Study of the potential of graph-based approaches in blockchains

Mohamed Aimen Djari

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *INFO*

Par
Mohamed Aimen DJARI

« Etude du potentiel des approches à base de graphes dans les blockchains »

« Study of the potential of graph-based approaches in blockchains »

Thèse présentée et soutenue le 06 décembre 2022
Unité de recherche : UMR CNRS 6074 Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA)

Rapporteurs avant soutenance :

Silvia BONOMI	Assistant professor, Université La Sapienza, Rome, Italie
Sébastien MONNET	Professeur, Université Savoie Mont Blanc

Composition du Jury :

Examinateurs :	Bruno SERICOLA	Directeur de recherche, INRIA
	Quentin BRAMAS	Maitre de conférences, Université de Strasbourg
	Pierre-Yves PIRIOU	Ingénieur-chercheur, EDF R&D
Dir. de thèse :	Emmanuelle ANCEAUME	Directrice de recherche, CNRS
Co-encadrante :	Sara TUCCI-PIERGIOVANNI	Cheffe de laboratoire, CEA List

Résumé en Français

Les blockchains sont des systèmes pair à pair dans lesquels les utilisateurs peuvent échanger des biens numériques sans autorité centrale de validation. Une blockchain est un grand livre distribué maintenu grâce à la communication entre les nœuds du réseau. C'est un registre sur lequel sont enregistrées toutes les opérations du réseau, ce qui contribue à sa transparence puisque chaque ajout dans la blockchain peut être lu par tous et pour toujours (tant que le réseau existe). En fonction de l'application souhaitée, le bon fonctionnement de la blockchain repose sur trois piliers communs : (i) la décentralisation, (ii) la sécurité et (iii) le passage à l'échelle aussi appelé scalabilité. Ces trois caractéristiques réunies donneraient naissance à la solution blockchain parfaite. Malheureusement, cette solution est aujourd'hui considérée comme une utopie que l'on appelle le trilemme de la blockchain. Il s'agit d'une croyance selon laquelle une crypto-monnaie ne peut pas combiner ces trois caractéristiques, elle doit nécessairement sacrifier l'un de ces trois piliers. Par exemple, la sécurisation d'un système décentralisé impliquerait trop de restrictions à son passage à l'échelle (par exemple, le Bitcoin, une monnaie sécurisée et entièrement décentralisée mais qui ne confirme que 7 txs/s en moyenne). La décentralisation d'un système sécurisé et scalable ne serait également pas possible car un système décentralisé prend plus de temps pour atteindre un consensus (par exemple VISA, une solution centralisée et sécurisée qui peut confirmer plusieurs milliers de transactions par seconde). Par conséquent, le passage à l'échelle d'une solution décentralisée et sécurisée ne serait pas possible car les restrictions nécessaires à sa sécurité ajoutées au délai de communication inhérent à sa décentralisation l'empêcheraient d'atteindre un consensus (par exemple Ethereum, qui a choisi de diminuer le délai entre deux blocs pour augmenter sa scalabilité au détriment de sa sécurité en augmentant le risque de forks[1]).

Au cours de cette thèse, les enjeux ont vite été ceux de la quête de performance, notamment en matière de scalabilité sans négliger pour autant les deux autres aspects du trilemme. C'est là l'un des apports de l'approche à base de graphes par rapport à la structure classique sous forme de chaine. En début de thèse, nous avons alors réalisé un état de l'art en étudiant les différents systèmes de blockchains à base de graphes existants [1, 2, 3, 4]. Parmi ces solutions figure Sycomore, une blockchain PoW non permissionnée immuable et sécurisée dont la structure est une structure particulière de graphe dirigé acyclique, appelée SYC-DAG. L'une des caractéristiques uniques de Sycomore est l'auto-adaptabilité de la taille du SYC-DAG à la charge (i.e. au taux de remplissage moyen des derniers blocs d'une chaîne) actuelle du système et le fait que la probabilité de fork diminue avec

[1] Si deux mineurs (i.e. membres du système qui participent au maintien de la blockchain) trouvent chacun un bloc valide dans un intervalle de temps réduit, alors ils diffusent leur solution au même moment, on dit alors qu'il y a fork.

l'augmentation du nombre de blocs feuilles dans le SYC-DAG. Une augmentation ou une diminution du nombre courant de transactions émises par les utilisateurs est gérée de manière dynamique par la création ou l'extinction progressive de chaines parallèles dans la structure du SYC-DAG. La décision d'adapter dynamiquement le nombre de chaînes du SYC-DAG à l'activité du système dépend de la charge des blocs récents du SYC-DAG. La vitesse à laquelle les blocs sont créés peut donc être considérablement augmentée sans encourir plus de forks que dans Bitcoin. Puisque la principale différence entre Sycomore et Bitcoin réside dans leur structure, nous avons voulu étudier le comportement de Sycomore afin de mieux cerner l'intérêt de l'utilisation d'une approche à base de graphes. Pour ce faire, et pour les besoins de toutes les études conduites au cours de cette thèse, nous avons utilisé un simulateur basé-agents dédié à la blockchain, appelé Multi Agent eXperimenter (MAX) [5] qui propose des bibliothèques génériques pour développer facilement des protocoles blockchain. En raison de la complexité informatique de nos modèles et des simulations impliquant un nombre représentatif d'agents, toutes les simulations ont été menées sur Grid'5000 [6], une grille de calcul flexible et à grande échelle pour la recherche expérimentale.

