For example, we can take back Alice and Charlotte, from the previous example to illustrate this. Let's suppose Alice wants to send 2.1 Bitcoins to XX, one of Charlotte's addresses. But she only has the following funds on her own addresses (where BTC stands for Bitcoin), see Fig. 2.3. The funds are expressed in BTC for convenience, but in the blockchain, the amounts are written in **satoshis**, which is the smallest amount of Bitcoin someone can own. The conversion is 1 BTC = 10^8 satoshis.

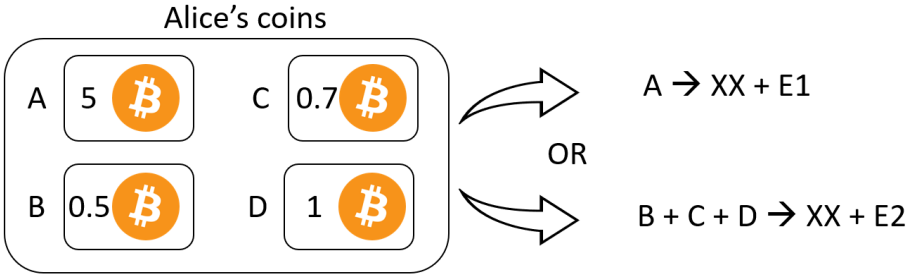


Figure 2.3: Alice's possible transactions to Charlotte (XX)

In order to send the 2.1 BTC to Charlotte, Alice could submit two possible transactions pictured in the figure above. In both cases, the letter E is used to represent a change address in the transaction. So in our case, both E1 and E2 are possible change addresses that could be used, depending on which transaction will be submitted by Alice. By looking at the content of Alice's addresses, we can easily observe that in the change address E1, Alice would recover 2.9 BTC, while in E2 she would get 0.1 BTC. In both situations, these Bitcoins belong to her and are just the changes associated with the two possible transactions.

It is not recommended to use the same addresses several times, **it is even advised to create a new address for every transaction to ensure better security and anonymity in the network**. Since creating addresses is particularly simple and easy, it is auspicious to use a brand new address each time a user expects to receive funds. This will help a lot to protect the user's anonymity which is one of the initial goal of Bitcoin. With this process, a malicious person will have a hard time determining what coins belong to a certain user with certitude by following the history of transactions. The anonymity aspect of the UTXO model will be further detailed in section 2.2.5.

We will now present the different objects needed, in addition to addresses, to build a real Bitcoin transaction. To start, we will look at a transaction object which is composed of many fields as shown in Table 2.1, in particular one or several inputs object(s) and output object(s). Then, to go into the details, we present in Table 2.2 below the different elements that compose an output object and in Table 2.3 the elements that we can find in an input object.

Element	Description
Version number	The type of transaction.
Number of inputs	The number of input objects of this transaction.
The inputs	All the inputs objects intervening in the transaction.
Number of outputs	The number of output objects of this transaction.
The outputs	All the outputs objects intervening in the transaction.
The locktime	A block height or a time, before which the transaction is not considered executed.

Table 2.1: Elements of a transaction object

Element	Description
The amount	The amount of Bitcoins to be associated to the output (in satoshis).
The locking script	The conditions to meet in order to spend the coins of this output. Also called *scriptPubkey* has it contains in particular the owner address. It is used during a transaction, at the moment when outputs are created. The locking script is executed to associate the UTXO with the address of the new owner.
The locking script size	The size of the locking script.

Table 2.2: Elements of an output object

Element	Description
The previous transaction hash	The hash of the transaction where this input/UTXO has been created.
The index	The index where the input is located among the outputs/UTXOs of the transaction.
Unlocking script	A script that meets the conditions set by the locking script. When a transaction is created, the unlocking script has to be placed in the input object in order to spend the coins of the corresponding UTXO. It is a proof for the actual ownership of the UTXO.
Unlocking script size	The size of the unlocking script.

Table 2.3: Elements of an input object

In a transaction, the inputs are not directly composed of the addresses. Instead, they are composed of the necessary information needed to retrieve the right UTXOs in the blockchain. We are going to clarify all those information about transactions, inputs and outputs objects with an example of two real transactions coming from the bitcoin blockchain explorer website [7] as depicted on Fig. 2.4.

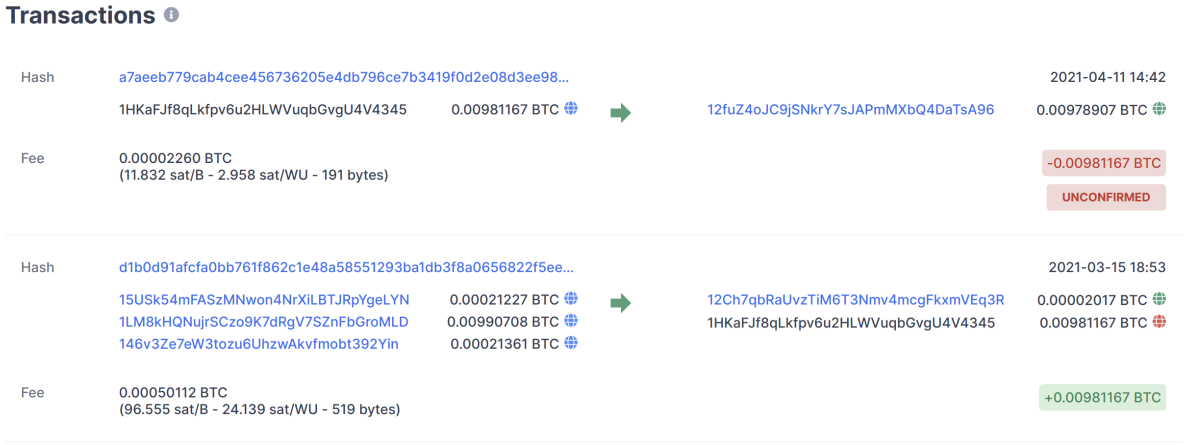


Figure 2.4: Transactions information taken from

In Fig. 2.4, the two transactions are represented with green arrows, with inputs on the left and outputs on the right. We can see that we have here one transaction (the bottom one) using three distinct inputs to send a total of 0.00981167 BTC to two distinct outputs. Then, one of those outputs is used in another transaction (the top one) to spend all of its funds. We can also see that the bottom transaction is already validated but not yet the top one.
The address `1HKaFJf8qLkfpv6u...` was first used as an output in the bottom transaction and then as an input in the not yet validated transaction (the other way around would be illogical). The top transaction says that it uses address `1HKaFJf8qLkfpv6u...` as input, but as explained above, a transaction does not have access to that precise information directly. It will only know which transaction has produced the outputs that should now be used as inputs. In order to find the address `1HKaFJf8qLkfpv6u...`, the website had to go to the transaction whose hash is written in the input object, also called the input transaction, which is here the transaction `d1b0d91afcfa0bb7...`. Then it uses the argument output index of the input object, to know which of the outputs addresses it has to use. In our case, the input object should have the following form:

Input object	
The previous transaction hash	`d1b0d91afcfa0bb7...`
The index	1
...	...

Table 2.4: The input object of transaction `a7aeeb779cab4cee...`

This way, we can find the address we are looking for: `1HKaFJf8qLkfpv6u...`, at the position 1 in the list of outputs of transaction `d1b0d91afcfa0bb7...`.

This section summarized how addresses and keys are used in transactions. It is also important to mention that transactions are most of the time accompanied by a **fee**. A fee will allow the transaction to be accepted and inserted into the Bitcoin network. This can be seen as a way to pay for the work that will be needed to process the transaction. The value of this fee is decided by the author of the transaction and should not be despised as an unrealistically low fee will result in a transaction being discarded by the network. This will be explained later, as the fee notion will lead us to the miner notion as well. Concretely the fee the user decides to pay is the difference between the funds present in the inputs and the funds in the outputs.
So far, we have seen the common transactions but there also exists another type of transactions, the **coinbase transactions**. Those transactions are used to introduce new coins into the network and also to recover the fees of common transactions [8]. That is why those transactions do not have inputs, only outputs where the new coins will appear for the first time in the network.

Many of the information in the section was taken from the very good book Mastering Bitcoin [9] and from the developer guide of Bitcoin [10].

2.2.2 Miners and Blocks

Once a transaction is sent into the network, its final goal is to be validated and added to the blockchain. As explained in a previous section, the blockchain is a large database shared among all nodes in the networks. The people that extend the chain are called **miners** and endorse several roles among which we can find the following.

1. Put transactions into blocks and append the block built this way to the chain.
2. Introduce new created Bitcoins into the network.
3. Ensure the validity and security of the network.

When miners compose blocks of transactions, they verify that the transactions are valid, namely that the total amount of funds in the inputs is at least equal to the total amount that should be sent to the outputs and that the inputs are not spent already. The situation where an input is spent twice, in two distinct transactions, is a well-known problem called the **double-spending** problem. Miners are full nodes, they are computers storing all the history of the transactions (which corresponds to the blockchain). With this, they can know whether an address has already spent or not its coins. It would be inefficient for a miner to scan the whole chain for each transaction in order to know whether its inputs are unspent. What miners do when they download the blockchain the first time, is that they keep in memory all the addresses containing money that has not been spent until the present. By doing so, they only have to check the validity of the inputs in these much smaller lists of UTXOs.
In return for the job done, miners expect to receive newly minted Bitcoins as well as the

fees of the transactions. Naturally, they will choose transactions with the bigger fees or at least a fee that allows them to be profitable. This is why the author of a transaction has to pay a realistic fee, otherwise, no miner will ever validate the transaction. So, the bigger the fee a user pays the most certain he can be that his transaction will be added to the ledger.

When a block is appended to the chain, it stores more information than just the transactions it contains. Here are the most important contents of a block, as explained by O'Reilly Media [11]:

- A block header which contains:
  - The previous block hash: this field is the hash of the header of the parent block. This is how blocks are linked. One could follow these links and hop from one block to another until reaching the very first block usually called **the genesis block**.
  - The hash of the Merkle tree's root: the Merkle tree is a data structure that allows determining very fast whether a transaction is in the block or not (more explanation are given in Appendix A).
  - A timestamp: the time at which the block has been created.
  - Difficulty and nonce: these numbers will be explained in the following section as they are key elements of the consensus protocol.
- A transactions number: the number of transactions contained in the block.
- The block size: the size of the block in bytes.

2.2.3 Consensus and longest chain rule

The real Bitcoin network contains many miners, each of them has a goal to turn a profit. Except that only one block can be added at a time in the blockchain, the lucky miner from which the block will be added to the blockchain is chosen via the **Proof of Work** consensus process (PoW).
Every miner can try to produce blocks with the transactions submitted to the network. But the only way for a miner to append his block to the blockchain is to have "proof that he did the work". This "work" corresponds to finding a value called a **nonce** such that the hash of the block's header is less than the fixed **difficulty** value. The smaller the difficulty, the harder it is to find the right *nonce*! The Bitcoin system was thought in a way to ensure a block should be produced every ten minutes on average. To always satisfy this property, we just have to adapt the *difficulty* to the total computing power of the network to make sure that even if the network becomes more powerful, a block is still validated only every ten minutes. With this limitation, the Bitcoin network can validate about 4.6 transactions per second. This is much less than other payment systems we use in our everyday life, it is for example much less than the 1700 transactions per second we can observe for VISA.
Remember that the *nonce* and the *difficulty* are part of the header, this implies that a change in the *nonce* will change the block header and its hash. So, once a correct *nonce* is

found, the block cannot be modified anymore because if even one single transaction in the block is modified, the hash of the block header will probably not satisfy the *difficulty* for that particular *nonce*. Thus, a new *nonce* has to be found.
An even worst situation is depicted in Fig. 2.5, where block 11 has just been validated and added to the chain. Now, a malicious miner decides to modify a transaction in block 9. It follows that the PoW for block 9 becomes invalid, so the attacker would have to redo it but then the PoW of block 10 becomes invalid since the *nonce* of block 10 was found for the header H9 and not H99. The same follows for block 11. Thus, we can see that it requires a lot of work to change a past block since we would have to redo all the work from there until the present.

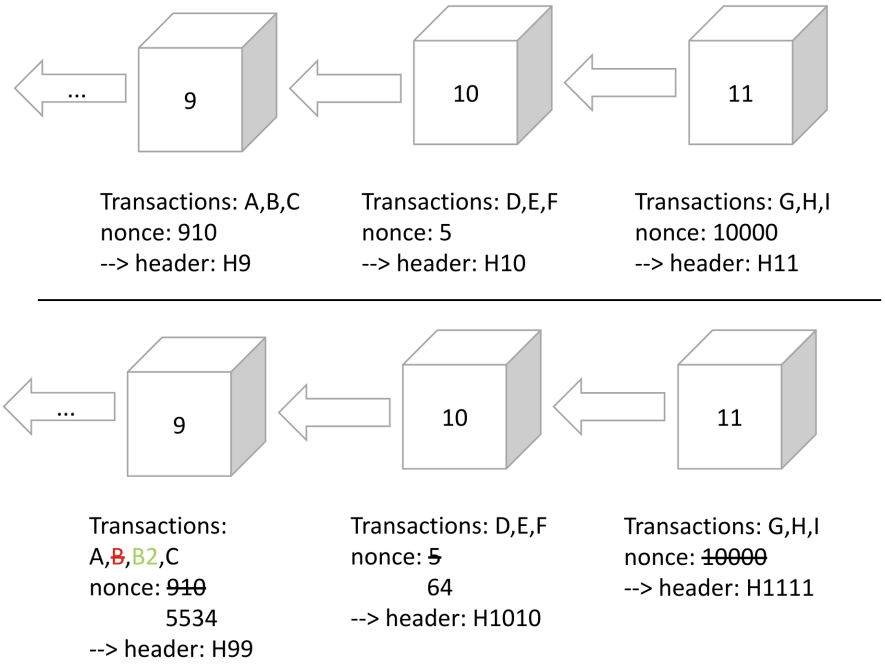


Figure 2.5: Blockchain modification example

The natural selection done between miners in order to decide which miner can add its block to the blockchain is a pseudo-random selection. It is the one able to produce a valid block with a valid *nonce* in the shortest time who will add its block to the blockchain. Hence, miners with more CPU power tend to have a bigger chance of being selected since they will probably find the solution sooner. Which makes sense since the more computing power a miner has, the fastest he will scan *nonce* values. So the only way for an attacker or a group of attackers to compromise the Bitcoin network for sure would be to own 51% or more of the total computing power of the network. This event is highly unlikely but non-negligible since a coalition of miners could be possible in order to reach that value. Such coalitions are called pools of miners or mining pools and are more and more popular as the difficulty to mine increases. The problem is that if one of them continues to grow faster than the others it could reach the 51% needed to compromise the network's security. Also in such a situation, the network would not be decentralized anymore.

Until here, we said that when a miner is the first to find a valid *nonce* value for its block, he just append it to the chain. Well, it is slightly more complicated. In fact, when a miner finds the right nonce, he can broadcast the block to all nodes. Nodes will accept the block only if the *nonce* is correct, and if it is, the nodes start working on a new block on basis of the block they just received.

Here is a small recap of the process from the original paper [1], describing the Bitcoin protocol:

1. New transactions are broadcast to all nodes.
2. Each node collects new transactions into a block.
3. Each node works on finding a difficult proof-of-work for its block.
4. When a node finds a proof-of-work, it broadcasts the block to all nodes.
5. Nodes accept the block only if all transactions in it are valid and not already spent.
6. Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

For example, if Charlotte submits a transaction to the network, it means that all the nodes, all the miners of the Bitcoin network, will try to include Charlotte's transaction into their own block. We assume here that Charlotte paid enough fees to ensure every miner is interested in mining her transaction. When the miners completed their block, they will have to find a proof-of-work for it, in order to show everyone the "work has been done". Yet, only the one who will manage to do it the fastest will be allowed to add it to the blockchain. This might not be the most efficient way to do but it is the way security is ensured by the Bitcoin PoW protocol. Coming back to the main steps of the original paper [1] mentioned above, we can see in the fifth step that the nodes check that transactions do not contain inputs already spent. This aims to avoid **the double-spending problem** introduced earlier.
What if two nodes find at the same time their respective nonce? In this case, a race between the two begins. In fact, they will both broadcast their block to the other nodes. Let's call the two broadcast blocks A and B. Some nodes will receive block A first and others block B first, and they will start working on the next block based on the first one they received and accepted. Yet, the nodes do not discard the second block, they save it. Two different branches may be building at the same time. At some point, one branch will become longer than the other one, and all nodes working on the smallest branch will start working on the longer one. Thus discarding the other. This is the **longest chain rule** and was thought to ensure the nodes still agree on what is the content of the blockchain as time goes on.
Because of that, a transaction that at first seemed to have been accepted in a validated block is never for sure embedded in the blockchain. This is why it is recommended to wait for several confirmations before considering a transaction carried out. A rule of thumb is waiting for six confirmations before considering a transaction was irreversibly added to the blockchain.

2.2.4 Transactions representations

Since our work will be based on a graph analysis, we present here the classical ways to represent Bitcoin transactions which is with graphs. Due to the huge size of the Bitcoin blockchain, it is not always easy to detect some properties just by looking at the graph itself. What is mostly done is advanced analysis like the ones done by Ron Dorit and Shamir Adi in "Quantitative analysis of the full bitcoin transaction graph" [12] which is the study of statistical and intrinsic properties of the graphs but also of some sub-graphs containing less information and which are therefore easier to process. We will now detail which graphs are usually used to model the Bitcoin network.

Firstly, we have the **transaction graph** for which nodes are transactions and a directed edge (u, v) between a transaction u and a transaction v represents the fact that u used v as input. We will refer to this graph as $G_1(V_1, E_1)$ where V_1 is the set of nodes and E_1 the set of edges of the transaction graph G_1. The direction of the edges might be reversed, it would only change the way we would interpret the graph. In the first case, an edge can be seen as a way to show what we used to build a new transaction and in the second case, the directions are reversed, the edges would show the direction in which the Bitcoins were sent. In both cases, it is a directed acyclic graph (DAG) because every transaction can happen only once. Indeed, we will never have the UTXOs of a given transaction used as input for this same transaction later. The opposite would mean that the transaction is not done yet since we are waiting for one input, but we use its output already as input. It does not make sense at all so we can conclude that the graph does not contain any cycle.

Then, another widely used graph, very similar to the transaction graph, is the one we will call the **address graph** and we will refer to it as $G_2(V_2, E_2)$ where V_2 is the set of nodes and E_2 the set of edges of the address graph G_2. Here, nodes are addresses and a directed edge from an address a to an address b represents the fact that a transaction happened between the two addresses and more precisely that Bitcoins were sent from the input address a to the output address b. A single transaction will therefore often create more than one edge in the graph. In opposition to the transaction graph, this one can contain cycles. This is because, even if it is not recommended for security reasons, a user can reuse the same address several times, for different transactions. What is usually recommended is to always use new addresses for each transaction, even if this means creating new addresses.

Finally, the last graph we can cite is the **user graph**. This one is a bit different from the first two because it does not represent directly Bitcoin transactions. In fact, it is a graph built from either a transaction graph or an address graph. The important thing is that we have to start from a graph representing directly Bitcoin transactions. From there, we will apply heuristics and clustering algorithms with the aim of identifying Bitcoin users. Even if it does not work perfectly (because otherwise, we would not have anonymity in the network), it can usually identify correctly the highly correlated regions of the graph. Once we have defined the clusters in the original graph, we have our user graph, even if the term user is not really correct, we generally call it so. Interesting analysis can be done on the user graph to detect particular behaviours or active/inactive entities. As mentioned, building a user graph is a step towards the deanonymization of users.

2.2.5 Anonymity in Bitcoin

One of the main reasons Bitcoin gained so much popularity around the world is the privacy it offers to its users. It was often described as fully anonymous since a user's real identity is never linked to anything happening in the blockchain. The only interaction a user has with the blockchain is through its keys and addresses. While in banks any transaction a user makes is linked to his real identity that includes his name, his age and so on.
So, with the growing popularity of cryptocurrencies and most particularly Bitcoin, researchers started to analyze its flaws from an anonymity standpoint and some interesting results appeared. In fact, the researcher started to qualify Bitcoin as pseudo-anonymous because basically a user interacts with the blockchain via addresses, which can be considered pseudonyms. If someone managed to link the real identity of a user with its pseudonyms then all actions done by this user can be tracked and analyzed since the ledger is publicly available.

First of all, it has been quite some time since people identified ways to list with high probability a lot of a user's addresses. As mentioned before, a user can have as many addresses as he needs but there is no reliable way to know whether two addresses belong to the same user or not. Trying to know such things is called deanonymization and aims at linking every user to the set of his addresses. The deanonymization was naturally studied by many people and two well-known heuristics today are **the multi-input heuristic** and **the change heuristic**, as described in the paper "A fistful of bitcoins: characterizing payments among men with no names" [13].

The multi-input heuristic is based on the assumption that a user makes a transaction using only his own coins as inputs. In fact, this heuristic tells us that all the addresses constituting the input of a transaction belong to the same entity, the same user. For example, we could have a transaction of the form $a + b \rightarrow c + d$, where a, b, c and d are coins coming from addresses of the same name. Then, by the multi-input heuristic, we can say that addresses a and b belong to the same entity. We could say for example that this transaction was submitted by Bob, who decided to send Bitcoins coming from its addresses a and b in a single transaction.
The change heuristic takes advantage of the change address that we discussed already. Indeed, it is quite uncommon for a user to have the exact amount of coins he needs to send in his addresses. So what he usually has to do, is to send a bit more than necessary. He will recover the excess amount via a newly created address called the *change address* (or one of the addresses he already owns but this is not safe). Usually, the wallet takes care of the creation of that new *change address*. This way, we can say that the change address belongs to the same entity as the inputs. However, identifying the change address is not as easy as it sounds. One way to do it is by relying on the fact that this address has been created by the wallet and de facto the owner is usually not aware he owns it if do not pay attention to this detail. Thus, never sharing it with anyone in order to receive money. Hence, coins will only arrive at this address thanks to the transaction for which it is the change address. So we can suppose that a *change address* is an output

address that has not more than one input transaction, discarding coinbase transactions for obvious reasons. In the event where multiple outputs only have one input, we cannot be sure which one is the change address. Another property of change addresses is that they should contain fewer coins than any of the inputs. Otherwise, the input containing fewer coins than the change address could just have been removed from the transaction.

Now, thanks to such heuristics, it is possible to build entities that correspond to lists of addresses belonging to the same user, without prior knowledge about these users. We just have to analyze the transactions. The only thing missing is a link to the identities of those users. The users we identify could be individuals but also larger entities, like companies or group of people and so on. Here are possible solutions to map the entities identified to real individuals:

- The network listening technique coming from the paper "Deanonymisation of clients in bitcoin p2p network" [14]:
  This attack is useful when the attacker knows the IP address of the user he wants to spy on. If he does so, this attack will listen to messages that might come from this IP and store the addresses from the transactions in those messages as belonging to that particular user.

- Use of web scrapping:
  This method simply consists in scrapping Reddit forums or any Bitcoin blog in order to find people posting their Bitcoin addresses on the web to receive donations. This way, we can make a parallel between an address and a real identity.

Finally, is it still worth using Bitcoin as "anonymity" can be violated? The answer is yes! While some of the methods above are well-designed, some of them have flaws.

- To distort the multi-input heuristic, there exists a simple solution, called **coinjoin transactions**. It is a transaction where several transactions from different users are combined into one. This way, the multi-input heuristic is wrong. There exist other coins mixing solutions but we will no go further on that matter, we can find them in this paper "Research on anonymization and de-anonymization in the Bitcoin system" [15].

- The change address heuristic can only be used if it is the only output having one input. It is extremely easy to create new addresses and it costs nothing, so a user just has to create transactions with two newly created addresses as output and the heuristic is not interesting anymore.

In addition to that, we also have to take into account who cares about anonymity. To know more in details how this could be analyzed, the paper "Do bitcoin users really care about anonymity? An analysis of the bitcoin transaction graph" [16] is worth mentioning. What we learn is that anonymity is not a big concern for most Bitcoin owners, or at least they assume enough anonymity is already ensured with the current protocol.

Conclusion To conclude this section, we will focus on *the drawbacks* of the Bitcoin protocol. We presented how it works, based on the blockchain principle but it is also important to mention its weaknesses as it is often criticized. Like we just said, perfect anonymity cannot be ensured by the network, so this is already a flaw of the Bitcoin protocol.
We also mentioned the fact that the mining pools tend to gain more and more computing power in the network and we could face a problem of centralization in the next years if they gained too much power.
Then, the consensus protocol based on proof-of-work introduces several drawbacks as well, even if it has some interesting properties to ensure a safe consensus. The first drawback is that the network is very inefficient from an energetic perspective since a lot of miners will work on the same transactions but only one version of it will be kept for the blockchain. Also, with the difficulty of this proof-of-work maintained to guarantee an average of one block produced every ten minutes, the network has a **low latency** and **throughput** and **does not scale with the number of nodes**. Since the size of a block is always about 1.3 MB and a block will only be produced every ten minutes, we have no way to improve the latency and throughput of the current Bitcoin protocol. Even by increasing the number of nodes and therefore the computing power available to mine transactions, it would not change anything, the difficulty would just be adapted to keep the same rate of block production.
All these drawbacks lead to interesting research about how a new scalable blockchain could be designed so that users like Alice, Bob and Charlotte could have their transactions processed faster and with security properties at least as good as the ones proposed by the Bitcoin protocol. This is what is presented in the next section.

2.3 Scalable blockchains : state of the art

The inherent problems of the Bitcoin protocol were soon addressed by a lot of researchers. The main problem being, as stated before, the scalability of the Bitcoin network, as well as its poor latency and throughput. Several families of protocols were therefore designed to counter these weaknesses and propose more advanced solutions.

2.3.1 A single-chain protocol

We will first present a protocol called **Bitcoin-NG** which is a single-blockchain protocol described by Eyal Ittay et al. [17]. Bitcoin-NG is based on the same trust model as Bitcoin and tries to scale optimally, with bandwidth limited only by the capacity of the individual nodes and latency limited only by the propagation time of the network.

