Yggdrasil: Secure State Sharding of Transactions and Smart Contracts that Self-adapts to Transaction Load

Aimen Djari, Yackolley Amoussou-Guenou, Emmanuelle Anceaume, Sara Tucci Piergiovanni, Antonella Del Pozzo

▶ **To cite this version:**

Aimen Djari, Yackolley Amoussou-Guenou, Emmanuelle Anceaume, Sara Tucci Piergiovanni, Antonella Del Pozzo. Yggdrasil: Secure State Sharding of Transactions and Smart Contracts that Self-adapts to Transaction Load. 2022. hal-03793291v1

HAL Id: hal-03793291
https://hal.science/hal-03793291v1

Preprint submitted on 30 Sep 2022 (v1), last revised 6 Oct 2022 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Yggdrasil: Secure State Sharding of Transactions and Smart Contracts that Self-adapts to Transaction Load

Aimen Djari
Université Paris-Saclay, CEA, List
Palaiseau, France

Yackolley Amoussou-Guenou
Université Paris-Saclay, CEA, List
Palaiseau, France

Emmanuelle Anceaume
CNRS / IRISA
France

Sara Tucci-Piergiovanni
Université Paris-Saclay, CEA, List
Palaiseau, France

Antonella Del Pozzo
Université Paris-Saclay, CEA, List
Palaiseau, France

ABSTRACT

Praesent imperdiet, lacus nec varius placerat, est ex eleifend justo, a vulputate leo massa consectetur nunc. Donec posuere in mi ut tempus. Pellentesque sem odio, faucibus non mi in, laoreet maximus arcu. In hac habitasse platea dictumst. Nunc euismod neque eu urna accumsan, vitae vehicula metus tincidunt. Maecenas congue tortor nec varius pellentesque. Pellentesque bibendum libero ac dignissim euismod. Aliquam justo ante, pretium vel mollis sed, consectetur accumsan nibh. Nulla sit amet sollicitudin est. Etiam ullamcorper diam a sapien lacinia faucibus.

PVLDB Reference Format:
Aimen Djari, Yackolley Amoussou-Guenou, Emmanuelle Anceaume, Sara Tucci-Piergiovanni, and Antonella Del Pozzo. Yggdrasil: Secure State Sharding of Transactions and Smart Contracts that Self-adapts to Transaction Load. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://anonymous.4open.science/r/Yggdrasil-11E5.

1 INTRODUCTION

Praesent imperdiet, lacus nec varius placerat, est ex eleifend justo, a vulputate leo massa consectetur nunc. Donec posuere in mi ut tempus. Pellentesque sem odio, faucibus non mi in, laoreet maximus arcu. In hac habitasse platea dictumst. Nunc euismod neque eu urna accumsan, vitae vehicula metus tincidunt. Maecenas congue tortor nec varius pellentesque. Praesent imperdiet, lacus nec varius placerat, est ex eleifend justo, a vulputate leo massa consectetur nunc. Donec posuere in mi ut tempus. Pellentesque sem odio, faucibus non mi in, laoreet maximus arcu. In hac habitasse platea dictumst. Nunc euismod neque eu urna accumsan, vitae vehicula metus tincidunt. Maecenas congue tortor nec varius pellentesque. Pellentesque bibendum libero ac dignissim euismod. Aliquam justo ante, pretium vel mollis sed, consectetur accumsan nibh. Nulla sit amet sollicitudin est. Etiam ullamcorper diam a sapien lacinia faucibus.

Praesent imperdiet, lacus nec varius placerat, est ex eleifend justo, a vulputate leo massa consectetur nunc. Donec posuere in mi ut tempus. Pellentesque sem odio, faucibus non mi in, laoreet maximus arcu. In hac habitasse platea dictumst. Nunc euismod neque eu urna accumsan, vitae vehicula metus tincidunt. Maecenas congue tortor nec varius pellentesque. Pellentesque bibendum libero ac dignissim euismod. Aliquam justo ante, pretium vel mollis sed, consectetur accumsan nibh. Nulla sit amet sollicitudin est. Etiam ullamcorper diam a sapien lacinia faucibus.

Praesent imperdiet, lacus nec varius placerat, est ex eleifend justo, a vulputate leo massa consectetur nunc. Donec posuere in mi ut tempus. Pellentesque sem odio, faucibus non mi in, laoreet maximus arcu. In hac habitasse platea dictumst. Nunc euismod neque eu urna accumsan, vitae vehicula metus tincidunt. Maecenas congue tortor nec varius pellentesque. Pellentesque bibendum libero ac dignissim euismod. Aliquam justo ante, pretium vel mollis sed, consectetur accumsan nibh. Nulla sit amet sollicitudin est. Etiam ullamcorper diam a sapien lacinia faucibus.

Praesent imperdiet, lacus nec varius placerat, est ex eleifend justo, a vulputate leo massa consectetur nunc. Donec posuere in mi ut tempus. Pellentesque sem odio, faucibus non mi in, laoreet maximus arcu. In hac habitasse platea dictumst. Nunc euismod neque eu urna accumsan, vitae vehicula metus tincidunt. Maecenas congue tortor nec varius pellentesque. Pellentesque bibendum libero ac dignissim euismod. Aliquam justo ante, pretium vel mollis sed, consectetur accumsan nibh. Nulla sit amet sollicitudin est. Etiam ullamcorper diam a sapien lacinia faucibus.

Praesent imperdiet, lacus nec varius placerat, est ex eleifend justo, a vulputate leo massa consectetur nunc. Donec posuere in mi ut tempus. Pellentesque sem odio, faucibus non mi in, laoreet maximus arcu. In hac habitasse platea dictumst. Nunc euismod neque eu urna accumsan, vitae vehicula metus tincidunt. Maecenas congue tortor nec varius pellentesque. Pellentesque bibendum libero ac dignissim euismod. Aliquam justo ante, pretium vel mollis sed, consectetur accumsan nibh. Nulla sit amet sollicitudin est. Etiam ullamcorper diam a sapien lacinia faucibus.

2 BACKGROUND AND BASIC DEFINITIONS

2.1 Blockchains

A blockchain constitutes a history that contains all the trades made between its users since its creation. This history is secure and distributed: it is shared by its various users, without intermediaries,

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

which allows each one to verify the validity of the chain. Permanently updated and distributed, the maintenance of the blockchain is based on cryptographic primitives that make any modification almost impossible, which increases its security. Transactions between users are thus immutable. Essentially, when a user broadcasts a transaction, it is received by all the other users of the network and stored in their *mempools*, a memory space where transactions awaiting validation are stored. Block creators will group these transactions into blocks. Once a block has been created, it is broadcast to the network and appended to the blockchain.

Permissionless Blockchains and Verifiable Elections. Permissionless blockchains are public blockchains where participants do not rely on a centralised registration system to take part to the blockchain construction. Indeed, every node can read the blockchain and take the rights to append a block in a decentralized way. Permissioned blockchains differ from permissionless ones in that they rely on predefined nodes to append blocks to the blockchain. The absence of such a predefined group of nodes in permissionless blockchains makes the election an essential element of their design. Election is usually pseudo-random and *verifiable*, i.e., it allows elected nodes to prove they have the rights to append a block (PoW [1], VRF [2], PVSS [3], Randao [4, 5]). Two main approaches exist: *leader-based* and *committee-based.* In *leader-based* approaches, a verifiable election aims at electing a single node, which can then append a block and prove that it has the right to do so. In *committee-based* approaches a large enough committee of nodes must be elected, and a block can be appended only if a *quorum* of the committee signs the block. A verifiable election grants rights to a committee of nodes, providing them with means to prove quorum's legitimacy.

Finalization and Transaction Confirmation. An abstraction called consensus ensures a clear and unambiguous ordering of valid blocks within the blockchain. Each block is valid if it has been created by respecting the rules of the blockchain construction (e.g., valid signatures for blocks) and contains only valid transactions, where valid is application dependent (e.g., no double spending, positive balances in case of cryptocurrencies). Consensus also guarantees the integrity and the consistency of the blockchain between any correct user. Leader-based permissionless blockchains guarantee weaker properties than the consensus abstraction by offering probabilistic finality [6]. That is the very last appended blocks of the blockchain may be revoked, i.e., pruned from the blockchain, in presence of conflicting blocks (e.g. a fork due to two concurrent appends) but the probability that a block is pruned decreases as it gets deeper into the blockchain. The term of *Nakamoto style consensus* is often used to refer to the properties of these blockchains, and solving Nakamoto style consensus may rely either on Proof-of-Work (PoW) or Proof-of-Stake (PoS) (e.g., [1, 3, 7]) for the election mechanism.

Committee-based permissionless PoS blockchains are generally grounded on variants of BFT Consensus [2, 4, 8, 9]. In systems like Cosmos and Tezos [8, 9] a verifiable election mechanism chooses a committee that, once elected, runs the Byzantine consensus protocol (i.e., Tendermint [10] and Tenderbake [11] respectively) to append a unique block to the blockchain. These blockchains are said to have deterministic finality, because conditions to determine if a block is finalized are deterministic and verifiable (e.g. in [8], as soon as a block is appended; in [9] as soon as an appended block is followed by another one). Finalization is certain, and a finalized block can never be revoked. Ethereum PoS also uses a committee to finalize blocks [12] generated by an underlying Nakamoto-style protocol. When a block is finalized (finalized with high probability in probabilistic models) all the contained transactions are said to be confirmed. Yggdrassil will adopt a PoS-based system that guarantees immediate deterministic finality, similar to [8, 9].

UTXO vs Account-based Models. Bitcoin introduced the first type of spending model in crypto-currencies, called UTXO (Unspent Transaction Output). An unspent transaction output is the result of transactions that a user has received and is able to spend in the future. An UTXO can be spent at most once, i.e., it must be debited in a single transaction. At that point, the UTXO is no longer unspent, meaning that it cannot be used again in the future. Thus, through a transaction a receiver gathers money in new UTXOs. A user can have numerous UTXOs at a time, which can be combined to reach a given amount of money to spend.