L'étude faite sur Sycomore est une étude expérimentale centrée sur la scalabilité et la résilience de Sycomore face à la présence d'adversaires qui chercheraient à nuire à l'efficacité du système. Cette étude nous a permis d'étudier le comportement, les limites ainsi que les perspectives d'amélioration de Sycomore. C'est au cours de cette étude que nous proposons Sycomore^{++}, un protocole blockchain basé sur Sycomore. Il consiste en une amélioration de Sycomore permettant une auto-adaptation de la difficulté du PoW à la structure du graphe. Nous avons ensuite évalué ce protocole via des simulations basées agents qui ciblent la capacité de Sycomore^{++} à relever le défi de la scalabilité dans différents contextes d'exécution. L'une des principales leçons tirées de ces simulations intensives est la capacité de Sycomore^{++} à réduire considérablement le temps de confirmation des transactions par rapport à Bitcoin et Sycomore, à garantir l'équité du traitement des transactions et à s'adapter aux variations du taux de création de transactions. Enfin, nous avons pu mettre en évidence certains scénarios d'attaques susceptibles de pénaliser l'efficacité de Sycomore^{++}. Grâce à nos simulations, nous avons pu montrer que Sycomore^{++} avait bel et bien une meilleure scalabilité que celle de Bitcoin et Sycomore avec des résultats 16 fois supérieurs pour les mêmes paramètres utilisés. Sycomore^{++} arrive à stabiliser la latence (i.e. le temps écoulé entre la soumission et la confirmation d'une transaction), et ce, malgré la fluctuation du débit de transactions, ce qui implique une accumulation de transactions non confirmées beaucoup moins importante que sur Bitcoin. Enfin, nos résultats analytiques et expérimentaux montrent que la sécurité de Bitcoin n'a pas été sacrifiée au profit de sa scalabilité et étant donné que le travail des mineurs est reparti entre les chaines du graphe, la décentralisation de Bitcoin a également été améliorée. publications dans des conférences internationales à comité de lecture. Les résultats de cette étude ont fait l'objet de plusieurs publications dans des conférences internationales à comité de lecture [7, 8] et une conférence francophone avec comité de lecture [9]. Ils nous ont permis de montrer le poten-

tiel des approches à base de graphes en termes de scalabilité tout en améliorant la décentralisation du pouvoir de décision et sans toutefois négliger leur sécurité.

Dans un second temps, après avoir montré l'apport d'une solution classique à base de graphes dans le trilemme blockchain, nous nous sommes penchés sur les solutions de *sharding* qui étant donné leur structure en graphe, nous semblaient être des solutions à base de graphes plus avancées et des plus prometteuses en termes de scalabilité. Le *sharding* étant à l'origine une technique avancée de gestion de base de données, il peut également être utilisé dans le domaine des blockchains. En effet, lorsque l'on fait face à un très grand volume de données comme c'est le cas pour les transactions de crypto-monnaies, il est intéressant, pour les traiter plus rapidement, de les séparer en plusieurs sous-ensembles. La technique du sharding sert dans ce cas à partitionner les données de la blockchain (notamment tout l'historique des échanges réalisés entre ses utilisateurs depuis sa création), et à ne pas concentrer sur les blocs d'une seule blockchain le processus de validation de chaque nouvelle transaction. Avec le sharding, il serait possible de traiter un plus grand nombre de transactions en même temps, car la blockchain serait partagée en plusieurs sous-parties autonomes. Ici encore, avant d'approfondir notre étude sur le sharding, nous avons fait un second état de l'art qui compare différentes solutions de sharding sur la base de critères préalablement sélectionnés. Parmi ces solutions figurent des solutions industrielles prometteuses [10, 11] ainsi que des solutions académiques intéressantes [12, 13, 14, 15, 16]. Cet état de l'art nous a permis de réaliser qu'il y avait un manque à combler dans les solutions existantes. C'est dans ce cadre que nous avons proposé Yggdrasil, une solution de *state-sharding* (i.e. sharding d'état) pour les blockchains non permissionnées qui supporte à la fois les transactions de paiement et les smart contracts.

Yggdrasil permet de diviser et de fusionner les shards de manière dynamique en s'appuyant sur des mécanismes décentralisés pour assigner les nœuds aux shards de manière sécurisée et non prévisible. Nous proposons également au cours de ce travail un nouveau protocole *2-Phase-Commit* permettant de garantir l'exécution de smart contracts distribués sur différents shards, et ce de manière atomique même lorsque les shards se réorganisent dynamiquement. La principale caractéristique d'Yggdrasil réside dans son auto-adaptation à la charge de transactions, de sorte que le nombre de shards s'adapte continuellement pour assurer une confirmation rapide des transactions. Yggdrasil permet aux shards de se réorganiser en cas de charge élevée en se scindant en nouveaux shards, puis de se regrouper à nouveau si la charge de transaction diminue. Alors que la cohérence locale de chaque shard repose sur une blockchain BFT locale, Yggdrasil assure la cohérence globale du système grâce à une blockchain globale connue par tous appelée *masterchain*. Cette blockchain ne contient que les informations nécessaires au maintien de la cohérence globale du système. Yggdrasil permet notamment d'assigner les utilisateurs et les smart contracts aux shards de manière transparente et décentralisée. Lorsque l'ensemble de shards change, les smart contracts et les utilisateurs sont automatiquement réassignés à un shard nouvellement créé (si nécessaire) de manière transparente et vérifiable. Lorsqu'un shard est divisé en deux nouveaux shards, il s'éteint et un résumé de

son état est transféré aux shards nouvellement créés. Yggdrasil garantit que chaque utilisateur soit assigné à tout moment à un seul shard, c'est-à-dire qu'un utilisateur ne peut pas soumettre de transactions à deux shards différents, ou s'il le fait, la transaction est rejetée par l'un des shards, car l'assignation d'un utilisateur à un shard est vérifiable. De la même manière, un smart contract est attribué à tout moment à un seul shard. Yggdrasil garantit la confirmation des transactions cross-shard mais également l'atomicité de l'exécution distribuée d'un smart contract (i.e. une exécution qui s'étend sur différents shards) grâce à un algorithme 2PC basé sur le verrouillage des contrats et la confirmation entre shards. Yggdrasil est tolérant à un adversaire adaptatif : en s'appuyant sur le *shuffle* (i.e. brassage aléatoire des noeuds dans différents shards), les validateurs sont régulièrement réassignés à des shards choisis au hasard pour se défendre contre un adversaire adaptatif. De plus, en utilisant un tirage aléatoire secret et vérifiable, l'assignation des validateurs est imprévisible. Enfin, afin de réduire le volume de transactions cross-shard, Yggdrasil permet aux nœuds de s'*incarner* dans plusieurs shards avec des comptes identifiés de manière unique, afin de réduire le volume de leurs transactions cross-shard. En effet, les nœuds peuvent être intéressés par un smart-contract particulier ou par des échanges avec des utilisateurs spécifiques, de sorte qu'ils s'incarnent uniquement dans le shard où ils échangent le plus et bénéficient d'un temps de confirmation de transaction rapide en s'évitant des transactions cross-shards.