The Bitcoin-NG protocol divides time into epochs during which nodes called the leaders are elected to process transactions and add them to blocks. These leaders will produce *key blocks* for leader election and *microblocks* that contain the ledger entries. Once a node produced a valid key block, it is designated as the leader and is allowed to produce microblocks at a maximum deterministic rate. The size of these microblocks is bounded and they do not need proof-of-work so their generation can be done fastly and cheaply. In

the case of a fork, the branch containing the most work is chosen but since microblocks do not contain any proof-of-work, they will not be taken into account when computing the weight of the chain, in the context of the longest chain rule. To still ensure some security, a leader needs to sign the microblocks he generates with his private key that should match the public key in the latest key block of the chain.

This protocol motivates miners to include transactions in their microblocks, to extend the heaviest chain and to extend the longest chain. The first objective is ensured because Bitcoin-NG includes restrictions on the possible fees and rewards of transactions in such a way that it would not be interesting for either a leader or a user to try not to add transactions into microblocks as they should. Then, extending the heaviest chain has similar goals as in the Bitcoin protocol. We prefer to trust the chain containing the most proof-of-work and assume there is a majority of honest nodes in the network in terms of computing power. Lastly, miners will tend to extend the longest chain because doing otherwise would not increase their revenues.

Bitcoin-NG is one of the many attempts to make the single-chain blockchain protocol scale. It addresses the scalability issues of the original Bitcoin protocol as described by Satoshi Nakamoto [1] not just by tuning its parameters since, as stated by Eyal Ittay et al. [17], it is possible to improve Bitcoin's consensus delay and bandwidth by tuning its parameters, but then its performance deteriorates dangerously on all security-related metrics.

Even if we have protocols like Bitcoin-NG, improving the scalability of the Bitcoin network, no matter how smart the protocol is, using a single ledger stays a strong limitation for its scalability if we do not want to compromise the security of the network. One solution for this is to have a more radical change to the original protocol that consists in the introduction of a level of parallelism. This is often referred to as sharding. The next section introduces this new approach that will interest us in the design of a more efficient solution.

2.3.2 Sharding of transactions

With single-chain protocols like Bitcoin-NG, all miners must still validate and replay transactions which is intrinsically unscalable. That is why from now on, we will focus on another promising concept which is **sharding**. The principle is to introduce some parallelism in the network by dividing the nodes into smaller groups of nodes. These sub-groups, called shards will have to validate transactions and build blocks just as the initial Bitcoin network but they will build their own ledger in each shard. The required work for transactions processing is divided among shards and the scalability of the network should therefore improve with the number of shards used. We will now take a look at what can currently be achieved thanks to sharded blockchains.

First, we have systems combining a central authority with a decentralized network. This is the case of **RSCoin**, a cryptocurrency framework described by Danezis George and

Meiklejohn Sarah [18]. RSCoin ensures strong transparency and auditability, as expected to keep the advantages of decentralized cryptocurrencies. Among the benefits of this pseudo-centralization, we can mention the elimination of wasteful hashing as well as the design of a scalable system for avoiding double-spending attacks. However, it was actually not efficient to prevent these double-spending attacks.

A solution to this was proposed with **Elastico**, a distributed agreement protocol for permissionless blockchains designed by Luu Loi et al. [19]. Byzantine adversaries are now allowed to control up to one-fourth of the total computation capacity of the network. The big improvement made with this system is that the transaction rate scales almost linearly with the available computing power used for mining. This means that if we have more computation power in the network, we will obtain a higher number of transaction blocks selected per unit time.

Since Elastico still contained some security vulnerability, Kokoris-Kogias Eleftherios et al. [20] designed **OmniLedger**, which can be seen as an improved version of Elastico. OmniLedger is a novel scale-out distributed ledger that preserves long term security and correctness under permissionless operation. It securely scales out to offer high throughput, scaling linearly in the number of active validators, and latency of seconds while preserving full decentralization and protecting against a Byzantine adversary.

Finally, the last step of this evolution if we can call it like that is **RapidChain**. We will now present this sharded blockchain in details since we would like to base our own work on it.

RapidChain

RapidChain [21] is a project aiming to improve the throughput and latency of existing blockchains like Bitcoin by sharding transactions into what they call in the paper, **committees**. Committees, also called shards, are groups of nodes that endorse the same role as miners do in the classical Bitcoin protocol. The only difference here is that each committee receives **randomly** different transactions they are responsible to process. All the work is therefore parallelized among the committees, the verification of a transaction's validity, the consensus among the nodes as well as the storage of the blockchain and additional information. This way, "the throughput of the system can scale proportionally to the number of committees unlike the standard Nakamoto consensus" as stated by Rapidchain [21], where the Nakamoto consensus is nothing else than the PoW consensus described by Nakamoto in the original bitcoin paper [1]. This property could satisfy the constant growth of blockchain-based applications.
The creators of RapidChain state that among all existing blockchain sharding mechanisms, many overlook several aspects and do not exploit all the advantages of sharding. They claim they are the first sharding-based blockchain that can achieve sublinear ($o(n)$) communication overhead per transaction while previous ones required $\Omega(n)$, with n being the number of participants (or nodes) in the network. Furthermore, they can ensure security in the network while a third of the network's nodes are dishonest, unlike previous works that can only tolerate as much as 12.5% to 25% corrupted nodes. They also provide a

reconfiguration method to guarantee protection against the slowly-adaptive Byzantine adversary and even allow new nodes to join in without a long interruption of the services. They inspired themselves from past works in order to accomplish an efficient cross-shard verification system. Additionally, they largely reduced the latency and overhead of consensus. Before going further, let's introduce some notation to ease the description of the model.

Notations

- n: Number of nodes in the network.
- t: Total number of corrupted nodes.
- m: A committee (size) if less than half of its nodes are corrupted.
- f: Percentage of corrupted nodes in a committee.

Protocol details RapidChain consists of three main phases, it starts with a bootstrapping, then the consensus and finally a reconfiguration step. Once the bootstrapping is done, it will never be done again, only the consensus and reconfiguration are repeated again and again during an endless succession of iterations (or epochs). We will now detail these phases more in detail to ensure a good comprehension of RapidChain.

1. Decentralized boostrapping

   This phase is an initialization phase and is only executed once, at the launch of RapidChain. The objective is to form committees that have no more than half of dishonest nodes with high probability. For that, we need to have a group of nodes for which we are certain that a majority of them are honest, this group is called the **root group**. This root group must be honest because he will have the responsibility to form the **Reference shard**, as we will explain in a moment.
   Once the nodes launched the initialization procedure, as illustrated with the START arrow in Fig. 2.6, each one of the nodes will build a **sampler graph**. A sampler graph is a random bipartite graph $G(L, R)$, where the vertices in L are all the nodes in the network and the vertices in R are groups of nodes. The edges between the two partitions represent the belonging of a node to a group of nodes and are selected independently and uniformly at random without replacement. The number of groups is given by $\frac{|L|}{d_R}$, with $d_R = \sqrt{|L|}$ being the degree of each group of R.

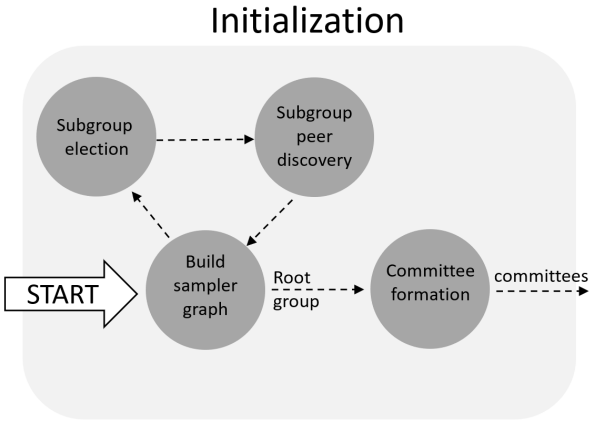


Figure 2.6: Decentralised bootstrapping

Then, we enter the second stage of the initialization process, where each newly formed group has to elect e nodes. This step is called the **subgroup election**. To do that, each group has to get a random string s via the DRG (for Distributed Random Generation) protocol. With s, each node computes $h = H(s||ID)$ with ID being a node identification ID and H being a hash function modelled as a random oracle. If a node satisfies the following condition $h < 2^{256-e}$, then it can notify the rest of the group about it. Among the nodes satisfying the condition, the group only keeps the e nodes with the smallest h value, and signs the tuple (ID, s) of the e nodes, to finally gossip those signatures in the group.

Next, the groups have to communicate with each other in order to learn the elected nodes of every other groups. This is the **subgroup peer discovery**. So, the elected nodes will gossip their ID and a proof consisting of $\frac{d_R}{2}$ different signatures on (ID, s). But, if groups receive more than e elected nodes from a single group they consider it as corrupted and do not take the corresponding group into account.

Afterwards, we repeat the process described above by building a sampler graph where the vertices in L are now the elected nodes from the previous step. After several iterations the sampler graph contains only one group in R, which means that we found the root group. This root group is in charge of forming the Reference shard (that we will call C_R). Finally, this honest reference committee will **form different committees** of the network by dividing randomly the set of nodes containing a majority of honest nodes.

2. Main phase

This brings us to the main phase where transactions are put into blocks and blocks are appended to the chain.

First of all, each committee starts by choosing randomly a leader which will be in charge of putting transactions together into a block. The transactions are divided

among the shard randomly.
Once the block is built, the leader gossips the block to the committee's members via a consensus protocol and waits for it to be accepted. If the block is accepted, then the objective of the leader is accomplished. If the block is pending, this means not enough nodes accepted the block and it cannot be added to the chain. In both cases, it is time to start a new iteration by selecting a new leader.
Furthermore, a pending block can be re-gossiped by another leader later, this was implemented by the authors of RapidChain to avoid a halving of the throughput and is represented on Fig. 2.7 by the optional path.

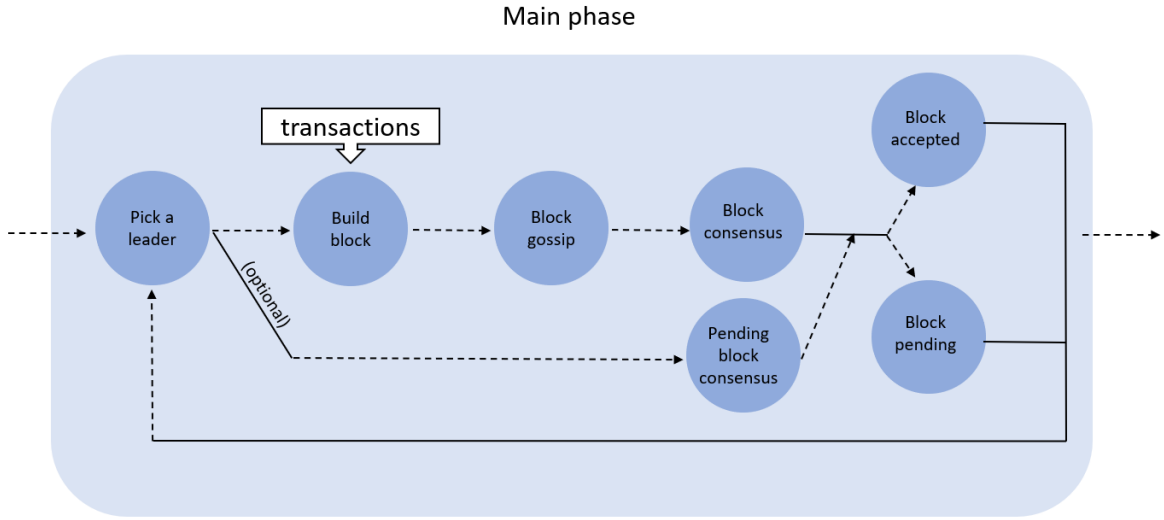


Figure 2.7: Transactions validations

Let's dive deeper into the way a consensus is reached. It consists of four synchronous steps and all the messages sent during those steps are tuples with first the header of the block to append H_i and then a tag. The header is the root of the Merkle tree and the iteration number.
The leader starts by gossiping to all nodes of the shard the message $(H_i, "propose")$. Then, the nodes that received the header from the leader, gossip it again to everyone in the committee but this time with the tag set to "*echo*". Next, if someone received several versions of the header for the same iteration, then the leader is catalogued suspicious and the corresponding node gossips to everyone the header with the tag set to "*pending*" and the root of the Merkle tree is replaced by null. If at some point a node received $mf + 1$ or more echoes, then it will gossip the header again but this time with the tag set to "*accept*" in addition to a proof consisting of all the echoes the node received, this proof is called tamper-proof.

On the scheme of Fig. 2.7, there is an optional path, telling that a leader can re-propose the pending blocks headers in addition to proposing a brand new block. This way, the throughput can be improved. So, how exactly does it work to re-propose a header? The authors call that **pipelining**. It states that the consensus can take several iterations to complete. So each node stores the number of echoes a pending

header has. When a pending header receives a sufficient amount of echoes, it is accepted. The header can be accepted several iterations after it was first proposed.

3. Reconfiguration

This phase is done at the end of every epoch and is used to prevent dishonest nodes from gaining a majority by corrupting other node(s). A good approach to prevent that is to regenerate all the committees from scratch at every epoch, however, this solution is time-consuming. So Rapidchain proposes the **Bounded Cuckoo Rule**. When a new epoch begins and a new node has to be added to the network, the Reference committee C_R partitions the other committees into two distinct groups, the set A of active committees and the set I of inactive committees. Then, C_R chooses one of the active committees, let's call it C_a, and add the new arriving node in it. This last step is done for every new node entering the network. In exchange, C_R displaces several nodes from the committees in A and places them into committees of I at random. That process ensures that the committees stay balanced while keeping a majority of honest nodes.

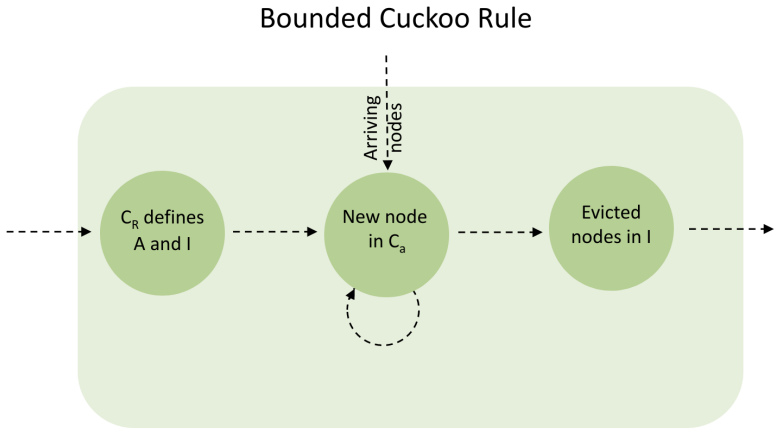


Figure 2.8: Reconfiguration

In RapidChain, new nodes entering a committee do not have to download the whole blockchain of the committee but only need to download the corresponding set of UTXOs.

Putting everything together To sum up, RapidChain will start with a decentralized bootstrapping as initialization to build the committees, then each committee will start validating transactions into blocks during the time of an epoch. At the end of an epoch, a reconfiguration of the committees is done, so that another epoch can start. This is summarized in Fig. 2.9. Then, the newly formed committees start validating transactions again, and so on.

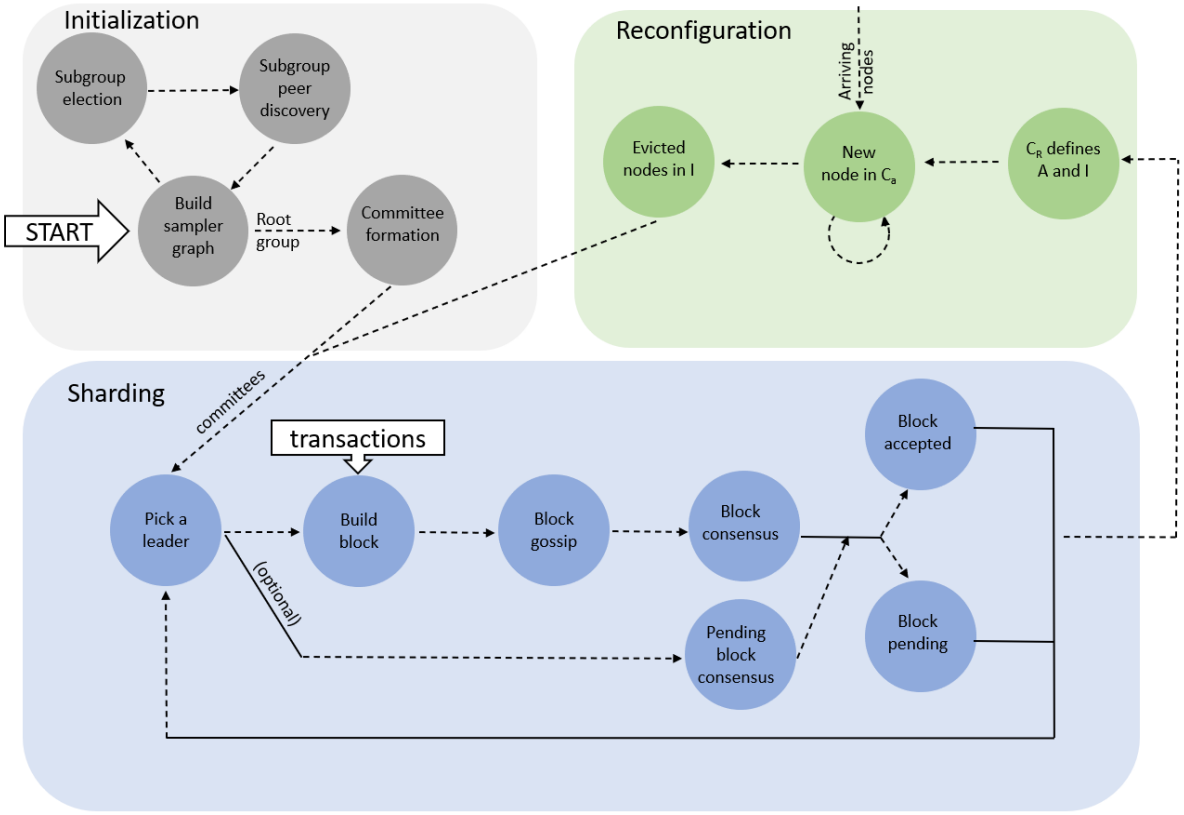


Figure 2.9: RapidChain global overview

Chainspace

Chainspace [22] is another attempt to improve the transaction throughput and reduce the latency, by dividing the state and execution of transactions into shards randomly. Each shard is made of several nodes which have the role of keeping track of the object's states, their validity and storing the history of committed transactions. Chainspace has Byzantine Fault Tolerance (BFT) and introduces a novel way of reaching consensus and satisfying consistency in the network called **S-Bac**. It is not an application-specific distributed ledger since it supports smart contracts, it can be used for many different purposes. A smart contract is first of all a contract where the middle man is eliminated. The contract is effectuated once both parties satisfy their end of the contract. We present this sharded blockchain system because we used an implementation of it for the evaluation of DAVE.

Globally In Chainspace, there are objects which are atoms that hold states, much like UTXOs. They are either active, inactive or locked. Active means that the object can still be used as input in a contract, unlike inactive ones. Locked means that the object is used in a contract that is being processed. Depending on whether the contract is accepted or not, the state of the object will become either "Active" or "Inactive". Usually, we find them in **contracts**, a contract can be used to make a transaction, the user only has to specify the inputs and output needed for the contracts and several other information, such as procedures. Procedures define how inputs are processed to produces certain outputs. So, a transaction is made primarily of inputs, outputs and procedures, and when all its

procedures are executed, the inputs become inactive and the outputs become active and the transaction is terminated. For a transaction to be valid, all its inputs and procedures need to be active, also transactions are **atomic** which means that all the procedures of a transaction complete or none of them do. If only one procedure of the transaction fails to be valid, then every other procedure of the transaction do not execute. Furthermore, transactions are **consistent** in the case where multiple transactions use the same input objects, only one will be committed, the others will be aborted.
The system is secure as long as there are no more than $\boldsymbol{f}$ faulty nodes per shard and each shard has a total of $\mathbf{3}\boldsymbol{f}\mathbf{+1}$ **nodes** which is a pretty common characteristic in the world of distributed sharded blockchains. All the nodes in the networks are distributed in several shards. Within a shard, the nodes have to work hand in hand to manage the state of objects, record transactions involving the objects and making sure that objects are used at most in one transaction as an input. The transactions are placed in shards in **a random manner**. Besides, all the nodes in a shard have got to agree on a transaction getting committed thus consuming objects and creating new ones. This process is called the intra-shard consensus and the one developed in Chainspace is called **S-Bac**.

Transactions intra-shard consensus (S-Bac) S-BAC is the acronym of Sharded Byzantine Atomic Commit and is based on two existing principles:

- Byzantine agreement
- Atomic commit

We will explain the protocol using an example transaction in Fig. 2.10. As a reminder, transactions can be made of input objects located in different shards.

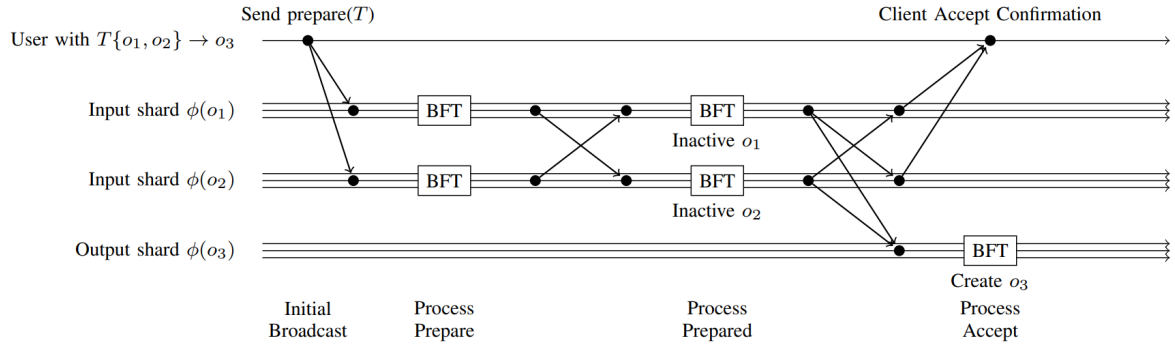


Figure 2.10: Illustration of the S-BAC protocol from Chainspace paper [22].

1. **Transaction T**: o1, o2 $\longrightarrow$ o3. The inputs o1 and o2 are in the state "Active" while o3 is not yet created.
2. **Initial broadcast**: The very first step is done by the user. When he sends the message 'prepare(T)' to one honest node in each shard concerned. As there might be f dishonest nodes per shard, the user will then send the message to $f + 1$ nodes in each shard. So that at least one honest node receives it. When we say "concerned" shards, we mean the shards that contained one or more of the input objects needed in T.

3. **Process prepare**: Then, the nodes having received the message start the Byzantine consensus protocol by sending the message to other nodes in the shard. The nodes receiving the message will all have the same decision since they all have the same history stored. However, there could be malicious nodes sending false decisions. Thus the shard sends to other concerned shards the decision which is either "prepared(T, commit)" or "prepared(T, abort)". In addition to this message are added at least $f + 1$ signatures in order to prove the validity of the decision. At this point, all the objects used by the transactions enter the "locked" state.

4. **Process prepared**: If a shard receives one or more abort messages from the other shard(s), then the shard will send "accept(T, abort)" to the other(s). But, if the shard receives as many commits as there are shards involved, then the shard will send the message "accept(T, commit)" to the other shard(s) involved. The shards also send the decision either to commit or to abort to the client so that he is informed that its transaction has been accepted or not and to the shard(s) concerned by the output(s) of the transaction T.

5. **Process Accept**: If the decision is to commit then the inputs o1 and o2 are moved to the state "Inactive" and the shard involved with the output o3 creates it. But if the decision is to abort, then the inputs are moved back to the state "Active" and the transaction T is discarded.

Optchain

Until now, the sharded blockchains presented place transactions into shards either in a random manner or based on the transaction's hash. These are probably the most straightforward and simple solutions since it is fast and it ensures the security of the network at the same time. But these solutions also come with non-negligible drawbacks. Taking the placement decisions randomly implies that a huge majority of transactions will be cross-shard. A **cross-shard transaction** being a transaction for which the set of its inputs and outputs are not altogether in the same shard. Unfortunately, this kind of transaction is a bottleneck for the performances of the network. They imply inter-shard communication (communication between different shards) which is always slower than intra-shard communication.

Optchain [23] is one of the first concrete approaches towards an efficient sharding method. The Optchain strategy, executed at the user side, tries to place transactions into the shards that will minimize the future number of cross-shard transactions while minimizing the load balance between the different shards.
More concretely, Optchain treats transactions as a stream of nodes in an online graph and uses a lightweight and on-the-fly transaction placement method to group both related and soon-related transactions into the same shards. This leads to an improvement in terms of cross-shard communication compared to the other sharding algorithms. What is also really interesting in this work is that, in addition to decreasing the number of cross-shard communication, OptChain maintains a temporal balance among shards to guarantee high parallelism.

To achieve such results, Optchain's decision for transactions placement is based on the analysis of a transaction graph from which two scores are built, a **Transaction-to-Shard** score (noted T2S) and a **Latency-to-Shard** score (noted L2S). These two scores form together what is called the **Temporal Fitness** score. A transaction will then be placed in the shard for which it has the highest Temporal Fitness score. As explained by Nguyen Lan N. et al. [23], "*The T2S scores between a transaction t and a shard S_i, measures the probability that a random walk from a node/tx t ends up at some node in S_i, i.e., hence, how likely the transaction should be placed into the shard without causing further cross-shard txs. The L2S score estimates the processing delay when placing the transaction to a shard. Importantly, the protocol to estimate the two scores is lightweight and is executed at the user's side.*" Let's take a look at how the T2S score is computed.

In order to shard a transaction t, we have to compute its T2S score for each of the k shards of the network, to be able to select the one fitting the best with it. Naturally, a T2S score will then be stored in a vector of k dimensions $p(t)$. The fitness between a shard i and our transaction t can be found in $p(t)[i]$, that is the i^{th} entry of the vector $p(t)$. This vector $p(t)$ is computed as follows:

$$p(t) = \alpha s(t) + (1 - \alpha) \sum_{v \in N_{in}(t)} \frac{p(v)}{|N_{out}(v)|} \tag{2.1}$$

Where α is a constant between 0 and 1, $s(t) = \{0\}^k$ if t is a new arriving transaction while if t is a known (and therefore already sharded) transaction, then $s(t) = \frac{1}{|S(t)|}$ at the entry corresponding to shard $S(t)$ and 0 elsewhere. The notation $S(t)$ is used to reference the shard in which a transaction t is placed and S_i for the i^{th} shard among the k present in the network. $N_{in}(t)$ (resp. $N_{out}(t)$) is the set of input (resp. output) transactions of t. In equation 2.1 we notice the T2S score of a transaction is based on all the history of its input transactions and also take into account the size of the shards.

