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

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

This paper presents Yggdrasil a sharding solution for permissionless blockchains that supports both payment transactions and general Ethereum-like smart contracts. Yggdrasil allows to split and merge shard dynamically leveraging decentralized mechanisms to assign nodes to shards in a secure way. A new 2PC protocol allows to guarantee the execution of smart contracts distributed across different shards even when shards dynamically re-organise. An experimental study confirms the capability of Yggdrasil to scale and to adapt to transaction load.

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 [1]. 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

Blockchains are peer-to-peer systems where users can exchange digital values without a central validation authority. Operationally, a distributed set of validators uses a consensus mechanism to validate transactions among users. More recently, with the advent of smart contracts, blockchains have become programmable: conditions ruling exchanges among two or more users can be encoded and executed in the blockchain. Thanks to smart contracts new decentralized applications beyond cryptocurrency (e.g. decentralized finance, traceability and audit of supply chains, decentralized digital identity, etc.) can be built in untrusted environments. Smart contracts can be implemented in different ways, but the most popular implementation is the one proposed by Ethereum, where a smart contract is a replicated service running in the blockchain, exposing methods that can be called by submitting transactions. A submitted transaction contains the remote method call and fees transferred to validators that execute the smart contract.

It is well-known that one of the main problems of blockchains is their lack of scalability [1]. Since all the validators must validate all the transactions, this can cause a huge computational and communication cost to validate and synchronize to a single consistent state, which degrades system performances. Recent academic works have addressed this issue by adopting sharding techniques [2–8]. In blockchains, sharding means partitioning transactions in disjoint sets, so that validators handle only a fraction of all transactions in parallel. Initially sharding solutions provided only transaction sharding (e.g. [4, 9]), where transactions where sharded in different sets, but validators contained the whole blockchain's state to verify those transactions. More recently, state-sharding solutions emerged (e.g. [3, 5, 8]), where not only the set of transactions is partitioned in different sets but the state of the blockchain is also chunked so that different validators maintain only a partial view of the system. In these systems a decentralized mechanism assigns validators and transactions to shards, and to adapt to varying transaction load, shards might need to be re-organized at run-time.

When devising a state-sharding solution there exists a trade-off among security and efficiency. Security is particularly important when we target permissionless blockchains, where users and validators can join the system at will. Indeed, dynamic re-organization of shards must ensure to leave in the shards enough validators to verify transactions and to shuffle them over time to protect them against adaptive adversaries. Efficiency, on the other hand, is mainly related to the ability of properly splitting the global state to maximize parallelization, and this over time. Since transactions may have dependencies among them, multiple shards might be involved in their verification. In that case shards need to coordinate through complex atomic protocols that may provoke a performance loss[3]. Up to now, sharding solutions in permissionless settings have mainly focused on cryptocurrencies or special classes of smart contracts managing payment transactions [3, 5, 6, 10]. Payment transactions need a weak form of atomicity called eventual atomicity [10]. Eventual atomicity ensures that if a payment is validated in the buyer's shard, then it will be also validated in the seller's shard. Intuition behind eventual atomicity is that if we assume that liquidity of the buyer is correctly verified in the first shard, then the second shard will also accept the transaction. However, to manage general smart contracts that call other smart contracts eventual atomicity is no more sufficient. In Ethereum invocations among smart contracts are managed in an atomic way: either all

[1]Yggdrasil has no relationship with the Yggdrasil sharding protocol project by THORChain.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

the smart contracts execute or all abort. To make an example consider a smart contract $SC0$ that calls two other smart contracts $SC1$ and $SC2$ in sequence, where both of them realize a transfer of 1 coin. Let's suppose the second transfer will fail (there are many reasons for that, for instance insufficient fees for the execution of the second transfer). In this case the first transfer must be cancelled. In a sharded system for performance reasons $SC1$ and $SC2$ may be assigned in different shards but in this case coordination through atomic-commit protocols can provoke a performance loss. In this paper we propose Yggdrasil, a new sharding system that securely ensures dynamic reconfiguration of shards to adapt to transaction load in a permissionless setting while ensuring consistency of the distributed smart contracts execution through a new two-phase commit (2PC) algorithm among shards. The 2PC algorithm is a type of 2PC working on the call graph generated by smart contracts calls, where shards do not fail but can be re-arranged dynamically during the algorithm execution. The algorithm is based on locking and query-commit transactions routed among shards through a masterchain, a minimal state blockchain globally maintained. Importantly, we have two levels of atomicity to maintain: eventual atomicity for transaction confirmation and atomicity of smart contracts, where 2PC is only used for smart contract atomicity. The adaptivity of the system is based on verifiable conditions on blocks and allows reactivity: as soon as conditions are met the system re-organizes. This allows to be more efficient compared to current solutions and to dynamically find a sweet spot among efficiency of general smart contract sharding and security.

The paper is organized as follows: Section 2 presents basic concepts and definitions, Section 3 the main building blocks and assumptions Yggdrasil relies on, while Section 4 presents Yggdrasil and Section 5 its implementation details. Sections 6 and 7 present an extensive performance analysis and evaluation respectively. Section 8 presents an analysis of the state of the art and Section 9 concludes the paper.

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, 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 [11], VRF [12], PVSS [13], Randao [14, 15]). 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. Consensus protocols are meant to provide 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). About consistency as perceived by users, leader-based permissionless blockchains usually guarantee probabilistic finality [16]. 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., [11, 13, 17]) for the election mechanism. Committee-based permissionless PoS blockchains are generally grounded on variants of BFT Consensus [12, 14, 18, 19] offering deterministic finality. In systems like Cosmos and Tezos [18, 19] a verifiable election mechanism chooses a committee that, once elected, runs the Byzantine consensus protocol (i.e., Tendermint [20] and Tenderbake [21] 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, verifiable and once a block is finalized it can never be revoked. .No matter the model used, 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 [18, 19], where a block is finalised as soon as it is appended to the blockchain.

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. Yggdrasil uses the account-based model 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), which simplifies sharding. This model is also the one adopted by Ethereum for supporting smart contracts [17].

2.2 The Many Faces of Sharding

Sharding, first used in databases, is a method of distributing data across multiple machines with a scaling objective. In blockchains, sharding means partitioning transactions, so that processes handle only a fraction of all transactions in parallel. As long as there is a sufficient number of nodes verifying each transaction for the system to maintain high reliability and security, dividing a blockchain into *shards* will greatly improve the throughput and efficiency of the system. In the ecosystem, sharding exists along three dimensions: network, transaction and state sharding [22].

Network sharding manages the way processes are grouped into shards. This technique is used to optimize communication by letting nodes in the same shard communicate directly with each other rather than having them communicating with the entire network. In this way, nodes only work with the messages sent to their shard(s), saving communication and computational resources.

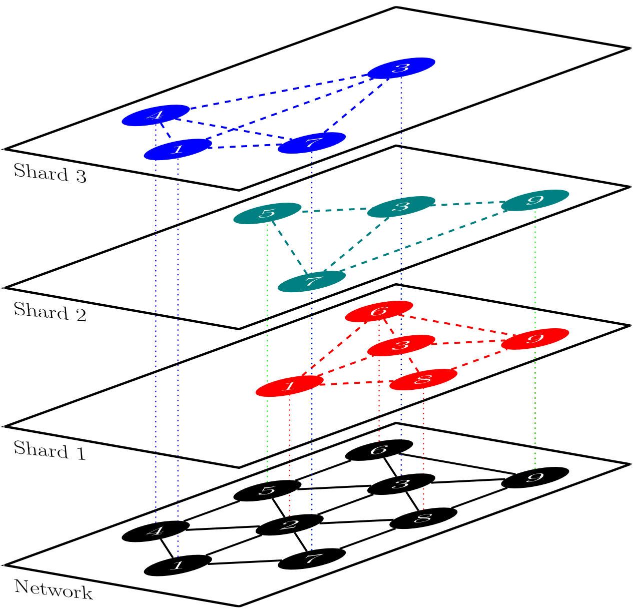


Figure 1: An example of network sharding where each node belongs to a different shard even though they communicate with the same peer-to-peer network: node 9 belongs to shards 1 and 2, node 7 belongs to shards 2 and 3; and node 3 belongs to all shards.

Transaction sharding manages the way transactions are assigned to the shards aiming at achieving parallel confirmation of transactions in multiple shards. It guarantees that a transaction belongs to a unique shard. In fact, transactions need to be confirmed only by nodes in the corresponding shard and not by all the nodes in the system. Moreover, they do not need data from the other shards to compute the validity of any transaction. Therefore, confirmation time can be faster, hence the improvement in throughput and latency.