Une étude analytique a d'abord été faite pour vérifier les propriétés d'Yggdrasil. C'est ainsi que nous avons prouvé la sécurité, ainsi que l'atomicité des transactions dans Yggdrasil. C'est dans ce cadre que nous avons par exemple prouvé la confirmation des transactions intra et cross-shard, la bonne assignation des utilisateurs dans un shard donné du système ainsi que l'intérêt du *shuffle* de validateurs dans la résilience du système face à un adversaire adaptatif. Dans un souci de complétude, nous avons également étudié Yggdrasil expérimentalement avec MAX, le simulateur basé agents que nous avons utilisé pour l'étude de Sycomore^{++}. Pour ce faire, nous avons implémenté Yggdrasil en choisissant Tendermint [17] comme moteur de consensus de chaque shard. Le but, entre autres, étant de comparer les performances d'Yggdrasil à celles de Tendermint afin de montrer l'intérêt de faire du sharding dans les blockchains. Au cours de cette étude, nous avons montré la capacité d'Yggdrasil à s'adapter rapidement au débit d'arrivée des transactions, et ce, encore plus efficacement que des solutions concurrentes de la littérature [13, 11]. Nous avons également pu montrer le potentiel d'Yggdrasil en termes de scalabilité avec des résultats très prometteurs, environ 85 fois le débit de Tendermint. Grâce à nos simulations, nous avons également pu montrer qu'Yggdrasil arrivait à stabiliser la latence de confirmation des transactions, et ce, malgré la fluctuation du débit de transactions, ce qui implique une accumulation de transactions non confirmées beaucoup moins importante que sur Tendermint. Nous avons également montré que l'auto adaptation du nombre de shards dans Yggdrasil lui permettait de garder une latence stable malgré un volume de transactions cross-shards grandissant. Pour finir, nous avons montré que l'utilisation de notre algorithme 2PC dans un environnement de sharding dynamique était certes plus coûteux que dans un environnement

sans sharding mais beaucoup moins que dans du sharding statique. Étant donné la capacité de scalabilité indéniable des solutions à base de sharding, nous montrons que ce surcoût lié au sharding n'est pas pénalisant. Grâce à cette technique de sharding d'état en plusieurs shards, nous avons pu obtenir des performances très prometteuses, meilleures que celles de Sycomore^{++} en termes d'évolutivité. Le principal avantage du sharding d'état étant de réduire les frais de communication et de stockage, les solutions qui la mettent en œuvre ont tendance à passer à l'échelle plus facilement. Un de ses autres avantages étant sa capacité à maintenir une forte décentralisation en donnant plus de pouvoir à plus de nœuds dans des shards séparés. De plus, notre étude analytique et expérimentale d'Yggdrasil nous a permis de vérifier sa sécurité. Au moment de l'écriture de ce manuscrit, les résultats de cette étude sur Yggdrasil ont été soumis pour publication à à la conférence internationale avec comité de lecture VLDB 2023 et un rapport technique présentant les résultats est accessible [18].

Pour conclure, notre travail sur les solutions de DAG et de *state-sharding* nous a permis d'établir que les solutions basées sur les graphes sont effectivement une solution très intéressante et prometteuse en termes de scalabilité. De plus, comme cette solution ne sacrifie ni la décentralisation ni la sécurité, elle semble être un bon équilibre entre les trois piliers de la blockchain, a savoir, la décentralisation, la sécurité et la scalabilité.

Mots-clés: Blockchain, Graphe, Scalabilité, Simulation basée agent, Sharding, Performance, Décentralisation, Sécurité, Trilemme.

Abstract

Blockchains are peer-to-peer systems in which users can exchange digital assets without a central validation authority. It is a distributed ledger maintained through communication between the nodes of the network. It is a ledger on which all operations are recorded, which contributes to its transparency since every addition in the blockchain can be read by everyone and forever (as long as the network exists). Depending on the desired application, the proper functioning of the blockchain relies on three common pillars: (i) decentralization, (ii) security and (iii) scalability. A solution that would bring these three characteristics together is currently considered a utopia that is known as the blockchain trilemma, a belief that a crypto-currency must necessarily sacrifice one of these three pillars.

During the course of this thesis, the issues at stake were quickly those of the quest for performance, particularly in terms of scalability without neglecting the other two aspects of the trilemma. We then started by studying Sycomore, an immutable and secure permisionless PoW blockchain with a graph-based structure. It is during the study of Sycomore that we propose Sycomore^{++}, a blockchain protocol based on Sycomore whose main feature is to dynamically self-adapt the number of blocks created to the current number of transactions submitted. The results of this study have been published in the proceedings of peer-reviewed conferences [7, 8, 9].

In a second step, after having shown the contribution of a classical graph-based solution in the blockchain trilemma, we looked at sharding solutions which, given their graph structure, seemed to us to be the most advanced graph-based solutions and the most promising in terms of scalability. It is in this context that we propose Yggdrasil, a state sharding solution for permisionless blockchains that supports both payment transactions and smart contracts. Yggdrasil allows for dynamic splitting and merging of shards by relying on decentralized mechanisms to assign nodes to shards in a secure manner. In this work, we also propose a new 2-Phase-Commit protocol to guarantee the execution of distributed smart contracts on different shards, even when shards dynamically reorganize. An experimental study confirms the ability of Yggdrasil to evolve and adapt to the transaction load with very promising performance, better than Sycomore^{++} in terms of scalability. Since the main benefit of state-sharding is to reduce communication and storage costs, solutions that implement it tend to scale more easily. Another advantage is its ability to maintain strong decentralization by empowering more nodes in separate shards without hindering its security. At the time of writing this manuscript, the results of this study on Yggdrasil have been submitted for publication to VLDB 2023 and a technical report presenting the results is available [18].