2.4 Problem statement

The best approach in the design of a scalable model is to use the concept of sharding to parallelize the work and construct independent ledgers rather than a single one. RapidChain is one of the best sharded blockchain we can find today that improves both throughput and latency of the network while ensuring its security. Optchain is also a very interesting approach because it addresses the problem of inter-shard transactions. We state in this section what challenges should face a sharding method aiming to improve the existing ones.

The decision to attribute a transaction to a particular shard is an important step in every sharding protocol. In Optchain, this is done at the user side but since we do not want heavy clients computation, we should rather do this assignment at the miner side. In addition to that, the placement decisions should not be too costly to make at the miners' side because we do not want it to impact the network performances.

We also have to ensure security in the network as well as verifiable placement decisions. We have to ensure that no shard will ever be overloaded on purpose or that no validator will ever be able to choose which transaction it validates. Indeed, an overloaded shard would compromise the latency of the network and a validator being able to choose which transaction it verifies, could voluntarily validate incorrect transactions and this could lead to problems such as double-spending. We also need the placement decisions to be verifiable because if we let miners decide on placement arbitrarily they can decide to place outputs in specific shards to be able to collect more fees later.

The solution to this is to have a verifiable deterministic sharding of transactions that will prevent a shard from being overloaded. Optchain almost proposes this. Almost because it is not possible to verify the placement of a transaction with Optchain. In fact, in order to verify whether a transaction has been placed in the correct shard or not, we have to recompute the scores at the exact moment the transaction was sharded. But there is no way to know this moment because there is no global order of the validation of the transactions. So Optchain does not meet the requirements. What we want is a deterministic sharding that would enable anyone to verify efficiently whether a transaction was validated by the shard it was supposed to or not.

One of the main problems we also want to address is the inter-shard transactions. We would like to minimize them as it is done in Optchain. To illustrate how they can impact the performances of a network, we simulated in Fig. 2.11 two situations on Byzcuit. Which is a sharded blockchain based on the principles of Chainspace. In one case we consider only cross-shard transactions (this is the *fullcross* line on the graphs) and in the second case, we consider only intra-shard transactions (this is the *nocross* line on the graphs). The results show that a network having to process only intra-shard transactions will be way more efficient than a system having to process inter-shard transactions.

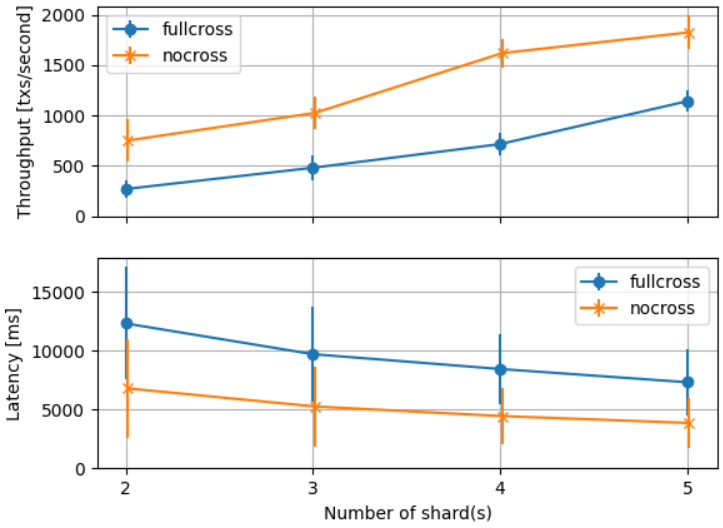
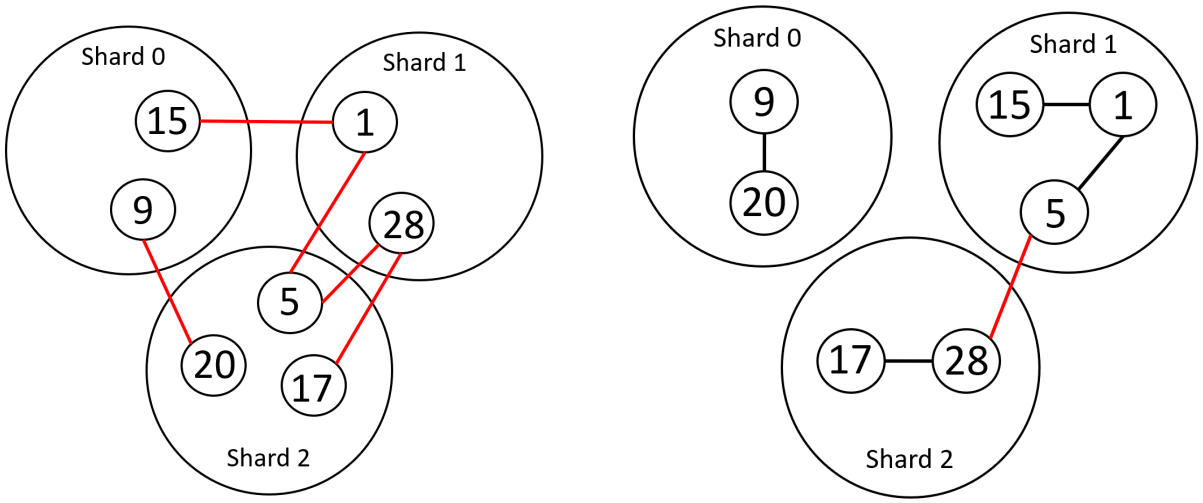


Figure 2.11: **Evolution of the throughput and latency in function of the number of shard(s) used**.

Naturally, having absolutely no cross-shard transaction is impossible in practice but the idea is that the fewer they are, the best will be the performances achieved by the network. The high percentage of cross-shard transactions compromises the good scalability of existing hash-based/random sharded blockchains. The conclusion of this is that this random solution is safe but inefficient.

We illustrate with the small example of Fig. 2.12 what would be an efficient transaction placement compared to the hash-based sharding used in some existing protocols. A small set of artificially created transactions are placed in three shards. The edges represent an input-output link between transactions and the red edges are the ones crossing shards. In Fig. 2.12a, the transactions were placed based on their number. We took the modulo of the transaction's number to choose which shard should validate it. This represents what kind of results can be achieved with hash-based sharding. On the other hand, Fig. 2.12b shows what could have been done if we can decide the clever placement of each transaction. This second solution leads to a lot fewer cross-shard transactions while respecting the same load balance between shards than the hash-based sharding.



(a) Hash-based sharding

(b) Clever sharding

Figure 2.12: Hash-based versus optimal transaction sharding

As mentioned, an approach to this problem was done with Optchain. The first objective of this sharding paradigm is to minimize the number of cross-shard transactions in the network. This can result in an almost twice faster confirmation time and throughput, which would be great for the overall scalability of the network. This approach is very promising and suggests it should be the next generation of sharding algorithms.

We end up with an interesting problem consisting in the design of an efficient sharding algorithm that handles the following main challenges:

- The decisions should minimize the percentage of cross-shard transactions in the network.
- The decisions should maintain an acceptable load balance between the shards.
- It should be possible to take the placement decisions at the miner side.
- The way we take the placement decisions should be a fast, deterministic and verifiable.
- It should be possible to verify transaction placements in a bounded amount of time and it should require a bounded amount of memory.
- The sharding algorithm itself should need only a limited amount of memory to work efficiently.
- The sharding algorithm should be easily introduced in a real network.

Our goal in the next chapter will be to describe DAVE, the sharding algorithm we designed to satisfy the above constraints. We will not try to optimize the way DAVE is implemented in this thesis but rather its intrinsic efficiency. The main challenge for us is to satisfy at the same time the objectives of load balance and cross-shard minimization. Indeed these two are in competition since being perfectly balanced between the shards would introduce a lot of cross-shard transactions while placing all the transactions in a single shard would lead to no cross-shard transaction at all with a terrible load-balance. In addition to that, we will have to respect the time and memory complexity constraints to build a verifiable deterministic sharding easily introduced in a real network.

2.4.1 Mathematical formulation of the objectives

Minimizing the number of cross-shard transactions while keeping an acceptable load balance among shards is a **bi-objective optimization problem** under constraints for which we can give a mathematical formulation. The first objective consists in minimizing the ratio of cross-shard transactions while the second objective is minimizing a score representing the unbalance between shards through time. In order to do so, we defined in eq.2.2 a variable N_{inter} for the number of cross-shard (or inter-shards) transactions and a variable N_{intra} for the number of not cross-shard (or intra-shard) transactions. We will explain what each variable below represent.

$$
\begin{aligned}
t_{uv} &= \begin{cases} 1 \text{ if } (u,v) \in E_1 \text{ such that } u \in S_i,\ v \in S_j \text{ and } i \neq j \\ 0 \text{ if } (u,v) \in E_1 \text{ such that } u \in S_i,\ v \in S_j \text{ and } i = j \end{cases} \\
count(u) &= \sum_{v \in N_{in}(u)} t_{uv} \\
cross(u) &= \begin{cases} 1 \text{ if } count(u) > 0 \\ 0 \text{ otherwise} \end{cases} \\
\mathrm{N_{inter}} &= \sum_{u \in H} cross(u) \\
\mathrm{N_{intra}} &= \sum_{u \in H} 1 - cross(u)
\end{aligned}
\tag{2.2}
$$

We first defined t_{uv}, equal to 1 if the transaction u used the transaction v as input and v was in another shard than u, and equal to 0 otherwise. As mentioned previously, E_1 is the set of edges of the transaction graph G_1. Then we use t_{uv} to define *count* as a function taking a transaction u as argument and returning the number of transactions used as input by u and being in a distinct shard than u. Remember, when we say a transaction b uses another transaction a as input, it means that transaction a has 1 or several inputs that are outputs in transaction b. In the expression of $count(u)$, N_{in} is a function taking a transaction u as argument and returning the set of input transactions of u. The last step before defining $\mathrm{N_{inter}}$ and $\mathrm{N_{intra}}$ was to create a function named *cross*, that takes a transaction u as argument and returns 1 if u has a cross-shard communication with one of its input transactions and 0 otherwise.
Finally, we can express $\mathrm{N_{inter}}$ and $\mathrm{N_{intra}}$ by using the previous variables on the whole set of processed transactions H. Our two objectives are then expressed as follows.

1. $\min \frac{\mathrm{N_{inter}}}{\mathrm{N_{inter}}+\mathrm{N_{intra}}}$
2. $\min \frac{1}{\mathrm{K}} \sum_{i=1}^{\mathrm{k}} |n_i - \tilde{n}|$

The first objective aims to reduce the ratio of cross-shard transactions as much as possible while the second one focuses on the load balance. Here we suppose we still work with K shards and n_i is the size of shard S_i while $\tilde{n}$ is the average size of the shards. By size, we mean the total number of transactions ever processed by a given shard. This second objective is not a mean square error on the size of the shards because we thought it would be more interesting to keep interpretable objectives. Written like this, the second objective can be interpreted as the average difference between a shard's size and the mean of all shards' sizes. This objective will in fact only help us to understand the average load of the shards but as explained in chapter 4, we minimized the maximum loads of the shards through time while choosing the parameters of our model.
To avoid trivial solutions, we should also add a constraint on the number K of shards to use. We will therefore add the constraint that K should be equal to a fixed constant strictly greater than one.

The load balance score as defined above is relevant as far as the total number of transactions processed stays small but it will not be the case anymore if this total number of transactions processed grows too much. In fact, with the second objective defined as above, we can ensure a good average load balance but not an acceptable load balance through time for every shard. What we can do to solve this problem is that we do not consider n_i as the total number of transactions ever processed by the shard S_i anymore but rather as the total number of transactions ever processed by the shard S_i among the last x transactions. Where x is defined as constant. Similarly, $\tilde{n}$ would then be defined as the mean of the new n_i's. Typically we fixed x to 50 times the number of shards used in our experiments, so 50 times K. The second objective will then be computed with a set T of 50 times K transactions. This still gives an indication of the average load of the shards but on a shorter amount of time. A more formal definition of this new load balance score will be given in chapter 4.
We will present in the next chapter how we managed to find an efficient solution to solve this bi-objective optimization problem and how it could be integrated into a real network.

Chapter 3

Solution

In this chapter, we present DAVE, our solution to the problem stated. While explaining the different parts of our solution, we detail how it satisfies the challenges of the initial problem.
We first describe how we took inspiration from Optchain to design scores enabling DAVE to place a transaction in the best shard possible. In this first part, we also introduce the heuristics used to enrich our address graph. Once this is done, we explain how this sharding algorithm could be introduced in a real network such as RapidChain. This part will mostly mention the changes that would be necessary to do so. We also show in a dedicated section that the placement decisions taken are verifiable and how this can be done efficiently. Finally, to conclude this chapter we mention how we managed to implement the sharding algorithm in order to obtain interesting results which will be presented and discussed in another chapter.

3.1 Clever sharding of transactions

3.1.1 An inspiration from Optchain

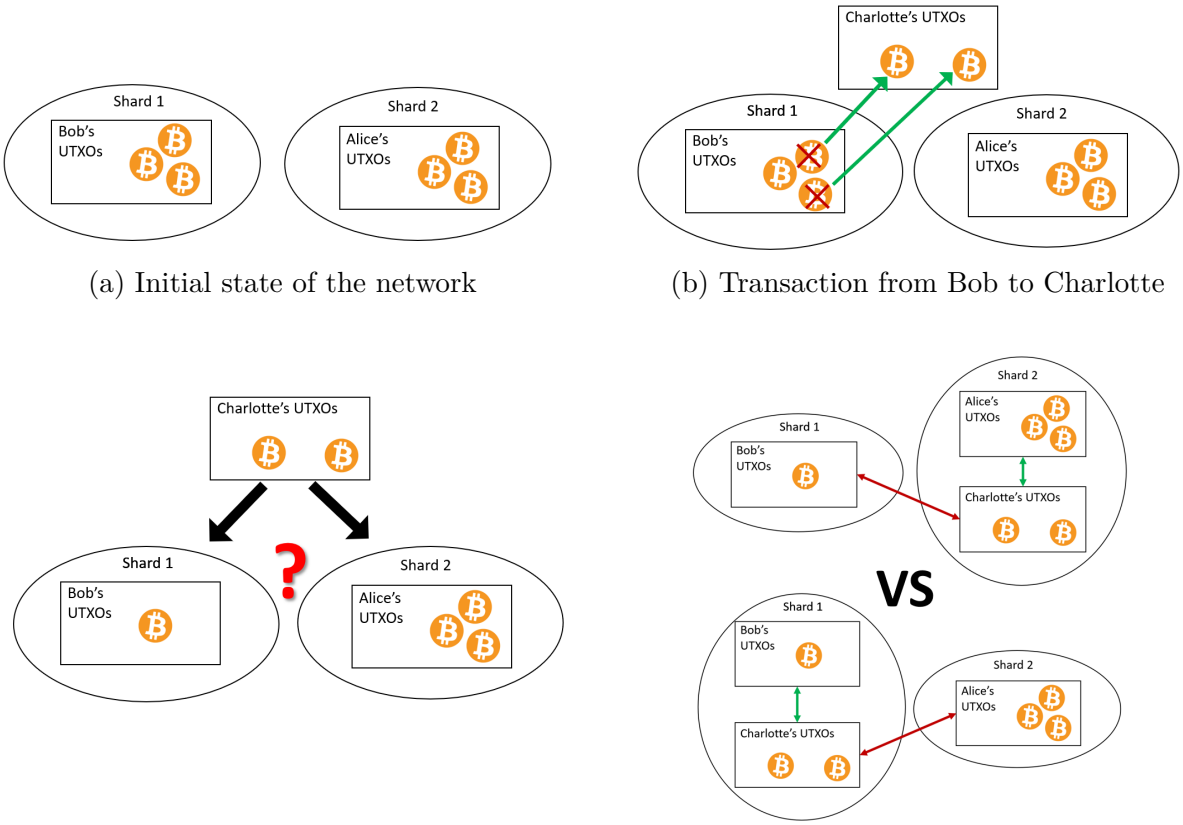
The techniques used in Optchain to reduce the number of cross-shard transactions and keep the shards balanced at the same time were a source of inspiration for DAVE. However, some changes were needed since, with Optchain, it is not possible to verify transactions placements efficiently. Furthermore, the decision should not be taken at the user side for safety reasons. DAVE will use addresses rather than transactions. The usage of addresses should indeed improve the sharding performances thanks to the addresses-based heuristics presented in the next section.
The T2S score of Optchain is the most useful for DAVE because it gives more attention to the number of cross-shard communication between well-balanced shards while the L2S score was directly related to real network applications and their latency.

3.1.2 Heuristics

DAVE should minimize the number of cross-shard transactions while keeping the shards balanced. These two objectives are clearly in opposition since ensuring a perfect load balancing between shards most of the time implies introducing new cross-shard trans-

actions. We present here heuristics based on addresses interactions aiming to help the reduction of cross-shard transactions. We will explain later how DAVE manages to ensure an acceptable load balance among shards.

Let's look at the small example illustrated in Fig. 3.1 to understand the problem. In this example, we have a network of two shards and its users Alice and Bob have respectively UTXOs in shard 1 and shard 2 as depicted in Fig. 3.1a. Bob wants to give some Bitcoins to a new user, Charlotte which does not have any cryptocurrency yet. Charlotte has to create a new address on which Bob can send the Bitcoins. This is the transaction represented in Fig. 3.1b. Once the address is created and the Bitcoin transaction is submitted to the network, the new UTXOs should be placed in one of the shards in order to avoid cross-shard transactions. This decision step is represented in Fig. 3.1c. Finally, in Fig. 3.1d, we have the possible final network configurations. Charlotte's new UTXOs can be placed in shard 1 since Bob sends it from shard 1. By doing so, cross-shard communication is avoided. The problem is that if later Charlotte wants to send UTXOs to Alice, it will introduce a cross-shard transaction. In this case, it would have been the same to place Charlotte's UTXOs in shard 2. In the future, if Alice and Charlotte appear to have a lot of such interactions and no interaction with Bob anymore, then it would even have been better to place Charlotte's UTXOs in shard 2 from the beginning, this is the second solution. In Fig. 3.1d, the red arrows represent cross-shard communication implying Charlotte that could happen in the future. In the same logic, green arrows represent Charlotte's possible future intra-shard communications.



(a) Initial state of the network

(b) Transaction from Bob to Charlotte

(c) Sharding step

(d) Possible final configurations

Figure 3.1: Problematic of cross-shard transactions

With an UTXO model, every input was first an output (except for coinbase transactions). So in the long term, it is reasonable to assume every transaction (or at least the wide majority of them) will first be output and then will be used as input, possibly with other transactions. To avoid cross-shard transactions DAVE should try to minimize the number of transactions placed in a different shard than the one containing their inputs. But for this, the inputs also need to already be placed in a clever way when new output transactions arrive. For example, if all the inputs are not in the same shard, it will be impossible to not introduce a cross-shard transaction. DAVE has to place transactions in a way that minimizes the number of cross-shard transactions in the long term. Predicting the future is complex, naturally, but heuristics can help in the prediction of the most probable events. The goal is here to identify in which shard a transaction will have the most activity in the future.

Addresses contain more precise information about the relations between transactions than transactions themselves. Therefore, applying heuristics on addresses will lead to more accurate predictions on the future activity in the network. The behaviour of users should be analyzed rather than their real identities so DAVE will focus on the addresses usages in transactions rather than on which addresses belong to the same user. Without any specific algorithm or deanonymization technique and without knowing the future, it is still possible to study the usage and interactions of addresses in already validated transactions. The Bitcoin blockchain contains all the interactions done between addresses of every

transaction ever validated. The key idea is that DAVE can then base its predictions on the past. The heuristics retrieve information about addresses from the blockchain and make the assumption that addresses with interactions in the past have a higher probability to interact again in the future.

The Optchain placement decisions were based on the information coming from the transaction graph. Similarly, our placement decisions will be taken using information from the address graph except that heuristics will help to improve the quality of the address graph analysis by augmenting it. We inspired our heuristics from the ones used by Nick Jonas David in his Master's Thesis [24] and by D. Ermilov et al. in "Automatic Bitcoin Address Clustering" [25]. The **Interactions heuristic** and the **Cooperation heuristic** we designed should enable DAVE to augment the address graph efficiently. Two nodes connected in this graph should have a higher chance to appear in the same future transaction. Let's now explain how these two heuristics work and illustrate them with small examples.

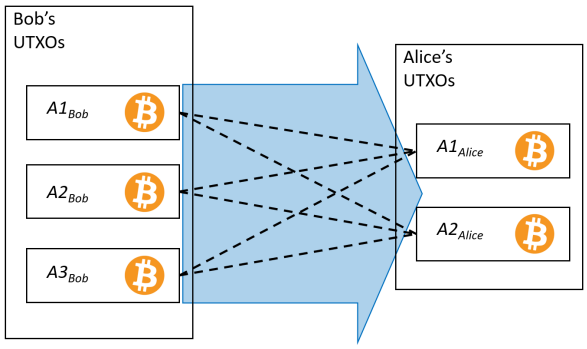
Interactions heuristic

> The Interactions heuristic describes mainly how we usually build an address graph. It adds an edge between each pair of input-output addresses of a transaction. Linking the input addresses to the outputs addresses already represents an interaction between these addresses since coins were sent from the first ones to the others. Concerning predictions about future interactions between addresses, this heuristic helps because two addresses exchanging Bitcoins will probably do it again in the future or they might also be used together as inputs (or outputs) in a new transaction.

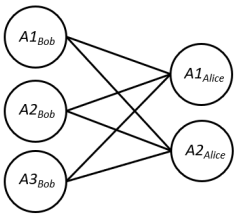
For example, Bob could send his coins stored in its Bitcoin addresses $A1_{Bob}$, $A2_{Bob}$ and $A3_{Bob}$ to Alice, on the addresses $A1_{Alice}$ and $A2_{Alice}$. In this case, thanks to the Interactions heuristic, we would add the following edges to our address graph:

$$\begin{array}{lll} A1_{Bob} \rightarrow A1_{Alice} & A2_{Bob} \rightarrow A1_{Alice} & A3_{Bob} \rightarrow A1_{Alice} \\ A1_{Bob} \rightarrow A2_{Alice} & A2_{Bob} \rightarrow A2_{Alice} & A3_{Bob} \rightarrow A2_{Alice} \end{array}$$

This is illustrated in Fig. 3.2. These added edges represent the fact that in the future Bob and Alice could exchange Bitcoins with these addresses again. For instance, if Bob wants to give more Bitcoins to Alice using the same addresses or if Alice wants to send Bitcoins back to Bob or even if they want to use their addresses together as inputs for a new transaction. In all these cases, it will be useful to already know that there is a link between the addresses of Bob and Alice.
A change address is used to receive the exceeding amount of cryptocurrencies in a transaction, it will hence appear as one of the outputs and belong to the same user as the input addresses. The heuristic would link input addresses to output addresses as usual and in this case, it would even link two addresses belonging to the same user.



(a) Transaction from Bob's to Alice's addresses



(b) State of the address graph after the Interactions heuristic was used for the Bob to Alice transaction

Figure 3.2: Illustration of the Interactions heuristic

One can directly see that using our Interactions heuristic is usually the way an address graph is built. This heuristic will enable us to build the "standard edges" of our address graph. However this is not enough, we want to augment the classical address graph to represent more information about the addresses and the probability of their interactions in the future, that is why we designed the second heuristic.

Cooperation heuristic

> This heuristic is based on the fact that if some addresses were used together as inputs (or together as outputs) in the same transaction, then they have a higher probability to be used again together in a transaction later. We called this link between them cooperation. In order to represent this in the address graph, we add an edge between each pair of input addresses of each transaction and we do the same for each pair of output addresses.

The justification of this heuristic is quite intuitive. In the example used for the Interactions heuristic, Bob sent coins to Alice by using several of its addresses as input at the same time, in one single transaction. So all the input addresses belong to the same user, Bob, and are directly related. The output addresses also belong to the same user, Alice. Therefore it is logical to link Bob's addresses together and Alice's addresses together. The use of the Cooperation heuristic, in addition to the Interactions heuristic, is illustrated for this example in Fig. 3.3a.

The strength of this heuristic is that, even if all the input addresses or all the output addresses belong to several users, it can be justified to link them. Indeed, if some users decide to execute a transaction together, it is not absurd to guess they are related and that they might do it again in the future or be implied together again in a future transaction. So if Bob, Alice and Charlotte receive Bitcoins in the same transaction or if they sent Bitcoins together in a transaction, it will assume their addresses are related. In both of these situations, their addresses would be linked together in the address graph by the Cooperation heuristic as represented in Fig. 3.3b. We would assume there was cooperation between them.

(a) Address graph with green edges added thanks to the Interactions heuristic and purple edges added with the Cooperation heuristic

(b) Address graph with edges added between Alice's, Bob's and Charlotte's addresses thanks to the Cooperation heuristic

Figure 3.3: Illustration of the Cooperation heuristic

These two heuristics that are the Interactions and the Cooperation heuristics, enable to augment the address graph which should therefore contain more interesting information concerning the placement decisions than before. We present in the next section how are designed the scores used by DAVE.

3.1.3 Scores

Address-To-Shard score

The heuristics presented above aimed to augment the address graph that will now be used to compute scores, like the transaction graph was used for the scores in Optchain.

> Inspired by Optchain, a **Address-To-Shard** score, called **A2S**, was designed for DAVE. The A2S score of a given transaction t is stored in a vector A_t of dimension K if the network contains K shards. The i^{th} entry of this vector contains the fitness between this transaction and the shard i. The shard i for which the value of $A_t[i]$ is the highest, is the one having the most affinities with the transaction t. If the placement decision was uniquely based on A2S, such a shard i would be chosen for transaction t.

A_t can be computed in two steps, with first a focus on the addresses relations to minimize the number of cross-shard transactions and then the size of the shards is also taken into consideration to balance them. The A2S score A_t of a transaction t for K shards is computed in lines 1 to 19 of Algorithm 3.1.
The first step is to initialize A_t with 0s at every entry. Then two sets of transactions, T_1 and T_2, are needed to store respectively the transactions whose input or output addresses are also input addresses of t and the transactions whose input or output addresses are neighbours in G_2 of the input addresses of t. In order to retrieve such transactions the following functions and data structures are used:

- $G_2(V_2, E_2)$ is the address graph composed of a set of addresses (nodes) V_2 and a set of edges E_2 where an edge (a, b) between an address a and an address b represents the fact that these two addresses interacted at least once in a same transaction.