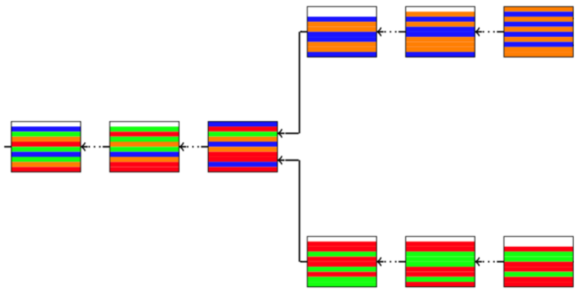


Figure 2: An example of transaction sharding. The colors of the bars in each block illustrate the transaction partition. This provides an intuitive way to see how transactions are partitioned over a DAG is Sycomore [23]: When the DAG is made of a single chain, each block contains transactions of all partitions, which explains the multitude of colors of the blocks. When the DAG becomes larger, the new appended blocks partition the transactions into multiple sets. This explains the partitioning of block colors in the chains.

State sharding aims at splitting the blockchain data structure in different shards. Operationally this implies that each node only maintains a portion of the blockchain data, saving storage and computational resources. This is the most challenging form of sharding, because of the presence of so-called cross-shard transactions, occurring when the transaction recipient does not share the same shard as the transaction sender. This is an issue specific to state sharding, and may require to find a trade-off between the number of shards and cross-chain transactions. Additionally, because shards have only partial views of the system, care must be taken to prevent inconsistencies such as double spending.

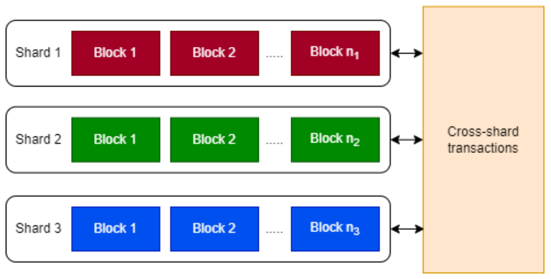


Figure 3: An example of state sharding where each shard keeps its own state and confirms its own transactions. However, when transactions involve more than one shard, cross-shard communication is necessary.

2.3 Smart Contracts

Popularized by Ethereum, many blockchains today (e.g., [19], [24]) 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 [25]. 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.

2.4 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 [26], 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 [10]. 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[2] [27]. 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. 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 observed in [28], specific classes of smart contracts, like ERC-20 contracts, can be divided into smaller ones as their states can be fragmented into non-interfering states, which may increase even more the parallel execution of the smart contract. However, independently from this optimisation, one needs to handle numerous interactions between smart contracts that do not execute in the same shard, or their invocation from users that do not belong to the shard of the smart contract. As detailed in the following, Ygdrasil 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 to dynamically adapt shards to reduce the overload generated by these protocols.

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[3]. 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

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

[3]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.

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 [29]. 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 4.8).

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.

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 [30] that provides deterministic finality [16]. Specifically, we assume that each blockchain is grounded on Tendermint [20], 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 ([12, 14, 18, 21]), 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 [18, 19] or Ethereum PoS [14]. 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 [31]. 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 [32], 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 [33]. 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 2.3) and atomic-commmit of smart contract distributed execution — execution that spans different shards – through a 2PC algorithm based on locking and eventual confirmation among shards. Yggdrasil ensures eventual atomic confirmation during re-organisations of the

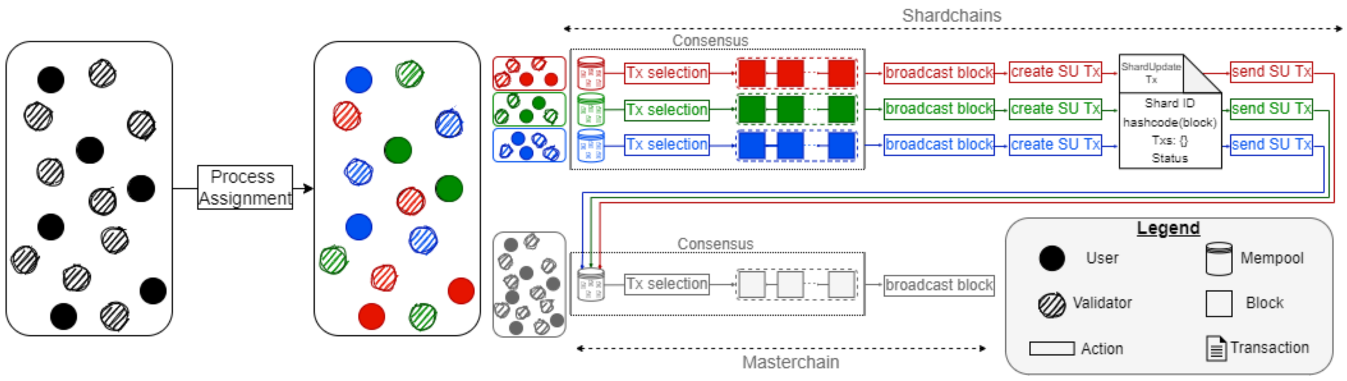


Figure 4: A simple overview of Yggdrasil.

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 contract 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.

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*[4], respectively. Yggdrasil manages all these transactions in a unified way as described below.

Transactions and state sharding. As will be detailed in Section 4.4, 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 [34], 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 4). Transactions are collected by the shard's validators[5], 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 4, 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.

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 Section 5.3.

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

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

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

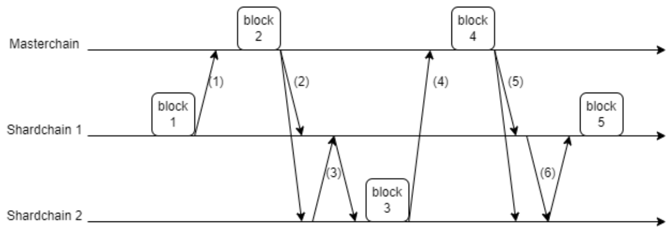


Figure 5: The different steps involved to confirm a cross-shard transaction.

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 [6]. As mentioned in Section 2.3, 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 illustrated by Figure 5 and 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). Implementation details of the confirmation of cross-shard transactions can be found in Section 5.4.

[6]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

In the following the definitions of the confirmation conditions for the different types of transactions.

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.*

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 the distributed execution when smart contracts involved live in different shards. Let us make an explanatory scenario to illustrate the algorithm. Let us suppose to have a user that calls, through 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, 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, front-end smart contract's shard coordinates commit and abort of other shards following an approach where shards committees emit special transactions throughout the process. Specifically, inside committees, validators propose blocks inserting specific transactions. Validators verify the block being sure that 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, 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 us 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 41) 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 a 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 sc_0 for inclusion in successive proposals. As soon as

$tx^{query}_{0,i}$ are confirmed, validators in the recipients shards s_i, read these query transactions (line 3). Note that, at that moment, 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 $tx^{vote}_{i,0}$. As soon as those transactions are confirmed, the coordinator shard compares results to decide either 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^{unlock}_0, while the decision is a cross-shard transaction $tx^{decision}_{i,0}$ 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 (smart contracts are deterministic). After the execution, an unlock intra-shard transaction is inserted in the block tx^{unlock}_i. 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 above, 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 intersection of multiple call graphs. Let us define an *active call graph* $\mathcal{G}$ as a call graph involved in a 2PC protocol that has not terminated yet, i.e. in the marsterchain we have the first QUERY cross-shard transaction issued by the front end smart contract for $\mathcal{G}$ but not yet the DECISION one. In Yggdrasil users can issue transactions that generate intersecting active call graphs, which, if not managed, might induce deadlocks. In our system, the marsterchain is in charge to prevent (when possible) and manage them. Let us remember that the marsterchain does not have the view of the whole active call graphs in advance. However, cross-sharding transactions, come with partial information about their relative call graph $\mathcal{G}$, e.g. the QUERY transactions from a sc_i to $sc_j, sc_z, \ldots$ are batched together in the same block, which gives partial information about $\mathcal{G}$. Leveraging on those information, the marsterchain might detect if a cross-shard transaction tx related to some active call graph $\mathcal{G}' \neq \mathcal{G}$ is targeting some smart contracts already involved in $\mathcal{G}$. In that case, the marsterchain keeps tx pending. tx is processed after that $\mathcal{G}$ is not active anymore. Notice that, even if another cross-shard transaction tx' arrives, starvation is not possible because tx appears in a corresponding shardUpdateTx SU in the marsterchain, which gives a total order among them. If the masterchain cannot prevent a deadlock, leveraging on the information in the masterchain, it can detect it. In that case, the masterchain applies a deterministic order among the active call graphs involved in the deadlock and make the necessary smart contracts revert (without aborting the whole call graph) to let the prioritized active call graph terminate, before resuming the 2PC protocol execution for the remaining active call graphs.

In such a way, the prioritized active call graph terminates before resuming the 2PC protocol execution for the remaining active call graphs.