In the account-based model each user has one account on which it can receive and spend money within the limits of the available funds. This model is akin to each individual wallet having a ledger of its own. After every transaction, the new balance is computed using basic arithmetics. Yggdrasil uses the account-based model due to its simplicity since a transaction with an arbitrary amount of money can be performed with one sending account and one receiving account (instead of multiple UTXOs on both sides). This model is used by Ethereum and it is thought to be better suited than UTXO for supporting smart contracts [7].

2.2 Smart Contracts

Popularized by Ethereum, many blockchains today (e.g., [9], [13]) provide smart contracts as a generic mechanism to make blockchains programmable. Smart contracts are sequential programs, composed of a set of methods and variables, that execute in the blockchain. Operationally, a smart contract is deployed in the blockchain by its creator, which submits to the blockchain a uniquely identified transaction containing the smart contract code. As soon as the submitted transaction is confirmed we say that the contract is deployed. Once deployed, the set of variables of the smart-contract assigned with initial values is defined as the initial state of the contract. In the general case, the execution of one of the smart contract methods results in a new state of the smart-contract, that is a new valuation of its variables. Users can interact with a smart contract by submitting transactions that are requests to execute one of the methods of the smart contract. These transactions are sent to the smart contract's address, which is deterministically generated using the creator's address and how many transactions he has sent [14]. For each transaction invoking a smart contract method, the issuer has to pay some fees just like normal payment transactions.

The smart contract executes in the blockchain network, i.e. each node of the network locally executes the called methods. Since smart contracts are deterministic, participants can unequivocally determine the state of the smart contract by simply executing all transactions submitted to it. Transactions are totally ordered by the blockchain via the underlying consensus mechanism. Thus any two nodes executing the smart contract will compute the same state. That is, for any *confirmed* transaction in the blockchain, either the transaction is successfully executed or not. In the former case

we say that the transaction is *committed*. In the latter case it is *aborted*: the execution failed and the state of the smart contract is not changed. An execution can fail for usual reasons like run-time errors or if the amount of fees sent by the caller does not cover the costs of executing the method call with the given input parameters. As a smart contract can call other smart contracts to complete a method execution, the whole computation originated by a single user invocation is represented as a *call graph* of smart contract invocations. Since semantics must be guaranteed to be sequential for smart contracts, then either the whole call graph is committed or aborted. In a given call graph, we denote with the term *front-end smart contract*, the unique smart contract invoked by the user.

2.3 Sharded smart-contracts and atomicity

In state sharding systems, each smart contract' address resides in a single shard. However, when a user invokes a smart contract, this smart contract may belong to another shard. The system must have a mechanism to route user's call to the smart contract. Routing calls to smart contracts residing in different shards must be done in a careful way to guarantee that if a balance is updated in the issuer's shard, the corresponding transaction will be eventually confirmed in the destination shard, no matter if the result is an abort or a commit. Differently from the general atomic commit problem [15], which must deal with the situation in which two different shards might not willing to both confirm or reject the transaction, for each cross-shard transaction, if the issuer's shard confirms, then the other shard will never reject the transaction. This is true only if the verification of transaction validity is a deterministic process and shards do not fail. Sharding systems usually make these hypotheses to rely on this weak form of atomicity [16]. More formally, *eventual atomicity of confirmation* guarantees that for each transaction between a user and a front-end smart contract, if one shard confirms the transaction then other shards will eventually confirm it.

Besides users, smart contracts themselves can call other smart contracts. The case of a smart contract calling smart contracts belonging to the same shard can be treated as in a non-sharded system, or, if the user invoking the smart contract is in another shard, by employing mechanisms to guarantee eventual atomicity as explained above. On the other hand, invocations crossing shards cannot be treated as internal invocations, like in the non-sharded case, but must be represented as cross-shard transactions. Then, we need to guarantee the *atomic commit of the distributed execution* of the front-end smart contract across shards, i.e., either cross-chain transactions in the call graph originated from a given user invocation are all committed or they are all aborted[1]. We also need isolation so that smart contract execution looks sequential [17]. Let us stress that this form of atomicity works on a commit and abort status of confirmed transactions because only confirmed transactions are part of the call graph as mentioned in the previous section. Since these confirmed transactions are cross-chain, eventual atomicity must be assured, as in the case of user to the front-end smart contract (which is the call graph root).

As will be detailed in the following, our solution, Yggdrasil, combines a 2PC protocol with a cross-chain confirmation mechanism to assure *atomic commit* of the distributed execution of smart contracts and *eventual atomicity* of confirmation.Moreover, adaptivity of Yggdrassil allows us to dynamically adapt shards to reduce the overload generated by these protocols.

[1]For sake of simplicity we consider that internal invocations in the same shard are collapsed in the call graph to a single vertex.

3 SYSTEM MODEL

Nodes, processes, users and validators. Yggdrasil is composed of an unbounded set of nodes $N = \{n_1, \ldots, n_i, \ldots\}$. Each node controls several processes. Each process p_i has a unique identifier id_i, and owns exactly one account of coins. The total sum of available coins in the system is limited and its current value is known by all. Each process has a well-defined role, that of user or validator. When a node joins the network, it creates a process with the role of user, and the identifier of that user is the public key of the node. Subsequently, a node can create other processes with the role of user whose identifiers are derived from the node's public key. To participate in the maintenance of Yggdrasil, a node creates processes with the role of validator, and stakes coins[2]. For sake of simplicity and without loss of generality we assume that we have as many validators as coins staked in the system. The set of processes is denoted by P, the set of validators is denoted by V and the set of users is denoted by U. We have $P = U \sqcup V$, where $\sqcup$ is the symbol of disjoint union.

Adversarial model. We suppose that at any time some processes can fail in any arbitrary manner. These processes are indifferently called *faulty* or *Byzantine* processes. Byzantine processes can "pollute" the computation (e.g., by sending messages with different contents, when they should have sent messages with the same content if they were not faulty). Processes that always follow the protocol are called *honest*. We model the behavior of faulty processes as a weakly adaptive adversary. We characterize the power of the adversary as follows [18]. The adversary has a bounded amount of stake, i.e., at any time, Byzantine validators possess less than a fraction $\tau \in [0, 1)$ of the total stake σ currently available in the system. Note that this does not guarantee that in each shard Byzantine validators possess less than a fraction τ of the shard stake. Indeed, the adversary may try to manipulate more than one third of validators in a specific shard. Yggdrasil provides a shuffling mechanism and a random uniform election mechanism guaranteeing that in any shard, no more than $\tau = 1/3$ of the stake (i.e., validators) are owned by the adversary (see Section 4.7).

The second assumption is related to the adversary's level of adaptability. The adversary can decide to corrupt more processes in a particular shard, but once a process is corrupted the adversary cannot change his mind before k units of times occurred. A time unit represents the maximal amount of time needed to build a block.

Users can also be corrupted by the adversary, but the only action corrupted users could carry out would be to create transactions and therefore incur costs (transaction fees). First, these costs imply that such an attack cannot be done infinitely often, and moreover, these costs would disincentives the adversary to attempt distributed denies of service (DDoS) attacks.

[2]Coin staking can be done through a special smart contract, as done in Eth2.0. We abstract those implementation details, and just assume that coins can be put in escrow for the whole validator lifetime.

Byzantine fault-tolerant consensus and selection of committees. Yggdrasil maintains in parallel several blockchains. Each blockchain is built thanks to a variant of Byzantine Fault Tolerant (BFT) Consensus [19] that provides deterministic finality [6]. Specifically, we assume that each blockchain is grounded on Tendermint [10], that provides immediate finality: a block is finalized as soon as it is appended to the blockchain. Any transaction is then confirmed as soon as it appears in the blockchain. As Yggdrasil is permissionless (see Section 2.1) we also need a verifiable election to elect the committee that once in place run the chosen BFT consensus protocol to build and sign the block to be appended to the blockchain. Among the different existing solutions ([2, 4, 8, 11]), we aim at those that elect a committee of fixed size to determine the quorum of two-third signatures needed to finalize a block, such as the ones provided in [8, 9] or Ethereum PoS [4]. Specifically, (i) a new validator joins a validator set through a confirmed stake transaction, (ii) the maximal size of the validator set is fixed at design time, (iii) the committee for each block is then chosen uniformly at random within the validator set by a shuffling function that makes a pseudo-random permutation of the validator members list at each election and returns the first n validators, where n is the size of the committee. The shuffling function takes as parameter the validator list and a random seed by reading the blockchain. The random seed is generated by applying the xor operation on the hashes of all finalized blocks. These operations being deterministic, this ensures that exactly one committee is elected. Note that a recent improvement to this mechanism makes shuffling secret and unpredictable [20]. In the following, for any blockchain b maintained by Yggdrasil, we assume the existence of a committee of validators Q_b elected among the current set of validators V_b thanks to the assumed election mechanism, where $Q_b \subseteq V_b \subseteq V$. Byzantine validators in the committee are maintained under $1/3$ threshold by the shard shuffling mechanism and the random uniform election. We say that a shard is honest if less than a fraction τ of the committee of validators is Byzantine.

Communication primitives. Processes communicate by sending and receiving messages via a best effort broadcast primitive, which means that when a honest process broadcasts a value, eventually all the honest processes deliver it [21], i.e., messages sent by honest processes cannot be lost. Note that messages sent by Byzantine processes are not guaranteed to be delivered to all honest processes. Such a primitive can be implemented through a peer-to-peer gossip-based diffusion mechanism, as usually done in blockchains. Messages contain a digital signature and we assume that digital signatures cannot be forged. When a process p_i receives a message from p_j, it is certain that p_j sent that message. We assume a partially synchronous environment where the maximum transmission delay is bounded but unknown by the processes [22]. Finally, communication among shards is as follows. When we say that a shard sends a message, we assume that the committee of validators inside the shard broadcasts the message to the system. Any receiving process will accept the message only if it is signed by a quorum of the corresponding committee. Because each shard is maintained under the $1/3$ Byzantine threshold by Yggdrasil, messages sent by a shard are never lost and are received by all honest processes.