- A_{in} (respectively A_{out}) is a function taking a transaction t as input and returning the set of t's input (respectively output) addresses.
- d is a dictionary for which the keys are addresses and the values are sets of transactions such that $d[e]$ contains the set of all the transactions in which the address e was ever involved, either as input or output.

Once the two sets T_1 and T_2 are computed, they are used to build the A2S score while respecting two rules:

1. A transaction can only be taken once into consideration.
2. A transaction already in T_1 will be ignored in T_2.

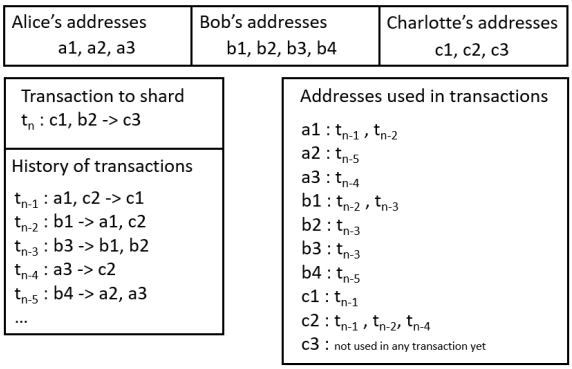
Then a last data structure is still needed to retrieve in which shards were placed the transactions from the T_1 and T_2 sets. This data structure is the following:

- s is a dictionary for which keys are transactions and values are vectors of size K (for K shards). $s[t]$ is the vector associated with transaction t and contains only 0s if t is a new arriving transaction or, if t is a known (and therefore already sharded) transaction, then $s[t]$ contains a 1 at the i^{th} entry if t was sharded in S_i and 0 elsewhere.

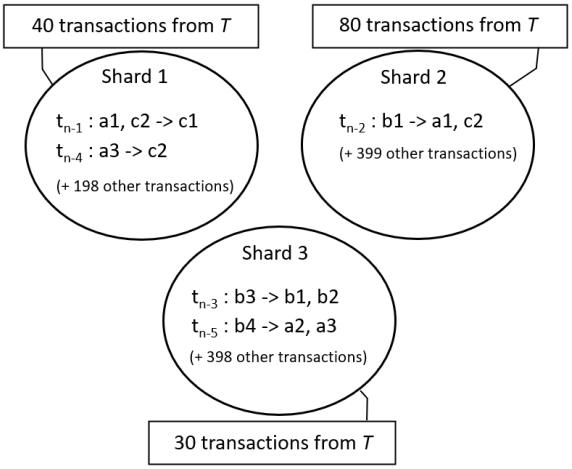
The vectors retrieved from s are added together to build the A2S score A_t. But the fact that transactions coming from T_2 are less related to transaction t we want to shard than the ones coming from T_1 has to be taken into account. For this, a parameter β has to be introduced. Its value is equal to 0.45 and it is used to weight the vectors retrieved from s and associated to transactions of T_2. In Algorithm 3.1, these additions of vectors from s correspond to the lines 11 to 12 for the transactions coming from T_1 and to the lines 13 to 14 for the transactions coming from T_2.

Finally, the last step will be to divide each entry of A_t by the size of the corresponding shard. The size of a shard being the total number of transactions ever placed in it. The larger the shard (in comparison to other shards), the worse it is for him to be selected for the reception of transaction t. The problem is that by doing so, we can only try to give the same number of transactions to each shard in the long term but not in the short term. This is due to the differences between the shards' sizes that would become less and less important in comparison to their total respective sizes as long as we shard more and more transactions. What we did to counter this weakness is that we simply compute the size of the shards among the set T of the last N sharded transactions where N is equal to 50 times the number of shards used. Lines 15 to 19 of Algorithm 3.1 illustrate how this can be done.

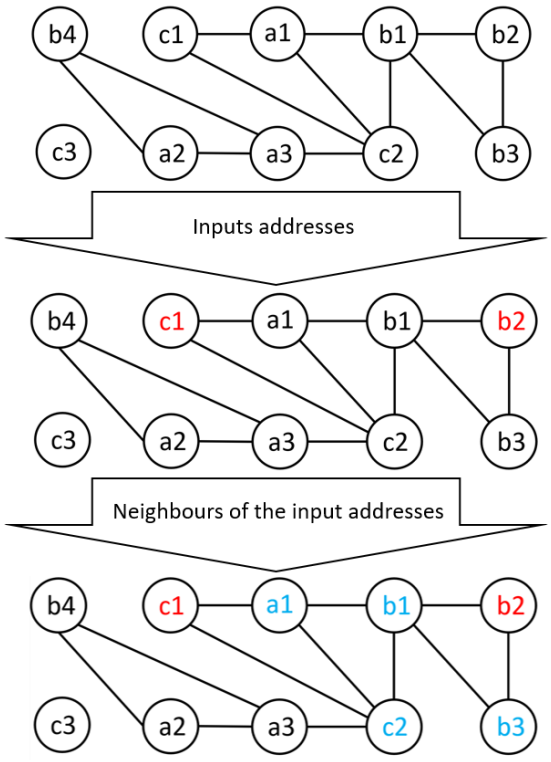
All the elements mentioned in these explanations are illustrated in the example of Fig. 3.4 to help with the visualisation of what was explained. It should already help to understand how an A2S score is computed.



(a) Initial data



(b) Current state of the shards



(c) Selection of nodes in the augmented address graph

$$A_{t_n} = \overbrace{[1,0,0]}^{c1} + \beta(\overbrace{[0,1,0]}^{a1} + \overbrace{[1,0,0]}^{c2}) + \overbrace{[0,0,1]}^{b2}$$
$$= [1+\beta, \beta, 1]$$

Division by the shards' sizes

$$A_{t_n} = [\frac{1+\beta}{40}, \frac{\beta}{80}, \frac{1}{30}]$$
$$= [0.036250, 0.005625, 0.033333]$$

(d) Computation of the score

Figure 3.4: The example illustrated in this Figure shows how an A2S score can be computed for a new transaction in the network. Here, in Fig. 3.4a and Fig. 3.4b, we described the current state of the network as well as the transaction Bob and Charlotte want to submit. Then in Fig. 3.4c we can see which addresses were selected for the score computation in the augmented address graph. While in Fig. 3.4d we have the final calculations leading to the A2S score of the transaction t_n. In conclusion, based on the A2S score only, we would choose shard 1 to validate the transaction t_n.

For the example of Fig. 3.4, we use a network of three shards in which Bob, Alice and Charlotte can submit transactions involving their respective addresses. In Fig. 3.4a, we have the users' addresses, the transactions in which each address was involved[1], a history of transactions and the transaction t_n we have to shard. This transaction is submitted by Bob and Charlotte who want to send Bitcoins coming from their $b2$ and $c1$ addresses together on Charlotte's $c3$ address. So this transaction t_n is written as $c1, b2 \rightarrow c3$, with the input addresses on the left of the arrow and the outputs on its right. This same notation is used to represent other transactions in Fig. 3.4a and Fig. 3.4b.

We suppose a total of 1000 transactions were already sharded[2], coming from known and unknown users. As we can see in Fig. 3.4b, from these 1000 transactions, 200 were validated by shard 1 and 400 by both shard 2 and shard 3. The three users are only involved in the last five sharded transactions (t_{n-1} to t_{n-5}). We will only look at T: the last 150 sharded transactions (50∗K where K = 3) to compute the size of the shards. Therefore we obtain the sizes written in Fig. 3.4b. So for the moment the shard 3 validated the least transactions from T and should therefore have more chance to be chosen as the next shard if we consider the load balance objective.

The first step to compute the A2S score of a transaction when we know the current state of the network is to retrieve from G_2 the addresses in which we are interested. Note that the address graph G_2 represented in Fig. 3.4c is only the connected component of G_2 containing the addresses of our three users. We do not have to pay attention to the remaining components of the graph since we start from Bob's, Alice's and Charlotte's addresses and these addresses never had any interaction with other users' addresses. We also already augmented the graph in Fig. 3.4c thanks to our two heuristics.
Starting from this augmented graph on the top of Fig. 3.4c, we found and colored the input addresses of t_n in red before doing the same with the neighbours of these addresses in light blue, as we see in the bottom graph of Fig. 3.4c.
Once this is done, we can identify the sets T_1 and T_2 and directly add the corresponding vectors from s to our A2S score A_{t_n}. We start with the input address $c1$ that was an output in transaction t_{n-1}, which is in shard 1. So due to $c1$, we add the vector $s[t_{n-1}] = [1, 0, 0]$ to A_{t_n}. Then we look at the neighbours of $c1$ in G_2 which are $a1$ and $c2$. These two neighbours were involved in t_{n-1}, t_{n-2} and t_{n-4} (we can know that thanks to the bottom right table of Fig. 3.4a) but since t_{n-1} was already considered once in the A2S score, we can ignore it. Consequently, we add the vectors $s[t_{n-2}] = [0, 1, 0]$ and $s[t_{n-4}] = [1, 0, 0]$ multiplied by β (which is equal to 0.45) to A_{t_n} since t_{n-2} is in shard 2 and t_{n-4} in shard 1 (visible on Fig. 3.4b).
We assume in Fig. 3.4d that the adding of $s[t_{n-2}]$ was due to $a1$ and not $c2$ only because we supposed $a1$ was examined first and a transaction can only be taken once into account for the computation of an A2S score. Then we can now do the same as we did for $c1$ but with the second input address, $b2$. This second input address was involved in t_{n-3}, which is in shard 3 and we will in fact only add $s[t_{n-3}] = [0, 0, 1]$ to A_t because every other

[1]This is the mapping between addresses and the transaction(s) in which they were involved, it corresponds to the data structure d mentioned previously.

[2]We did not apply our algorithm to shard these transactions, we suppose here we have to start from this given sharding.

transaction in which $b2$ or its neighbours were involved in was already taken into account in the computation of the score.
Now, we can divide each element of A_{t_n} by the corresponding shard's size. This lead to the final A2S score $A_{t_n} = [0.036250, 0.005625, 0.033333]$ suggesting that shard 1 should be chosen for t_n.

The conclusion is that even if shard 3 was the smallest one in the sense it was the least chosen for the last 150 transactions, our A2S score still considered that it was more clever to place t_n in another shard. This is due to the past interactions we have between the addresses of Bob, Alice and Charlotte. This shard choice means that at this step, it was still more important to limit the number of cross-shard communication rather than to reach a perfect load balance.

Use-of-the-Shard score

For the moment, the heuristics and the A2S score focused mostly on the minimization of the number of cross-shard transactions. The other objective is to maintain an acceptable load balance among shards over time. This is a bit taken into account in the A2S score but the fitness score between a transaction and a given shard is only divided by the length of this shard. In fact, by only using the A2S score, DAVE will have no way to enforce the load balancing.

> A new score will help to enforce the load balance among shards. It is called **U2S** to stay in the same logic as the T2S, L2S and A2S scores mentioned above. Even if "U2S" stands for "**Use-of-the-Shard**", it was easier to keep similar names for scores. The U2S score of a given transaction t is a vector U_t of size K (for K shards) containing negative integers. Its i^{th} entry is equal to minus the number of transactions placed in shard S_i among the last N sharded transactions (N is again equal to 50∗K, where K is the number of shards used). Negative numbers enable to give the least chosen shard the highest score. A transaction t will therefore tend to be placed in the shard S_i for which the i^{th} entry of U_t is the highest. This score can be computed as in lines 20 to 23 of Algorithm 3.1.

Fitness score

The next step in the design of DAVE's scores is to manage to use the A2S and U2S scores together to obtain the best overall performances. These two are first normalized then optimally combined in a weighted sum. By normalization, we mean a min-max normalization, which is a bijective mapping where the highest value of the vector is mapped to 1 while its smallest value is mapped to 0 and other values to decimals between 0 and 1. This normalization does not change the property that in both A2S and U2S the entry with the largest value gives the best shard. The scores need to be normalized to ensure they are always in the same order of magnitude before combining them. A vector can be normalized like this by using the Min-max normalization function from section 3.1.4.

The combination of the A2S and U2S scores is a weighted sum of two terms so only one coefficient is needed to give more weight to one or another term.
The coefficient used is called $U2S_{coef}$ because it multiplies the U2S score. The value of $U2S_{coef}$ is 0.0005, this choice will be justified in the evaluation of DAVE.

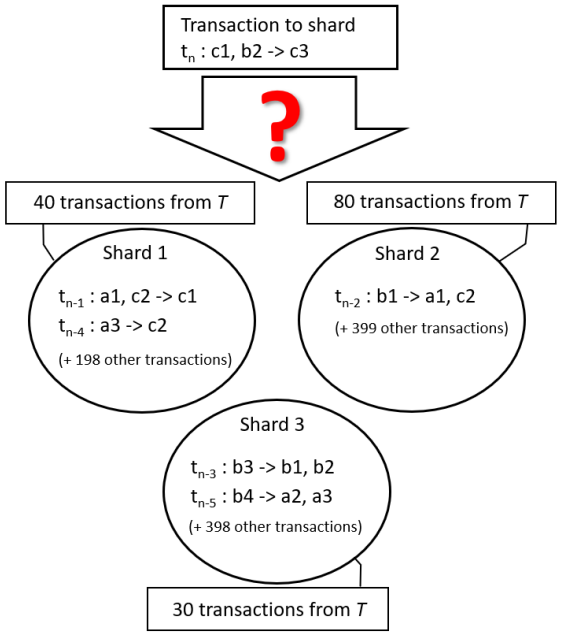
The result of the sum is a **Fitness score** called **FS** and since A2S and U2S are vectors of dimension K (for the K shards in the network), so will be FS. The FS score of a transaction t is therefore a vector F_t of size K for which the i^{th} entry is the fitness between a transaction t and a shard S_i. As it was the case for both previous scores, the entry of F_t having the highest value (or highest fitness), indicates which shard to select for transaction t. This score is computed in line 26 of Algorithm 3.1.

More precisely, Algorithm 3.1 shows how to compute the list L of the indexes of the shard(s) having the best (highest) FS score. Indeed, several shards can have the same fitness with a given transaction t. So L can be computed as in lines 27 to 33 of Algorithm 3.1 but then the question is: "What should DAVE do if there is more than one shard corresponding to the best fitness score for a given transaction? How should DAVE choose one single shard in such a situation?". In this case, there is a procedure to avoid making a random choice. To keep a deterministic and verifiable sharding algorithm no randomness can be tolerated. Below are the steps done to decide in which shard a transaction should be placed, including both the cases of one or several shards having the highest fitness score for a transaction t.

1. Select the list L of the indexes of the shard(s) having the best (highest) FS score.
2. If this list L only contains one element, then we have our solution.
3. Otherwise select among L the shards with the smallest size. Indeed, in case of elements with equal value in the FS score, DAVE places t in the shard that received the least transactions in the past.
4. If the remaining list only contains one element, then we have our solution.
5. If DAVE still has to decide between several shards, a hash function is used on the transaction hash to decide which shard should be picked among the remaining ones. What is done in reality is that, for a choice among R remaining shards, DAVE takes the modulo with respect to R of the result returned by the hash function. This gives a number between 0 and R-1 and a direct mapping with the shards can then be done to make a deterministic and verifiable choice.

This strategy ensures to make no random choice so that everything can be verified afterwards. At first sight, the last step is not ideal because it is not based on the minimization of the cross-shard communication or the load balance among shards. However, this is not exactly true because a pre-selection has already been done before reaching this step. Hence, a "blind" choice is still a good choice since DAVE chooses blindly among the best shards selected.
In addition to that, using the hash of the transaction should ensure a quite good load balance among shards if done on a lot of transactions.

The Fig. 3.5 takes back the example of Fig. 3.4 and should help to give a global understanding of what is done to select a shard when a transaction arrives in the network. It illustrates how to compute the Fitness score and the list L as in Algorithm 3.1.



(a) Current state of the network

$$A_{t_n} = [0.036250, 0.005625, 0.033333]$$
$$U_{t_n} = [-40, -80, -30]$$

Normalization

$$A_{t_n} = [1.0, \quad 0.0, \quad 0.904751]$$
$$U_{t_n} = [0.8, \quad 0.0, \quad 1.0]$$

Fitness score

$$F_{t_n} = A_{t_n} + \text{U2S}_{\text{coef}} * U_{t_n}$$
$$= [1.36, \quad 0.0, \quad 1.354751]$$

(b) Computation of the scores

Figure 3.5: Computation of a FS score

Fig. 3.5 is a complete example showing how a decision is made to shard a transaction, based on its fitness score. So as in Fig. 3.4, we start with the transaction t_n that we want to shard again in the same network. This is illustrated in Fig. 3.5a. Then the A2S score for t_n can be computed as in the previous example to obtain A_{t_n}, as shown in Fig. 3.5b. What is new here is the computation of U_{t_n}. This score is computed based on the placement of the last N transactions sharded. So in our case, these are the last 150 transactions sharded among which we assume 40 were in shard 1, 80 in shard 2 and 30 in shard 3. Knowing that, it is trivial to deduce the vector U_{t_n} on top Fig. 3.5b.
Once A_{t_n} and U_{t_n} are computed according to their definitions, they can be normalized. That is what is done in the middle step of Fig. 3.5b. Here we can see that the maximum values are indeed mapped to 1.0 while the minimum ones are mapped to 0.0, as expected. Finally, we can compute the fitness score F_{t_n} using the coefficient U2S_{coef} equal to 0.0005. The largest element(s) of F_{t_n} can be added to the list L which becomes $L = [1]$ since there is only one shard that has the largest score. In our case, it is the shard 1 as we can see from the list L. Therefore, transaction t_n will be sent to the shard 1.

This example shows that even if shard 1 was not selected the least in the sharding of the last 150 transactions, it will still be chosen for the validation of t_n. It means that the U2S score of t_n was not enough to change the decision suggested by the A2S score. In fact, we should have two elements of A_{t_n} very close, even more than in the given example, to make the U2S score decisive. The role of the U2S score will indeed be to help the selection made by the A2S score in case of an almost (or total) ex-aequo situation. This is when

A_{t_n} has two or more equal (or almost equal) elements. In such situations, when A2S does not help to make a decision, using a score as U2S should lead to a decision optimizing the load balance among shards and this is exactly what we want.

3.1.4 Resulting algorithm

In this section, we first define a function `MinMaxNorm` taking a non-empty vector as input and returning the same vector normalized according to a min-max normalization. This function is used to normalize our A2S and U2S scores as explained before in the Fitness score part of section 3.1.3. Then we present the Algorithm 3.1 regrouping the different scores needed to build the list L. Based on that list, DAVE will be able to choose which shard should validate a transaction t. In the pseudo-codes below, we assumed vector's indexes start at 1.

Function Min-max normalization function

Input: A non-empty vector X of length K.
Output: The vector X normalized according to a min-max normalization.

```
1  Function MinMaxNorm(X):
2      M ← X[1]  // Variable used to store the maximum element of X
3      m ← X[1]  // Variable used to store the minimum element of X
4      for i ← 2 to K do
5          if X[i] > M then
6              M ← X[i]
7          if X[i] < m then
8              m ← X[i]
9      if M = m then
10         for i ← 1 to K do
11             X[i] ← 1/K
12     else
13         for i ← 1 to K do
14             X[i] ← (X[i]−m)/(M−m)
15     return X
```

For the Algorithm 3.1, we define two operations on vectors:

- **Multiplication by a scalar** (symbol: *) - when a vector P is multiplied by a scalar α, we note it $\alpha * P$ and it means every entry of P is multiplied by α.
- **Addition of two vectors** (symbol: +) - when we add two vectors P_1 and P_2 supposed to have the same size K, we note it $P_1 + P_2$ and it means we add them entry by entry to obtain a new vector P_3 also of size K such that $\forall i \in [1, \mathrm{K}] : P_3[i] = P_1[i] + P_2[i]$

We use in Algorithm 3.1 some data structures, functions and parameters already mentioned before but we still recall their definitions to make sure everything is clear. All this data is used/accessed by the algorithm to obtain the list L as result.

- A_t is a vector of dimension K (for K shards) used to store the A2S score of the transaction t. The i^{th} entry of this vector contains the fitness between this transaction and the shard i.
- T_1 and T_2 are set of transactions containing respectively, the transactions whose input or output addresses are also input addresses of t and the transactions whose input or output addresses are neighbours in G_2 of the input addresses of t.
- A_{in} (respectively A_{out}) is a function taking a transaction t as input and returning the set of t's input (respectively output) addresses.
- $G_2(V_2, E_2)$ is the address graph composed of a set of addresses (nodes) V_2 and a set of edges E_2 where an edge (a, b) between an address a and an address b represents the fact that these two addresses interacted at least once in a same transaction.
- d is a dictionary for which the keys are addresses and the values are sets of transactions such that $d[e]$ contains the set of all the transactions in which the address e was ever involved, either as input or output.
- $S = \{S_1, S_2, ..., S_{\mathrm{K}}\}$ is the set of K shards.
- s is a dictionary for which keys are transactions and values are vectors of size K (for K shards). $s[t]$ is the vector associated with transaction t and contains only 0s if t is a new arriving transaction or, if t is a known (and therefore already sharded) transaction, then $s[t]$ contains a 1 at the i^{th} entry if t was sharded in S_i and 0 elsewhere.
- β and $\mathrm{U2S}_{\mathrm{coef}}$ are parameters of the model.
- T is the set of the last N transactions sharded, with N equal to 50 times K in our model.
- U_t is a vector of size K (for K shards) containing negative integers and used to store the U2S score of the transaction t. Its i^{th} entry is equal to minus the number of transactions placed in shard S_i among the last N sharded transactions
- F_t is a vector of size K (for K shards) used to store the FS score of the transaction t and for which the i^{th} entry is the fitness between a transaction t and a shard S_i.
- L is a list of indexes from the shards suiting best transaction t.

This algorithm is only there to understand how our scores can be first computed and then used together to find the best shards possible where to place a given transaction t. Some parts of it are indeed implemented more efficiently in practice. For example, the lines 16 to 18 aim to compute the number of transactions in the shard S_i among the set T. But in our implementation, this information is not re-computed at every transaction

sharded, it is kept in memory and updated in linear time with the number of transactions processed. What is important here is to understand the algorithm principle. The details on how everything is implemented can be found in our GitHub repository[3] and some of them will also be presented in section 3.4. We also recall that our goal was not to obtain the most efficient implementation possible but rather to have a working implementation of the algorithm designed.

[3]https://github.com/alxd112/UTXO_Graph_Analysis.git

Algorithm 3.1: How to build the list L of indexes from the shards suiting best transaction t.

Data:

A_{in}	s	S	β	$G_2(V_2, E_2)$
A_{out}	d	T	$\mathrm{U2S}_{\mathrm{coeff}}$	t

Result: A list L of shard indexes from the shards suiting the best transaction t.

```
/* Compute t's A2S score like in the Address-To-Shard part of section 3.1.3    */
for i ← 1 to K do
    A_t[i] ← 0
T_1 ← ∅ // Set of transactions whose input or output addresses are also input
 addresses of t
T_2 ← ∅ // Set of transactions whose input or output addresses are neighbours in
 G_2 of the input addresses of t
foreach e : e ∈ A_in(t) do
    foreach t_e : t_e ∈ d[e] ∧ t_e ∉ T_1 do
        T_1 ← T_1 ∪ {t_e}
    foreach y : y ∈ V_2 ∧ (y, e) ∈ E_2 ∧ y ∉ A_in(t) ∧ y ∉ A_out(t) do
        foreach t_y : t_y ∈ d[y] ∧ t_y ∉ T_2 do
            T_2 ← T_2 ∪ {t_y}
foreach t_1 : t_1 ∈ T_1 do
    A_t ← A_t + s[t_1]
foreach t_2 : t_2 ∈ T_2 ∧ t_2 ∉ T_1 do
    A_t ← A_t + β * s[t_2]
for i ← 1 to K do
    size ← 0
    for t_j : t_j ∈ T ∧ t_j ∈ S_i do
        size ← size +1
    A_t[i] ← A_t[i] / size

/* Compute t's US2 score like in the Use-of-the-Shard part of section 3.1.3    */
for i ← 1 to K do
    U_t[i] ← 0
    for t_j : t_j ∈ T ∧ t_j ∈ S_i do
        U_t[i] ← U_t[i] − 1

/* Normalization of A_t and U_t like in the part Fitness score of section 3.1.3 */
A_t ← MinMaxNorm(A_t)
U_t ← MinMaxNorm(U_t)

/* Compute the FS scores of t like in the part Fitness score of section 3.1.3  */
F_t = A_t + U2S_coeff * U_t

/* Build the list L of shard indexes from the shards suiting best transaction t,
   like in the part Fitness score of section 3.1.3                             */
L ← []
L.append(1)
for i ← 2 to K do
    if F_t[i] > F_t[L[1]] then
        L ← [i]
    else if F_t[i] = F_t[L[1]] then
        L.append(i)
```

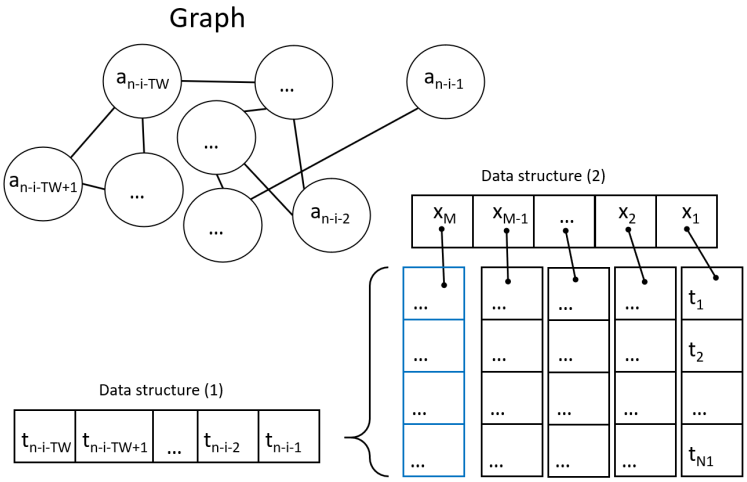
3.1.5 Remaining challenges

Thanks to the scores used, the algorithm already satisfies some of the initial challenges. DAVE can focus both on the minimization of the number of cross-shard transactions and the load balance among shards. The decisions taken are also ensured to be deterministic but there are still other constraints that need to be satisfied. The time and memory complexities of the algorithm, as well as its verifiability, are some of these.