Keywords: Blockchain, Graph, Scalability, Agent-based Simulation, Sharding, Performance, Decentralization, Security, Trilemma.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Begin at the beginning," the King said, very gravely,"and go on till you come to the end: then stop."

– Lewis Carroll, *Alice in Wonderland*

Contents

The community is defined as an extended family, as opposed to society, which exists from the moment when communities exchange with each other. Society thus stems from the need to exchange because communities cannot live in autarky. However, the exchange has dangerous limits for the social link, it is always based on trust.

Any transfer of value (money or any asset) usually requires a central entity that plays the role of a trusted third party. For any money transfer, one must go through a bank, which makes sure its client has enough money. After a car accident, one must goes through an insurance company, which judges its client's responsibility before compensating him. When voting to elect the next president of a country, a concerned citizen must go through the government, which checks his rights before letting him vote. And the list goes on... All these innocuous activities are centralized. For any money transfer, what actually happens is that the bank first seizes money and then transfers it to the recipient(s). This implies that there is a more or less long period of time where the money does not belong to neither the sender nor the receiver but to the bank. Usually, all the mechanisms necessary for everyday life follow this centralized pattern, which poses a problem, who is this authority? How much trust should be given to it? Is it worthy of this trust? And more importantly, what choice do we have but to trust it?

This last question has been answered in 2008 with the appearance of Bitcoin. A cryptocurrency solution that aimed to give power back to the "people". No more need to go through a central authority or any trusted third party. On Bitcoin, everyone has to go through a whole network of people who decide in full transparency. At that time, Bitcoin and the technology it is built on, the blockchain, already represented what could allow us to transform our daily actions by diluting our trust in an

individual or entity into a whole network. Unfortunately, to get there, blockchains must be accepted. And to do so, they must satisfy users by meeting their criteria. Among the adoption criteria appears *scalability*, i.e. the ability of a system to adapt its performance in response to increasing demand.

To put it more simply, let's imagine you are in a traffic jam. Instead of making a trip in 10 minutes, you make it in an hour and are annoyed. If we transpose this to cryptocurrencies, imagine waiting for hours for your coffee payment in Bitcoin to be validated. Long-waiting Bitcoin transactions and traffic jams stem from the same problem. A lot of demand at the same time but since the infrastructure was to meet a much smaller demand, users wait longer. In other words, scalability is crucial for mass adoption. If blockchains cannot scale, they will not be able to compete with traditional systems in terms of throughput and latency.

Finally, while scalability emphasizes the best, it is *security* that prevents the worst from happening. Usually, when people are willing to give up their freedom, they do so for their security. This is what has happened with the surrender of individual power to centralized systems that can guarantee the security of exchanges if necessary. Being a new technology, blockchain is subject to many attacks or vulnerabilities. Indeed, due to its transparent nature, it is easier to target it as the different attacks that have occurred so far on blockchains can testify. This shows the importance of such an issue and how it can impact blockchain's approval.

In a nutshell, blockchains' mass adoption will be defined by these three properties: decentralization, security and scalability. In this manuscript, we will focus on these properties with a particular emphasis on the latter and how to improve it without sacrificing any of the other two.

In the following, we describe the context of this thesis in Section 1.1, as well as our contributions in Section 1.2.

1.1 Context

Based on the work of Stuart Haber, W. Scott Stornetta, and Dave Bayer [20], cryptocurrencies were created by Satoshi Nakamoto in 2008. The goal was to create a new decentralized cryptocurrency, Bitcoin. This new digital currency is based on a distributed and public ledger. To this day, we still don't know if Satoshi Nakamoto really exists, his identity (or their identities if it's a group of people) remains unknown. Still, his creation, Bitcoin, was the first digital currency to be able to solve double spending without having to rely on any trusted central authority.

The main features of Bitcoin come from its infrastructure, the blockchain. It is a distributed ledger maintained through communication between nodes in a peer-to-peer network. It is said to be append-only, which means that information can be added to it at the end and it is supposed to be impossible to delete or change it, so the data entered is considered unchangeable. It is a register on which all the operations of the network are recorded, which contributes to its transparency since each addition in the blockchain can be read by everyone and for ever (as long as the

network still exists).

As said before, the blockchain allows us to emancipate ourselves from any controlling body. The power is given to the users who are distributed in the network. Each user executes a precise protocol, the functioning of the blockchain is said to be tolerant to human errors but also to breakdowns. However, human actions are still behind all attacks in Bitcoin, which is still very resilient to corruption since it is no longer a question of corrupting a cog in a system but a large part of it. Because of its decentralized nature, it is more difficult to attack. Indeed, attacking a central system is easier since it is located in one place. However, attacking a decentralized system requires a lot more resources and is therefore not easy to achieve.

1.1.1 Blockchain Pillars

Depending on the desired application, the proper functioning of blockchain relies on certain common pillars. Among the pillars of blockchain, we can find (i) decentralization, (ii) security and (iii) scalability.

1. **Decentralization** specifies how the power is distributed among the nodes in the network. The more power is concentrated at a node or group of nodes, the less decentralized the system is. Decentralization balances the power of decision among all participants. This is in contrast to a centralized system where the power would belong to one person or group of people.

2. **Security** of a blockchain lies in its ability to withstand unexpected events, such as breakdowns or malicious attacks. In general, the security of a blockchain is guaranteed up to a certain centralization threshold. If this threshold is reached, the security is no longer guaranteed. For example, systems where decision power is controlled by very few actors that are known and trusted are said to be more secure than more permissionless systems where not all nodes are trusted.

3. **Scalability** is a term that describes the ability of a system to adapt to a change in the order of magnitude of demand in terms of transaction confirmation, in particular its ability to maintain its functionality and performance under high demand.