Addressing the dynamicity of the call graph. In Yggdrasil 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 modify the protocol as follows. Firstly, in the call graph, we treat all the calls between smart contracts 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 he immediately processes it. In this way, we avoid to add latency in the processing of an invocation between two smart contracts living in the same shard.

4.3 2PC Correctness proofs

In the following we abstract away the complexity given by the cross-sharding communications. For conciseness, we abuse our notation to say that a "smart contract issues a transaction", meaning that the shard in which the smart contract lives sends that transaction (after being written in the shardchain and confirmed). In the same spirit, we say that a "smart contracts" delivers a transaction meaning that, the shard in which the smart contract lives received that transaction from the memPool and the state of Yggdrasil chains.

Lemma 5. *Given a call graph $\mathcal{G}$, let $sc_i, sc_j \in \mathcal{G}$ take a decision respectively dec_i and dec_j belonging to the set $\{COMMIT, ROLL-BACK\}$. Then $dec_i = dec_j$.*

Proof. We proceed by construction. Let us first consider that a smart contract sc_i takes a decision in two cases: (i) sc_i is the front end smart contract sc_0 and delivered all the required votes to take a decision dec_0 and inserts it in a transaction DECISION $tx^{decision}_{0,i}$; (ii) $sc_i \neq sc_0$ and delivered a transaction DECISION $tx^{decision}_{j,i}$ carrying the decision dec_i. We need to prove that for all $sc_i \neq sc_0$, we have $dec_i = dec_0$. Since shards are correct, the 2PC protocol is correctly executed, hence sc_0 issues the same $tx^{decision}_{0,i}$ toward all its smart contracts children. Each smart contract child recursively does the same toward its smart contracts children until the while call graph $\mathcal{G}$ is covered. This concludes the proof. □

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:    $ttx \leftarrow tx.targetTx$
            /* query received, target transaction $ttx$ extracted */
5:    if(!$isLocked(ttx.sc)$ $\vee$ ($isFromSameCallGraph(tx)$)) 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:        $targetTxs \leftarrow getTargetTxs(ttx)$
9:        $block\_proposal.insertQueryTxs(ttx, targetTxs, tx)$
10:     else
            /* call graph reaches a leaf */
11:       $res \leftarrow$ 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:   $dtx \leftarrow tx.destTx$;
            /* vote received, dest transaction $dtx$ extracted from $tx$ */
18:   if( $isReadyToCompute(dtx) \wedge isLocked(dtx.sc)$ 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:     $votes \leftarrow getVotes(getAllVoteTxs(tx))$
20:     if($noAbort(votes)$)
21:       $res \leftarrow$ exec($dtx, getResults(getAllVoteTxs(tx))$)
22:     case 1 ($res! = null \wedge isLockOnInvoke() \wedge noAbort(votes)$)
            /* the $dtx$ is the root, a decision is sent */
23:       $insertDecisionTxs(COMMIT, getAllVotesTxs(tx))$
24:       $insertUnlockTx(dtx.sc)$
25:     case 2 ($res! = null \wedge !isLockOnInvoke() \wedge noAbort(votes)$)
            /* the $dtx$ is not root, a vote must be sent to the parent */
26:       $prevQuery \leftarrow tx.queryTx.previousQ.last()$
27:       $insertVoteTx(PREPARE, res, prevQuery, tx)$
28:     case 3 (($res = null \vee !noAbort(votes) \wedge isLockOnInvoke()$)
            /* the $dtx$ is the root, a decision is sent */
29:       $insertDecisionTxs(ROLLBACK, getAllVoteTxs(tx))$
30:       $insertUnlockTxs(dtx.sc)$
31:     case 4 ($res = null \vee !noAbort(votes)) \wedge !isLockOnInvoke()$)
            /* the $dtx$ is not the root, a vote must be sent to the parent */
32:       $prevQuery \leftarrow tx.queryTx.previousQ.last()$
33:       $insertVoteTx(ABORT, res, prevQuery, tx)$
34: for each $tx$ such that ($tx.type = DECISION$) then
35:   $dtx \leftarrow tx.targetTx$;
            /* 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(!$isLocked(tx.sc) \wedge hasCrossShardCalls(tx)$) then
43:     $blockProposal.insertLockTx(tx.sc)$
44:     $targetTxs \leftarrow getTargetTxs(tx)$
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: $callPath \leftarrow prevQueryTx.callPath.add(sourceTx)$
2: $previousQ \leftarrow prevQueryTx.previousQ.add(prevQueryTx)$
3: for each $targetTx \in targetTxs$
4:   $queryTx \leftarrow createTx(QUERY, callPath, targetTx, previousQ)$
5:   $blockProposal \leftarrow blockProposal.add(queryTx)$
```

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

```
1: $destTx \leftarrow queryTx.call\_path.last$
2: $previousV \leftarrow prevVoteTx.previousV.add(prevVoteTx)$
3: $voteTx \leftarrow createTx(VOTE, vote, res, destTx, queryTx, previousV)$
4: $blockProposal \leftarrow blockProposal.add(voteTx)$
```

Algorithm 4 *insertDecisionTxs(decision, voteTxs)*

```
1: for each $voteTx \in votesTxs$ then
2:   $targetTx \leftarrow voteTx.queryTx.targetTx$
3:   $prevVoteTx \leftarrow voteTx.previousVotes.last()$
4:   $decisionTx \leftarrow createTx(DECISION, decision, targetTx, prevVoteTx)$