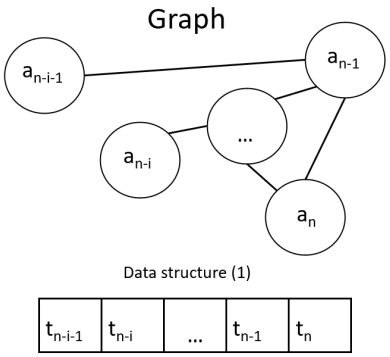
In order to execute the algorithm 3.1, several data structure are needed, which are listed next to the name *Data* on algorithm 3.1. It might seem acceptable to keep all those information in memory but the problem is that the amount of information that needs to be stored in the data structures only increases with time since it is directly related to the total number of transactions ever submitted to the network. Storing so much information would be a too important requirement for the miners.

Fortunately, DAVE uses a **time window** that will slide as time goes on. Details about the unit of this time window and how it is used will be given later. The main point is that only a limited amount of information is kept in memory by DAVE to perform its sharding. By doing so, the size of all the data structures used is ensured to not increase to infinity. This time window enables to satisfy several constraints at the same time. By using only a subset of preceding transactions, the decisions will be taken fast and will only require a limited amount of memory. Therefore it will also be possible to verify the placement of transactions in a bounded time and using a bounded memory. Adding all this together makes our solution easy to introduce in a real network. The next section details how our algorithm can be introduced in a real network so more explanations on how this time window can slide in a real network will be given.

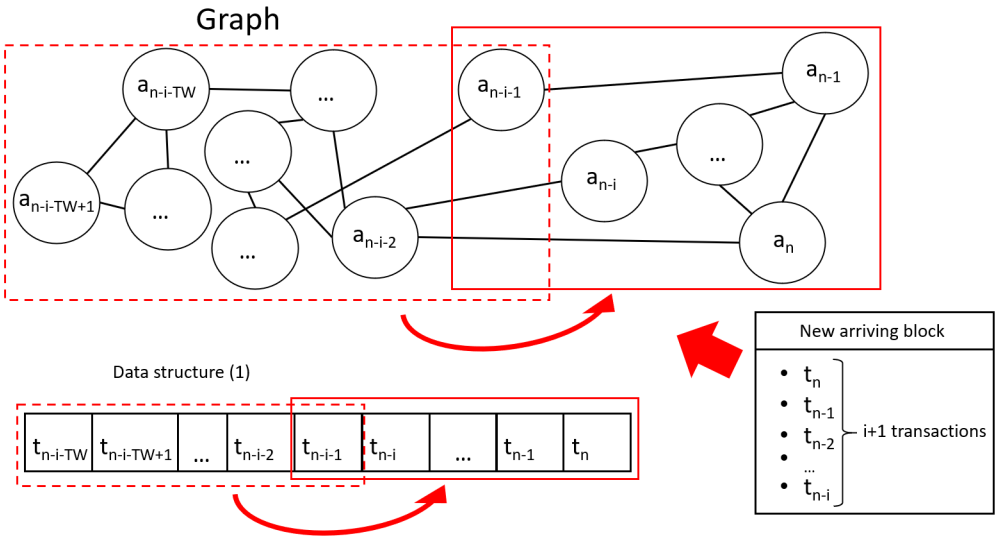
About the network, it is only important for now to know that one entity of the network will take the placement decisions and make the time window slide while other entities will validate transactions.
The entity in charge of taking the placement decisions can only use information about already validated transactions and this information arrives in the form of blocks. This means our time window slides at every block received. Every time a block of transactions is built, we update the information we can use to respect the memory limitation of the time window. Fig. 3.6 illustrates how a time window operates on our data.



(a) Initial data structures



(c) Data structures after the time window slid



(b) Transition of the time window on data structures at the arrival of a new block

Figure 3.6: Evolution of our data structures through time with a time window of TW transactions. The different data structures go from some initial state in Fig. 3.6a to their updated state in Fig. 3.6c after one slide of the time window. Fig. 3.6b illustrates the transition between the initial and the updated states.

To ease the comprehension of the example, we assumed the size of the time window to be arbitrarily set to TW transactions. For instance, the graph of Fig. 3.6a, could be our address graph G_2, for which each node is a given address and edges are added according to our heuristics. We just have to consider a node a_n as an address for which the last transaction it was involved in is the n^{th} sharded transaction.
The data structure (1) of Fig. 3.6a is simply a list of transactions while the data structure (2) is a list of such lists.
Starting from these data structures, we now consider a new validated block is arriving. This is the situation depicted in Fig. 3.6b. The new block contains $i + 1$ new transactions we have to take into account in our data structures. All the data structures are updated accordingly, to include these new transactions. However, we cannot allow them to contain

more data than the maximum allowed by the time window TW. The red dotted line is the time window before the arrival of the new block and the red plain line is the time window after the arrival of the new block.
As we see, we cannot just add in memory everything coming from the new block, we have to slide the time window in order to only keep the most recent data. That is what is illustrated in Fig. 3.6b. Before the arrival of the new block, we had information about transactions $n - i-$TW to $n - i - 1$ and now we will only keep information about transactions $n - i - 1$ to n. This means that we had TW-1 transactions inside the new block. Finally, we only keep what is covered by the time window after it slid and delete the remaining data, to end with the data structures as represented in Fig. 3.6c.

3.2 Our sharding algorithm in a real network

Until now, we presented DAVE, a way to intelligently place transactions among different shards. So here, we are going to explain how this method could be used in a real network. We present a protocol, or more precisely we present the modifications to an existing sharded blockchain that would be needed to support the introduction of DAVE.

First of all, the sharded blockchain we would use is based on the one proposed in RapidChain [21] (see section 2.3.2 for more details about RapidChain). We chose it because it was very well detailed, it could easily integrate our sharding method and also because it uses a Reference shard to manage the shards. We introduced **3 adaptations** in order to be able to make it work with DAVE, these adaptations are detailed below.

Sharding In RapidChain, the main phase consists of the election of a leader by each shard, followed by the construction of blocks of transactions by the leaders, then a consensus in each shard is done before appending the blocks to their chains. All this is repeated until the end of an epoch. The Reference shard is inactive during all this main phase, so *we decided to use it for computing the best shard where to place a transaction in according to the method we developed above* (1^{st} adaptation). Therefore, when a client submits a transaction into the network, instead of being sent to one shard randomly, the transaction will be routed directly to the Reference shard C_R. In the Reference shard, the transaction will be taken care of by one of the Reference shard's members that will run the DAVE algorithm in order to know where the transaction should be placed. To speed up the process, each member of the Reference shard works individually on different transactions that will then be sent to the best shard as shown in Fig 3.7.

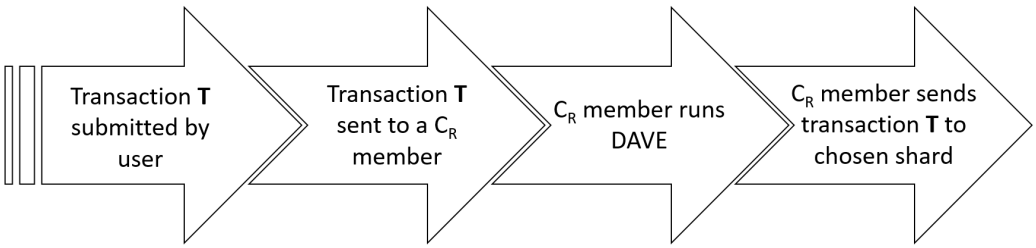


Figure 3.7: Scheme of the sharding || C_R: Reference shard

When a shard receives a transaction T to validate like in Fig. 3.8, it initiates *a transaction intra-shard consensus protocol* (2^{nd} adaptation) called S-BAC (this has already been explained in section 2.3.2 and is explained again differently in the appendix C). The protocol will make sure the transaction is valid and that the nodes of the shard agree to accept it or not. We added this step ourselves and it is largely inspired by what ChainSpace uses with slight modifications to suit our case. Once transaction T is accepted, it is added to block B of the shard. When B is full, the leader runs *a block intra-shard consensus protocol* on the block B (described in section 2.3.2). Finally, B is added to the shard's chain.
The 2^{nd} adaptation had to be made because RapidChain's method to deal with cross-shard transactions is not compatible with DAVE. In fact, when a transaction has inputs that are in different shards than the outputs, RapidChain bypasses the difficulty by creating additional transactions that move the inputs from their original shards to the same shard as the outputs.

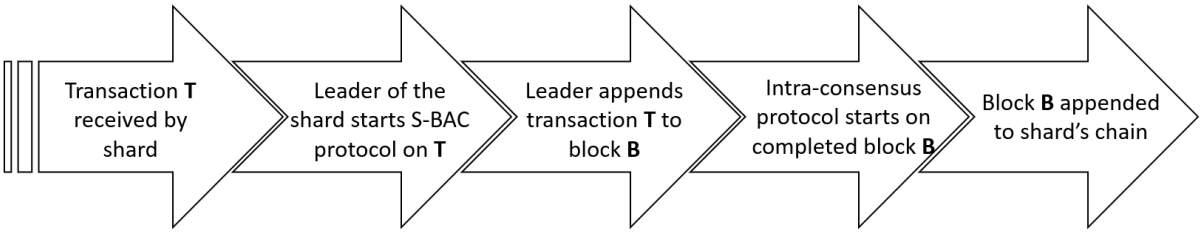


Figure 3.8: Main phase

As explained in the previous section, our algorithm is based on the notion of **time window**. This concept enables the algorithm to take a placement decision based only on a limited history of transactions (a window) that evolves in time. We introduced this concept in order to avoid having a history of transactions growing indefinitely which would be costly in term of memory but would also have bad consequences on the performances and verifiability properties of our method.
As in our solution, the Reference shard C_R is given the role of sharding the incoming transactions, it will also be required to update its history of transactions. To do that, it needs to be aware of when a new block has been accepted by a shard. So when this is the case, the newly created block is sent by its shard to C_R. C_R then updates the information it knows about the network with the data contained in the block received and deletes some old information to respect the time window limits made by our algorithm as depicted in Fig. 3.9 (3^{rd} adaptation). To update all the data structures, we defined four time windows. These time window limits for the history of C_R are the ones below, for which G1OldestBlock (resp. G2OldestBlock) is the number of the oldest block from which we can find information in G_1 (resp. G_2).

1. Number of transactions in $G_1 \leq 10000$
2. Number of addresses in $G_2 \leq 15000$ | [4]

[4]This case is special. When an new block arrives, the C_R will delete of its history the oldest blocks one by one until being left with a number of addresses inferior or equal to 15000.

3. For dictionaries involving transactions keep only information from blocks whose number are ≥ (G1OldestBlock - 25)

4. For dictionaries involving addresses keep only information from blocks whose number are ≥ (G2OldestBlock - 25)

The Reference shard C_R will update itself according to each limits. The choices for these constraints will be discussed in section 4.

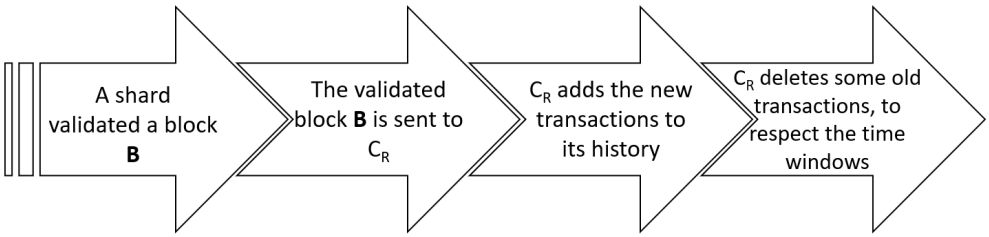


Figure 3.9: Reference shard update

Meanwhile, the shard whose block got accepted marks all the UTXOs used in the block as input as *spent* for those he stores and for those he does not, he will notify the shard holding them.

Now that we explained how to integrate the DAVE in a real sharded blockchain. We will explain in the next section a key property of our method which is verifiability. As it is mainly based on the time window concept, further examples about it will be displayed at the same time.

3.3 Verifiability

One of the main objectives we were trying to reach when working on this new sharding technique was to maximize its security. In addition to being tolerant to byzantine attacks, we wanted to let anyone verify the placement of transactions. This way people can be sure that the placement of their transactions or from other users has been done according to the rules. Therefore verifiability is a key point in our sharding method to detect possible attacks on the protocol and possibly prevent them. Indeed, an attacker could do some important damages to our system if he was able to choose in which shard to place his transactions. For example, an attacker could send a lot of transactions to a particular shard in order to flood it. Which will slow down a lot the processing of other users' transactions that have to be validated in this shard.

Since the placement of every transaction has been computed based on a window of previous transactions, we can verify the placement of the transaction by using the exact same window of previous transactions like the one used for the initial placement choice. The question is how to retrieve this window of transactions?

The first idea was based on a common time shared among all nodes of the Reference shard C_R. We were using the time of validation stored in the transaction object, to be

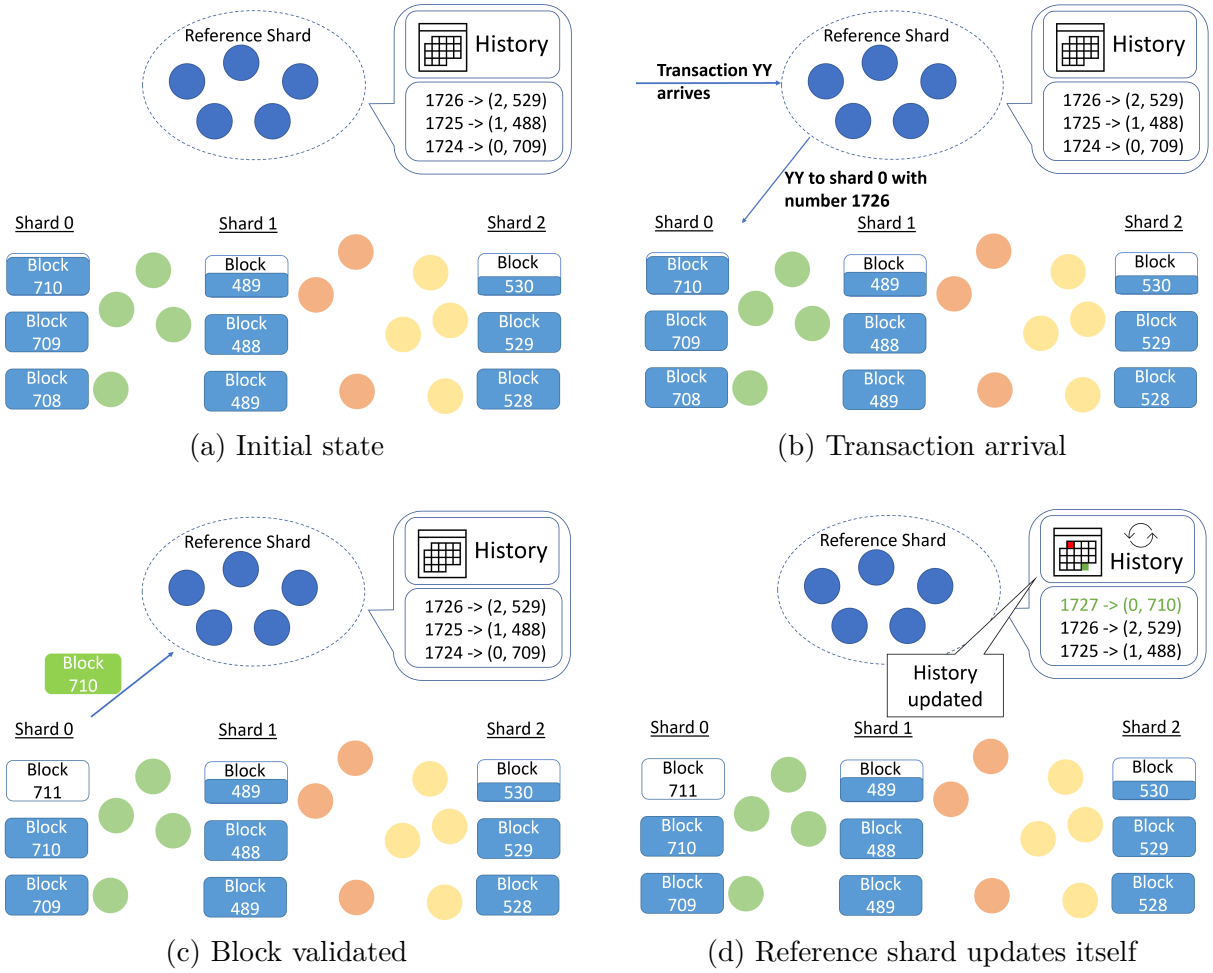
able to have a global order of the validation of transactions and so their placement into shards. This way, a user who wants to verify a transaction simply has to find the last X transactions that were sharded before the concerning transaction. In this case, a user only has to run the algorithm once using a small number of transactions. Unfortunately, this method is rather optimistic since we were informed by our supervisors that it is really hard for independent nodes to share the exact same time in a distributed system, especially if the nodes are geographically far from each other.

Then, we decided to go forward with a more realistic solution:

> The Reference shard associates a unique number N to each new validated block received so that the earlier a block has been validated, the smallest the number it will be associated with. This way, the Reference shard keeps track of the order in which the blocks arrived to it. Also, when a new block gets associated with a number by the Reference shard, this number will serve as a key in a dictionary stored by the Reference shard to keep in memory from which shard this block came. The values of this dictionary are tuples of the form (S, H) where S is the shard number where the block is permanently stored in a blockchain and H is the height of the block in this shard's blockchain. When a transaction is sharded, it is accompanied by the number of the most recent block stored by the Reference shard. The Reference shard makes its decisions according to a window of the latest transactions validated. So, a user would need the information of this window in order to verify the placement of the transaction. Thanks to the global order of the blocks in the Reference shard, we can deduce the window of transactions used by the Reference shard to make the decision and thus replicate it.
>
> In order to retrieve the window of transactions used to shard a transaction, we simply look a the number associated with the transaction we want to check, let's say the number is Y. As explained, this number corresponds to a block. The steps to retrieve the window are as follow:
>
> 1. We take 10000 transactions of this block, if it does not contain enough, we take the rest from the block Y-1, Y-2 and so on until reaching 10000 transactions. We fix G1oldestBlock to the oldest block we took transactions from.
> 2. We take 15000 different addresses into account from block Y, Y-1, Y-2 and so, stopping a the block right before reaching 15000 addresses. We fix G2oldestBlock to the oldest block we took addresses from.
> 3. We take all information of block Y to the block numbered G1oldestBlock - 25 necessary for the dictionaries involving transactions.
> 4. We take all information of block Y to the block numbered G2oldestBlock - 25 necessary for the dictionaries involving addresses.

Let's now see what happens concretely in the Reference shard when Bob sends a transaction (called 'YY') into the network. The initial state of the network is represented in Fig. 3.10a. Then, Bob sends his transaction 'YY' on Fig. 3.10b. At some point, the

transaction will be routed to the Reference shard, which will place it into a shard according to a history of transactions. The transaction is attributed to the shard 0 and associated with the number 1726. This number signifies that 'YY' has been placed when the newest block in the history of C_R was block 1726.
The block of shard 0 is completed, thus it is sent to C_R as depicted in Fig. 3.10c which will update its information according to the four limits. For the example, we fixed the limits to 20 transactions, 30 addresses, 1 block for the transactions and addresses dictionaries. In Fig. 3.10d, the Reference shard updates its history of transactions by adding the new information and deleting old information in order to satisfy the limits.



(a) Initial state

(b) Transaction arrival

(c) Block validated

(d) Reference shard updates itself

Figure 3.10: Example showing the system when a transaction is sent and a block is completed.

Now, we are going to describe a concrete example of the verification of a transaction. We start with Fig. 3.11, where Alice wants to verify that the transaction 'YY' has been placed in the correct shard. In order to do that, she first has to locate transaction 'YY' in the chains. Once this is done, she can retrieve the associated number which is 1726. With this, she knows that 'YY' has been placed when the newest block in the history of C_R was block 1726. Then she uses the mapping stored by the Reference shard, which will tell her that block 1726 from the Reference shard is actually from shard 2 and is the block 529 in this shard (see Fig. 3.11).

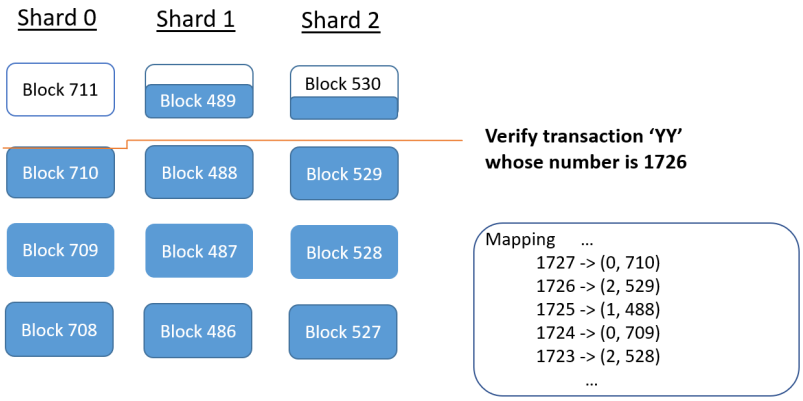


Figure 3.11: Verify transaction 'YY'

So, Alice has to take the most information into account according to the **time windows limits** that the Reference shard used. This way she should be able to recover the history of transaction used by C_R to make its choice. Alice will start taking information from the transactions in block 529 of shard 2 (which is block 1726 in C_R) starting at the end of the block (top left corner of Fig. 3.11). Fig. 3.12a represents the transactions inside block 529 of shard 2, each transaction (represented by a contract icon), has a number written on it, that corresponds to the number of new/not yet seen addresses in the corresponding transactions.

Alice starts by taking all the block 529, which brings her to 9 transactions, 9 addresses, 1 block for both the transactions and addresses dictionaries. Now, Alice will do the same with the next block, which is 1725, thus block 488 of shard 1 see Fig. 3.12b.

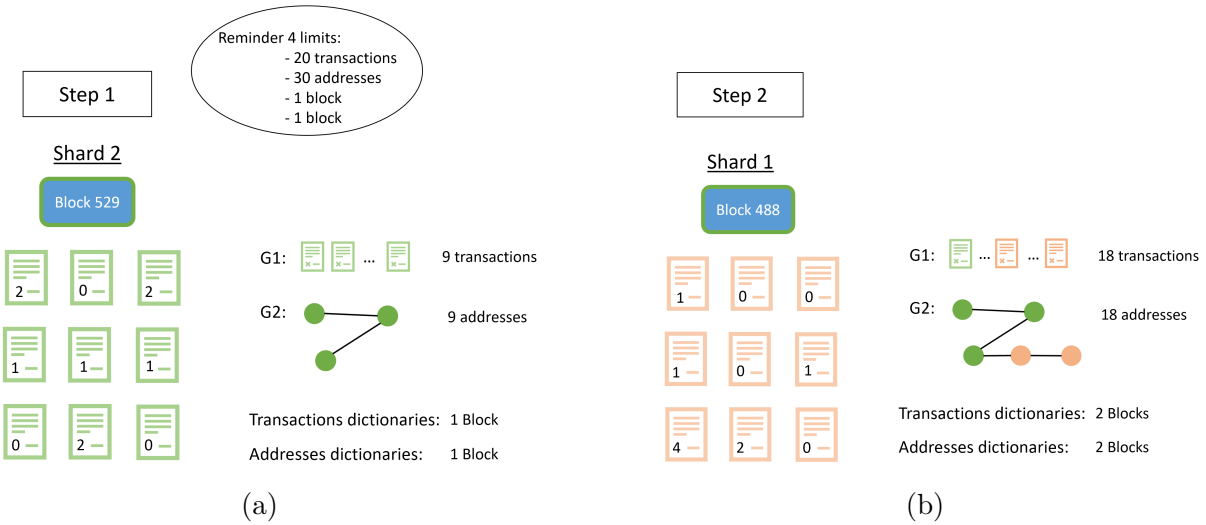


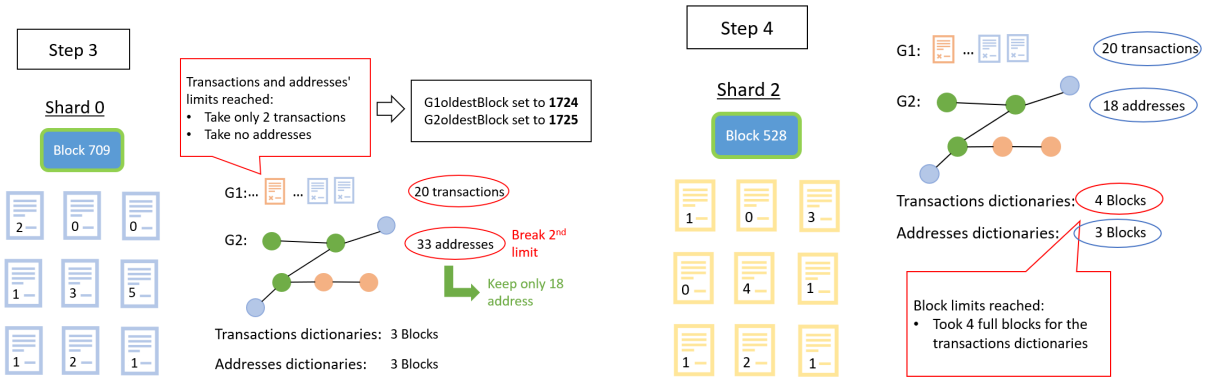
Figure 3.12: Alice takes as many transactions as she can into account while respecting the four rules. **Step 1**: At this moment, she has not taken anything yet, so she can take all the block 529. **Step 2**: She takes all the block 488, thus she has now a history made of 18 transactions, 18 addresses, 2 blocks for the transactions dictionaries and 2 blocks for the addresses dictionaries.

Alice has now taken 18 transactions, 18 addresses, 2 blocks for both the transactions and addresses dictionaries. Alice keeps going with block 709 of shard 0, but can only take the two oldest transactions, otherwise, she would break the first limit (see Fig. 3.13a). Therefore, she now can fix the variable G1oldestBlock to 1724. The third window condition becomes:

3. We take all information from block 1726 to block 1724 - 1 necessary for the right dictionaries (involving transactions).