4 YGGDRASIL PROTOCOL

The main feature of Yggdrasil lies in its self-adaption to transaction load, so that the number of shards continually adapts to provide fast transaction confirmation in average. Yggrdrasil allows shards to re-organise under high load by splitting into new shards, and later re-merge if transaction load reduces. Notably, Yggdrasil provides a way to assign processes and smart contracts to shards seamlessly with respect to shard dynamics. Smart contracts and processes are automatically re-assigned to a newly created shard (if needed) in a transparent and verifiable way. When a parent shard splits in two new shards, the parent extinguishes itself while a summary of its state is transferred to the newborn shards.

While the local consistency of each shard relies on a local PoS committee-based BFT blockchain (Section 3), Yggdrasil provides global consistency of the system. Yggdrasil ensures that each user is assigned at any time to only one shard, i.e., a user cannot submit transactions to two different shards, or if he does so, the transaction is rejected by one of the shards, because user-to-shard assignment is verifiable. In the same way a smart contract is assigned at any time to only one shard. As for user transactions crossing shards, Yggdrasil safely ensures eventual atomic confirmation (Section 2.2) and atomic-commmit of smart contracts distributed execution — whose execution spans different shards – through a two-phase commit protocol based on locking and eventual confirmation among shards. Yggdrasil ensures eventual atomic confirmation during reorganisations of the system (split or merge operations). This is achieved by shards labeling mechanism, guaranteing that there always exists only one shard at time t that is the closest to any transaction, thus responsible of the transaction processing.

Yggdrasil is tolerant to an adaptive adversary: By relying on random shuffling, validators are regularly assigned to randomly chosen shards to defend against a weakly adaptive adversary. Furthermore, by using a secret and verifiable random draw, validators' assignment is unpredictable.

Last but not least, Yggdrasil allows nodes to incarnate themselves in multiple shards with uniquely identified accounts, to reduce the number of their cross-shard transactions. Indeed nodes can be interested in some particular smart contracts or to trade with specific users, so to incarnate themselves only in the shard where they trade more and benefit for fast transaction confirmation time.

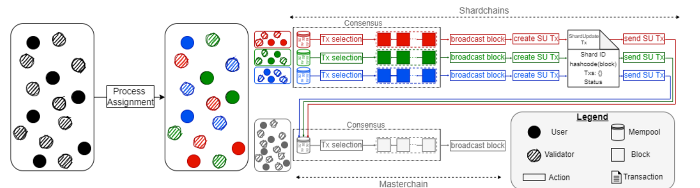


Figure 1: A simple overview of Yggdrasil.

4.1 Transaction Life-Cycle through Sharding

An Yggdrasil's process with the role of user can transfer coins to another user, deploy smart-contracts, invoke smart contract methods, or deposit coins to become a validator as realized in common PoS-based blockchains. For each of these actions different user transactions are submitted to Yggdrasil, i.e., *payment transactions*, *smart*

contract deployment transactions, *smart contract method invocation call transactions*, and *stake transactions*[3], respectively. Yggdrasil manages all these transactions in a unified way as described below.

Transactions and state sharding. As will be detailed in Section 4.3, Yggdrasil assigns each process to exactly one shard in a verifiable way, where a process can be either a user (submitting transactions) or a validator (validating transactions). Since the assignment is unique at any point of time, transaction sharding is realised by assigning all the transactions of a user to this user's shard. This also implies that any smart contract is assigned to the shard of the user that deploys the smart contract, through the smart contract deployment transaction. To realise state sharding, Yggdrasil maintains a blockchain for each shard, called *shardchain*. Since a trusted third party is needed to achieve synchronization between two or more blockchains [23], Yggdrasil also maintains a synchronization blockchain, called *masterchain*. Each shard locally builds a shardchain to validate its own transactions. When needed, shards coordinate to handle the creation of new shards or the merging of some of them, and cross-shard transactions. To coordinate themselves, shards submit to the masterchain special transactions called *shard update transactions*. The masterchain validates shard update transactions submitted by shards and serves as a gateway for processes that want to stake coins to become validators. To build a blockchain (i.e., a shardchain or the masterchain), a committee (quorum) of validators is elected after each block through modalities described in Section 3. Each process in Yggdrasil locally manages, i.e., stores, reads and updates, the masterchain. On the other hand, shardchains are managed solely by the processes assigned to them. Each process has access to the state of both the masterchain and its shard, where the state is defined as follows:

Definition 1 (State of a blockchain). *The state of a blockchain is the current value of accounts and smart contracts that can be computed by reading the blockchain.*

Transaction processing. A user submits transactions within its shard (see Figure 1). Transactions are collected by the shard's validators[4], and locally stored in their memory pool (a.k.a mempool). To create a block, validators being part of the current committee invoke the Byzantine fault-tolerant consensus protocol with a set of transactions from their mempool. Transactions are validated and embedded in the next block of the shard's shardchain. Once a block is appended to the shardchain, validators send a summary of the block to the masterchain via the shard update transaction (denoted by SU in Figure 1, and whose content is detailed later). Validators of the masterchain verify that each shard update transaction has been created and sent by the issuer shard.

For space reasons implementation details and pseudo-codes of blockchain creation in each shard and verification of the shard update transaction by the masterchain can be found in the full version of the paper [?].

Transaction confirmation and atomicity of cross-shard transactions. Yggdrasil introduces its own notion of transaction confirmation to guarantee the global consistency of the system. Specifically, all the transactions processed by the masterchain, i.e. *shard update transactions* and *stake transactions*, are immediately confirmed once they appear in a block appended to the masterchain. These two types of transactions are confirmed in the masterchain because they have a system-wide scope: they need to be seen from any shard to correctly manage shards membership, shard dynamics and cross-shard transactions. The level of confirmation of the other user transactions depends on whether or not they are intra-shard or cross-shards. In the case of intra-shard transactions, both the issuer and the recipient entities of the transaction (i.e., users or smart contracts) are assigned to the same shard. Any intra-shard transaction is *confirmed* as soon as it appears in a block of the shardchain *and* the corresponding shard update transaction sent by the shard to the masterchain, notifying its confirmation in the shardchain, is confirmed in the masterchain.

In the case of cross-shard transactions, the issuer and the recipient entities of the transaction are assigned to two different shards [5]. As mentioned in Section 2.2, to avoid inconsistent situations or double spending, it is sufficient to guarantee the eventual atomicity of cross-shard transactions confirmation. This is because (i) the check of the issuer balance, which is done in the issuer's shard, is the only condition to confirm or reject a transaction and (ii) shard's behavior, as a whole, is honest. Yggrdrasil ensures that if the issuer is honest then her transaction is eventually confirmed. For both payment and smart contract invocations, cross-shard transactions are managed by relying on the masterchain. The different steps involved to confirm a cross-shard transaction tx_1 from shard s_1 to shard s_2 are explained in the following. First, validators of s_1 create block b_1, containing tx_1, and broadcast a ShardUpdateTx SU_1 (containing uniquely the Merkle roots of the transactions of the block containing tx_1); SU_1 is then added in a masterchain block. When validators of s_2 see SU_1, they ask for b_1. After receiving it, they extract tx_1, add it in a block b_2, append b_2 to their shardchain, and broadcast a ShardUpdateTx SU_2. SU_2 is then added in a masterchain block. In case tx_1 is the call of a smart contract deployed in s_2, validators of s_2 create a new transaction tx_2 containing the results of the call and send it to s_1 in the same ShardUpdateTx as tx_1 (SU_2). After receiving it, s_1 asks for b_2, extracts tx_2 and puts it in its shardchain (e.g. in block b_3). Due to lack of space, implementation details of the confirmation of cross-shard transactions is deferred to the full version of the paper [**?**]. The following definitions summarize the confirmation conditions for the different types of transactions in Yggdrasil.

Definition 2 (Masterchain transactions confirmation). *Any stake and shard update transactions is confirmed when it appears in a block of the masterchain.*

Definition 3 (Intra-shard transactions confirmation). *An intra-shard transaction tx assigned to shard s is confirmed when tx is embedded in a block of s's shardchain* ***and*** *the shard update transaction notifying tx is confirmed.*

[3]When a user submits a stake transaction tx, the user's node creates a new process with the role of validator identified by tx.

[4]Users can also store blocks and transactions if they want to but since they are not responsible of building blocks, this is not mandatory.

[5]For a payment transaction, the two involved entities are user's accounts. For smart contracts invocations, the two entities are a user account and a smart contract account. Of course, smart contracts can call in their turn smart contracts in another shards. Nested calls generate cross-shard transactions that are managed by the 2PC protocol presented in Section 4.2

Definition 4 (Cross-shard transactions confirmation). *A cross-shard transaction is confirmed if and only if it is confirmed as intra-shard transaction by both involved shards.*

4.2 2PC for distributed smart-contracts

This section provides a 2PC algorithm to guarantee atomic-commit of distributed execution when insmart contracts involved live in different shards. To illustrate the algorithm, let us to make an explanatory scenario. Let us suppose to have a user that calls, trough a transaction tx_0 a smart contract sc_0, which calls, in the body of the called method, two other smart contracts sc_1 and sc_2 in sequence. If sc_1 and sc_2 live in two different shards, then Yggdrasil generates a cross-shard transaction for each call, let us say tx_1 and tx_2. Note that eventual confirmation guarantees that the two transactions are added to the call graph, however, if their execution is left independent we could have the situation in which tx_1 is committed and tx_2 is aborted. To be atomic, since tx_2 failed, the tx_0 as a whole should be aborted and tx_1 's effects reverted. The 2PC algorithm we propose prevents tx_1 to commit in this scenario.