These three characteristics put together would give birth to the ultimate blockchain solution. Unfortunately, this solution is today considered as a utopia since such a solution cannot exist. This is what is called the blockchain Trilemma, a belief that a cryptocurrency cannot combine all three characteristics, a blockchain must necessarily sacrifice one of these three pillars. Indeed, securing a decentralized system would imply too many restrictions to scale (e.g. Bitcoin, a secure and fully decentralized currency but which can only confirm 7 txs/s). The decentralization of a secure and scalable system is not possible since a decentralized system takes longer to reach a consensus (e.g. VISA, a centralized and secure solution that can confirm

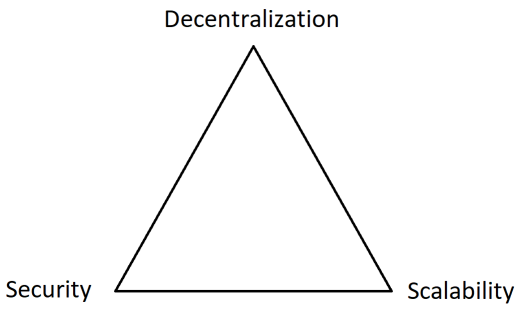


Figure 1.1 – The Blockchain Trilemma.

several thousand transactions per second). Consequently, the scaling of a decentralized and secure solution would not be possible since the necessary restrictions to its security added to the communication delay inherent to its decentralization would prevent it from reaching consensus (e.g. Ethereum, which has chosen to lower its inter-block delay to increase its scalability at the expense of its security by increasing the risk of forks).

1.1.2 Blockchain Fields

Blockchain is a decentralized solution that revolutionized numerous fields. Its particular design has inspired many new solutions since it represents an alternative to the traditional system. This has given rise to new applications, not only in the field of finance with crypto-currencies, but in many diverse fields.

1. **Finance** The use of blockchain in money transfer is one of the most widespread applications. It is generally cheaper and faster than traditional money transfers (e.g. Western Union). This is especially true when the two actors of the transfer are in different countries with different rules. We are talking about several days for classic systems and several minutes or even hours for the slowest blockchains. Still in the field of finance, blockchain can also be used for loan management. Indeed, the appearance of smart-contracts has allowed the implementation of hundreds of "smart" applications. Depending on certain events, actions will be executed automatically. These actions can be for example the payment of a rent, the redemption of a credit... In this way, the management of loans is faster and less expensive.

2. **Insurance** Thanks to the emergence of smart-contracts, the blockchain allows today a better management of insurance and related risks. Indeed, in addition to the reduced processing speed, the insurances offer more transparency regarding their decisions and the registration of all the information in the blockchain would also prevent anyone from wanting to defraud by declaring

the same accident twice for example or by subscribing to two different insurances for the same property and thus receiving the double of the compensation.

3. **Real estate** Real estate transactions can be long and costly. The use of blockchain in this area would reduce processing times related to changes in ownership and increase confidence in title deeds that would be forgery-proof, mainly through NFTs and smart contracts.

4. **Voting** Blockchain could be a great help to democracy. Today, voting implies a considerable waste of time and a significant processing delay. Coupled with digital identity (i.e. personal information kept on the blockchain), blockchain can accelerate these processes while guaranteeing their authenticity (e.g. one vote per person).

5. **Government benefits** Blockchain can also improve the management of social rights by automatically assigning the rights related to each person while avoiding fraud.

6. **Medicine and Health** The management of medical records on the blockchain would allow a faster treatment but also a more interesting global follow-up for the patient. Indeed, having ones medical file available for consultation by doctors would allow a more precise and thorough follow-up of its examinations. Coupled with the management of insurance, even the payments would be done automatically on the blockchain.

7. **Art** Using the blockchain to manage authors' rights when it comes to art is a way to compete with labels that would take a big part of the money intended for the artists. This way, the artist would receive all his rights in full transparency.

8. **Logistics and supply chain tracking** Using blockchain for product traceability is also one of the most widespread applications. This technology would allow to follow the life of each part of a product and thus give more information about its origin but it would also allow a better management of the life cycle of the equipment and a more fluid communication between suppliers and customers.

1.2 Contributions

During these three years, we have studied the veracity of the blockchain trilemma by focusing on the scalability issue. More precisely, we have tried to answer the following questions:

- Is it possible to improve the scalability of existing blockchain solutions while maintaining their decentralization and security?
- Are graph-based blockchains a solution to blockchain's lack of scalability?

This manuscript is organized as follows. Chapter 2 contains the state of the art of what exists in the areas covered by this thesis work. In Chapter 3, we present the tools used to evaluate the performance of our proposals presented in chapters 4 and 5. Chapter 6 will discuss the observations inherent to our results and present the different perspectives associated with them.

Graph-based blockchains A recent evolution in blockchain technology seeks to address the performance issue of permissionless chain-based ledgers, in particular the small number of transactions confirmed per second – around $7tx/s$ for Bitcoin. While some new efforts are dedicated to replace the proof-of-work (PoW) consensus mechanisms with mechanisms such as proof-of-stake and BFT consensus (such as [17, 21, 22, 23]), reaching $10^2 - 10^3$ tx/s, it is undeniable that Bitcoin has shown a great longevity, validating on the ground its good design and security properties in these last 10 years. For this reason other proposals are exploring how to leverage the same design principles, and in particular the simplicity, of Bitcoin protocol. In this line of works some proposals, including [24, 25, 26], called second-layer protocols, propose to implement a protocol on top of Bitcoin that hits the blockchain only from time to time. In this way second-layer transactions are handled at the Internet speed, while only special transactions, needed occasionally to open/close sessions and solve disputes, are translated into Bitcoin transactions. While the idea of off-loading transactions is interesting, these proposals do not specifically address the problem of scalability of the ledger-based PoW itself. In this respect, and to the best of our knowledge (see Section 2.5), Sycomore [27] [1] has been the first ledger protocol, relying on Bitcoin design principles, that addresses Bitcoin's scalability issues: Its graph-structure design allows for the "parallel" creation of valid and durably appended chains of blocks.

In Chapter 4, we analyze the properties of Sycomore and propose Sycomore^{++}, a scalable Nakamoto-style Proof-of-Work protocol based on Sycomore. We present our results and conclusions about its performances in order to define whether or not it improves the scalability of Bitcoin without neglecting its decentralization or security. This would give a definite insight into the potential of graph-based blockchains in terms of scalability.