Then, Alice tries taking the new addresses of block 709 but realizes that she would get 33 addresses, which surpasses the limit. Then, she goes back and stops at the preceding block, which is in this case the block 1725 (this is block 488 of shard 1). Thus, Alice can fix the variable G2oldestBlock to 1725. The last window condition becomes:

4. We take all information of block 1726 to the block 1725 - 1 necessary for the right dictionaries (involving addresses).



(a) (b)

Figure 3.13: **Step 3**: Alice takes only the two oldest transactions of the block because of the limit of 20 transactions. She does not take any addresses from this block otherwise she would broke the second limit. Alice takes all the new addresses of this block as well as all the information of the block for the dictionaries involving transactions and addresses. The 4^{th} limit is reached because the block 709 of shard 0 has the unique number 1724. **Step 4**: Alice takes all information of this block for the dictionaries involving transactions and stops there. As the third limit has been reached, as block 528 of shard 2 has the number 1723 in C_R.

In Fig. 3.13b Alice goes through the next block, 1723 (block 528 from shard 2). Alice does not add anything to G_1, G_2 or the addresses dictionaries since their corresponding limits have already been met. But we can take the information of this block into account for the different transactions dictionaries. However, it will be the last block she needs, since the third limit tells us so.

With all this, Alice managed to retrieve the history of transactions that the Reference shard used to shard the transaction 'YY'. She can now run the DAVE algorithm on this history and check the results.

3.4 Implementation

This section contains details about how we implemented the DAVE algorithm. We present first some points about the project organization and the algorithmic considerations we had to make while designing DAVE. Then we also explain how we tested its performances in an existing network.

3.4.1 Sharding algorithm

Project organization The first difficulty encountered when thinking about the implementation of an algorithm like DAVE is to find access to the real Bitcoin blockchain in order to design and test the sharding algorithm. We used a Bitcoin blockchain parser [26] developed by Antoine Le Calvez (using the pseudo *alecalve* on GitHub) to read correctly all the data contained in the Bitcoin blockchain. This way, we can use the transactions and block information coming directly from this Bitcoin blockchain to test our algorithm. We assumed this was a realistic way to proceed and we will evaluate this assumption in the Evaluation section.

About the code itself, it was written in python and the whole project is available on our GitHub repository[5]. We recall the main goal was to write a code simulating a working algorithm but its implementation could still easily be optimized.
We implemented different sharding algorithms in different files. First, we have DAVE, but we also implemented a random sharding and a relaxed sharding to compare their performances with the one used by DAVE. A random sharding simulates what could be done by existing sharded blockchains while a relaxed sharding shows how the performances of DAVE could be improved by relaxing some constraints. For example, by updating the time window at every transaction received instead of block by block. However, we noticed that this kind of sharding did not outperform significantly DAVE. In addition to that, it is unrealistic so we will not present it in the Evaluation section.
In opposition to this, the random sharding is realistic and will be used as a baseline for the evaluation of DAVE.

Algorithmic considerations We will here discuss how is organized the implementation of DAVE. It uses a set of global variables initialized to default values to store the data structures needed in the algorithm. Then the algorithm itself is decomposed into several parts and each one of these parts is implemented in a dedicated function. These functions are all called by a `main` function coordinating the sharding that must be done with the transactions it gets from the Bitcoin blockchain. More precisely, we can list some of these key functions used by DAVE as follows.

- A function `create_G1` (respectively `create_G2`) to construct the transaction graph G_1 (respectively the address graph G_2).
- Two functions `compute_A2S_score` and `compute_U2S_score` to compute respectively the A2S and U2S scores of a given transaction.

[5] https://github.com/alxd112/UTXO_Graph_Analysis.git

- A function `sharding` which can be seen as the second-highest level function after the `main` one, it executes the sharding by implementing the Algorithm 3.1 and then calls the `choose_shard` function.
- A function `choose_shard` to choose the most appropriate shard to place the incoming transaction, based on the obtained scores. It implements the strategy used to make a deterministic choice in the list L of shard indexes.
- A function `nbrCrossShard` that counts the total number of cross-shard edges and cross-shard transactions DAVE could not avoid. It calls the function `count_crossSansG1` which counts the number of cross-shard edges a single transaction has.
- Three functions `timeWindowGraph1`, `timeWindowGraph2` and `timeWindowDicos` that update the data structures used every time new information has to be taken into account to make sure we respect the assumption of the time window with the four limits to be respected by C_R.
- A function `ValidateBlock` to add a validated block to the blockchain and update the data structures used for the sharding in C_R accordingly.

We will now explain some of the implementation choices we made in the design of DAVE. First, to implement the transaction and address graphs G_1 and G_2, we used a list of transactions for G_1 and a graph from the library networkx for G_2. In fact, we do not need a whole graph for G_1 as we do not use directly such information in our algorithms so we opted for a list as this is probably more efficient since it can be used as a FIFO queue. Indeed, transactions only happen once and arrive in chronological order.
We also stored in the variables `G1_oldesBlock` and `G2_oldestBlock` the number of the oldest block respectively in G_1 and in G_2. By oldest block, we mean the oldest block from which a transaction or an address comes from and is still in G_1 or G_2. These two structures are updated differently to satisfy the time window constraints. For G_1 we can simply keep the most 10000 recent transactions directly since we have a queue to update. But for G_2 we work with addresses so the limit will be 15000 addresses. To only keep the 15000 most recent ones, we would have to define a total order among them. It was more realistic to associate an address with the block in which it appeared. So to update G_2, we used a property of networkx enabling us to retrieve all the nodes of the graph with a given condition on their attributes. The condition was here the block number, we remove first the transactions coming from the oldest block in G_2.
The conclusion is that for G_1 we satisfy the limit of 10000 transactions perfectly while for G_2 we remove addresses block by block until the limit of 15000 addresses is satisfied.

Another aspect we had to simulate is the creation of blocks in our algorithm. This had to be done to enable C_R to update its history correctly. For this, we defined a parameter **blockSize**, fixed to 400, that defines the maximum number of transactions the algorithm can put in a single block and the parameter **timeBlock**, fixed to 300, expressed in seconds and representing the maximum time a block can spend being built. These choices will be justified in the Evaluation section.

We also designed an interesting feature that enables us to take the prices of transactions into account. If the intra-shard transaction is the cheapest, which should normally always be the case, then we make the sharding as expected. However, when the inter-shard transaction becomes the cheapest one, we have to take this into account because it means it is not as important to minimize cross-shard communication. We decided in such a situation to make a choice based uniquely on the hash of the transaction with a probability directly related to the ratio between the defined prices. This will increase the percentage of cross-shard transactions while improving the load balancing. It is important to understand the choice introduced is not random and will, in addition to making the sharding even faster, preserve the deterministic and verifiability properties of the algorithm.
For the evaluation of DAVE, we always worked with fixed prices for transactions such that intra-shard transactions are always cheaper than inter-shard ones like it is the case in real networks. This ensures no random transaction placement will ever be done but only the efficient placement made by the DAVE algorithm, as presented before.

3.4.2 Integration with Byzcuit

We wanted to try the DAVE strategy in a real network in order to see whether or not it improved its throughput and latency.
Initially, we wanted to try our sharding algorithm in RapidChain, but sadly we were told that this project is not public anymore. Fortunately, we were informed by our supervisors that their colleagues developed another sharded blockchain system that could help us.

This system is **Byzcuit** [27], it is very similar to Chainspace, Byzcuit is actually an implementation of the fundamental principles of Chainspace. The good thing with Byzcuit is that we can run tests without specifying transactions' objects (= addresses). We only need to provide a file with the shards where we want an object of a transaction to be located. For example, one line of the file could be $1, 2$. Byzcuit will then build a transaction with inputs being an object from shard 1 and an object from shard 2, and with the output(s) being either an object from shard 1 or 2 chosen randomly. This manner of testing is very convenient since we can retrieve the transactions sharded by DAVE to write them in a text file, and then simply feed it to Byzcuit.

Before actually running all this, we tried to understand the code, so we contacted one of the contributors of the project to ask questions about how to launch the program since no readme was available at first. Right after we sent our email, the contributor answered us and told us he pushed a readme so that we could easily use the project. We found several errors in the readme and informed the author about them. Later, we also wrote a more detailed readme about every single line of code that needed to be changed in order to run the project correctly and shared this with the author.

Regarding the code itself, it makes use of AWS services. So, Byzcuit creates shards that are constituted of nodes. Every single node is an independent instance on AWS. Furthermore, the transactions are sent by clients which are also independent AWS instances.

Basically, the program can be used by following these steps (everything is well available on the GitHub of Byzcuit or ours):

1. We create the number of instances for the nodes of the shards and the number of clients desired. Those will be seen *running* on the AWS website.

2. We execute the command that will initialize those clients with the correct versions of packages and that will clone the GitHub project on each instance.

3. We run the program with the file of transactions we want to test and with the correct argument values, like the number of shards used, the number of transactions and the output files.

4. When the test is done, we *stop* the instances to avoid wasting money, they can be restarted at any time without needing to do the initialization again. We *terminate* the instances when we will not need them for a long time, to avoid paying storage fees.

New transaction maker mode

The program until here only supported transactions where only the inputs can be chosen, the outputs are chosen randomly from the inputs. This is not sufficient to represent a sharding produced by DAVE, we needed a more deterministic way to describe transactions. So we decided to write our own **transaction maker mode** where we could specify inputs as well as outputs. A user simply has to set the `inputobjectargument` to 5 in the file `tester.py` to use this new mode. This way, we can feed to Byzcuit a text file made of transactions under the following format:

$$SI1, SI2, ..., SIN : \underbrace{SO, SO, ..., SO}_{\text{M times}}$$

Where SIi is the shard where the input i of the transaction is located and SO is the shard where the outputs are located (all outputs goes into the same shard by the definition of DAVE). This format is illustrated in Fig. 3.14.

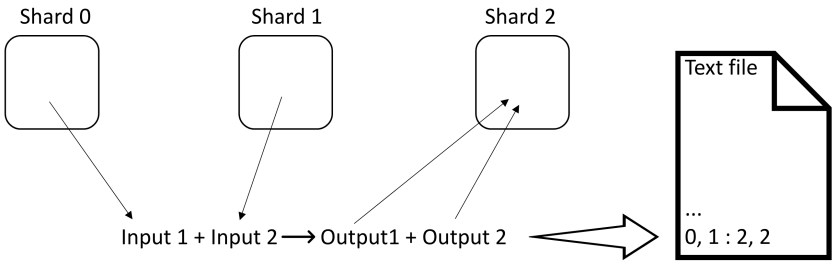


Figure 3.14: This example shows how we write the transaction into the text file (transactions file).

Before going any further, we have to mention the fact that some transactions cannot be written into the text file. Indeed, we do not write coinbase transactions into this text file because it does not have the right format "input(s):output(s)" required by this new mode. In addition to that, since we will do our tests on a range of blocks right in the middle of the blockchain, there will be a lot of transactions for which DAVE does not know their inputs. These are considered as coinbase in the simulations. Therefore, such transactions cannot be written in the format described above. If we decided to simply not write those transactions into the text file, we realized that this was problematic. Because, during the sharding, DAVE took all the transactions, including the ones considered as coinbase, into account for both the cross-shard minimization and the load balance. Therefore, we observed that by giving to Byzcuit a text file that does not contain the same transactions like the ones processed by DAVE, we got a terrible load balance for Byzcuit. Simply because some of the transactions used by DAVE to optimize the load balance were missing in the text file.

We decided for these simulations to modify slightly the load balancing part of the DAVE algorithm, we decided to not take into account transactions that appeared to be considered as coinbase by DAVE: namely real coinbase transactions and transactions that have no know input addresses. These transactions cannot be written in the text file so it was logical to avoid them in DAVE as much as possible. We managed to not take them into account for the computation of U2S but we still use them partially for the score A2S. All in all, the results obtained with this modification should be well representative of the performances of DAVE. It will give us a lower bound of its actual performances since coinbase transactions are never cross-shard and can even be used to improve the overall load balance.

Chapter 4

Evaluation

This chapter presents the evaluation of our model in two parts. First, we look at the sharding performances of DAVE. We get back to the original bi-objective problem and verify our solution manages to both reduce the percentage of cross-shard transactions and keep an acceptable load balance between the shards. We also ensure the other challenges stated in the problem statement are solved with DAVE. Then, we present the performances we could have when integrating DAVE into Byzcuit.

4.1 Evaluation of the sharding algorithm

In this section, we start by analysing how realistic arrival times were simulated for each transaction. Then we explain how we choose the values of the parameters used by our model as well as what results can be obtained with such parameters. This section ends with a few words on the complexity of our sharding algorithm.

4.1.1 Simulation of the transactions' arrival times

The simulations were done with transactions coming from the real Bitcoin blockchain, as explained before. Each block of transactions contains a timestamp telling when was the block added to the ledger. However, for the simulations, an arrival time is needed for each transaction rather than for each block. So the solution was to split uniformly the interval of time separating a block B from the preceding one between the transactions of B. In case there is no preceding block, all the transactions will get the timestamp of the block from which they come from[1].

In the implementation of DAVE, the transactions' arrival times are not simulated by a real waiting time, the time is only a variable. Indeed, the placement decisions will not be impacted by the time waited between two incoming transactions but only by the order in which they arrive. The order of the transactions is always the same which is the order

[1]Sometimes it happens that the timestamps of Bitcoin blocks are not relevant. For example, the timestamp of a block might be strictly smaller than the one of the preceding blocks. In this quite uncommon situation, a solution was to just keep the preceding timestamp for all the transactions of the block to ensure the time simulated never "sends a transaction back in the past".

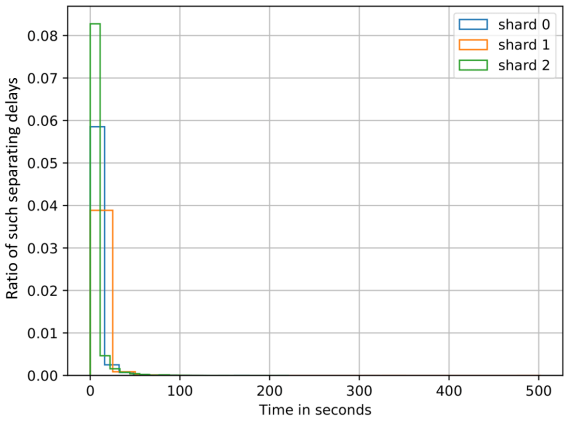
defined by the blockchain.

These transaction arrival times are used to simulate the creation of blocks in the algorithm. The created blocks will then be sent to the Reference shard C_R which will update correctly its history with the time window constraints. More precisely, it was explained in section 3.4 that a new block is created when one of the two following constraints cannot be satisfied anymore.

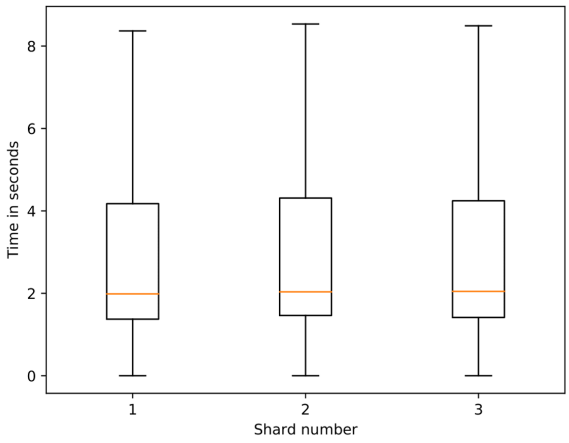
- A block cannot contain more than 400 transactions.
- A block cannot be built in more than 300 seconds.

These constraints were fixed to have fewer transactions per block than in the original Bitcoin protocol because, with DAVE, more blocks should theoretically be produced in a shorter amount of time. A block has to be validated in a maximum of five minutes to keep a short expected validation time for a given transaction and to prevent shards from keeping unvalidated blocks for too long.

We will now verify if the artificial arrival times of transactions are realistic and well handled by our sharding. First, in Fig. 4.1, we focus on the time separating the arrival of two successive transactions and then in Fig. 4.2, we observe how the number of transactions sent to each shard evolves with time.



(a) **Distribution of the delays separating the arrival of two successive transactions in each shard.** This long tail distribution shows that a wide majority of transactions arrive quite quickly after the preceding one in each shard. One can also note that the maximum time a shard has to wait is about 500 seconds.



(b) **Boxplots of the delays separating the arrival of two successive transactions in each shard.** We can directly see with these boxplots that all the three shards observe similar arriving delays behaviours with an average waiting time between two transactions of two seconds.

Figure 4.1: These two plots illustrate the fact that we do not introduce too much waiting time between two successive transactions in most of the cases but also that we end up with very similar arriving delays patterns between shards.

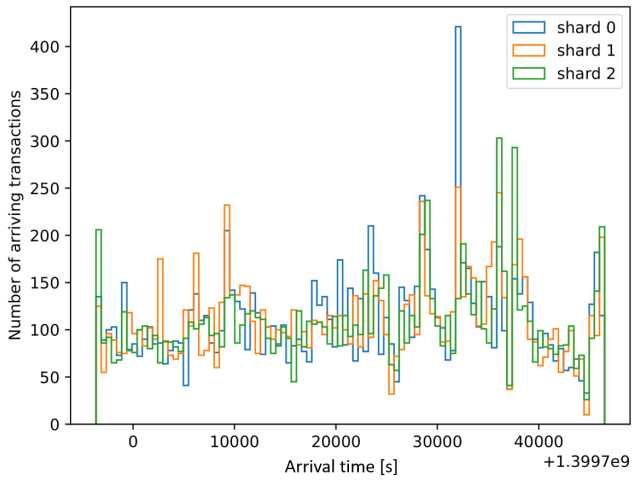


Figure 4.2: **Number of transactions arriving in each shard according to a discrete-time range.** We can learn from this graph that transactions' arrival times follow a distribution close to a uniform one. There is no "hot" moment when more transactions are sent except once for the shard 0. This is probably explained by the fact that sometimes we give the same time for all the transactions of a single block, leading to a large arrival of transactions at this precise time.

Fig. 4.2 is a third graph illustrating how we simulate an arriving time for the transactions coming from the Bitcoin blockchain. Like the two previous experiments, this one was done with a set of more than 32000 transactions coming from the blocks 300000 to 300100 of the real Bitcoin blockchain. These transactions are relevant because quite recent and representative of the real Bitcoin transactions users make currently and we do not have almost empty blocks in this interval as it is the case for the beginning of the blockchain. The graph on the left shows that we managed to simulate well-distributed arrival times, respecting the original order of transactions. Another important observation is that these artificial times do not introduce discrimination between shards. Also, since our data comes from the real Bitcoin blockchain, we can assume the times used were quite realistic.

All these experiments were done with three shards to keep the graphs readable but the conclusions were the same with more shards. We will now see how we can choose the parameters of our sharding algorithm.

4.1.2 Parameters and main results of the sharding algorithm

The two main parameters, U2S_{coef} and β were optimized one by one while keeping the other fixed. In what follows, we observe the impact of modifying one parameter while the other stays at its final chosen value. Before finding these final values, the parameters were first set to quite acceptable values during the design of the algorithm and first experiments.

The U2S_{coef} parameter should help to reach an ideal load balance while β should be used to improve the number of cross-shard transactions. U2S_{coef} was the first parameter fixed.
For the experiment aiming to find the U2S_{coef} parameter leading to the best load balance, 32000 transactions were used as in the previous experiment. However, they were now sharded among six shards to make sure any problem in the load balance would be easily detected. The division of transactions among shards is analysed on 50 intervals containing 654 transactions each. As a consequence of this, the perfect number of transactions processed by one shard during an interval should be 109. The purpose of these experiments is to find a coefficient that would minimize the maximum load of the shards during the sharding to avoid congestion in the network. Fig. 4.3 was obtained with a random sharding.

A random sharding will simply choose a shard randomly for each incoming transaction.

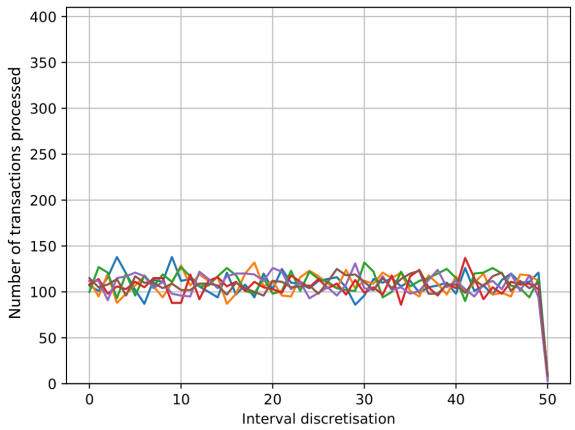
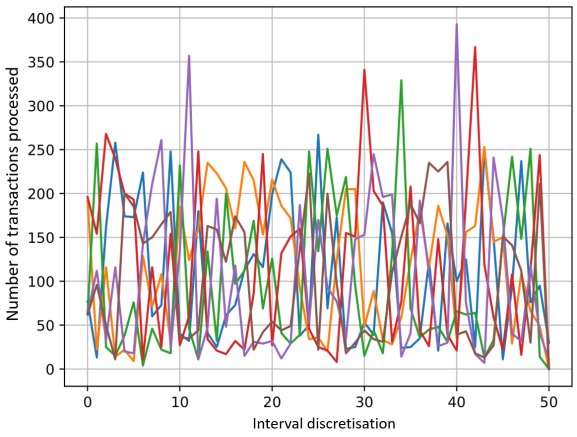


Figure 4.3: **Load balance of the shards through time for a random sharding.** The number of transactions processed by one shard during an interval varies from 90 to 120 most of the time.

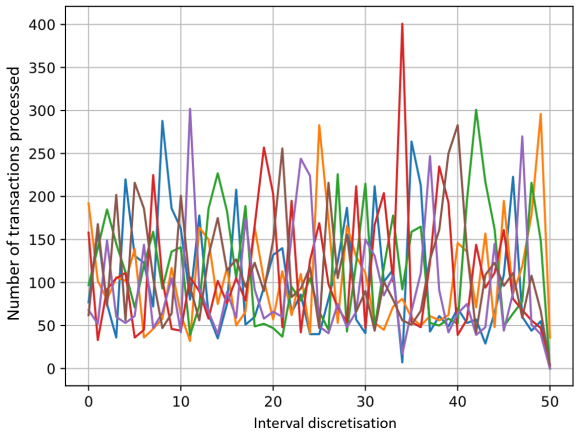
As we can see in Fig. 4.3, with a random sharding, a really nice load balance between shards is obtained. There is not exactly 109 transaction for each shard in every interval since the sharding is random and not a deterministic one optimizing the load balance. In fact, it should still be almost optimal if the random function of python used is really random. Any shard is supposed to have the same probability of being chosen for each transaction, making the probability distribution uniform. We can therefore consider this graph as an optimal objective we will not be able to reach with our sharding algorithm in term of load balance since our probability distribution is based on scores rather than being uniform.

We will now observe how, on the same data, DAVE behaves compared to a random sharding in terms of load balancing depending on the value of U2S_{coef}. This is depicted in the six graphs of Fig. 4.4[2].

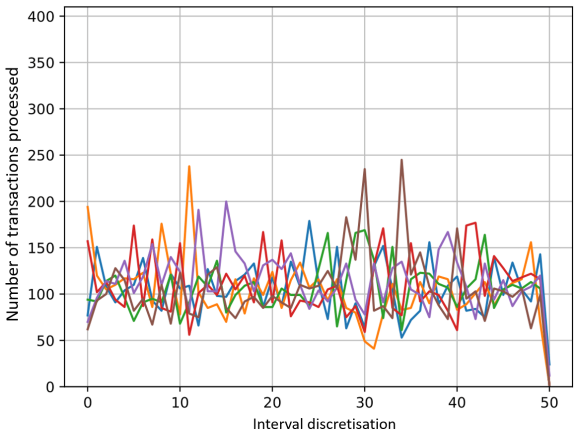
[2]The number of transactions processed by a given shard during the intervals is represented with a specific color but there is not necessarily a relation between the colors of different graphs.



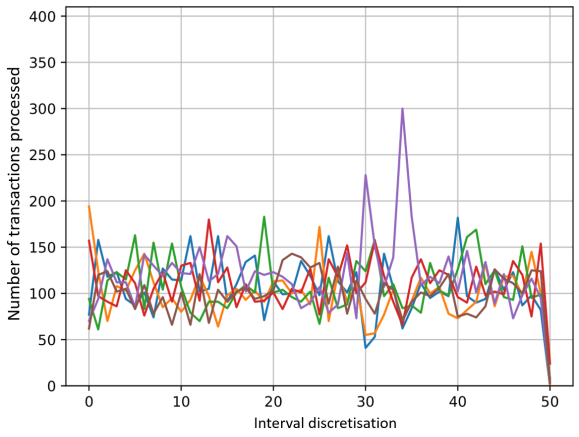
(a) **Load of the shards through time for a U2S$_{\text{coef}}$ of 50.**



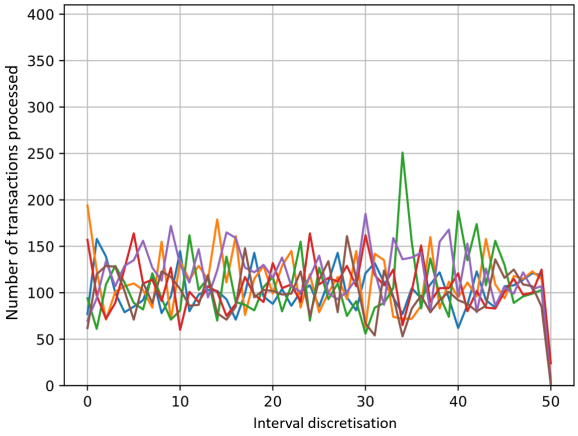
(b) **Load of the shards through time for a U2S$_{\text{coef}}$ of 5.**



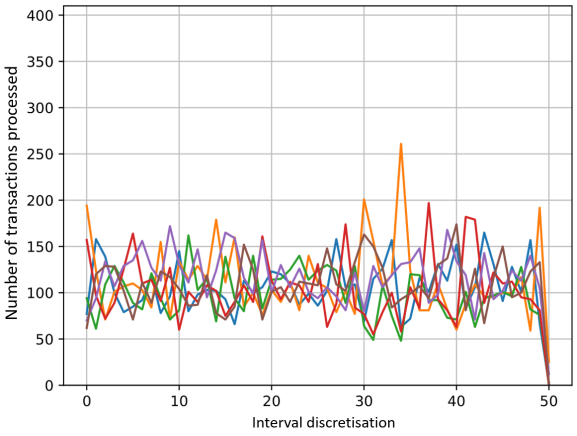
(c) **Load of the shards through time for a U2S$_{\text{coef}}$ of 0.5.**



(d) **Load of the shards through time for a U2S$_{\text{coef}}$ of 0.05.**



(e) **Load of the shards through time for a U2S$_{\text{coef}}$ of 0.005.**



(f) **Load of the shards through time for a U2S$_{\text{coef}}$ of 0.0005.**

Figure 4.4: Evolution of the load balance for different values of the U2S$_{\text{coef}}$ parameter in our sharding algorithm.