In the algorithm the shard, where the front-end smart contract lives, coordinates commit and abort of other shards following an approach where shards committees emit special transactions throughout the process. More specifically, inside committees, validators propose blocks inserting specific transactions. Validators verify the block being sure the the algorithm has been followed before accepting it. Once accepted (signed by a quorum), any other validator in subsequent committees can resume the algorithm if the previous committee did not complete it, just by looking at blocks in shardchains and masterchain. In other terms, the state of the algorithm is fully recorded in the shardchains and the masterchain, which allows to have dynamic committees that rely on the total order of all transactions (intra and cross) to determine the state of the algorithm. The pseudo-code is depicted in Algorithm 1. Each proposer that selects a transaction in the MemPool (line 40) verifies, before inserting it in a block, if it is an invocation to a front-end smart contract sc_0 spanning different shards. If the smart contract is not already locked, the proposer prepares and inserts in the proposed block a intra-shard transaction of type *lock* tx_0^{lock} and a cross-shard transaction $tx_{0,i}^{query}$ of type *QUERY* for each outgoing call crossing the coordinator shard reaching shard s_i. The query transaction contains transactions to call the recipient smart contract and the calling one.

When the block is confirmed by the committee of the sc_0 (coordinator shard), then the lock becomes effective. Each validator in the coordinator shard sees that the smart contract has been locked by reading the blockchain, and stops to consider other transactions directed to the locked smart contract sc_0 for inclusion in successive proposals. As soon as $tx_{0,i}^{query}$ are confirmed, validators in the recipients shards s_i, read these query transactions (line 3). Note that when these shards receive the query transaction, the lock of sc_0 is already effective. If the execution of the incoming transactions do no involve other smart contracts in other shards, then proposers pre-execute the called transaction (if the smart contract is not already locked). More specifically, the block proposer pre-executes the result against the state of the blockchain till the previous finalized block. Result of this pre-execution can be abort or prepare-to-commit. In both cases the proposer prepares and insert in the proposal a cross-shard vote transaction towards the coordinator shard. Cross-shard transaction towards the coordinator shard are denoted as $tx_{i,0}^{vote}$. As soon as those transactions are confirmed, the coordinator shard compares results to decide if roll-back or commit (line 16). In case of no abort in the votes received, the proposer of the coordinator shard executes the transaction tx_0 (line 21), then compute the decision (lines 23 and 29) and then unlock. The unlock is an intra-shard transaction tx_0^{unlock}, while the decision is a cross-shard transaction $tx_{i,0}^{decision}$ for each shard s_i. At receiver side, all the shards commit or roll-back accordingly with the decision. Roll-back is implicit, the validator does nothing in this case. In case of commit, the computation must be redone by the new proposer (the proposer might have changed since the last pre-execution). Since the state of the smart contract did not change from the last prepare-to-commit because of the lock, the result is the same as in the pre-execution phase (let us remind you that smart contracts are deterministic). After the execution, an unlock intra-shard transaction is inserted in the block tx_i^{unlock}. Note that the whole process is recursive to explore the whole call graph. In case of loops in the call graph, to avoid deadlocks a locked smart contract can accept incoming calls when originating by the same root of the call graph that caused the smart contract to be locked (line 5). Let us stress that the call graph is distributed among shards. To cope with that call paths, which are added at each outgoing invocation in the call graph, allow to trace back the path till the root and find if there is a common root.

As mentioned, locking a smart-contract consists in ignoring future transactions that could modify the state of this contract (until this smart contract is unlocked), however for stateless smart-contracts (i.e., smart-contracts that do not have a state to maintain), it is useless to lock the contract.

Addressing the dynamicity of the call graph. Notice that, we can have merges and splits during the 2PC protocol, e.g., two smart contracts that are on the same shard at the beginning of the protocol can live on two different shards at the end of it, splitting at some arbitrary moment. To make the dynamic sharding seamless to the protocol, we can modify the protocol as follows. Firstly, in the call graph, we treat all smart contract transactions as cross-shard smart contract transactions, i.e., the call graph has at its vertices all the involved smart contracts, independently whether two adjacent ones are on the same shard or not. Secondly, when a validator inserts in a block a cross-shard transaction that targets another smart contract on the same shard, then it immediately process it. In this way, we avoid to add latency in the processing of an invocation between two smart contracts on the same shard.

4.3 Process-to-shard assignment

Shards are uniquely identified by their label l (the computation of shards' label is described in Section 4.4). At any time, any process is assigned to the (unique) shard whose label minimizes the distance with the process's identifier.

Definition 5 (Distance function). *[24] Let $a = a_0 \dots a_{d-1}$ and $b = b_0 \dots b_{d'-1}$, for any $d, d \geq 1$, be any two bit strings, and $s = max(d, d')$. Note that the bit numbering starts at zero for the most significant bit. The distance between a and b, denoted by $D(a, b)$*

Algorithm 1 Distributed-Graph 2PC for any shard block proposer

```
1:  upon block proposal fetch MemPool and state of Yggdrassil chains
2:  fetch all tx from confirmedTransactionSet in state
            /* confirmed transactions till the previous block in the shardchain */
3:  for each tx such that(tx.type = QUERY) then
4:    ttx ← tx.targetTx
            /* query received, target transaction ttx extracted */
5:    if(!isLocked(ttx.sc) ∨ (isFromSameCallGraph(tx)) then
            /* isFromSameCallGraph() returns true if the query comes from the same call graph as the query
            transaction that provoked the lock of ttx.sc. This means that the call path at the lock time is a
            prefix of the call path of ttx. False otherwise. */
6:      if(hasCrossShardCalls(ttx)) then
            /* call graph goes one level deeper */
7:        block_proposal.insertLockTx(ttx.sc)
8:        targetTxs ← getTargetTxs(ttx)
9:        block_proposal.insertQueryTxs(ttx, targetTxs, tx)
10:     else
            /* call graph reaches a leaf */
11:       res ← exec(ttx, state)
12:       if(res! = null) then
13:         block_proposal.insertLockTx(ttx.sc)
14:         block_proposal.insertVoteTx(PREPARE, res, tx, tx_v0)
            /* tx_v0 is a root vote transaction with all values to empty */
15:       else block_proposal.insertVoteTx(ABORT, null, tx, tx_v0)
16: for each tx such that (tx.type = VOTE)
17:   dtx ← tx.destTx;
            /* vote received, dest transaction dtx extracted from tx */
18:   if( isReadyToCompute(dtx) ∧ isLocked(dtx.sc) then
            /* isReadyToCompute() checks if, in this shard (the dtx.sc's shard) all the votes, for which the query
            tx.queryTx has been issued, have been gathered */
19:     votes ← getVotes(getAllVoteTxs(tx))
20:     if(noAbort(votes))
21:       res ← exec(dtx, getResults(getAllVoteTxs(tx)))
22:     case 1 (res! = null ∧ isLockOnInvoke() ∧ noAbort(votes))
            /* the dtx is the root, a decision is sent */
23:       insertDecisionTxs(COMMIT, getAllVotesTxs(tx))
24:       insertUnlockTx(dtx.sc)
25:     case 2 (res! = null∧!isLockOnInvoke() ∧ noAbort(votes))
            /* the dtx is not root, a vote must be sent to the parent */
26:       prevQuery ← tx.queryTx.previousQ.last()
27:       insertVoteTx(PREPARE, res, prevQuery, tx)
28:     case 3 ((res = null ∨!noAbort(votes) ∧isLockOnInvoke())
            /* the dtx is the root, a decision is sent */
29:       insertDecisionTxs(ROLLBACK, getAllVoteTxs(tx))
30:       insertUnlockTxs(dtx.sc)
31:     case 4 (res = null∨!noAbort(votes))∧!isLockOnInvoke())
            /* the dtx is not the root, a vote must be sent to the parent */
32:       prevQuery ← tx.queryTx.previousQ.last()
33:       insertVoteTx(ABORT, res, prevQuery, tx)
34: for each tx such that (tx.type = DECISION) then
35:   dtx ← tx.targetTx;
            /* decision received, dest transaction dtx extracted from tx */
36:   if(isLocked(dtx.sc)) then
37:     if(isCommit(tx)) then exec(dtx)
38:     if(tx.prevVoteTx! = tx_v0) then
            /* dtx is not a sink transaction in the call graph */
39:       insertDecisionTxs(tx.decision, getAllVotesTxs(tx))
40:     insertUnlockTx(tx.sc)
41: for each tx such that(tx.type = INVOKE from user) then
42:   if(!isLocked(tx.sc) ∧ hasCrossShardCalls(tx)) then
43:     blockProposal.insertLockTx(tx.sc)
44:     targetTxs ← getTargetTxs(tx)
45:     blockProposal.insertQueryTxs(tx, targetTxs, tx_q0)
            /* tx_q0 is a root query transaction with all values to empty */
46:     else blockProposal.insertMemPoolTxsInBlock(tx)
            /* insert all other invoke transactions from the MemPool in the block */
47: propose block
```

Algorithm 2 *insertQueryTxs(sourceTx,targetTxs,prevQueryTx)*

```
1: callPath ← prevQueryTx.callPath.add(sourceTx)
2: previousQ ← prevQueryTx.previousQ.add(prevQueryTx)
3: for each targetTx ∈ targetTxs
4:   queryTx ← createTx(QUERY, callPath, targetTx, previousQ)
5:   blockProposal ← blockProposal.add(queryTx)
```

Algorithm 3 *insertVoteTx(vote, res, queryTx, prevVoteTx)*

```
1: destTx ← queryTx.call_path.last
2: previousV ← prevVoteTx.previousV.add(prevVoteTx)
3: voteTx ← createTx(VOTE, vote, res, destTx, queryTx, previousV)
4: blockProposal ← blockProposal.add(voteTx)
```

Algorithm 4 *insertDecisionTxs(decision, voteTxs)*

```
1: for each voteTx ∈ votesTxs then
2:   targetTx ← voteTx.queryTx.targetTx
3:   prevVoteTx ← voteTx.previousVotes.last()
4:   decisionTx ← createTx(DECISION, decision, targetTx, prevVoteTx)
5:   blockProposal ← blockProposal.add(decisionTx)
```

is the numerical XOR between a and b and is computed as follows.

$$D(a,b) = D(a_0 \ldots a_{d-1}.0^{s-d}, b_0 \ldots b_{d'-1}.0^{s-d'}) = \sum_{i=0}^{s-1} 2^{s-1-i} 1_{a_i \neq b_i}$$

where notation 0^{s-d} represents $s-d$ digits set to 0, and 1_A denotes the indicator function, which is equal to 1 if condition A is true and 0 otherwise.