5:   $blockProposal \leftarrow blockProposal.add(decisionTx)$
```

Lemma 6. *Given a call graph $\mathcal{G}$, let $sc_0 \in \mathcal{G}$ be the front end smart contract of $\mathcal{G}$ and $sc_i \in \mathcal{G}$ be the other smart contracts. If all $sc_i \neq sc_0$ vote for PREPARE, then sc_0 decides for COMMIT.*

Proof. Let us proceed by construction. We need to show that, if all $sc_i \neq sc_0$ vote for PREPARE then, sc_0 collects all those votes and decides accordingly. A smart contract sc_i votes for PREPARE in two cases: upon delivery of a QUERY transaction $tx_{j,i}^{query}$ in the case where sc_i is a leaf of $\mathcal{G}$ or after having collected VOTE transactions (for PREPARE) from all its children in case sc_i is not a leaf of $\mathcal{G}$. In the former case, and by assumption of the proof sc_i votes for PREPARE and issues a transaction $tx_{i,j}^{vote}$ toward its parent sc_j. In the later case, and by assumptions all its children vote for PREPARE, i.e. sc_j receives $tx_{i,j}^{vote}$ from all its children. Hence sc_i votes for PREPARE and issues the transaction $tx_{j,z}^{vote}$ toward its parent sc_z. The procedure continues up to sc_0, which collects all the votes from its children and decides for COMMIT. □

Lemma 7. *Given a call graph $\mathcal{G}$, let $sc_0 \in \mathcal{G}$ be the front end smart contract of $\mathcal{G}$ and $sc_i \in \mathcal{G}$ the other smart contracts. If at least a sc_i votes for ABORT, then sc_0 decides for ROLL-BACK.*

Proof. The proof follows the same spirit as the one of Lemma 6. Let us proceed by construction. Let sc_i be the smart contract that votes ABORT. sc_i issues $tx_{i,j}^{vote}$ where j is the parent smart contract. sc_j upon receipt of ABORT can stop waiting for other votes from its children and issues a transaction $tx_{j,z}^{vote}$ toward its parent sc_j, where the vote is ABORT. The procedure continues up to sc_0, which collects all the votes from its children and decides for ROLL-BACK. □

Theorem 4.1. *Given a call graph $\mathcal{G}$, if there exist some sc_i that votes for ABORT, then all $sc_j \in \mathcal{G}$ decides for ROLL-BACK, otherwise all $sc_j \in \mathcal{G}$ decides for COMMIT.*

Proof. The proof follows from Lemmas 5, 6 and 7. □

Lemma 8. *Given a call graph $\mathcal{G}$, let $sc_0 \in \mathcal{G}$ be the front end smart contract of $\mathcal{G}$. If sc_0 issues a QUERY transaction relative to $\mathcal{G}$ then each $sc_i \in \mathcal{G}$ delivers a QUERY transaction.*

Proof. We proceed by construction. Upon receipt of an INVOKE transaction from an user (Line 41 of Algorithm 1), sc_0 invokes *insertQueryTxs* (Line 45 of Algorithm 1). This function prepares and inserts in the proposed block all the transactions $tx_{0,child}^{query}$ that have to be delivered by the sc_0's children sc_{child} in the call graph $\mathcal{G}$. When some sc_{child} delivers a QUERY transaction (Line 3 of Algorithm 1), the algorithm first checks if the smart contract is already locked. In the affirmative, the transaction is not treated at that moment, except if that transaction results from the same call graph $\mathcal{G}$ as the transaction that previously locked sc_{child}. In this particular case, the algorithm checks if the target transaction ttx on sc_{child} induces a call to another smart contract or not, i.e., if sc_{child} is a leaf of $\mathcal{G}$ or not. If the former case we are done. In the latter case, sc_{child} executes the same steps as sc_0 does. sc_{child} invokes *insertQueryTxs* (Line 6 of Algorithm 1). This function prepares and inserts in the proposed block all the transactions $tx_{child,child's\ child}^{query}$ that have to be delivered by the sc_{child}'s children $sc_{child's\ child}$ in the call graph $\mathcal{G}$. The process continues recursively, and all the vertices in the call graph deliver a QUERY transaction relative to $\mathcal{G}$. □

Lemma 9. *Given a call graph $\mathcal{G}$, let $sc_0 \in \mathcal{G}$ be the front end smart contract of $\mathcal{G}$. If sc_0 issues a decision transaction (either COMMIT or ROLLBACK) relative to $\mathcal{G}$ then all $sc_i \in \mathcal{G}$ deliver it.*

Proof. The proof follows the same spirit as the one of Lemma 5, having that if sc_0 issues a decision, i.e. a transaction decision $tx_{0,j}^{decision}$, then such a transaction is delivered by all smart contract sc_j that are children of sc_0, which recursively propagate a decision transaction toward their children, covering the whole call graph $\mathcal{G}$. □

Theorem 4.2. *Given a active call graph $\mathcal{G}$, then eventually $\mathcal{G}$ is not active anymore, i.e. the 2PC protocol terminates.*

Proof. Termination of the 2PC protocol is proved as follows. Once the front end smart contract issues a QUERY transaction, then eventually a QUERY transaction is propagated to all the smart contracts in $\mathcal{G}$ (Lemma 8). Since all smart contracts receive a QUERY transaction, then all of them eventually lock and vote. For the smart contract leaf this is immediate. Other smart contracts need to wait for their children votes. Since all leaf smart contracts deliver a QUERY transaction then eventually all their parent will vote and recursively up to the front end smart contract sc_0. Finally, since sc_0 collects all the votes, then it can issue a decision and unlock. By Lemma 9 all the smart contracts in $\mathcal{G}$ deliver the decision, apply it and unlock. This concludes the proof. □

4.4 Process-to-shard assignment

Shards are uniquely identified by their label l (the computation of shards' label is described in Section 4.5). At any time, any process is assigned to the (unique) shard whose label minimizes the distance with the process's identifier.

Definition 10 (Distance function). *[23] 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 $D(a, b)$ is the numerical XOR between a and b and is computed as follows.*

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

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 11 (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.*

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

By construction of the shard labels mechanism (see Section 4.5) 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 $D(id_i, S_\ell) < D(id_i, S_{\ell'})$.

The pseudo-codes executed by a newly created process and its assignment to a shard are moved to Section 5.2.

4.5 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 12 (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 re-assigned 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.

4.6 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.

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

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:

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

4.7 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 their volume. We 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 [7]. 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: $<id_{incarnation}, id_{node}, \_>$.

4.8 Dealing with an adaptive adversary

So far, we 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 [8]. 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.

Algorithm 5 describes our procedure to shuffle validators. The reassignment function is parametrized by k. k is the number of blocks that need to be created in a given shard before the adversary is capable of corrupting a new validator in it. 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 reassignment 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 shard it should re-assign to. To do that, the validators first calculate the distance (see Definition 10) 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

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

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

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. Yggdrasil's properties and proofs are presented in Section 6.

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 10. */
```

5 IMPLEMENTATION DETAILS

5.1 User transaction types and structures

Any user transaction tx has a structure <*sender, receiver, payload*> where *sender* and *receiver* are addresses, *payload* is a float or a binary depending on the transaction type:

- **Payment**: A transaction of type payment corresponds to a sending of tokens from one account to another. eg: <A, B, 10> corresponds to sending 10 tokens from account A to account B. This kind of transaction can be of two types: UTXO and accounts.
- **Stake**: A transaction of type stake corresponds to the sending of one token from one account A that wants to put the token in stake to a global address known as STAKEHOLDER. i.e. : <A, STAKEHOLDER, 1> which means A staking 1 token.
- ***Smart Contract* Deployment**: A transaction of type deployment corresponds to the creation of the contract by p_i. eg: <A, *nil*, data>, where data contains the information about the deployed SC. Please note that smart-contracts have their own identifier which derives from their creator's identifier. This is done in order to have the smart-contracts deployed in the same shard as their creator.
- ***Smart Contract* method invoke**: A transaction of type method invocation corresponds to the use of a *Smart Contract* method by the process. ex: <A, SC, data>, where data contains the information about the SC method invoked and its parameters.
- **ShardUpdateTx**: A transaction of type ShardUpdateTx is sent by a shard committee after each newly created block. It contains the hash of the block, the status of the shard and the merkle roots of all transactions contained in the block.

5.2 Joining the network

When a process p_i connects to the network, it follows the general routine as described in Algorithm 6. First, it enters the Yggdrasil network by calling a *join()* function, which aims at synchronizing its state with that of the other processes. During the execution of the *join()* procedure, p_i calculates its assignment to one of the existing shards. Once the process is assigned to a shard, it can participate in the shard by sending/receiving transactions if it has the role of a user, or, if it is a validator, by maintaining the state of the shard with blocks creation and management of split and merge mechanisms.

Algorithm 6 General Routine

```
1: $h_i^m$ := 0
2: $h_i^\ell$ := 0
3:
4: upon arrival in the network
5:   join()
        /* join() is the first action $p_i$ executes when it enters the system, it initializes
        the main structures, connect to others and ask for synchronization. */
6:   ($h_i^m$, $h_i^\ell$) := updateLocalVariables()
7:
8: upon receive block from $C_m(h_i^m + 1)$
        /* $C_m(h)$ being the masterchain committee at height $h$. */
9:   addBlockToMasterchain($h_i^m$ + 1)
10:  $h_i^m$ ++
11:  setProcessAssignment()
12:  getCrossShardTxs($block$)
13:
```

Algorithm 7 Process Assignment

```
1: Input : $\mathcal{S}$
2: Output : /
3:
4: action setProcessAssignment()
5: if (shardOf($p_i$) = nil) then
        /* shardOf($p_i$) returns the label of the shard $p_i$ belongs to or $nil$ if it does
        not belong to any shard. */
6:
7:   $p_i.shard \leftarrow getClosestShard(p_i.identifier, \mathcal{S})$
        /* getClosestShard() returns the closest shard between $p_i$ label and the shard
        labels in $\mathcal{S}$ using the distance function defined in Definition 10. */
```

5.3 Transaction sharding and processing

Let us recall that our system is composed of a dynamic set $\mathcal{S}$ of shards s_ℓ. Each shard maintains its own blockchain, called shardchain. A small dynamic (i.e. its composition changes during execution) set of processes (with validator role) constitutes the committee responsible of maintaining the shardchain, it is denoted by $C^\ell(h)$. As stated earlier, processes are assigned to a shard depending on their identifier and the label of the shard. When a transaction (other than stake) involving a process is broadcast, it is assigned to a given shard s_ℓ. In fact, since process assignment is computable by anyone, any process can calculate the position (in which shard) of another using its identifier (contained in the transaction). Being assigned to s_ℓ, the transaction is then processed and confirmed in one of the blocks of the shardchain maintained by s_ℓ.

As soon as it appends a block to its shardchain, s_ℓ must send a ShardUpdateTx to the masterchain committee (Algorithm 8) in order for it to be eventually added to the masterchain. As said earlier, ShardUpdate transactions contain the latest updates about the shard such as the the hash of the newly appended block, the cross-shard transactions and its shard status.

At masterchain side, since shards work in parallel and independently, we propose here a verification algorithm (Algorithm 9) to check if the received updates are coherent with the current information to maintain a consistent overall state and reject them if they are not. This is done to avoid possible synchronization problems between the shards and thus allow them to evolve correctly. As an example, if a shard has split in the last update, it does not exist anymore, so it can not send updates before its two child shards merge. As shown by Algorithm 9, we consider an update valid iff the label of the shard transmitter is included in the shards set $\mathcal{S}$, which is calculated using the previous blocks of the masterchain.