What we see in Fig. 4.4 is that the load balancing tends to improve as $U2S_{coef}$ decreases. This might be counter-intuitive because this coefficient is supposed to give more weight to the load balancing criterion. However, there is an explanation for this. The U2S score we designed does not fit well with the A2S score. The A2S score already takes into account how many times a shard was selected for the last transactions. The best way to use the U2S score is when the A2S score is not helpful at all. This explains why $U2S_{coef}$ has to be so small. It is because as soon as we have an A2S score, the U2S score should not interfere. A transaction will have a useless A2S score when it has no history (or not a useful one in terms of sharding) in the time window. In such situations, we have no clue on how to reduce the number of cross-shard transactions. The best strategy is to use the U2S score at this precise moment to optimize the load balance when the number of cross-shard transactions cannot be optimized.

We have in Fig. 4.5 a proof that the U2S score is only used when the A2S score is not. Indeed we reduced here even more the value of the parameter $U2S_{coef}$ and the sharding does not change at all. This is also true for smaller values of $U2S_{coef}$. So it was logical to fix the parameter $U2S_{coef}$ to 0.0005. Then the value of the β parameter still needs to be fixed. Since from now the A2S and U2S scores are separated in the placement decisions, the improvement in load balance achieved thanks to the U2S score will not be impacted by the choice of β. Therefore a Pareto graph should help to identify which β has the best impact on the two objectives.

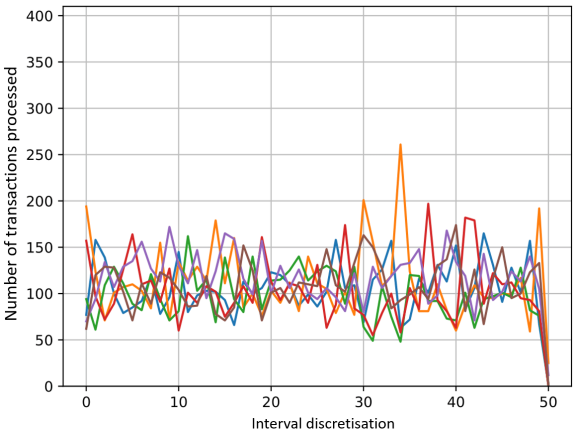


Figure 4.5: **Load of the shards through time for a $U2S_{coef}$ of 0.00005.**

The two objectives of the Pareto graph will be the minimization of the cross-shard percentage as well as the load balance score mentioned in the problem statement. We will now precise in equation 4.1 how exactly a load balance score is computed, based on the idea given in the problem statement. The idea was to use subsets of transactions rather than directly compute a score on the whole set of transactions sharded. We will then build a load balance score based on all the subsets with the following variables.

- K : number of shards used
- S : set of K shards = $\{S_1,S_2,...,S_K\}$
- I : number of subsets of transactions
- T_i : i^{th} subset of 50∗K transactions
- n_{ij} : number of transactions from T_i placed in the j^{th} shard = $\sum_{t \in T_i \wedge t \in S_j} 1$
- $\tilde{n_i}$: average of the n_{ij} on the number of shards = $\frac{1}{K} \sum_{j=1}^{K} n_{ij}$

Then we can define the load balance score as follows:

$$\text{Load balance score} := \frac{1}{I}\sum_{i=1}^{I}\frac{1}{K}\sum_{j=1}^{K}|n_{ij} - \tilde{n_i}| \tag{4.1}$$

So this is a global indicator of the average load balance among shards. But considering the choice of the U2S_{coef}, we ensured that it would minimize the maximum shard loads.

Several Pareto graphs were built on several ranges of 200 blocks to see if some coefficient β was better on average. Fig. 4.6, is one example of the kind of graphs we obtained. For this experiment, we used a range of 200 blocks from the block 300000 to the block 300200 and containing more than 60000 transactions. The parameter U2S_{coef} was fixed to 0.0005 and we used six shards. With the A2S score, the placement decision of a transaction is based on its neighbors in the address graph G_2 but also the neighbors of these neighbors. The coefficient β represents the relative weight we want to give to these second-order neighbors in comparison to the first-order ones. Therefore, we tested values between 0 and 1 for β since it seemed logical to give less weight to the addresses less related to the transactions we are sharding.

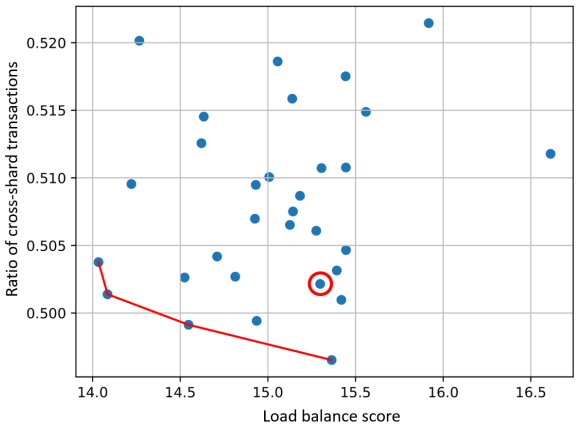


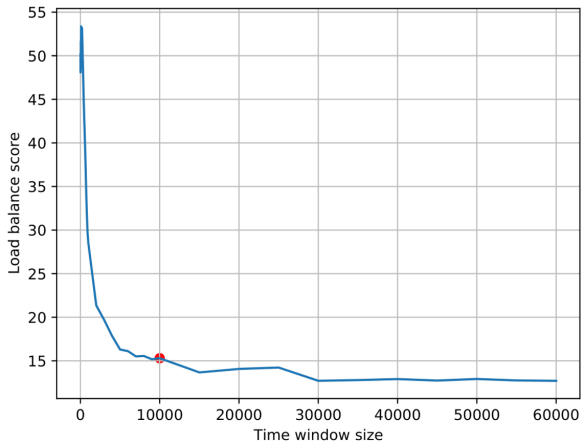
Figure 4.6: **Pareto optimization**

The obtained Pareto graphs showed that the best value of β was case-dependent. Indeed, we did not always get the same results for different ranges of blocks. However, what we could observe is that the choice of β did not influence by a lot our results in terms of both cross-shard percentage and load balance. So we decided to keep a value of β that was not too specific, to avoid overfitting but also that lead to average results in these simulations. That is why we fixed our coefficient β to 0.45, which corresponds to the dot circled in Fig. 4.6.

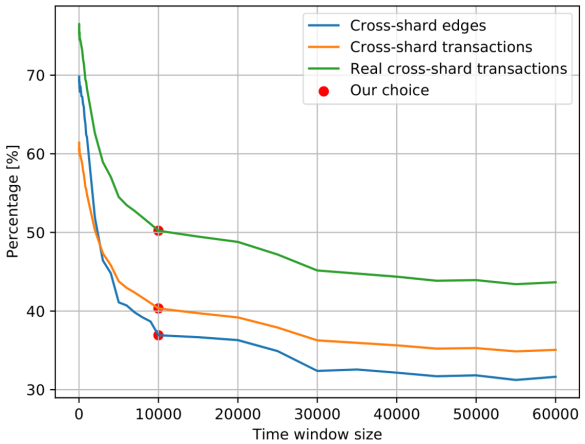
All the scores parameters are now fixed, so U2S_{coef} to 0.0005 and β to 0.45, but the time window parameter of the model itself still needs to be fixed. As mentioned in section 3.2, we have four time window constraints to respect. In fact, these time windows are all expressed based on the first one, the maximum number of transactions that G1 can contain. The time window used for addresses aims to keep in memory 1.5 times more addresses than the maximum number of transactions. It seems logical to do so since several addresses are used in one transaction (at least two but an address might theoretically be reused). The maximum number of transactions was fixed to 10000 transactions so this gives a maximum of 15000 addresses.
Then the time window based on blocks for dictionaries has a value equal to the maximum number of transactions in memory divided by 400, the maximum size of a block. This window is applied in addition to the first ones, this means that the dictionaries based on blocks will keep in memory all the information about transactions and addresses already kept in memory in other data structures but also from the previous 25 blocks.

Since the two last windows are expressed in function of the first ones, we just had to choose the size of the main window, which is a number of transactions. We analyzed the consequences of a variation of this time window size in Fig. 4.7 to make our choice of 10000 transactions. This choice corresponds to the red dots on the graphs. The experiments were done with six shards on more than 60000 transactions.
The curves of Fig. 4.7b represent different percentages of crossings between shards. We have the percentage of cross-shard edges between transactions. These edges are the ones we would find in a transaction graph. Then the percentage of cross-shard transactions is computed by considering a transaction as being cross-shard if one or more of its input transactions are in other shards. Finally, we also have what we called the real cross-shard transactions percentage. It is computed by ignoring transactions from which we did not know the input transactions and which were therefore processed as coinbase transactions. We think it is interesting to analyze the percentage of cross-shard transactions without considering such transactions simply because since we do not know their inputs, they will naturally not be cross-shard. It is important to note that we talk here about transactions for which we do not know the inputs, not the ones for which the inputs are not in the time window anymore. Some input transactions are unknown because we start our sharding in the middle of the real Bitcoin blockchain and so we have to ignore the history preceding our starting block.

Naturally, as the time window size increases, the performances of DAVE in terms of load balancing and inter-shard minimization will only get better, as we can see in Fig. 4.7. Indeed with more transactions taken into account, we can take better decisions however they require more time and memory. The time needed increases linearly with the size of the time window.
To have a good compromise between the time and memory needed and the results obtained, we fixed the main time window to 10000 transactions. This choice is justified by the fact that with a time window larger than 10000 transactions, we get much less improvement in the scores for the same increase in the time window size.



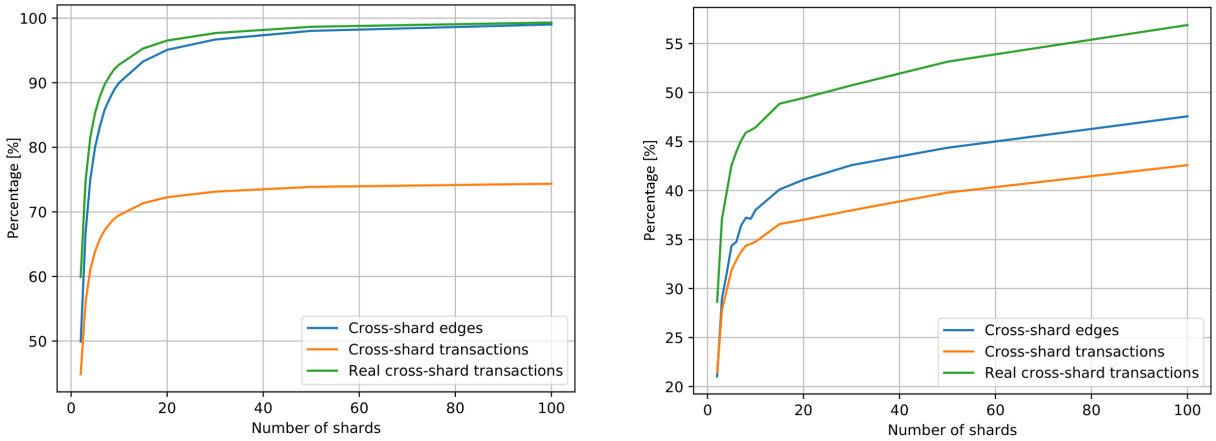
(a) **Evolution of the load balance score depending on the size of the time window.**



(b) **Evolution of the percentages of crossings depending on the size of the time window.**

Figure 4.7: Impact of the time window size on our results.

With the graphs from Fig. 4.7, we see that with the main time window fixed to 10000 transaction, we managed to obtain with DAVE a load balance score of **15** and only **50**% of real cross-shard transactions. This can be compared to a load balance score around **5.15** and **90**% of real cross-shard transactions for a random sharding to understand how important are the improvements done. The conclusion here is that with DAVE we only get a slightly poorer load balance to reduce the percentage of cross-shard transactions by 1.8. In the next experiment, we evaluate how the load balance score and the percentage of cross-shard transactions evolve with the number of shards used and again we compare these results with a random sharding. The cross-shard percentages analysis is done in Fig. 4.8.



(a) **Percentage of crossings in function of the total number of shards used in a random sharding.**

(b) **Percentage of crossings in function of the total number of shards used by DAVE.**

Figure 4.8: Comparison of the percentage of cross-shard transactions obtained with DAVE algorithm or a random placement depending on the number of shards used.

We made the experiments of Fig. 4.8 with six shards and the range of blocks going from block 300000 to 300100, so with more than 32000 transactions. We can observe the number of cross-shard transactions tends to grow with the number of shards used in both algorithms. Yet the main important thing is that we always manage to reduce the number of cross-shard transactions by the factor of almost 1.8 with our algorithm in comparison to a random sharding. This is a huge gain that could improve the performances of existing sharded blockchains a lot.
Then, even if this is not represented with graphs, we can also analyze on the same range of blocks what are the corresponding load balance scores of both DAVE and a random sharding depending on the number of shards. The load balance score tends to stay constant as the number of shards increases because we take the averages on more and more shards. The values obtained were very similar to the ones obtained before, so 5.15 for a random sharding and 15 for our algorithm. DAVE keeps a larger load balance score than the random one but this was expected since a random sharding is supposed to maintain an almost optimal load balance between shards no matter how many shards are used.

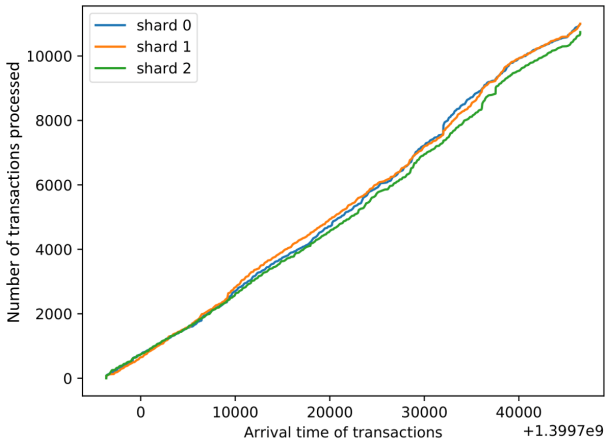
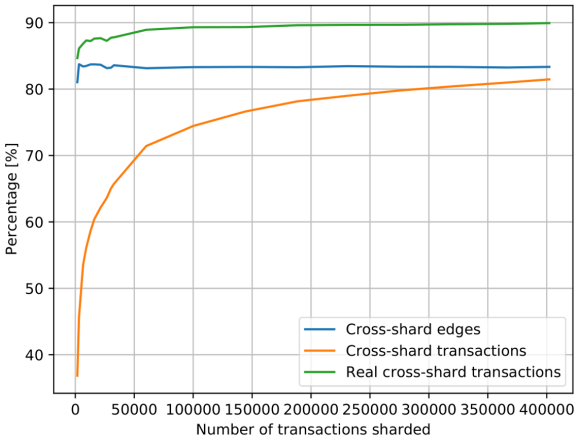


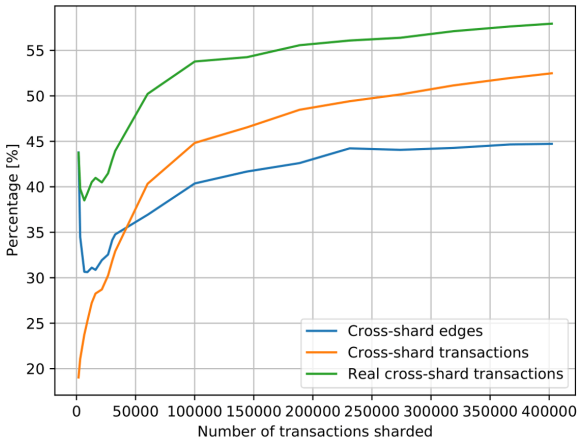
Figure 4.9: **Evolution of the number of transactions processed by each shard through time.**

Fig. 4.9 is another way to see how well our sharding keeps an acceptable load balance between the shards. We made here an experiment with three shards again on our range of 100 blocks. What we can see is that at the beginning the load balance is excellent. Indeed at this time, we can optimize it because we do not have a lot of history yet. Afterwards, we start to optimize more efficiently the number of cross-shard transactions but the load balance stays totally acceptable as the three curves always stay quite close to each other.

A last interesting experiment will be to analyze the results of our algorithm depending on the number of transactions sharded and compare this with a random sharding. We use six shards to obtain Fig. 4.10 which analyses the cross-shard percentages and shows that again DAVE significantly outperforms a random sharding. However, the load balance score obtained with DAVE tends to increase with the number of transactions sharded while the one obtained with a random sharding stayed around 5.15 no matter how many transactions were sharded. Yet, this should not impact the performances of DAVE significantly since the load balance may not be perfect but still stays acceptable.



(a) **Percentage of crossings depending on the number of transactions sharded randomly.**



(b) **Percentage of crossings depending on the number of transactions sharded by DAVE.**

Figure 4.10: Comparison of the sharding performances in term of crossings.

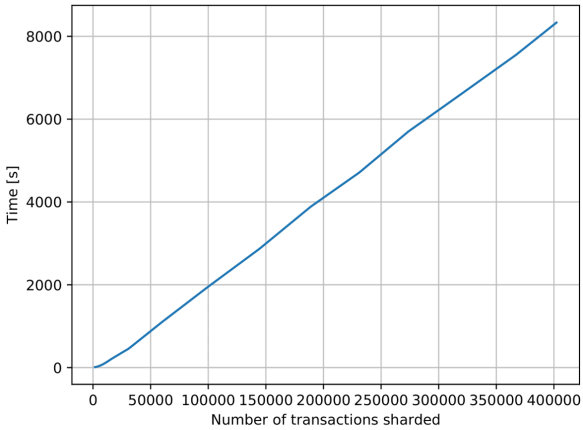
4.1.3 Complexity of the sharding algorithm

Now we will talk about the complexity of our algorithm. The key is that it should be used in a real system so we cannot allow too slow decisions. The miners and more precisely the

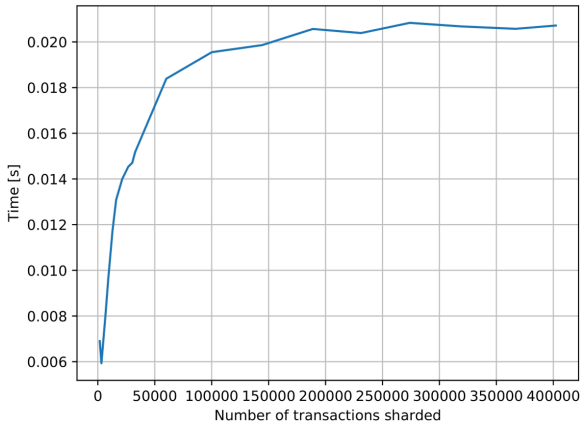
Reference shard's members should be able to execute DAVE in a very short amount of time to shard one transaction. Slow decisions could affect both the throughput and the latency of the network, which would make our work useless.

DAVE is invoked once every time a transaction needs to be sharded. Thus, it is interesting to analyze the average time it takes to take one placement decision as well as the total time needed to shard a given amount of transactions. Fig. 4.11a shows that the time needed to execute our sharding increases linearly with the number of transactions to shard while in Fig. 4.11b we notice the average time needed to shard a single transaction tends to stabilize around 20 milliseconds. In order to obtain those results, we used the transactions coming from blocks 300000 to 301000. This is a representative subset of the whole blockchain, comparable to what we could expect today, which does not contain empty blocks. The total time needed should therefore continue to increase linearly as time goes on since it is cumulative. However, for the time needed to shard a single transaction, we can see that it tends to stay constant after a stabilization period. During this period, the very first transactions sharded still have an impact on the average time. Indeed when we shard the first transactions, the window used by the Reference shard is almost empty and placement decisions are then taken much faster. In the long term, DAVE should continue to shard a transaction on average every 20 milliseconds.

These simulations were executed on a computer with an Intel Core i7-6700HQ processor running at 2.6 GHz using 8 GB of RAM, running Ubuntu version 18.04.5 LTS. The algorithm could be executed much faster on more efficient machines like the ones miners usually work with. In addition to that, further code optimization can be performed. These results could easily be outperformed if the code was written in a language like C but also with more suited data structures. Knowing this, we can conclude that a linear time complexity combined with an upper bound of 20 millisecond per transaction is enough to claim DAVE could be used in a real network.



(a) **Total time needed by DAVE depending on the number of transactions to shard.**



(b) **Average time needed by DAVE to shard one transaction depending on the total number of transactions sharded.**

Figure 4.11: Analysis of the time complexity of our algorithm.

Now we also have to ensure the algorithm does not need too much memory to be used efficiently. It is not acceptable for the miners to store a large amount of data just to take a placement decision. DAVE should need only a bounded amount of memory to make an efficient sharding. The size of our graphs and dictionaries should be the only point to verify since, in comparison to these, other variables used in our code should not impact the memory significantly.
We know the time window constraints limit the sizes of these data structures. Both the graphs' and the dictionaries' sizes are bounded and therefore we can ensure our algorithm will never need more than a fixed amount of memory. Its space complexity is then constant with the number of transactions sharded. Again this enables us to claim that our algorithm could be used in a real network.

4.2 Simulation of sharding in a real network

In this section, we further investigate our algorithm by running real-world experiments. As explained before, we used a sharded blockchain called Byzcuit, which we can feed with a text file specifying all the transactions we want to use in our test. This file explicitly states the input and output shards of each transaction. The steps followed to run the tests are illustrated in Fig. 4.12, first, we have to execute DAVE on the real blockchain, then we can produce a text file defining the transactions as sharded by DAVE (= transaction file), so that finally we can launch Byzcuit with this transaction file.

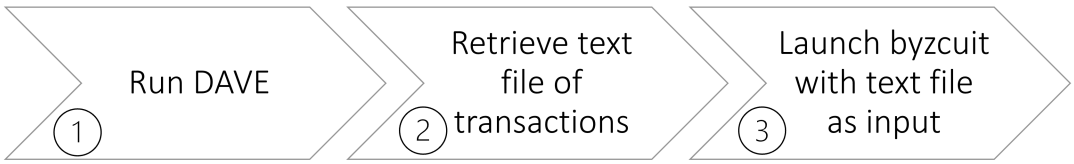


Figure 4.12: Steps of simulation

All the tests of this section have been realized with the following parameters: we set the `awsregion` to eu-west-2, the `ImageId` of the instances is ami-e5a35782 and the `Imagetype` is t2.medium. All our tests, simulations and results for this section can be found in our dedicated GitHub repository[3].

The very first results we are going to show is a comparison of a random placement of transactions into shards and a placement done by DAVE. In figure 4.13, we can observe that as expected, augmenting the number of shards will increase the performance.
Each point on the graph was obtained by running DAVE on the range of blocks 300000 to 300100, which corresponds without the transactions considered as coinbase by DAVE to 24512 transactions. We took the average of the results on 10 runs of Byzcuit. Those 10 runs correspond to redoing 10 times the steps 3 in Fig. 4.12, so keeping the same

[3]https://github.com/alxd112/byzcuit.git

transaction file each time and therefore the same sharding of the same transactions. The vertical lines correspond to the standard deviation of the 10 runs. As we use the real transactions from the blockchain, the number of inputs and outputs per transaction is variable. We thought it might be interesting to mention that the more inputs and outputs there are in a transaction the harder it is to process. This analysis is available in the appendix F.

We can also observe that for the throughput the blue line (DAVE placement) is most of the time above the other one (random placement) while for the latency the blue line is mostly below the random line. In fact, for 10 shards, DAVE increases the throughput by **11.6**% and reduces the latency by **6.17**% compared to the random placement. Such results are encouraging regarding the work presented in this thesis since the goal was to increase the throughput and reduce the latency perceived by users. However, we can observe that the standard deviation (represented on the graph by the vertical lines) is non-negligible.

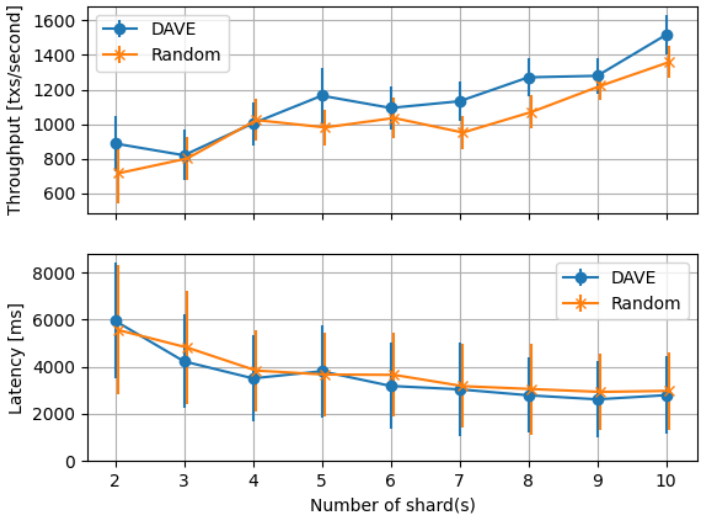


Figure 4.13: **Evolution of the throughput and latency in function of the number of shard(s) for a random sharding and DAVE**. The throughput and latency on the graphs are the mean throughput and latency.

This is due to the fact that Byzcuit has randomness involved in the way transactions are processed. Furthermore, after discussing with most of the people that were involved in the project, we learned that the Byzcuit project was not developed with the objective to run in production thus, they did not spend a lot of time to make it perfect in term of performances and reproducibility. This is why a lot of variances is observable. The distribution of the results is available in the appendix E.

We can observe another interesting result in the graph of Fig. 4.14, we see that the more shards used, the more difference there is between a random sharding and DAVE in terms of cross-shard transactions. This might explain why in the graph of Fig. 4.13 we observe that globally DAVE perform better than random the more shards there is.
Concerning the latencies, we observe that there are no real difference between random and DAVE. We just observe that DAVE is slightly better when more than five shards are used.