Property 6 (Process Assignment). *Let id_i be the identifier of process p_i and $\mathcal{S}$ be the set of shards, then the shard S_ℓ to which p_i is assigned satisfies relation 1.*

$$S_\ell = \arg\min_{S \in \mathcal{S}} D(id_i, S) \quad (1)$$

By construction of the shard labels mechanism (see Section 4.4) shard S_ℓ is unique with respect to id_i, that is, for any shard $S_{\ell'} \in \mathcal{S}$ with $\ell' \neq \ell$, then $D(id_i, S_\ell) < D(id_i, S_{\ell'})$.

Due to lack of space, the pseudo-codes executed by a newly created process and its assignment to a shard are moved to the full version of the paper [?].

4.4 Dynamic management of shards

The number of shards in Yggdrasil self-adapts to the actual rate at which transactions are submitted to Yggdrasil. This is achieved by two operations, namely the *split* and the *merge* operations. Specifically, when the last blocks of a shardchain become overloaded (i.e., the average ratio between their number of bytes and the maximal number of bytes contained in a block exceeds a given threshold), then the committee of validators of the overloaded shard triggers a split operation. Note that this assumes that the size of the committee is greater than twice the minimal size of a Byzantine tolerant committee. In the negative the overloaded shard does not split into two smaller shards. Now, when a shard is under-loaded (i.e., the average ratio between their number of bytes and the maximal number of bytes contained in a block falls short of a given threshold), or the size of its committee of validators is close to the minimal size of a Byzantine tolerant committee, then the committee of validators triggers a merge operation with the shard closest to theirs.

Operationally, each shard maintains an attribute called *status* that can be set to *Splittable*, *Mergeable*, or *Regular* depending on the conditions mentioned above. This attribute is also included in the shard update transactions sent from shards to the masterchain to globally share information about all the shards status.

We formally express the status of a shard as follows:

Definition 7 (Shard's Status). *We denote by $V_\ell(t)$ the set of validators assigned to s_ℓ at time t. At time t, a shard is in one of the following three status.*

- ***Splittable****: A shard is considered splittable at time t if $|V_\ell(t)|$ goes above a certain threshold Φ* ***and*** *block load goes above another threshold Γ.*

- ***Mergeable**: A shard is considered mergeable a time t if $|V_\ell(t)|$ goes below a certain threshold ϕ **or** block load goes below another threshold γ.*
- ***Regular**: A shard is considered regular if it is neither splittable nor mergeable.*

Note that at each split/merge operations, the label of the newly created shard(s) is derived from its parent's label. Initially, Yggdrasil is made of a single shard labelled with the empty binary string $\ell = \epsilon$. If Yggdrasil needs to replace a splittable shard s_ℓ labelled with ℓ by two new shards, they respectively inherit the label of the overloaded shard suffixed with 0 and 1, i.e., $s_{\ell.0}$ and $s_{\ell.1}$. If two shards $s_{\ell.0}$ and $s_{\ell.1}$ are concomitantly Mergeable, they are replaced by a single shard s_ℓ whose label is equal to the maximum prefix shared by the two Mergeable shards, i.e., ℓ. Processes are automatically reassigned to the newly created shards according to their identifiers.

State transfer between shards. As the split and merge operations lead to the creation of new shards, this gives rise to the creation of new shardchains and the extinction of old ones. The state of a newly created shardchain is initialized with a summary of its parent(s)' state. This summary is the genesis block of the new shardchain.

Each split or merge operation automatically re-assigns validators to their new shard. This assignment is verifiable in the masterchain. The genesis block of each new shardchain is produced by committees pseudo-randomly selected upon the validators assigned to the shard. The pseudo-random selection is based on public information contained in the masterchain.

Processes maintain the set of shards $\mathcal{S}$ by reading the information contained in the masterchain's blocks. Specifically, upon receipt of a masterchain's block, processes append it to their local copy of the masterchain and update $\mathcal{S}$ using the information contained in it.

4.5 Shards update transactions details

We are now able to detail shard update transactions. A shard update transaction contains the latest information related to a shard, namely, the hash of the last block created, the status of the shard, and information about outgoing cross-shard transactions. When a shard validates a cross-shard transaction in its shardchain, it must notify the receiving shard s'. It includes in its shard update transaction the Merkle Root m' of the cross-shard transactions that involve the shard s' (if any) associated to the label ℓ' of s'. More formally, a shard update transaction SU sent by shard s is defined as follows.

$$SU = (\ell, h(b), \mathcal{T}, \theta), \quad (2)$$

where ℓ is the label of shard s, $h(b)$ is the cryptographic hash of the latest block b created in s, $\mathcal{T}$ represents the set of cross-shard transactions contained in b that involves r corresponding shards, and θ represents the status of s. Note that $\mathcal{T}$ is a key-value list where the keys are the labels ℓ^j of involved shards s^j, by involved shards, we mean the shards that have to confirm at least one of the cross-shard transactions contained in b. The value associated to each ℓ^j in $\mathcal{T}$ is the merkle root m^j of the transactions (contained in b) involving s^j as a recipient, it is defined as:

$$\mathcal{T} = \left\{(\ell', m'), \ldots, (\ell^j, m^j), \ldots, (\ell^{(r)}, m^{(r)})\right\} \quad (3)$$

4.6 Reducing cross-shard transactions volume

Cross-shard transactions are very expensive in terms of latency (i.e. since a cross-shard transaction needs to be processed by two shards, users have to wait longer for it to be confirmed), therefore, it is essential to limit the volume of these transactions. We propose to allow any node to create several users (not necessarily when the node joins), one for each shard of interest, to make transaction processing local to each shard. We call this optimization *incarnation*. Each of these incarnations is a user with one account. Any two incarnations have two different accounts.

Incarnations get identifiers allowing nodes to position themselves in the targeted shard. Specifically, an incarnation is identified by the label of the targeted shard concatenated to the public key of the node. Concretely, suppose that when a node joins the networks there exist 3 shards respectively labelled 0, 10, and 11 and the node's public key is 1001 [6]. Based on its node's public key, the default user incarnation would be identified by $id_{node} = 1001$, therefore, would be assigned to shard 10. However, if the node intends to repeatedly interact with another user or with a smart contract located in shard 0, Yggdrasil enables it to incarnate in s_0, with the identifier $id_{incarnation} = 0.id_{node} = 01001$. Operationally, to create an incarnation, initial funds must be deposited into its account by sending a cross-shard transaction tx_1: $< id_{node}, id_{incarnation}, _>$. If the node wants to withdraw its funds from its incarnation it must send a transaction tx_2: $<id_{incarnation}, id_{node}, _>$.

4.7 Dealing with an adaptive adversary

So far, we have considered a deterministic and static assignment for all processes in a shard. However, to deal with an adaptive adversary, validators must be moved to random shards from time to time (quickly enough to prevent the adversary from poisoning the shard by progressively compromising more than a fraction τ of the validators committee). This mechanism is known as shuffling [7]. However, shuffling validators (committee members or not) introduces some synchronization overhead, i.e., the time it takes for moved validators to download the latest state. To avoid downtime during the synchronization procedure, it is imperative that for each shard, each resynchronization involves a subset of the validators of the shard, and to defend against a weakly adaptive adversary, the new assignement must be random, and unpredictable.

Our procedure to shuffle validators is presented in Algorithm 5. The reassignment function is parametrized by k, k being the number of blocks that need to be created in a given shard before the adversary is capable of corrupting a new validator in the shard. Operationally, our reassignment function consists in computing a new identifier id_v^h for each validator v and each new height h of the masterchain. Input values of the reassignement function are: *(1)* the validator identifier id_v and *(2)* the hash of the latest masterchain block $hash(b_{h_i^m})$. Note that the adversary cannot guess $hash(b_{h_i^m})$ prior this block is created which limits its adversarial strategies. This results on an output value id_v^h, a binary string the validators use to (i) define if they need to move and (ii) in which

[6] Here, we reduce the size of the public key for simplicity of the example. In reality, the public key is 256 bits long

[7] Note that users do not need to be shuffled as they have no decision power, i.e., they have no voting power.

shard it should re-assign to. To do that, the validators first calculate the distance (see Definition 5) between their identifier id_v and id_v^h $D(id_v^h, id_v)$. The validator is allowed to move if its $D(id_v^h, id_v)$ is below a threshold such that the probability for the validator to be shuffled is equal to $1/k$ (line 3 of Algorithm 5). Then, the validators calculate $D(id_v^h, \mathcal{S})$ and assign the validator to the closest shard to id_v^h (line 4 of Algorithm 5). Note that the distance function could return the same shard the validator was in for the last height, thus, it would not move. Hence, the probability of of having a different shard than the former shard the validator was in would be $1 - \frac{1}{|\mathcal{S}|}$.

In this way, for each masterchain block, we have a probability $\approx 1/k$ for a validator to be re-assigned and each process in the system could compute its assignment.

Algorithm 5 Validators Reassignment

1: ***upon*** *receive block* ***from*** $C_m(h_i^m + 1)$
   /* $C_m(h)$ being the masterchain committee at height h. */
2: $id_v^h \leftarrow f(id_v, hash(block))$.
3: ***if*** $(D(id_v^h, id_v) < \frac{D(id_v^h, \overline{id_v^h})}{k})$
   /* where $\overline{id_v^h}$ is the binary complement of id_v^h */
4: $v_i.shard \leftarrow getClosestShard(id_v^h, \mathcal{S})$
   /* getClosestShard() returns the closest shard between id_v and the shard labels in $\mathcal{S}$ using the distance function defined in Definition 5. */

5 SECURITY ANALYSIS

Due to space limitations, Yggdrasil's properties and proofs are presented in the full version of the paper [**?**].