Algorithm 8 Shardchain Block creation

```
1:  Input: $h_i^\ell$
    $\mathcal{S} := \{\varepsilon\}$
2:
3:  if ($p_i \in C^\ell(h_i^\ell + 1)$) then
4:     $shardBlock \leftarrow$ createShardchainBlock()
5:     $C \leftarrow \{\}$
6:     for each ($s \in \mathcal{S}$)
7:        $MR \leftarrow$ getMerkleRootOf(getCrossShardTxsWith($s, shardBlock$))
              /* getCrossShardTxsWith(s, $block$) returns the list of cross-shard transactions involving s contained in $shardBlock$.
              getMerkleRootOf() returns the merkle root of a list of transactions. */
8:        $C \leftarrow C \cup \{[$getClosestLabelTo($s$), $MR]\}$
              /* getClosestLabelTo(s) returns the closest label to shard s in $\mathcal{S}$ */
9:     send <ShardUpdateTx, hashcode($block$), C, getStatus()> to all processes in $C_m(t)$
              /* hashcode($shardBlock$) returns the hashcode of $shardBlock$.
              getStatus() returns the status of the shard (S/M/R).
              $C_m(t)$ being the masterchain committee at time t. */
10:
```

Algorithm 9 Reception of ShardUpdateTx

```
1:  Input: $h_i^m$, $\mathcal{S}$
2:  upon receive $ShardUpdateTx$ from any shard
3:
4:     if ($p_i \in C_m(h_i^m + 1) \wedge$ emittingShardOf(ShardUpdateTx) $\in \mathcal{S}$) then
5:        addTxToNextMasterchainBlock(ShardUpdateTx)
6:
```

5.4 Cross-shard transactions' confirmation

In the following, we suppose that the shard where p_i belongs is called shard transmitter s^t. The receiving shard, denoted by s^r, is then notified using a ShardUpdateTx confirmed in the masterchain. More operationally, cross-shard transactions are divided in two. First, the s^t processes the transaction as an intra-shard payment transaction and confirms the financial capabilities of the process for the operation. Then, it sends to the masterchain a ShardUpdate transaction SU_1 containing the cross-shard transaction. Upon receving SU_1, s^r asks for the block referenced in it. After its reception and depending on the cross-shard transaction involving another process or a smart-contract, s^r puts different transactions in its shardchain. In the case of a payment transaction, the only added transaction is the original payment transaction. In the case of a smart-contract method invoke, another transaction containing the results of the invoke is put in the block in addition to the original payment transaction. After the creation of the block containing this/these transaction(s), s^r sends a ShardUpdate transaction SU_2 to the masterchain. After receiving it via a masterchain block, shard s^t will also ask for the transactions contained in the SU_2 and in case of a method invoke, put the resulting transaction in its shardchain thus making available the result of the invocation. The cross-shard transaction confirmation process is illustrated in Figure 6 and in the Algorithm 10. Since confirming a cross-shard transaction is a lengthy process, one or both shards may no longer exist at some point in the confirmation process. To cope with this, all messages are sent to the shard with the closest label (using the distance function defined in Definition 10) to the one of the sending/receiving shard.

Algorithm 10 The actions of p_i for confirming a received cross-shard tx.

```
1:  Initialization : $\mathcal{S} := \{\varepsilon\}$
2:
3:  action getCrossShardTxs()
4:  upon receive $block$ containing < CrossTx, _, shardOf($p_i$) > from $C_m(t)$
              /* $C_m(t)$ being the masterchain committee at time t */
5:  for each ($CrossTx \in block$)
6:     if ($CrossTx.label == p_i.label$) then
7:        send <GETBLOCK, CrossTx.label, $getBlockHashOf(CrossTx)$> to all processes belonging to the shard with the closest label to CrossTx.label
              /* $getBlockHashOf(CrossTx)$ returns the hashcode of the shardchain block containing the cross-shard transaction referenced in the masterchain block. */
8:
9:  action confirmCrossShardTxs()
10: upon receive $shardBlock$ as reply to <GETBLOCK, _, _>
11: for each ($CrossTx \in getMyCrossShardTxs(shardBlock)$)
              /* $getMyCrossShardTxs()$ returns the list of cross-shard transactions in $shardBlock$ that involve $p_i$ shard. */
12:    if (isValid($CrossTx$)) then
13:       addTxToMempool($CrossTx$)
              /* $addTxToMempool(CrossTx)$ puts $CrossTx$ in its mempool in order to propose it for future shard blocks. */
14:
```

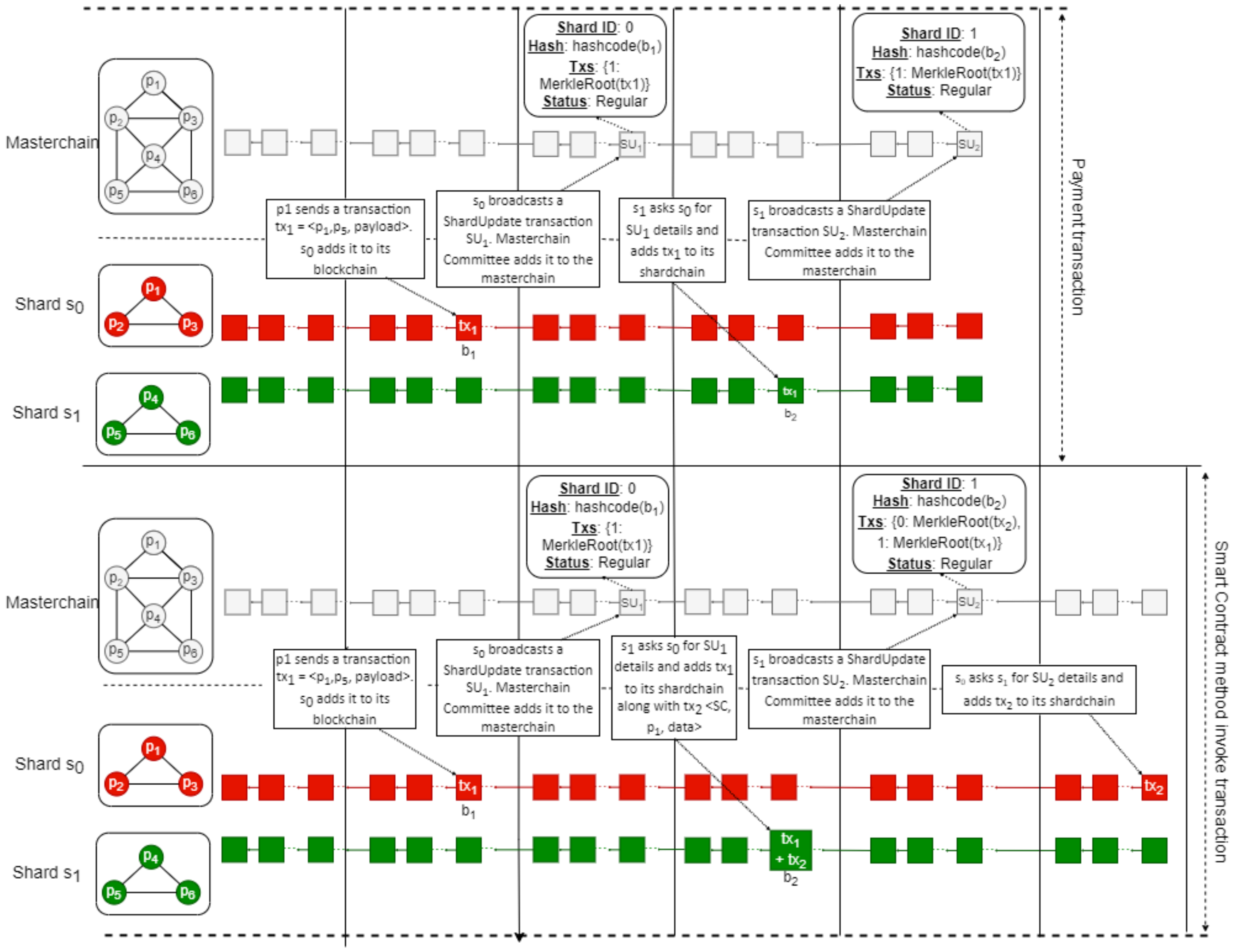

Masterchain
Shard s0
Shard s1
Shard ID: 0
Hash: hashcode(b1)
Txs: {1: MerkleRoot(tx1)}
Status: Regular
Shard ID: 1
Hash: hashcode(b2)
Txs: {1: MerkleRoot(tx1)}
Status: Regular
p1 sends a transaction tx1 = <p1,p5, payload>. s0 adds it to its blockchain
s0 broadcasts a ShardUpdate transaction SU1. Masterchain Committee adds it to the masterchain
s1 asks s0 for SU1 details and adds tx1 to its shardchain
s1 broadcasts a ShardUpdate transaction SU2. Masterchain Committee adds it to the masterchain
Payment transaction
Shard ID: 1
Hash: hashcode(b2)
Txs: {0: MerkleRoot(tx2), 1: MerkleRoot(tx1)}
Status: Regular
s1 asks s0 for SU1 details and adds tx1 to its shardchain along with tx2 <SC, p1, data>
s0 asks s1 for SU2 details and adds tx2 to its shardchain
Smart Contract method invoke transaction
tx1
tx2
tx1 + tx2
b1
b2
SU1
SU2

Figure 6: Handling cross-shard transactions: Yggdrasil relies on the masterchain to handle the confirmation of cross-shard transactions. When the cross-shard transaction tx_1 is broadcast, *(i)* tx_1 is confirmed by the sending shard's committee, *(ii)* a ShardUpdateTx containing the merkle root of tx_1 is sent to the masterchain by the shard committee and *(iii)* it is confirmed by the receiving shard. If tx_1 involves a Smart-Contract, the Smart-Contract's response must be contained in a different transaction tx_2 and sent to the masterchain as a ShardUpdateTx.