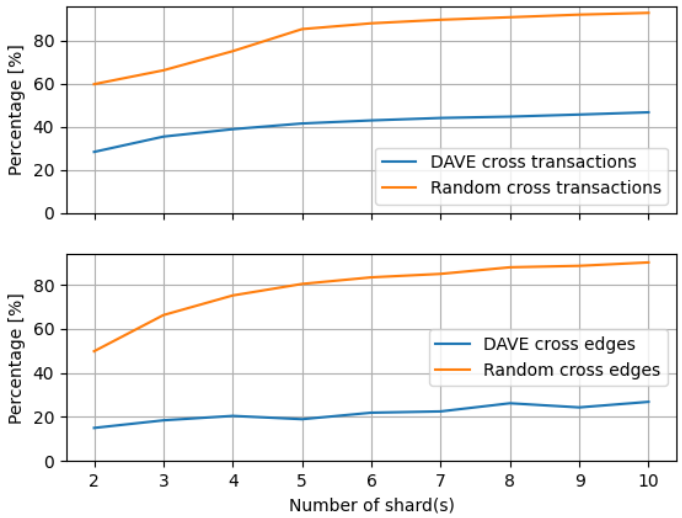


Figure 4.14: **Evolution of the percentages of cross-shard transactions and edges in function of the number of shard(s) used for a random sharding or for DAVE**. The results in these graphs are linked to the graph of Fig. 4.13, as each point of the graphs is based on the same transaction file.

In addition to that, we wanted to see what result we could get if we managed to have an algorithm that gives the same percentage of cross-shard transactions but with an excellent load balancing (comparable to a random sharding). To obtain the results as shown in Fig. 4.15, we artificially created for Byzcuit text files of transactions with a given percentage of them being inter-shard. Each point on this graph is the result of 10 runs on the same transactions file made of 50000 transactions. The blue curve named *DAVE percentages* corresponds to the results of an artificial placement of transactions that has the same percentage of cross-shard transactions as DAVE but with a much better load balancing.

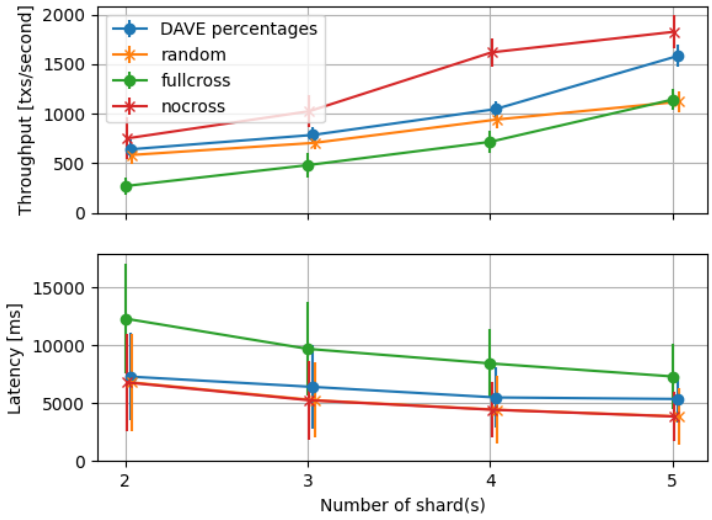


Figure 4.15: **Evolution of the throughput and latency of various modes in function of the number of shard(s)**.

Then, the green curve named *fullcross* corresponds to a sharding where every transaction is cross-shard transactions. While the red curve *nocross* corresponds to a sharding where every transaction is intra-shard transactions. What we see here is that the more shards we use, the more cross-shard transactions there are logically, and the more the *DAVE percentage* is efficient compared to the random sharding.

With all these observations, we might conclude that DAVE is more interesting than a simple random sharding. Dave significantly reduces the number of cross-shard transactions by almost 50% while ensuring an acceptable load balance among shards. This improves the overall throughput and latency of the network. All of this while being **deterministic**, **fast**, **verifiable** and computed by the miners.

Chapter 5

Conclusion

We presented in this thesis a new efficient sharding method. It improves the scalability of the original Bitcoin protocol, just like the existing sharded blockchains while being more efficient than them and offering interesting security properties.

To develop such a solution, we used an Interactions and a Cooperation heuristic to predict future interactions of transactions. Then we also took inspiration from the scores used in Optchain to design our scores. These scores being an Address-To-Shard score A2S and a Use-of-the-Shard score U2S that we combine to obtain a Fitness Score FS. It enables to reduce the number of cross-shard transactions by almost 50% while maintaining an acceptable load balance between the shards. The most important thing is that these results are achieved thanks to deterministic placement decisions. This, in addition to the use of a time window updated at each validated block, ensures DAVE's decisions to be verifiable. Furthermore, both the placement decisions and their verification are done fast and do not require an important memory usage.

We justified our choices concerning the algorithm conception and illustrated the results achieved through several experiments. Then we also explained how DAVE could be introduced in a real network already using sharding like RapidChain. It would involve small modifications which should not impact the performances of the network. We tested with Byzcuit what performances could be expected if our algorithm was used to shard transactions. What we could see is that even if the network was unstable and we only used an approximation of what could be done, we already obtained better performances in term of both throughput and latency than classical random shardings.

Given the results, we believe some companies could consider doing more research based on DAVE. First, by analyzing its performances in a real network, a network that would have been developed to run in production, unlike Byzcuit. By doing so, the measures would be more reliable. Then, increasing the time-window limits could also be an option to improve the performances of DAVE. All this could be done so that at the end, they could use DAVE for their sharded blockchain. On the other hand, DAVE works only with UTXO based blockchains and is therefore not compatible with other models so this could be a limitation.

In a few words, we proposed here a new way to place transactions into shards deterministically and in a verifiable manner. The placement decisions are taken in a bounded amount of time and require a bounded amount of memory, just as the verification of these decisions. The number of cross-shard communication is reduced by a lot in comparison to previous work and the load balance among shards stays acceptable, which leads to an improvement in both throughput and latency.

Bibliography

[1] Satoshi Nakamoto. A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[2] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. In Alfred J. Menezes and Scott A. Vanstone, editors, *Advances in Cryptology-CRYPTO' 90*, pages 437–455, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[3] ESPACE EMERAUDE. https://www.espace-emeraude.com/chaine-de-levage-3t200-20m.html#productCaracteristics.

[4] FOTOMELIA. https://fotomelia.com/?download=amitie-ami-friend-friends-copain-copine-equipe-groupe-images-gratuites#.

[5] JavaTpoint. Blockchain tutorial. https://www.javatpoint.com/blockchain-tutorial, 2011-2018.

[6] Indeed. What is a peer-to-peer network? https://www.indeed.com/career-advice/career-development/what-is-a-peer-to-peer-network, 2020.

[7] Nicolas Cary Peter Smith, Ben Reeves. Blockchain.com. https://www.blockchain.com/, 2011.

[8] D. Di Francesco Maesa, A. Marino, and L. Ricci. Uncovering the bitcoin blockchain: An analysis of the full users graph. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 537–546, 2016.

[9] Andreas M. Antonopoulos. *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*. O'Reilly Media, Inc., December 2014.

[10] Tushar Raj Cornelius Schumacher, Will Binns and small contributors. bitcoin developer. https://developer.bitcoin.org/devguide/transactions.html, 2019.

[11] O'Reilly Media. Chapter 7. The Blockchain. https://www.oreilly.com/library/view/mastering-bitcoin/9781491902639/ch07.html, 2021.

[12] Dorit Ron and Adi Shamir. Quantitative analysis of the full bitcoin transaction graph. In *International Conference on Financial Cryptography and Data Security*, pages 6–24. Springer, 2013.

[13] Sarah Meiklejohn, Marjori Pomarole, Grant Jordan, Kirill Levchenko, Damon McCoy, Geoffrey M Voelker, and Stefan Savage. A fistful of bitcoins: characterizing payments among men with no names. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 127–140, 2013.

[14] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. Deanonymisation of clients in bitcoin p2p network. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 15–29, 2014.

[15] QingChun ShenTu and JianPing Yu. Research on anonymization and de-anonymization in the bitcoin system. *arXiv preprint arXiv:1510.07782*, 2015.

[16] Anil Gaihre, Yan Luo, and Hang Liu. Do bitcoin users really care about anonymity? an analysis of the bitcoin transaction graph. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 1198–1207. IEEE, 2018.

[17] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *13th {USENIX} symposium on networked systems design and implementation ({NSDI} 16)*, pages 45–59, 2016.

[18] George Danezis and Sarah Meiklejohn. Centrally banked cryptocurrencies. *arXiv preprint arXiv:1505.06895*, 2015.

[19] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30, 2016.

[20] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[21] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 931–948, 2018.

[22] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778*, 2017.

[23] Lan N Nguyen, Truc DT Nguyen, Thang N Dinh, and My T Thai. Optchain: optimal transactions placement for scalable blockchain sharding. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 525–535. IEEE, 2019.

[24] Jonas David Nick. Data-driven de-anonymization in bitcoin. Master's thesis, ETH-Zürich, 2015.

[25] D. Ermilov, M. Panov, and Y. Yanovich. Automatic bitcoin address clustering. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 461–466, 2017.

[26] Antoine Le Calvez. python-bitcoin-blockchain-parser. https://github.com/alecalve/python-bitcoin-blockchain-parser.

[27] Alberto Sonnino, Shehar Bano, Mustafa Al-Bassam, and George Danezis. Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 294–308. IEEE, 2020.

[28] miningPools.com. Consensus protocols: The algorithms that the blockchain runs on. https://miningpools.com/consensus-algorithms/.

[29] Amy Castor. A (short) guide to blockchain consensus protocols. https://www.coindesk.com/short-guide-blockchain-consensus-protocols, 2017.

Appendix A

Merkle Tree

The Merkle tree is essentially a binary tree associated with a given block. Every leaf is the hash of a transaction of the block and every parent in the tree is the hash of the combination of the hashes of the two children. In Fig. A.1, let's say that TX is the hash of the transaction number X in a block and H is a hashing function.

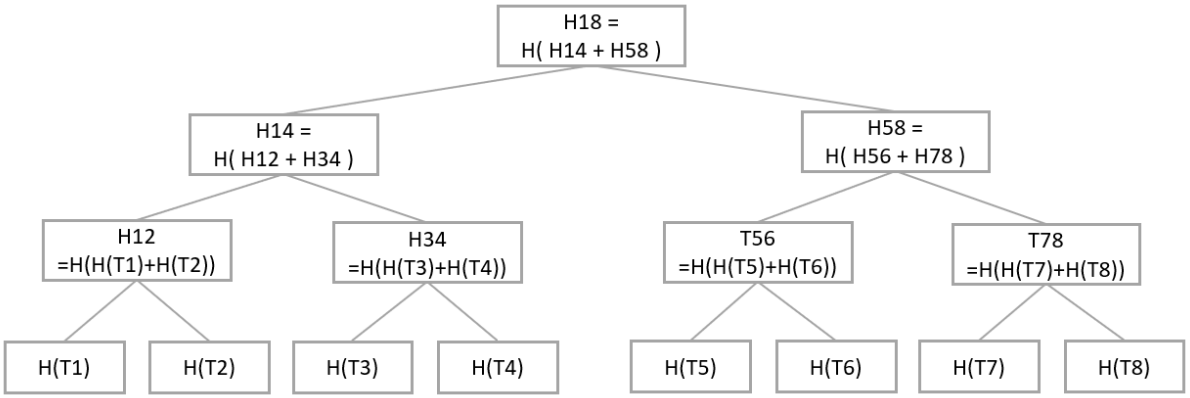


Figure A.1: Example of Merkle tree

From the explanation of the construction of such a data structure, it is obvious that if a transaction was to be changed in the block, its hash and several intermediary nodes of the Merkle tree would also change, therefore leading inevitably to a change of the root. Thus, it is very simple to check that a block did not change simply by remembering the root of the corresponding Merkle tree.
In addition to that, the Merkle tree offers the ability to check whether a transaction is in the associated block or not, very efficiently $\mathcal{O}(log_2(n))$ via a **proof of inclusion**, which is a tuple made of three elements: the hash of the transaction to verify, the hash of the root and finally all the siblings of the node in the path from the root to the transaction. On the Fig A.1, the proof of inclusion of the transaction T4 would be (H(T4), H18, [H(T3), H12, H58]). This way, we have all the information needed to recompute all the steps from the transaction to the root. If the computation of the intermediary nodes does not lead to the same root, this means the transaction is not in the block.

Appendix B

Alternative consensus mechanisms

Up until here, we have seen only one type of consensus method, the proof-of-work consensus. A series of other protocols exist. The PoW protocol required that every node compete to solve a hard puzzle. This allowed the network to stay secure as long as no attacker controlled 51% or more of the network computational power. The main disadvantage that results from this is that all the miners have to race to validate their block which consumes a huge amount of energy. One solution to this is to use the **proof of stake** consensus protocol (PoS [28]). In this protocol, only one miner will be allowed to create the next block and to work on it, all the others do not have to do anything. This implies a lot of energy savings for everybody. But, here the choice of the miners is a random choice among all the miners with a bigger weight to the miners holding a lot of the currency of the network. So, a miner holding 2% of the currency will have more chance to be selected than a miner with only 1% of the currency. Once a miner has been chosen, it has to put a part of its currency holdings at stake namely if he does something not ethical while creating the next block he will lose the currency he put at stake. Thus, another advantage of this protocol is that it is secure as long as no attacker control 51% or more of all the currency, which is not likely to happen.

Another interesting consensus protocol is the **proof of elapsed time** developed by Intel. It works similarly to PoW except that it consumes way less energy. Amy Castor explained that "instead of having participants solve a cryptographic puzzle, the algorithm uses a trusted execution environment (TEE) – such as SGX – to ensure blocks get produced in a random lottery fashion, but without the required work" [29]. The problem with this approach is that you are forced to trust Intel, which is not a good idea. Not because Intel is not a good company or anything but rather because we do not want to trust a third party.

Appendix C

Intra-shard consensus protocol for transactions (S-Bac) applied to our transaction placement strategy

This step is based on the S-BAC protocol developed for Chainspace which was explained in section 2.3.2 but we introduce several updates. When a transaction T is sent by C_R to shard 3 for example, the latter needs to make sure that the UTXOs the transaction uses cannot be double-spent. Thus, shard 3 sends the message "$prepare(T)$" to $f + 1$ nodes of the shards involved in this transaction because they contain the inputs of the transactions as shown in Fig. C.1. Let's say the transaction has the following form:

$$i1 + i2 \rightarrow o3$$

where i indicates an input and o an output, and the number after the letter indicates the shard in which the UTXO is stored (or in the case of output, in which it will be stored).

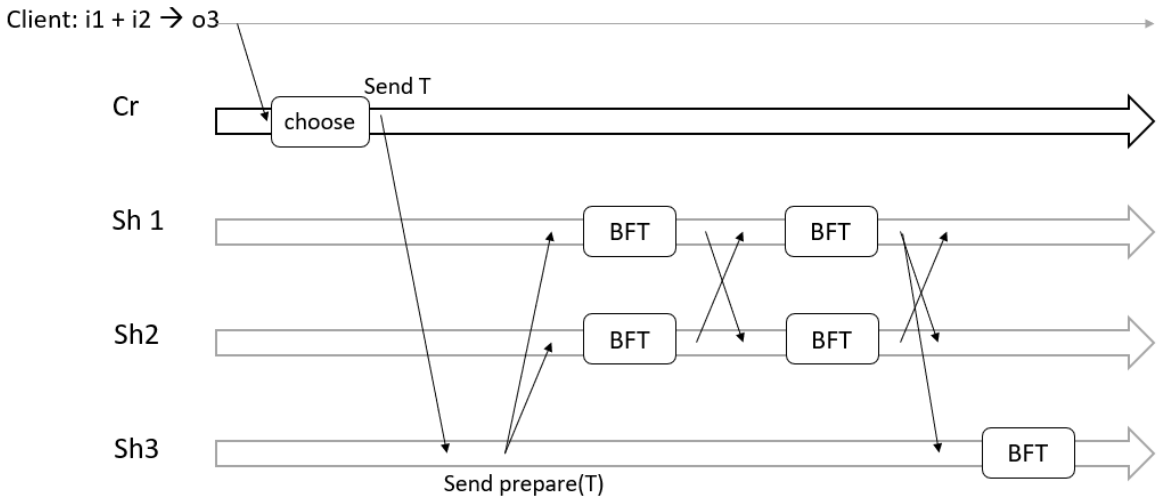


Figure C.1: Transaction example (inspired by Chainspace)

In this case, shard 3 will send "$prepare(T)$" to $f + 1$ nodes of shard 1 and 2, so that at least one honest node received the message as we guarantee the system is secure up to f dishonest nodes. As every node in a shard has the same history, their answer will be the same, either "*commit*" or "*abort*". If the decision is committed, the shard broadcasts the message "$prepared(T, commit)$" with $f + 1$ signatures proving the collective decision within a shard, to all other shard(s) concerned by the transaction T. In addition to that, the shard moves the states of the UTXO from *active* to *locked*, to prevent double-spending of the funds. If a shard decides to abort, whether it is because the UTXO is inactive, locked or does not exist, then the shard will let the other know about it with the message "$prepared(T, abort)$", which will make the other shards unlock their respective UTXOs for future transactions.
In our situation, let's say shard 1 and 2 accept the transaction. Thus, both of them let the other know about that, by sending the message "$prepared(T, commit)$" and they both move respectively $i1$ and $i2$ to the *locked* state.
Afterward, the shards receive the message to either abort or commit from the other shards involved. If there is even one single *abort* message received from another shard, then the shard will discard the transaction and move the states of its UTXOs back to *active*. But if it receives as many commits as there are shards concerned, then the shard will send the message "$accept(T, commit)$" to the other concerned shards so that they can move their respective UTXOs to the *inactive* state, meaning it has been spent. The shard also sends the message "$accept(T, commit)$" to the shard concerned by the output of the transaction, to let it know that it has to create the corresponding object and store it. In our case, this would be shard 3.
This paragraph explained how shards process transactions, in order to avoid double-spending problems.

Appendix D

IDA-Gossip protocol and sparsification

RapidChain presents an efficient communication protocol called IDA-Gossip that uses sparsification. It is inspired by the standard IDA protocol. Let's imagine a node wanting to send a message M to the d other nodes in its committee where $\phi\%$ of these nodes are corrupt. The protocol will divide the message into $(1-\phi)K$ chunks, to later obtain K chunks by using an erasure coding scheme. By doing so, even with ϕn corrupt nodes, the honest ones will be able to recover the message sent since the message can be reconstructed with only $(1-\phi)K$ chunks.
Now, in practice the kind of messages users will need to send during a consensus is composed of the Merkle hash tree. So, a user that would want to send the Merkle tree made of the following leaves: $M-1, M_2$,...,M_K would send all the leaves as well as a Merkle proof to each of its d neighbors (which are the other members of the committee) under the form of a set of $\frac{K}{d}$ chunks. Then, the neighbors will share the chunks they received between each other and in the end, the honest nodes will have received at least $(1-\phi)K$ chunks, which will allow them to recover the original Merkle tree. During the gossiping, each node can verify the validity of the received message by using the Merkle tree and the Merkle proof.

However, the authors of RapidChain did not stop there, they realized that this protocol was sending the hashes of the nodes close to the root in the Merkle tree very often. Remember, a Merkle proof has the following form $(H(i), H(root), path\ from\ H(i)\ to\ root)$ where the path is the biggest element. So, they decided to send every single intermediary node of the tree to only a subset of nodes of the network, ideally $(1-\phi)K$ nodes, such that at least one honest node has all the intermediary hashes. Then, those nodes will have to send the hashes to the remaining nodes so that they can verify their Merkle proof too. Thereby, the sparsification manages to reduce the communication but on the other hand, it introduces a slight risk of unsuccessful gossip of the message.

All the communication in the network uses the IDA-Gossip protocol. The authors say that sparsification should no be used for gossiping blocks as it is essential but they mention that instead it could be used by users when they send their transactions to the network. Because if the gossip failed, the user will just have to resend the Merkle Tree nodes later.

Appendix E

More Byzcuit results

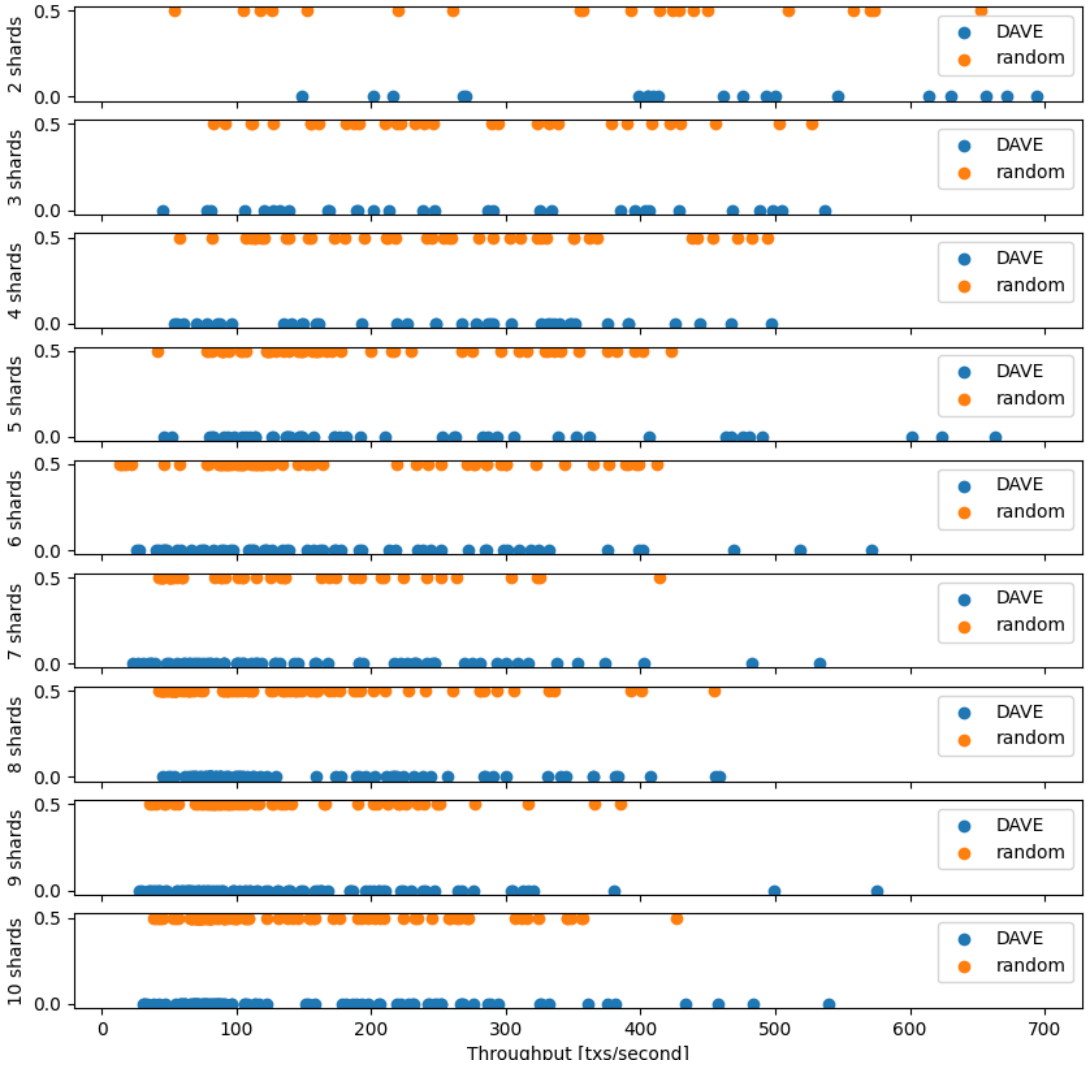


Figure E.1: **Distribution of the throughput observed in function of the number of shard used.** The values of these graphs are directly linked to the results of Fig. 4.13. Each subgraph shows the distribution of the throughput measured when simulating with the number of shards indicated on the y-axis.

Appendix F

Variation of the number of inputs and outputs for the Byzcuit simulations

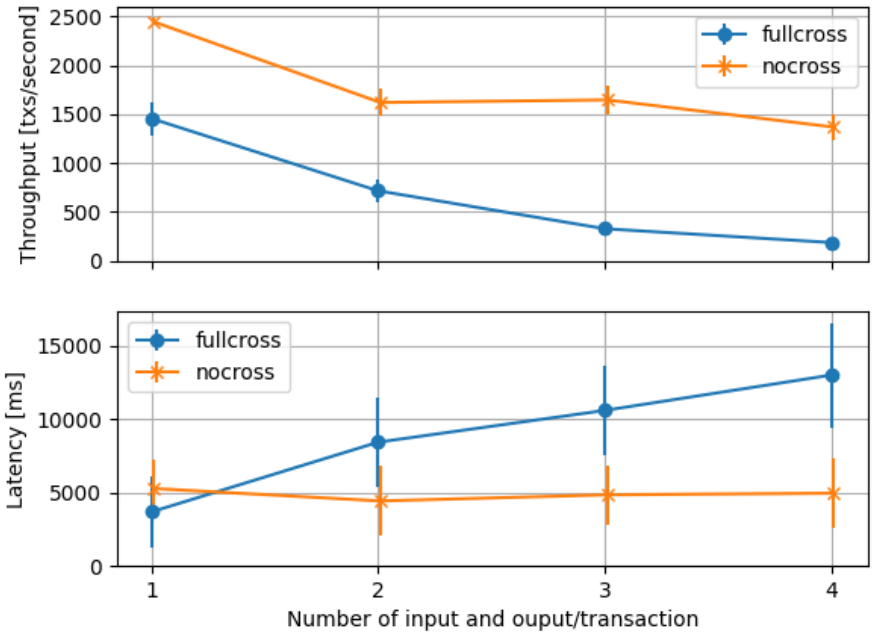


Figure F.1: **Evolution of the throughput and latency in function of the number of inputs and outputs in a transaction**. Those results have been obtained using four shards. The orange curve named *nocross* correspond to a network with only intra-shard transactions while the blue curve named *fullcross* corresponds to a network with only inter-shard transactions. We can see that the throughput and latency worsen when more inputs and outputs are used in a transaction. More importantly, we can also notice that the more inputs and outputs per transaction, the bigger the difference between intra-shard and inter-shard transactions.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**