6 PERFORMANCE EVALUATION

The objective of this section is to evaluate the performances of Yggdrasil against the following properties: (i) scalability, i.e., capacity to scale during a peak of transaction load in terms of *block/transaction throughput and latency*, (ii) *reactivity in terms of number of shards in the system*, against a sudden and abrupt transaction fluctuation, i.e., during and after a burst of transactions and (iii) the impact of cross-shard transactions.

We evaluate Yggdrasil scalability using 500k historical Ethereum transactions [25] contained in 10k blocks; between the $14,700,000^{th}$ and the $14,710,000^{th}$ blocks created between 02/05/2022 at 20:54:24 and 04/05/2022 at 10:47:00.

For reactivity we consider realistic fluctuations (by scaling time from real-time minutes to simulation seconds) and we compare Yggdrasil to time-driven approaches.

For cross-shard transaction we use a synthetic scenario, to evaluate performance under ever increasing proportion of cross-shard volume (from 0% to 100%).

The source codes of these protocols as well as all the scripts of the experiments are publicly accessible [26].

6.1 Simulator and Experimental Environment

We have used an agent-based simulation framework dedicated to blockchain systems, called Multi-Agent eXperimenter (MAX) [27] based on the MaDKit framework [28]. MAX offers generic libraries to easily develop distributed ledger protocols and a large range of simulation scenarios. The simulator is a discrete event simulator, where the unit of simulation time is referred to as a tick. Message-passing libraries allow us to configure different types of communication schemes and message delays. In this work, the communication schema is configured as a reliable broadcast with configurable delay to reflect assumptions on our reliable broadcast (see section 3 for more details). Impact of message losses is left for future works. All the experiments have been run on Grid'5000, a large-scale and flexible test-bed for experiment-driven research [29]. Due to the computational complexity of simulation models and experiments involving a representative number of agents, each experiment presented in this paper takes in average 24 hours.

6.2 Simulation Model

6.2.1 Block creation model. Miners create blocks by following an implementation [30] in the simulator of the Tendermint protocol[10], a BFT-consensus protocol that allows the creation of immediate finality blockchains.

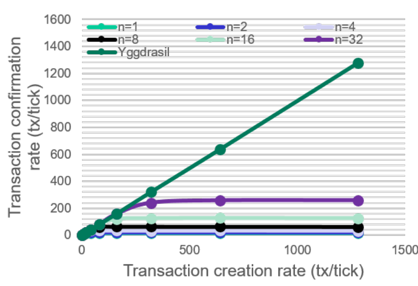
6.2.2 Common parameters of the simulations. For all the experiments presented in the paper we have fixed some common parameters as follows:

- The block capacity, that is the maximal number of transactions a block can embed, is set to 100 transactions (to avoid the simulator overload). Note that while in general, the block capacity is approximately equal to 4,000 transactions [31], reducing the block capacity does not affect the behaviour of the protocols.
- A transaction is confirmed when the block this transaction belongs to is appended to a blockchain and referenced in the masterchain.
- c_{min} is set to 1. Impact of c_{min} on the structure of the system (number of shards) and its performances is left for future works.
- For each experiment, we have run sufficiently many simulations to get a confidence interval equal to 5 ± %.

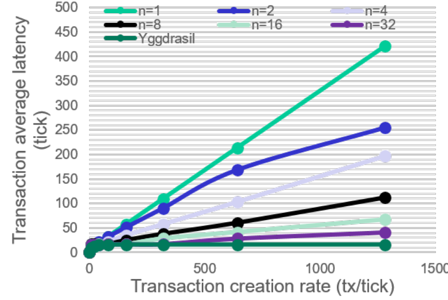
6.3 Scalability

This section studies the capability of Yggdrasil to handle high transaction submission rates. Specifically, we evaluate the transaction confirmation rate, the number of unconfirmed transactions and the transaction latency, i.e., the average time elapsed between the submission of a transaction in the network and the time at which the transaction is confirmed. We compare the performance of Yggdrasil to solutions with static sharding such as Monoxide[16] with a number of shards n throughout the simulation.

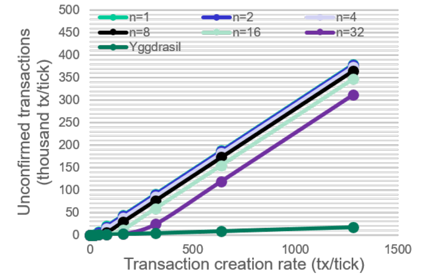
6.3.1 Experiment setting. The overload threshold Γ which conditions a shard splitting is fixed to 90% for Yggdrasil. Note that when $\Gamma = 100\%$, splits never occur and thus Yggdrasil reduces to Tendermint (n=1). The submission rate of transactions f_t, which represents the number of transactions submitted per tick of simulation, is set at the beginning of each experiment. f_t varies from 1 to 1280 txs/tick. Let us remark that we get in expectation one block created every 10 ticks. This means that in Tendermint $f_t = 10$ txs/tick already exhausts the system transaction treatment capacity, as the system creates one block every 10 ticks in expectation and one block contains 100 transactions. From this observation, we might expect that for $f_t > 10$ txs/tick, pending transactions will accumulate over time in, at least, Tendermint ledger. Note that to



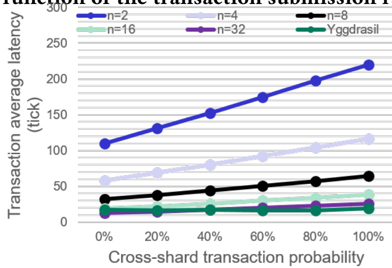
(a) Transaction confirmation rate as a function of the transaction submission rate.



(b) Transaction average latency as a function of their submission rate.



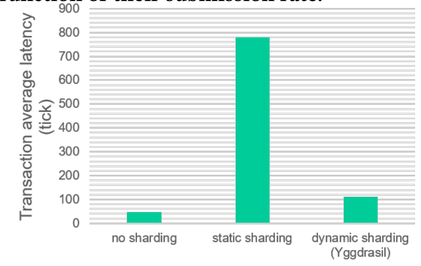
(c) Number of unconfirmed transactions as a function of their submission rate.



(d) Transaction average latency as a function of cross-shard transaction probability.

Solution	Yggdrasil	rp=10	rp=20	rp=50	rp=100	rp=500	rp=1000	rp=1440
Rate (txs/tick)	375	253,5	189	120	60	15	15	15
Latency (tick)	35,8	71,4	85,5	114,5	123,9	132	132	132

(e) Maximum rate and average latency of Yggdrasil and time-driven solutions in presence of a peak of load. Note that 1440 **ticks corresponds to a day, which is the reconfiguration period used by Elrond [32] and Omniledger [33].**



(f) Transaction average latency with 2-phase commit algorithm.

Figure 2: Performance evaluation of Yggdrasil.

avoid the overload of the simulator we were limited to $f_t = 1280$ txs/tick. Anyway, setting f_t up to 1280 txs/tick allows us to severely stress Tendermint and Yggdrasil. Similarly to Bitcoin Core client, validators give priority to old transactions in our implementations of Tendermint and Yggdrasil.

6.3.2 Experiment results. The main results of our experiments appear in Figures 2a, 2b and 2c. Note that in all the graphs, points are linked together with lines. This is only for readability reasons.

Figure 2a shows the confirmation rate of transactions as a function of their submission rate f_t. The main observation regarding static sharding solutions is that whatever the number of shards n is, they show a limited transaction confirmation rate (e.g. approximately 200 txs/tick for $n = 32$ shards). On the contrary, this rate is auto-adaptive for Yggdrasil which reaches more than 1.200 txs/tick while Tendermint ($n = 1$) reaches only 15 txs/tick (85 times less powerful) which confirms the interest of dynamic sharding when it comes to scalability. The implemented static-sharding solution does not allow to reach such good performances even with $n = 32$ shards. In order to better understand our simulation results, let us give a correspondence between our simulated system and what would give us a real system. According to [34], Tendermint has a transaction confirmation capacity of approximately 500 txs/s. Proportionally and taking the same basic parameters such as block size and inter-block delay, Yggdrasil would be able to confirm about 42.000 txs/s. Note that the ability of Yggdrasil to match its transaction confirmation capacity to the arrival rate of these transactions already allows us to glimpse its scalability potential. Figure 2b illustrates the average transaction latency as a function of f_t. In contrast to all the other experiments, transaction latency has been measured as follows: transactions are submitted at f_t for a while, then f_t is set to 0, and simulations stop once all the submitted transactions have been confirmed. For static sharding solutions, latency is increasing in average but reaches lower values as the number of shards n increases (450 tick/tx for $n = 1$ and 50 tick/tx for $n = 32$). On the other hand, Yggdrasil with its dynamic sharding shows a stable and lower latency (16 tick/tx). Figure 2c shows the average number of transactions that accumulate at the end of the simulation before being embedded in blocks. The number of unconfirmed transactions confirms our hypothesis about the rate. Yggdrasil has a better confirmation capacity than systems with a static number of shards. It is shown by a least number of pending transactions at the end of the simulation. Note that the number of pending transactions is close to 0 but not null for the simulated scenarios of Yggdrasil because simulations are interrupted while transactions are still arriving, thus not confirmed yet by newly created blocks.

6.4 Reactivity

This section aims at assessing the capacity of Yggdrasil to react to sudden and abrupt fluctuations in the creation transaction rate. Additionally, we compare Yggdrasil, which is event-driven, to the time-driven adaptability some solutions of our related-work provide (e.g, Elrond [32], Omniledger [33]). We thus study the reactivity of solutions that adapt the number of shards at specific reconfiguration periods rp.