6 YGGDRASIL ANALYSIS

6.1 State-sharding

Before analyzing main properties of Yggdrasil, we show that *Yggdrasil implements state sharding*. It means that in Yggdrasil, at any time, when there are a least two shards, two processes in different shards do not maintain the same state. Moreover, the union of the states of the different shards is the state of the whole system. Yggdrasil ensures that if a node is in a shard, it keeps only track of its shardchain, and of the masterchain.

To prove that Yggdrasil implements state-sharding, we need to have a formal definition of what the state sharding is. First, let us define the notion of *state*. The current state of a blockchain system corresponds to the current value of accounts and smart contracts in the system. It can be obtained by the sequential modifications/updates (confirmed transactions) applied to its initial state (genesis block).

As in Definition 1, recall that the state of a shard is the current value of the variables and accounts (smart contracts, users' balance) in this shard. The current state of a shard is the result of the successive modifications of the initial state of this shard. These modifications are the application of the blocks added to the shardchain.

Let us assume that the current shardchain s consists of the chain $b_0^s, \ldots, b_k^s$. The initial state of s is the valuation of the variables and account set in b_0^s. The current state state of s is the current valuations of the variables and accounts after applying blocks $b_1^s, \ldots, b_k^s$ to its initial state. By abuse of language, we say that state′ is the prefix of the state state of shard s, and we denote it state$' \sqsubseteq$ state, if either state$'$ is the initial state of s, or state$'$ is obtained after applying blocks $b^s_1, \ldots, b^s_{k'}$ to the initial state of s, with $k' \in \{1, \ldots, k\}$.

Definition 13 (State-sharding).

- *The state of a node is a prefix of the state of the shard he is member of:*
  $\forall p_i \in s_l, \text{state}_i \sqsubseteq \text{state}_{s_l}$*, where* state_{s_l} *is the state of the shard* s_l.
- *When two nodes are members of the same shard, one's state is necessarily the prefix of the other's state:*
  if $p_1, p_2 \in s_l$*, then* $\text{state}_1 \sqsubseteq \text{state}_2$ *or* $\text{state}_2 \sqsubseteq \text{state}_1$.
- *When two nodes are not the in the same shard, then their state are not prefix of one another. Moreover, the intersection of their states is a prefix of the state all nodes should share. If there is no such state to be shared by all nodes, then the intersection should be empty:*
  If $p_1 \in s_1$ *and* $p_2 \in s_2$ *such that* $s_1 \neq s_2$*, then* $\text{state}_1 \cap \text{state}_2 \sqsubseteq \text{state}_{\text{SharedKnowledge}}$. *When there is no structured shared information between shards,* $\text{state}_{\text{SharedKnowledge}}$ *is always empty.*

Lemma 14. *Yggdrasil implements state-sharding.*

Proof. Before trying to prove that Yggdrasil implements state-sharding, let us recall that in blockchain with state sharding, the state corresponds to the current value of accounts and smart contracts in the system. It can be obtained by the sequential modifications/updates (confirmed transactions contained in confirmed blocks) applied to its initial state (genesis block). One can think that nodes in different shards do not share any information but nodes from different shards can share some information represented in our work with the masterchain (see section 2.2 for more details).

To prove that Yggdrasil implements state-sharding, we prove here each point of definition 13:

- Let p_1 be a process assigned to s_1. p_1 maintains its own copy of s_1's shardchain, which corresponds to its state $state_1$. p_1 receives the blocks of s_1 after a transmission delay δ. It is therefore δ seconds behind the most advanced state of s_1, $state_{s_1}$. p_1's state $state_1$ is therefore prefix of $state_{s_1}$.
- Let p_1 and p_2 be processes assigned to s_1. p_1 and p_2 both maintain their own copy of s_1's shardchain. One of them, let's say p_1 necessarily receives the blocks and therefore updates its state δ seconds before p_2. p_1's state $state_1$ is therefore prefix of p_2's state $state_2$.
- Let p_1 and p_2 be processes respectively assigned to s_1 and s_2. p_1 and p_2 do not maintain copies of the same shardchain, however, they both maintain copies of the masterchain. Therefore, their states $state_1$ and $state_2$ have nothing in common except the masterchain which represents the shared knowledge.

□

6.2 Safety of the assignment

The safety of Yggdrasil ensures that transactions and processes are well assigned and that the assignments are verifiable by any process. More in detail, we have that (i) *Each intra-shard transaction is assigned to a unique shard*, (ii) *Process assignment is verifiable by any other process*, and (iii) *At any time, each process is assigned to exactly one shard*. Thanks to these properties, no inconsistency can happen due to the assignments. Each process knows which shard it is in and can compute the shard of any other process. Additionally, each transaction is assigned to the shard of its emitter. It means that conflicting transactions will be managed by the same shard, hence preventing the risk of inconsistencies.

Lemma 15. *Each intra-shard transaction is assigned to a unique shard.*

Proof. Let tx be a non-cross-shard transaction that has one sender and at most one receiver (in the case of a smart-contract deploy, there is no receiver). tx is assigned in the shard(s) of the sender and receiver if any. Since at time t, each user is assigned to a single shard and tx is not a cross-shard, then both nodes are necessarily in the same shard s so the transaction is only assigned to a unique shard s. □

Lemma 16. *Process assignment is verifiable by any other process.*

Proof. Let p_i be a process. First, let us consider that p_i is a user, its assignment is static and computed using the ID of the user and the set of shards at a given masterchain height h^m_i (line 11 of Algorithm 6 then line 7 of Algorithm 7). The ID of a user is public information, and the set of shards is computable using the masterchain state (public information). All parameters used to compute p_i's assignment are public, therefore user assignment is verifiable by any process in the system.

Now, let us consider the case of p_i as a validator, its assignment is done using a VRF. As explained in section 4.8, VRFs allow us to verify its output using the public key of the validator (public information), the stake transaction that identifies the validator process (public information) and the unforgeable proof (generated by the VRF) the validator has to send with all its messages. All parameters used to compute p_i's assignment are either public or provided by the process itself, therefore validator assignment is verifiable by any process in the system.

Since a process can either be a user or a validator and its assignment is verifiable in both cases, then process assignment is verifiable. □

Lemma 17. *At any time, each process is assigned to exactly one shard.*

Proof. Let p be a process and id its identifier. Its assignment is computed using the getClosestShard() function (see line 7 of Algo. 7) as specified in definition 11. It uses the distance function (see Definition 10) with id and the set of all shards (computed deterministically at a height h of the masterchain) as input parameters. The result of the distance function is a single shard s_ℓ among those given as input.

Note that the assignment shard s_ℓ is unique because:

- Shards labels satisfy the non-inclusion property [35], which means that a shard cannot be part of another shard.
- The XOR function has the property that for any point a, there exists one and only one point b such that b is at a certain distance d from a.

□

6.3 Eventual confirmation

The liveness property of interest for Yggdrasil is that *all valid transactions are eventually confirmed.* Any intra-shard transaction is assigned to one shard that manages it. If it is valid, it will be confirmed by the shard. On the other hand, cross-shard transactions are managed by two shards. However, if such a transaction is confirmed in the first shard, there is no conflict, and the transaction is correct. Since the transaction is valid, therefore, it will be confirmed by the target shard too.

Lemma 18. *Valid intra-shard transactions are eventually put in a block of the corresponding shard.*

Proof. Let us assume that property P3 is satisfied (see section 7.3). We say that the system is scalable, which means that the average transaction confirmation rate is roughly equal to the average transaction submission rate. If we assume that transactions are processed in order of arrival, then no transaction is processed before an older transaction.

Since property P3 is satisfied and transactions are processed in order of arrival, therefore, all transactions are eventually processed.

More precisely, at time t, a shard processes all its intra-shard transactions submitted at time $t' \leq t - \delta$ (where δ finite is the time of transfer and require to process a transaction), and if they are valid, the shard puts them in its shardchain. □

Lemma 19. *Valid cross-shard transactions are eventually confirmed.*

Proof. A cross-shard transaction is confirmed by the system, if it is confirmed in both shards involved (Definition 4).