After having studied the Sycomore protocol, we were convinced of the real potential of graph-based approaches to improve the scalability of blockchains. We were therefore interested in other solutions that used the same principle of graph structure, but in a more advanced way. That's when we realized that Sycomore was doing an underdeveloped form of *sharding*, a solution widely used in databases allowing to subdivide a set into several subsets. Consequently, we became interested in other forms of sharding that would further improve the performance of blockchains.

[1] Sycomore is the french word for sycamore, a large broad-leaved tree tolerant to wind.

Sharded blockchains It is well-known that one of the main problems of blockchains is their lack of scalability [28]. 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 [29, 13, 12, 16, 30, 31, 14]. 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. [12, 32]), 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. [14, 13, 16]), 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 between 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[13].

In Chapter 5, we propose Yggdrasil [2] a new sharding system that securely ensures dynamic reconfiguration of shards to adapt to transaction load in a permissionless setting.

[2]Yggdrasil, in Norse cosmology, is an immense and central sacred tree. Around it exists all else, including the Nine Worlds.

Chapter 2

State of the Art

"If you want to know the future, look at the past"

– Albert Einstein

Contents

In order to improve science and society in general, we have always tried to improve the existing. In order to do this, we need to learn from our mistakes, see what went wrong and improve or replace it. Thus, it is necessary to determine what has been done or not done in the chosen field, a *state of the art*. The objective of this chapter is to lay the foundations of our work through a state of the art of what has been done in the literature in the blockchain field and more precisely in

graph-based and sharded blockchains, which constitute our main interests in this manuscript.

This chapter will be structured as follows, we will start by defining the most common consensus models (Section 2.1), the two well-known transaction models (Section 2.2) and the most widespread incentives for honest participation (Section 2.3).

Then, we will focus on some interesting blockchain solutions, whether they are classical, graph-based or sharding-based, respectively in sections 2.4, 2.5 and 2.6. Finally, we present the most interesting solutions found in the literature that deal with the experimental approach chosen to carry out this research in section 2.7 then finish this chapter with a conclusion in Section 2.8.

2.1 Consensus Models

An abstraction called consensus ensures a clear and unambiguous ordering of valid blocks within the blockchain. Each block is valid if it has been created by respecting the rules of the blockchain construction (e.g., valid signatures for blocks) and contains only valid transactions, where valid is application dependent (e.g., no double spending, positive balances in case of cryptocurrencies). Consensus also guarantees the integrity and the consistency of the blockchain between any correct user.

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 [33], VRF [22], PVSS [21], Randao [34, 35]). 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. It is important to note that leader-based blockchains do not guarantee that exactly one leader is elected at any point of time. Committee-based blockchains, on the other hand, can guarantee that only one committee is elected at any time.

In this section, we focus on both leader-based (Section 2.1.1) and committee-based solutions (Section 2.1.2). Finally, in Section 2.1.3, we provide a thorough comparison of these consensus models.

2.1.1 Leader-based

In this section, we present some of the leader-based consensus models we found most interesting in the literature.

2.1.1.1 Proof-of-Work (PoW)

Applied to some cryptocurrencies since the emergence of Bitcoin [33], Proof-of-Work can be defined as an expensive computer calculation, also called "mining", which must be performed by "miners" in order to create a new block. Essentially, Proof-of-Work requires members of a community to solve challenging puzzles. It is a piece of data that is hard and costly to produce, but easy to verify once it's been generated. In order to solve the puzzle, a miner must resolve an equation, and to do so, it has to do intensive calculations. Additionally, the work that goes into solving the puzzle generates rewards for whoever solves it. More operationally, a miner must make sure that the hash of the block is below a certain target value. To do this, he has to modify the content of the block. Usually, they modify the nonce, a numerical value that serves no other purpose than to modify the internal structure of the block and thus its hash. It will then be necessary to reproduce the operation until having a "valid" hash. Please recall that the hash is a one-way function, one of its essential properties is that it is practically impossible to reverse, that is to say that the hash of a data is simply calculated but on the other hand the reverse calculation of a data from the hash is almost impossible to realize.

Difficulty readjustment. Resolving the Proof-of-Work becomes more difficult as the network grows and becomes more powerful. The more powerful the network is, the faster it creates blocks. To avoid the inter-block delay being too short (to prevent forks for example), the difficulty is periodically readjusted so that there is always an average of 10 minutes between blocks (i.e. the inter-block delay decided by Bitcoin). In Bitcoin, the difficulty is readjusted every 2016 blocks, i.e. every 2 weeks. The recalculation is based on the time spent to create the last 2016 blocks. If the difficulty respected this average of 10 mn between blocks, it would take 2 weeks to create these blocks. Thus, if we had spent less time, the difficulty would have to be revised upwards and vice versa, if we had spent more time, the difficulty would have to be revised downwards. The adjustment of the difficulty is proportional to the time spent over or under the 2 weeks planned to create the 2016 blocks.

Advantages and disadvantages of Proof-of-Work. Because it is very hard to do the work, PoW reduces the risk of a 51% attack [36]. It doesn't rely on a single trusted third party, which builds a "trustless" and transparent network. It also sets a limit on how many new blocks of data can be generated (e.g. one Bitcoin block every 10mn in average).

On the other hand, miners are incentivized to create as many blocks as possible to maximize their rewards. Unfortunately, by using their computing power to try

to create the next block, they consume energy. Since they are competing, a lot of this energy is wasted, which has a significant financial and ecological impact. Additionally, under certain conditions, it can happen that two blocks are created by two different miners at the same time, which leads to a *fork*. Eventually, one of these branches is chosen and the other deleted, which wastes the energy spent on the discarded chain.