6.4.1 Experiments setting. As briefly presented in Section 4.4, when f_t shrinks, the system reacts by progressively decreasing the under loaded sibling shards, and thus the number of created blocks. Thus each merge divides by almost two the number of blocks subsequently created. By the randomness of transaction identifiers, if one shardchain becomes under loaded, then soon after, all the shardchains become under loaded too, and thus merges occur in cascade. Initially, $f_t = 500$ txs/tick during 10 ticks to mimic a transaction

peak load, and then at tick $t = 12$, $f_t = 0$ txs/tick. Split parameters Γ and T are set respectively to 90% and 5, while merge parameters γ and τ are set respectively to 10% and 2. As for the time-driven parameters, rp is set to 10, 20, 50, 100, 500, 1000 and 1440 ticks. The latter matching the reconfiguration period of Omniledger [33] and Elrond [32] (a day).

6.4.2 Experiments results. Figure 2e shows the reactivity of Yggdrasil in presence of a load peak (constant function from $t = 1$ to $t = 11$ ticks at $f_t = 500$txs/tick). Yggdrasil initially undergoes a series of splits, it reaches a maximum transaction confirmation rate of 375 txs/tick in order to lower latency to 35 ticks. Then, it progressively moves on to a series of merge up to converging to a single shard.

The time-driven solution, on the other hand, performs less well since it does not adapt its number of shards automatically. Indeed, at low values of rp such as 10 or 20 ticks, the system still manages to increase the confirmation rate (190-250 txs/tick) to absorb the increase in throughput thus lower latency (70-85 ticks). At medium values such as 50 or 100 ticks, the system reacts late and many transactions are already passed at a lower confirmation rate (60-120 txs/tick) and therefore with a higher latency (115-125 ticks). For our highest values rp $> 100 ticks$, the system does not even realize that there has been an increase in the incoming transaction rate and does not react, therefore, all transactions are confirmed in one shard, with a low rate (15 txs/tick), thus a high latency (132 ticks) unlike Yggdrasil which shows optimal performance with a reactive confirmation rate, thus a lower latency.

6.5 Cross-shard volume

This section studies the impact of various cross-shard transactions volumes on the performances of Yggdrasil. The volume of cross-shard transactions is defined as the ratio of the number of cross-shard transactions to the total number of transactions at a given time. We vary this ratio to observe its impact on the scalability performances of the system.

6.5.1 Experiment setting. Additionally to the experiments settings defined in section 6.3.1, we vary the cross-shard transaction probability p_c from 0 to 1 to observe the impact of cross-shard transactions. Note that p_c represents the probability that each time a transaction is created, it involves two users from two different shards. Transaction creation rate is set to $f_t = 640$ txs/tick.

6.5.2 Experiment results. The main results of our experiments appear in the graphs of Figure 2d. Note that in all the graphs, points are linked together with lines. This is only for readability reasons. Figure 2d shows the average transaction latency as a function of the cross-shard transaction probability p_c. The main observation is that with dynamic or static sharding solutions whatever the number of shards n is, latency increases as p_c increases. It also decreases as n increases and is extremely low for Yggdrasil (as shown in Section 6.3) since the number of shards in this specific scenario depends on the transaction arrival rate.

6.6 2PC algorithm

This section studies the performance impact of our newly presented 2PC algorithm for distributed smart-contracts (see Section 4.1). This algorithm allows to lock a smart-contract while exchanging with other shards during one of its methods' execution. We study the impact of this lock on transaction latency under different scenarios.

6.6.1 Experiment setting. Additionally to the experiments settings defined in section 6.3.1, we study transaction latency (i.e. time spent between creation and confirmation of a transaction) of Yggdrasil while using our 2PC algorithm under three different configurations: (i) no-sharding (ii) static sharding and (iii) dynamic sharding. Transaction creation rate is set to $f_t = 160$ txs/tick. Transactions are all sent to SC_1 which calls SC_2. The addresses of SC_1 and SC_2 have been created so that these two smart-contracts can not be assigned to the same shard (if there is more than one). In this way, in a sharded configuration (at least 2 shards), any call between SC_1 and SC_2 would inevitably trigger our 2PC algorithm.

6.6.2 Experiment results. The main results of our experiments appear in the graph of Figure 2f. It shows the average transaction latency for the three different configurations presented above. The main observation is that in no-sharding solutions (Ethereum for instance), latency is the lowest (50 ticks). When the ledger is state-sharded, the smart-contract needs to be locked for each invoke, which makes transactions wait longer, thus a higher latency. Please note that as said before, only cross-shard calls involve the use of our algorithm, thus smart-contract lock and higher latencies (as can be seen in the static sharding configuration, i.e. 800 ticks). Finally, dynamic sharding solutions such as Yggdrasil allow to have a stable and low latency (110 ticks) despite smart-contract locking. This is a side-effect of our split-merge mechanism. When our system is sharded, only one transaction can be put in a block because this transaction locks the contract which would have to wait for a return from the other smart-contract located in another shard. This underfills leads to shards merging. On the other hand, when our system is not sharded, blocks can be fulfilled because no transaction requires smart-contract locking. This overfill leads to shards splitting. In other words, our system alternates splitting and merging. By doing so, it can confirm transactions in less time than in static sharding solutions but in more time than solutions with no sharding in this particular scenario. Note that in this experiment, there are no financial transaction that could fill blocks, which could hinder a merge. In this case, Yggdrasil would have the same transaction latency as static sharding solutions.

7 RELATED WORK

In this section we compare main sharding solutions against Yggdrasil along six criteria as shown in Table 1.

State sharding support. Most recent solutions in PoS settings aim at implementing state sharding [16, 32, 33, 36, 38, 39], as Yggdrasil does. All these solutions must provide support to cross-shard transactions. Omniledger relies on a two-phase atomic commit protocol driven by the client, where shards do not communicate to each other. Note that the need of a two-phase commit stems from the fact that Ominledger transactions follow a UTXO model where each financial transaction must be verified retrieving all the parent transactions, possibly distributed in different shards. Other solutions relies on inter-shard communication to confirm cross-shard transactions, like Rapidchain [36], Monoxide [16] and Brokerchain

			Elastico [35]	Omniledger [33]	Rapidchain [36]	StakeCube [37]	TON [38]	Elrond [32]	Monoxide [16]	BrokerChain [39]	Yggdrasil
State-sharding	Support		No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
	Smart-Contract	Support	No	No	No	No	Yes	Yes	No	No	Yes
		Atomic-Commit	/	/	/	/	No	No	/	/	Yes
Node-to-Shard Assignment	Model		PoW	PoW/PoX	Offline PoW	UTXO Ownership	PoS	PoS	PoW	PoS	PoS
	Predictability		No	No	No	No	No	No	Yes	Yes	No
Adaptability			Time-driven	Time-driven	Time-driven	Event-driven	Event-driven	Time-driven	Static	Time-driven	Event-driven
Type of Protocols	Intra		BFT	BFT	BA	BA	BFT	SPoS	PoW	PBFT	BFT
	Inter		/	BFT	/	/	BFT	SPoS	PoW	PBFT	BFT
Security Asumptions	Network		Partially synchronous	Synchronous	Synchronous	As required for the BA[8]	Synchronous	Synchronous	Partially synchronous	Synchronous	Partially synchronous
	Failure	Adaptive	Weakly	Weakly	Weakly	Weakly	N/A	Weakly	N/A	N/A	Weakly
		Threshold	25%	25%	33%	33%	33%	33%	50%	33%	33%
Cross-shard transaction reduction			No	No	No	No	No	No	Discussed	Yes	Yes

Table 1: Comparison table of blockchain sharding solutions.

[39], which use *special users* which exist in multiple shards and act as relays between shards. Other approaches [32, 38] use a globally-shared blockchain named masterchain (or metachain) to maintain synchronization between shards and thus confirm cross-shard transactions. Yggdrasil uses a masterchain-based solution to confirm cross-shard financial transactions. As for general smart contracts, atomicity is guaranteed via a two-phase commit protocol among shards. Note that a two-phase commit protocol is not needed for financial transactions in Yggdrasil because of the account-based nature of transactions. As for smart contract support, only [32, 38] manage smart contracts. The support however is only related to the management of smart contract-to-shard assignment but there is no support for atomicity of smart contracts in the general case. To the best of our knowledge, Yggdrasil is the sole academic proposal managing smart contracts in a sharded environment offering a 2PC protocol to assure their atomicity.

Node-to-Shard Assignment. In permissionless settings, node-to-shard assignment must be unpredictable. To this end, the selection and assignment of processes can be based on PoW [16, 33, 35, 36], PoS [32, 38], often coupled with decentralized partitioning (identifier-based, DHT, etc.). Some solutions propose to re-assign regularly nodes to cope with an adaptive adversary [37, 38]. Yggdrasil embraces the same approach. Note that Brokerchain uses a public globally known predictable heuristic to re-assign nodes.

Adaptability (Time/Event-driven). Adaptability refers to the adaptation of the number of shards to a given parameter specified in the protocol, e.g. computational power of the system [35]. We categorize how solutions manage the number of shards according to whether their adaptability is *(i)* static, i.e., the number of shards is fixed [16, 40], *(ii)* time-driven, i.e. the set of shards changes at specific instants of time [32, 33, 35, 36, 39], or *(iii)* event-driven, the set of shards changes automatically when appropriate conditions are met [37, 38]. Yggdrasil falls in the event-driven category, proposing for the first time in the realm of state-sharding solutions a split/merge method to adapt to transaction load without jeopardising the security of shards.

Type of protocols. Intra-shard protocols are typically consensus protocols used to create blocks and elect block creators in each shard. Mostly used consensus protocols in permissionless settings are BFT Consensus [10, 11], Byzantine Agreements (BA) [2] and Nakamoto-style consensus [1, 3, 7]. These protocols are typically used with no or small adaptations in sharded blockchains (see Table 1). Election mechanisms, always in place to establish the nodes that have rights to append blocks, are either based on PoW [16, 35, 36] or on PoS [32, 37, 38]. Yggdrasil relies on the partially synchronous BFT consensus of Tendermint for block creation (which gives immediate block finality and transaction confirmation), while committees are elected with a PoS-based method.