Let tx_0 be a valid cross-shard transaction involving shards s_1 and s_2. We prove here that tx_0 is necessarily confirmed in the system. Let tx_1 and tx_2 be the two components of tx_0 concerning respectively shards s_1 and s_2. Since tx_0 is valid, then tx_1 and tx_2 are both valid. To prove that the cross-shard transaction tx_0 involving s_1 and s_2 is confirmed, we prove in the following that tx_1 is confirmed by s_1, and tx_2 is confirmed by s_2.

By lemma 18, if tx_1 is valid, it is eventually put in a block, say b_1 in the shardchain of s_1. Once b_1 is appended to the shardchain of s_1, a ShardUpdateTx SU_1 containing tx_1 is sent to the masterchain (cf. line 9 of Algo. 8). Since SU_1 is necessarily valid, by Lemma 18, it will be put in the masterchain, which confirms b_1, and by extension tx_1. As defined in equation 2, SU_1 contains the label of the shard s_1, the hash of b_1, the status of the shard s_1 and the set of cross-shard transactions contained in b_1).

Thanks to the presence of SU_1 in the masterchain, s_2 is notified of the presence of a cross-shard transaction in block b_1 (line 4 of algorithm 10). s_2 then asks to receive b_1 and thus tx_1 (line 7 of algorithm 10) then inserts tx_2 in a newly created block b_2 appended to its shardchain. In the same way as b_1, b_2 is then referenced in the masterchain using a ShardUpdateTx SU_2, which confirms tx_2.

We have that tx_1 is confirmed by s_1 and tx_2 is confirmed by s_2. Therefore, tx_0 is confirmed in the system (Definition 4).

□

6.4 Security

The security properties concern the guarantee Yggdrasil provide against adversaries. Concretely, we have that in Yggdrasil, (i) *Validators (re-)assignment is unpredictable in advance (before a new block is appended to the masterchain)*, (ii) *Validators are dynamically re-assigned*, (iii) *No validator has control on how it is (re-)assigned.* More importantly, *in Yggdrasil, with high probability, at any time, no shard is corrupted.* Thanks to these properties, the adversary cannot predict in which shard a validator would be. Therefore, it will be complicated to target a given shard to compromise it. These properties hold thank to the impredictability of the seed used for the (re-assignement), since validator can predict it in advance.

7 PERFORMANCE EVALUATION

We evaluated 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 [36] 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 [37].

7.1 Simulator and experimental environment

We used an agent-based simulation framework dedicated to blockchain systems, called Multi-Agent eXperimenter (MAX) [38] based on the MaDKit framework [39]. 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 [40]. 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.

7.2 Simulation model

The Yggdrasil protocol has been implemented in the simulator on top of an implementation [41] of the Tendermint protocol[20], used for each shard. As for the generated workload we used for the scalability study a set of historical ethereum transactions containing records of the past 2 years, namely from 13/03/2020 to 14/03/2022

[36]. For reactivity and the impact of 2PC algorithm, synthetic workloads have been generated.

For all the experiments presented in the paper 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 [42], reducing the block capacity does not affect the behaviour of the protocols.

For each experiment, we have run sufficiently many simulations to get a confidence interval equal to 5 ± %. For each experiment,

7.3 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[10] with a number of shards n throughout the simulation.

7.3.1 Experiment setting. The overload threshold Γ 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 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.

7.3.2 Experiment results. The main results of our experiments appear in Figures 7a, 7b and 7c. Note that in all the graphs, points are linked together with lines. This is only for readability reasons.

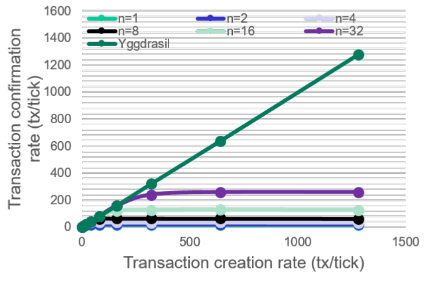
Figure 7a shows the confirmation rate of transactions as a function of their submission rate f_t. The main observation regarding static sharding solutions is that whatever the number of shards n is, they show a limited transaction confirmation rate (e.g. approximately 200 txs/tick for $n = 32$ shards). 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 [43], 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 7b 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 7c shows the average number of transactions that accumulate at the end of the simulation before being embedded in blocks. The number of unconfirmed transactions shows that Yggdrasil has a better confirmation capacity than systems with a static number of shards shown by a lower number of pending transactions at the end of the simulation. Note that the number of pending transactions is close to 0 but not null because simulations are interrupted while transactions are still arriving, thus not confirmed yet by newly created blocks.

7.4 Reactivity

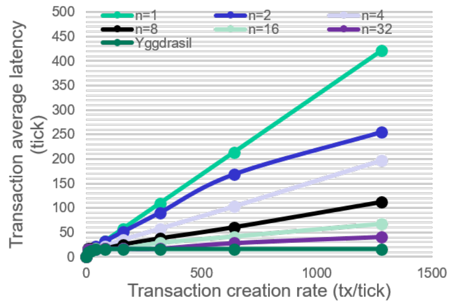
This section assesses the capacity of Yggdrasil to react to sudden and abrupt fluctuations in the creation transaction rate. Additionally, we compare Yggdrasil, which is event-driven, to the time-driven adaptability some solutions of our related-work provide (e.g, Elrond [22], Omniledger [3]). We thus study the reactivity of solutions that adapt the number of shards at specific reconfiguration periods rp.

7.4.1 Experiments setting. As briefly presented in Section 4.5, 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 [3] and Elrond [22] (a day).

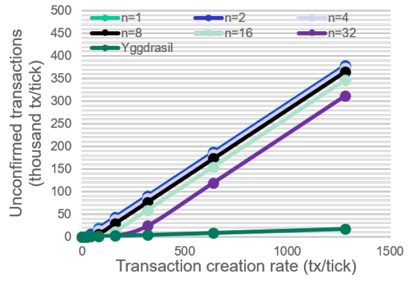
7.4.2 Experiments results. Figure 7e shows the reactivity of Yggdrasil in presence of a load peak (constant function from $t = 1$ to $t = 11$ ticks at $f_t = 500$txs/tick). Yggdrasil initially undergoes a series of splits, it reaches a maximum transaction confirmation rate of 375 txs/tick in order to 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



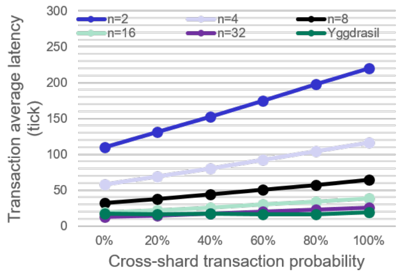
(a) Transaction confirmation rate as a function of the transaction submission rate.



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



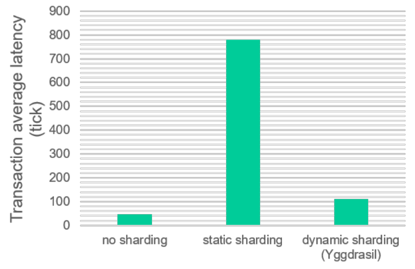
(c) Number of unconfirmed transactions as a function of their submission rate.



(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 [22] and Omniledger [3].**



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

Figure 7: Performance evaluation of Yggdrasil.

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.

7.5 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 system scalability.

7.5.1 Experiment setting. Additionally to the experiments settings defined in section 7.3.1, 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.

7.5.2 Experiment results. The main results of our experiments appear in the graphs of Figure 7d. Note that in all the graphs, points are linked together with lines. This is only for readability reasons. Figure 7d 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 7.3) since the number of shards in this specific scenario depends on the transaction arrival rate.

7.6 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 2PC on transaction latency.

7.6.1 Experiment setting. Additionally to the experiments settings defined in section 7.3.1, 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.

7.6.2 Experiment results. The main results of our experiments appear in the graph of Figure 7f. 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.

8 RELATED WORK

8.1 Evolution of sharding in blockchains

In the past few years many sharding solutions have been proposed to improve blockchain performance.

RScoin [2] is one of the first protocols that implements transaction sharding with the objective of controlling monetary supply: a central bank maintains complete control over the monetary supply, but relies on a distributed set of authorities, or *mintettes*, to collect transactions and prevent double-spending using a two-phase commit protocol. No consensus mechanism is used since mintettes are known and trusted, which makes the protocol strongly permissioned.

Elastico[4] was the first to provide a sharded solution in permissionless settings. Elastico proposes a PoW-based sharded blockchain that combines both network and transaction sharding to scale transaction rates almost linearly with available computational power. PoW-based state sharding has been proposed by Omniledger[3] and Rapidchain[8], showing optimized performances via parallel transaction processing. More recently, Monoxide [10] proposed a state-sharding solution proposing asynchronous synchronization among shards. Each shard maintains its blockchain through a PoW Nakamoto consensus. Authors aims at implementing *eventual atomicity* of cross-chain transactions, i.e. atomicity is guaranteed only if neither the source nor the target shards' blockchain fork.