2.1.1.2 Proof-of-Stake (PoS)

Since the appearance of Bitcoin and the first usage of Proof-of-Work in cryptocurrencies, researchers around the world struggle to find a way to reach consensus without it. The Proof-of-Work, as robust as it is, is also very energy consuming. It has an energy consumption comparable to that of a state like Netherlands. In view of the current ecological situation (as of writing this manuscript), such consumption cannot be maintained, which is why new approaches have been proposed, including the Proof-of-Stake (PoS) [37]. In Proof-of-Stake, contrary to Proof-of-Work, users do not have to do any calculation to create a block. The computational power of Proof-of-Work is here replaced by users' "wealth". Generally, in this model, users who wish to participate in the creation of blocks are called validators (this name may change depending on the cryptocurrency). They have the same role as miners in Bitcoin, i.e. to select transactions to be validated and to put them in blocks common to all so that all nodes agree on the same account status. In order to become a validator, a user must "lock in" some or all of their wealth, known as "stakes". These are actually funds that a user stakes in order to be eligible to create blocks and thus become a validator. As soon as a user stakes certain funds, they still belong to him, but he can no longer use them, as they serve as collateral.

In Proof-of-Work, the competition between miners is a kind of race where the first one to solve the cryptographic puzzle wins. In Proof-of-Stake, the competition is represented by a comparison of wealth. Indeed, the selection of the next block's creator is random with a selection probability proportional to its number of tokens in stake. As this information is public, everyone can know whether or not they are selected to be the next block creator.

The advantages and disadvantages of Proof-of-Stake. One of the advantages of Proof-of-Stake is its low power consumption. Although PoW is highly secure, it is very energy intensive due to the high demand for computing power. PoS, which does not have huge power requirements, offers an energy-efficient alternative. Furthermore, since PoS does not require any hardware investment, this method is accessible to a larger number of users. This means greater robustness, since in a distributed system, the more nodes there are, the more secure the system is. PoS also offers better performance; it is faster. Since no tedious calculations are required, transactions are validated more quickly. It is also more suitable for massive use with a large number of transactions to be validated, unlike PoW, which always needs more computing power and therefore hardware and energy. To put this

in perspective, during the early years of Bitcoin, one could mine a few blocks using its personal computer. Today, only large groups with dedicated facilities can afford to create blocks quickly. Despite the advantages of PoS over PoW, there are some obstacles that need to be clarified and corrected in order to achieve widespread and sustainable adoption of PoS. The main drawback that can be cited is the security gap. Since PoS is faster and more open than PoW, it is easier for malicious individuals to get the upper hand and work for their own interest. Nevertheless, solutions are being proposed every day to improve PoS and thus offer a powerful, efficient and secure system. One of these solutions, *slashing* [38], consists in punishing a malicious validator, for example, by withdrawing his stakes.

Types of Proof-of-Stake. Let's now look at the different variations of Proof-of-Stake. Indeed, this consensus model being very popular today, new variants appear every day.

- **Delegated Proof-of-Stake (DPoS)**: with this method, token holders can delegate part of their staked assets to validators who have a better chance of being selected. The user who delegates his stakes to another is also considered responsible for the latter's actions, which is why he receives a part of the reward if the validator is selected and acts correctly, but can also be punished if he does not.
- **Liquid Proof-of-Stake (LPoS)**: In LPoS, owners can delegate their voting rights without transferring their funds to a given validation node. This amounts to keeping one's cryptos in one's wallet while transferring the equivalent of decision power to the validator of one's choice. In this way, the token holder is excluded from any penalty for violating the security rules and the delegate is the only one to bear the penalty.
- **Hybrid Proof-of-Stake (HPoS)**: This is a compromise between Proof-of-Work and Proof-of-Stake. This model exploits the strengths of these two consensus mechanisms to increase the security level of the blockchain. Usually, in this type of configuration, miners produce new blocks with PoW, and PoS validators then vote on their validity.

2.1.1.3 Other Proof-of-* models

Other models, a little further away from the basic concept of previously presented models but which can still be considered as variants have emerged.

Proof of Elapsed Time (PoET). Proof of Elapsed Time (PoET) [19] is a consensus model created by Intel (Based on SGX (Software Guard Extensions)). The main idea is that each node waits a random amount of time before it can broadcast a block. This amount of time is generated using a distribution previously specified by the system. SGX helps the node creating a block to generate a proof of the waiting

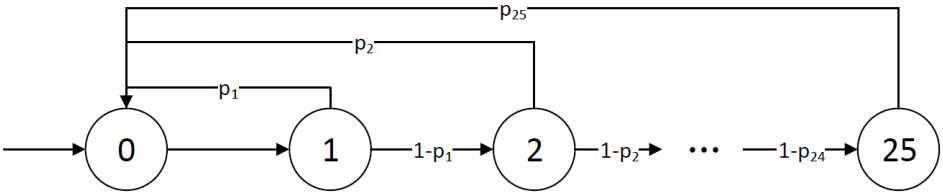


Figure 2.1 – Proof of Elapsed Time [19].

time. This proof can be easily verified by other nodes with SGX. A statistical test is used to determine whether the waiting time indeed follows the specified distribution. This approach has two advantages compared to PoW: (i) less energy consumption since no computation is needed, and (ii) a better fairness since PoET achieves the goal of "one CPU one vote". In PoET, each user has to wait before it can create a block, this waiting time follows a probability distribution F (previously specified by the system), the procedure works as follows:

Definition 1 (PoET)**.** *Each node first uses a formula to generate a number which represents its temporary waiting time. It is said to be in state 0. Whenever a node creates a block, it has a probability p to change its state (1− > 25[1]) thus to use the same temporary waiting time (for 25 times in a row at most). Otherwise, it calculates a new temporary waiting time.*

$$\begin{aligned} &localAverageWait = X \times Y \\ &waitTime = minimumWait - localAverageWait \times log(r) \end{aligned} \tag{2.1}$$

with minimumWait, a fixed system parameter, localAverageWait, calculated by multiplying X (constant value, which corresponds to the most recent blocks to estimate the number of active nodes) by Y (constant value). Finally, r is real number derived from the hash value of the node's previous certificate (uniformly distributed in [0,1]).

Every node has to register in the system both (i) its public/private key pair, which remains unchanged and (ii) the temporary waiting time, which can be updated using Equation 2.1. More operationally, each node uses a finite state machine to control the waiting time updating process. Each node starts from state 0, where it computes a waiting time. Whenever a node goes back to state 0, it updates its waiting time by recomputing a new number. At state $i \in \{1, 2, ..., 25\}$, the node first generates a block with the newest waiting time, and goes to state 0 (with probability p_i) or state i+1 (with probability $1 - p_i$). Probabilities p_i are the same for all nodes and satisfy the condition that $p_1 < p_2 < ... < p_{25} = 1$.