For state-sharding solutions, *inter-shard protocols* can require a BFT protocol [33], an asynchronous communication protocol (e.g. [16]), or synchronous protocols (e.g. BFT synchronous [38], stake-based Nakamoto-style [32]). Yggdrasil uses asynchronous protocols for cross-chain transaction confirmation, using masterchain-based communication among shards.

Security Assumptions. The security of the each solution lies in the robustness to an adversary that can take control of both network and nodes resources. Because of the need of Consensus, which requires partially synchronous networks to function properly, the best possible protection that blockchains can offer is tolerance to an adversarial network affected by temporary network partitions. Note that synchronous solutions [32, 33, 36, 38] are not robust to an adversarial network. As for processes corruptions, the best possible threshold that partially synchronous solutions based on BFT Consensus can tolerate is the 33% threshold. Security in permissionless blockchains is ensured by solutions coping with an adaptive adversary [32, 33, 35–37]. Yggrdrasil relies on a partially synchronous network, tolerates 33% threshold of corrupted validators in each shard and provides security against an adaptive adversary, being the sole (to the best of your knowledge) state-sharding PoS solution providing the proper level of security in a permissionless setting.

Cross-shard transaction reduction. Cross-shard transactions are inevitable when state sharding is used. A simple way to reduce the burden of cross-shard transactions is to let users choose the shards they are interested in, i.e., the ones containing accounts of their sellers and preferred smart contracts. This way transactions do not have to cross different shards. This approach has been mentioned in [16] but without providing any method to implement it. Yggdrasil offers a complete specification of the method. Brokerchain [39] uses a different approach: it proposes a shard formation heuristic to maximize the probability that users interactions take place inside a single shard. The heuristic, however, is fully public, which does not guarantee unpredictability and the required security level in a permissionless setting.

8 CONCLUSION

Incitations des validateurs (ajout txs cross-shards en priorite ?) Assignation basee sur le comportement (SCs préférés) Praesent imperdiet, lacus nec varius placerat, est ex eleifend justo, a vulputate leo massa consectetur nunc. Donec posuere in mi ut tempus. Pellentesque sem odio, faucibus non mi in, laoreet maximus arcu. In hac habitasse platea dictumst. Nunc euismod neque eu urna accumsan, vitae vehicula metus tincidunt. Maecenas congue tortor nec varius pellentesque. Pellentesque bibendum libero ac dignissim euismod. Aliquam justo ante, pretium vel mollis sed, consectetur accumsan nibh. Nulla sit amet sollicitudin est. Etiam ullamcorper diam a sapien lacinia faucibus.

REFERENCES

[1] S. Nakamoto, "Bitcoin : A peer-to-peer electronic cash system," 2009.

[2] J. Chen and S. Micali, "Algorand: A secure and efficient distributed ledger," *Theor. Comput. Sci.*, vol. 777, pp. 155–183, 2019.

[3] B. M. David, P. Gazi, A. Kiayias, and A. Russell, "Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain," in *EUROCRYPT*, 2018.

[4] "Ethereum proof-of-stake consensus specifications." [Online]. Available: https://github.com/ethereum/consensus-specs/tree/52a741f7c6d3bec98e04df3441bc8e7681480877/specs/altair

[5] V. T. Hoang, B. Morris, and P. Rogaway, "An enciphering scheme based on a card shuffle," in *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, ser. Lecture Notes in Computer Science, vol. 7417. Springer, 2012, pp. 1–13.

[6] E. Anceaume, A. D. Pozzo, T. Rieutord, and S. Tucci Piergiovanni, "On finality in blockchains," *CoRR*, vol. abs/2012.10172, 2020. [Online]. Available: https://arxiv.org/abs/2012.10172

[7] V. Buterin, "Ethereum white paper: A next generation smart contract & decentralized application platform," 2013. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper

[8] "Cosmos: The internet of blockchains." [Online]. Available: https://github.com/cosmos/cosmos

[9] M. Bourgoin, "An overview of the tezos blockchain."

[10] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," *CoRR*, vol. abs/1807.04938, 2018. [Online]. Available: http://arxiv.org/abs/1807.04938

[11] L. Astefanoaei, P. Chambart, A. D. Pozzo, T. Rieutord, S. Tucci-Piergiovanni, and E. Zalinescu, "Tenderbake - A solution to dynamic repeated consensus for blockchains," in *4th International Symposium on Foundations and Applications of Blockchain 2021, FAB 2021, May 7, 2021, University of California, Davis, California, USA (Virtual Conference)*, ser. OASIcs, V. Gramoli and M. Sadoghi, Eds., vol. 92. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 1:1–1:23.

[12] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *CoRR*, vol. abs/1710.09437, 2017. [Online]. Available: http://arxiv.org/abs/1710.09437

[13] "Cardano." [Online]. Available: https://github.com/input-output-hk/cardano-node

[14] "Pyethereum." [Online]. Available: https://github.com/ethereum/pyethereum/blob/782842758e219e40739531a5e56fff6e63ca567b/ethereum/utils.py

[15] D. Skeen, "Nonblocking commit protocols," in *In Proceedings of the 1981 ACM SIGMOD international Conference on Management of Data (SIGMOD)*, 1981, pp. 133–142.

[16] J. Wang and H. Wang, "Monoxide: Scale out blockchains with asynchronous consensus zones," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 95–112. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/wang-jiaping

[17] P. Robinson and R. Ramesh, "General purpose atomic crosschain transactions," in *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE, 2021, pp. 61–68.

[18] I. Abraham and D. Malkhi, "The blockchain consensus layer and BFT," *Bulletin of the EATCS*, vol. 3, no. 123, pp. 1–23, 2017.

[19] L. Lamport, R. Shostak, and M. Pease, *The Byzantine Generals Problem*. New York, NY, USA: Association for Computing Machinery, 2019, p. 203–226. [Online]. Available: https://doi.org/10.1145/3335772.3335936

[20] "Whisk: A practical shuffle-based ssle protocol for ethereum." [Online]. Available: https://ethresear.ch/t/whisk-a-practical-shuffle-based-ssle-protocol-for-ethereum/11763

[21] L. A. Rodrigues, J. Cohen, L. Arantes, and E. P. D. Jr., "A robust permission-based hierarchical distributed k-mutual exclusion algorithm," in *IEEE 12th International Symposium on Parallel and Distributed Computing, ISPDC 2013, Bucharest, Romania, June 27-30, 2013*, N. Tapus, D. Grigoras, R. Potolea, and F. Pop, Eds. IEEE, 2013, pp. 151–158.

[22] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, p. 288–323, apr 1988. [Online]. Available: https://doi.org/10.1145/42282.42283

[23] A. Zamyatin, M. Al-Bassam, D. Zindros, E. Kokoris-Kogias, P. Moreno-Sanchez, A. Kiayias, and W. J. Knottenbelt, "Sok: Communication across distributed ledgers," in *Financial Cryptography and Data Security*, N. Borisov and C. Diaz, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2021, pp. 3–36.

[24] E. Anceaume, A. Guellier, R. Ludinard, and B. Sericola, "Sycomore: A permissionless distributed ledger that self-adapts to transactions demand," in *Proceedings of the IEEE 17th International Symposium on Network Computing and Applications (NCA)*, 2018.

[25] E. API, 2022. [Online]. Available: https://docs.etherscan.io/api-endpoints/accounts

[26] Yggdrasil, "Source code," https://anonymous.4open.science/r/Yggdrasil-11E5.

[27] MAX, "Source code," https://gitlab.com/cea-licia/max/.

[28] O. Gutknecht and J. Ferber, "The madkit agent platform architecture," in *Workshop on Infrastructure for Multi-Agent Systems*, 2000.

[29] D. Balouek et al., "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science (CLOSER)*, 2013.

[30] MAX, "Source code," https://gitlab.com/cea-licia/max/models/ledgers/max.model.ledger.tendermint_v2.

[31] J. Göbel and A. Krzesinski, "Increased block size and bitcoin blockchain dynamics," in *2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*, 2017, pp. 1–6.

[32] T. E. Team, "Elrond - A Highly Scalable Public Blockchain via Adaptive State Sharding and Secure Proof of Stake," Tech. Rep., 06 2019.

[33] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "Omniledger: A secure, scale-out, decentralized ledger via sharding," Cryptology ePrint Archive, Report 2017/406, 2017, https://ia.cr/2017/406.

[34] D. Cason, E. Fynn, N. Milosevic, Z. Milosevic, E. Buchman, and F. Pedone, "The design, architecture and performance of the tendermint blockchain network," in *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, 2021, pp. 23–33.

[35] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. Association for Computing Machinery, 2016, p. 17–30.

[36] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: Scaling blockchain via full sharding," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. Association for Computing Machinery, 2018, p. 931–948.

[37] A. Durand, E. Anceaume, and R. Ludinard, "Stakecube: Combining sharding and proof-of-stake to build fork-free secure permissionless distributed ledgers," in *Networked Systems: 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19–21, 2019, Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2019, p. 148–165. [Online]. Available: https://doi.org/10.1007/978-3-030-31277-0_10

[38] N. Durov, "Telegram Open Network," Tech. Rep., 03 2019.

[39] H. Huang, X. Peng, J. Zhan, S. Zhang, Y. Lin, Z. Zheng, and S. Guo, "Brokerchain: A cross-shard blockchain protocol for account/balance-based state sharding," in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, 2022.

[40] H. Tian, P. Luo, and Y. Su, "A centralized digital currency system with rich functions," in *Provable Security: 13th International Conference, ProvSec 2019, Cairns, QLD, Australia, October 1–4, 2019, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2019, p. 288–302. [Online]. Available: https://doi.org/10.1007/978-3-030-31919-9_17