In the realm of Proof-of-Stake (PoS) permissionless settings, StakeCube[9] combines network and transaction sharding in such a way that the number of shards scales sub-linearly with the total number of active UTXOs. Shards validate transactions in parallel, and a Byzantine agreement protocol, run among subsets of shards, collects shard contributions (set of validated transactions) to create blocks. The periodic re-assignment of UTXOs owners to the shards relies on a randomized shuffling technique which allows StakeCube to defend against an adaptive adversary. Shuffling is a form of re-assignment that guarantees that the adversary cannot predict the shards in which nodes will sit.

As for state sharding in PoS settings, some popular cryptocurrencies, such as TON[44] and Elrond[22], propose a state sharding blockchain capable of adapting the number of shards at run-time. Both solutions, however, rely on synchronous network assumptions while security assumptions are unclear. Brokerchain [5] focuses on cross transactions issues, proposing a cross-sharding blokchain protocol that aims at reducing the volume of cross-shard transactions by clustering nodes on the basis of their past exchanges. To this aim BrokerChain proposes a state-graph partitioning algorithm that is executed by a main chain in charge of sharing out nodes among shards.

Gramoli et. al. [45] present a blockchain-agnostic shard management mechanism applicable to sharded blockchain solutions. For their experimentation, they evaluate their solution on a blockchain called CollaChain [46]. The solution presented a novel idea to reconfigure sharding without disrupting the blockchain service through dedicated smart contract invocations.

8.2 Comparison of existing solutions to Yggdrasil

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 [3, 5, 8, 10, 22, 44], 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 [8], Monoxide [10] and Brokerchain [5], which use *special users* which exist in multiple shards and act as relays between shards. Other approaches [22, 44] 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 2PC protocol among shards. Note that a 2PC protocol is not needed for financial transactions in Yggdrasil because of the account-based nature of transactions. As for smart contract support, only [22, 44] 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 [3, 4, 8, 10], PoS [22, 44], often coupled with decentralized partitioning (identifier-based, DHT, etc.). Some solutions propose to re-assign

			Elastico [4]	Omniledger [3]	Rapidchain [8]	StakeCube [9]	TON [44]	Elrond [22]	Monoxide [10]	BrokerChain [5]	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[9]	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.

regularly nodes to cope with an adaptive adversary [9, 44]. 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 [4]. 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 [2, 10], *(ii)* time-driven, i.e. the set of shards changes at specific instants of time [3–5, 8, 22], or *(iii)* event-driven, the set of shards changes automatically when appropriate conditions are met [9, 44]. Yggdrasil falls in the event-driven category, proposing 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 [20, 21], Byzantine Agreements (BA) [12] and Nakamoto-style consensus [11, 13, 17]. 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 [4, 8, 10] or on PoS [9, 22, 44]. Yggdrasil relies on the partially synchronous BFT consensus of Tendermint for block creation while committees are elected with a PoS-based method. For state-sharding solutions, *inter-shard protocols* can require a BFT protocol [3], an asynchronous communication protocol (e.g. [10]), or synchronous protocols (e.g. BFT synchronous [44], stake-based Nakamoto-style [22]). Yggdrasil uses inter-shard asynchronous protocols 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 [3, 8, 22, 44] 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 [3, 4, 8, 9, 22]. Yggrdrasil relies on a partially synchronous network, tolerates 33% threshold of corrupted validators in each shard, being robust against an adaptive adversary, both at network and node level in a permissionless setting.

Cross-shard transaction reduction. 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 approach has been mentioned in [10] but without providing any method to implement it. Yggdrasil offers a complete specification of the method. Brokerchain [5] 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 the required level of unpredictability.

9 CONCLUSION

In this paper we presented Yggdrasil, the first adaptive and secure sharding solution for general smart contracts in a permissionless setting. By combining verifiable decentralized techniques for dynamic sharding and a novel 2PC algorithm, we demonstrated the feasibility of sharding in such a challenging system paving the way to inter-blockchains distributed applications in the future. As future work, it would be interesting to further optimize cross-shard protocols and study how incentives and fees could be re-designed in a system where transactions are routed over multiple shards.

REFERENCES

[1] D. Agrawal, A. El Abbadi, M. J. Amiri, S. Maiyya, and V. Zakhary, "Blockchains and databases: Opportunities and challenges for the permissioned and the permissionless," in *European Conference on Advances in Databases and Information Systems*. Springer, 2020, pp. 3–7.

[2] 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

[3] 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.

[4] 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.

[5] 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.

[6] Y. Tao, B. Li, J. Jiang, H. C. Ng, C. Wang, and B. Li, "On sharding open blockchains with smart contracts," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020, pp. 1357–1368.

[7] M. J. Amiri, D. Agrawal, and A. El Abbadi, "Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019, pp. 1337–1347.

[8] 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.

[9] 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

[10] 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
[11] S. Nakamoto, "Bitcoin : A peer-to-peer electronic cash system," 2009.
[12] J. Chen and S. Micali, "Algorand: A secure and efficient distributed ledger," *Theor. Comput. Sci.*, vol. 777, pp. 155–183, 2019.
[13] B. M. David, P. Gazi, A. Kiayias, and A. Russell, "Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain," in *EUROCRYPT*, 2018.
[14] "Ethereum proof-of-stake consensus specifications." [Online]. Available: https://github.com/ethereum/consensus-specs/tree/52a741f7c6d3bec98e04df3441bc8e7681480877/specs/altair
[15] 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.
[16] 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
[17] V. Buterin, "Ethereum white paper: A next generation smart contract & decentralized application platform," 2013. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper
[18] "Cosmos: The internet of blockchains." [Online]. Available: https://github.com/cosmos/cosmos
[19] M. Bourgoin, "An overview of the tezos blockchain."
[20] 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
[21] 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.
[22] T. E. Team, "Elrond - A Highly Scalable Public Blockchain via Adaptive State Sharding and Secure Proof of Stake," Tech. Rep., 06 2019.
[23] 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.
[24] "Cardano." [Online]. Available: https://github.com/input-output-hk/cardano-node
[25] "Pyethereum." [Online]. Available: https://github.com/ethereum/pyethereum/blob/782842758e219e40739531a5e56fff6e63ca567b/ethereum/utils.py
[26] D. Skeen, "Nonblocking commit protocols," in *In Proceedings of the 1981 ACM SIGMOD international Conference on Management of Data (SIGMOD)*, 1981, pp. 133–142.
[27] 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.
[28] G. Pîrlea, A. Kumar, and I. Sergey, "Practical smart contract sharding with ownership and commutativity analysis," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1327–1341. [Online]. Available: https://doi.org/10.1145/3453483.3454112
[29] I. Abraham and D. Malkhi, "The blockchain consensus layer and BFT," *Bulletin of the EATCS*, vol. 3, no. 123, pp. 1–23, 2017.
[30] 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
[31] "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
[32] 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.
[33] 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
[34] 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.
[35] E. Anceaume, R. Ludinard, A. Ravoaja, and F. V. Brasileiro, "Peercube: A hypercube-based P2P overlay robust against collusion and churn," in *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2008, 20-24 October 2008, Venice, Italy*, S. A. Brueckner, P. Robertson, and U. Bellur, Eds. IEEE Computer Society, 2008, pp. 15–24. [Online]. Available: https://doi.org/10.1109/SASO.2008.44
[36] E. API, 2022. [Online]. Available: https://docs.etherscan.io/api-endpoints/accounts
[37] Yggdrasil, "Source code," https://anonymous.4open.science/r/Yggdrasil-11E5.
[38] MAX, "Source code," https://gitlab.com/cea-licia/max/.
[39] O. Gutknecht and J. Ferber, "The madkit agent platform architecture," in *Workshop on Infrastructure for Multi-Agent Systems*, 2000.
[40] D. Balouek et al., "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science (CLOSER)*, 2013.
[41] MAX, "Source code," https://gitlab.com/cea-licia/max/models/ledgers/max.model.ledger.tendermint_v2.
[42] 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.
[43] 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.
[44] N. Durov, "Telegram Open Network," Tech. Rep., 03 2019.
[45] D. Tennakoon and V. Gramoli, "Dynamic Blockchain Sharding," in *5th International Symposium on Foundations and Applications of Blockchain 2022 (FAB 2022)*, ser. Open Access Series in Informatics (OASIcs), S. Tucci-Piergiovanni and N. Crooks, Eds., vol. 101. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 6:1–6:17. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2022/16273
[46] D. Tennakoon, Y. Hua, and V. Gramoli, "Collachain: A bft collaborative middleware for decentralized applications," 2022. [Online]. Available: https://arxiv.org/abs/2203.12323