[1]Note that 25 is the number chosen to limit the number of times the same waiting time can be used

Proof of Burn. Proof-of-Burn (PoB) [39] was initially created to solve some of the PoW cons, especially concerning energy consumption. Instead of using computational resources, it uses the idea of burning coins. This method does not have any dependency on computational power and energy. In PoW, a potential miner needs to works for a certain amount of time in the hope of being the first to broadcast a given block. In PoB, it burns coins, which could be immediate. However, since it still needs coins to be generated in order to burn them, Proof-of-Burn needs another way to create blocks. In order to do so, other consensus models such as Proof-of-Work can be used as a money supply for the Proof-Of-Burn to burn it. Since PoB blocks can be produced immediately, one or more PoB blocks can be produced between two PoW blocks, which could improve scalability. Note that blocks created using PoB or PoW are concerned by the same verification conditions. Their only difference being the consensus model used for their creation, PoB blocks are created immediately while PoW blocks need some time.

More operationally, coins are burnt by sending a transaction to a predetermined address (in the same way as stakes are deposited in some protocols). It is impossible to get them back once they are burnt (unlike stakes, we don't get the money back.). In Proof-of-Burn, coins can be used to create blocks only when they are burnt. Once the burning transaction has enough depth in the blockchain (it is confirmed), it can be used to calculate burn hashes.

Definition 2 (PoB)**.** *Each burn transaction has a burn hash, it is calculated as follows:*

$$BurnHash = multiplier * [InternalHash] \tag{2.2}$$

the multiplier is different for each transaction, it follows an expo growth parameterized by the number of proof-of-work blocks found since the creation of its corresponding burn transaction. Note that the multiplier is used to decay the value of burnt coins over time.

2.1.2 Committee-based

In previously presented models, "leader-based" models, a single user is responsible for the creation of a block. In committee-based consensus models, blocks are created and broadcast by a group of validators called *committee*. In order to have an immediate finality [40] , committee-based blockchains satisfy the following properties:

1. **Termination** : Each node must have decided a value by the end of the algorithm.
2. **Integrity** : No node can vote more than once.
3. **Agreement** : If at the end of the algorithm, a node decides on a value, all nodes must have decided on the same value.
4. **Validity** : If a node decides on a value, it must be valid.

One of the main advantages of committee-based consensus is that it is considered fault-tolerant since the power is in the hands of several entities and not just one as in leader-based consensus. Usually, systems that tolerate failures belonging to the Byzantine Generals problem [41]. This problem is one of the most difficult in the literature since it involves no restrictions and makes no assumptions about the type of behavior a node can have, e.g. it can transmit and post fake transactions while pretending to be an honest user, which questions the reliability of the blockchain. The ability to keep an application running smoothly despite some failing, malicious, or simply non-protocol compliant machines is called Byzantine Fault Tolerance (BFT).

Byzantine problem In the case of peer-to-peer networks, consensus is reached when the majority of the nodes in the network agree on a given state. For this to work, nodes must act correctly according to their protocol. For instance, when a correct node receives a message, it forwards it to its neighbors without manipulating it. If all the nodes in the network do this then the system works. Except that among these nodes of the network, there are nodes that we call Byzantine. This term refers to nodes that might deviate from the protocol by delivering false messages or by remaining silent for example. By definition, these nodes have no limits and can harm the proper functioning of the system. A BFT blockchain is therefore tolerant to this kind of nodes. It can work despite their presence in the system by trying to reduce their impact through the dilution of their power in a committee. BFT has had many implementations over the years, the most widespread being the Practical Byzantine Fault Tolerant (PBFT) systems.

Practical Byzantine Fault Tolerance (PBFT) The PBFT [42] has emerged as one of the most interesting implementations of the BFT.

This implementation allows to create a state replication system resilient to byzantine faults in an asynchronous environment. It also provides excellent performance in terms of scalability. Its operation is based on simple principles. The nodes are ordered according to a certain common parameter verifiable by anyone (e.g. stakes). Each node in turn is considered as the leader that must propose a new state to the other users. The outcome of a new state relies on the communication between these nodes under the assumption that the proportion of Byzantine nodes does not exceed 33% of the committee size. Note that in some implementations, the size of the committee corresponds to number of nodes in the committee and in others, it is the number of stakes held by the whole committee.

More operationally, pBFT works in 4 phases:

1. The leader proposes a new state [2] to the other nodes of the committee.

[2] It is important to note that the nodes must have the same common state before deciding on a new state. And the calculation of this state must be deterministic so that all nodes arrive at the same result.

2. The other nodes check this state and give their verdict (i.e. if they approve or not this state).
3. Members of the committee wait for at least $f + 1$ answers (with f being the maximum number of Byzantine in a committee of size n).
4. Members of the committee update their state or change leader if the consensus was not successful. (practical BFT uses a round-robin type format for modifying the leader node in each view.) In this way, if the leader was byzantine, a new, perhaps honest, user can replace it.

Proof of Authority (PoA). Unlike the above solutions, Proof-of-Authority [43] is a permission-based solution. Instead of working or staking coins to prove its right to create a block, PoA simply gives the right to certain nodes in the network to create blocks. These nodes are called *sealers*. In this way, the decision power is a bit more centralized but the security of the network (generally small in the case of permissioned blockchains) is maintained. Let's note that the solutions previously mentioned are absolutely not suitable for such small systems. For instance, using PoW in a permissioned system would make it vulnerable to attacks by powerful users. As for PoS, one would not be safe from a malicious user who could quickly accumulate the majority of stakes and take control over the network. The fact that in PoA, users are known and can be held accountable makes it more appropriate for permissioned uses of the blockchain.

2.1.3 Leader-based vs Committee-based

Let us provide in this section a quick comparison of the previously presented consensus models.

Leader-based solutions are the first solutions that appeared in blockchain solutions along with PoW and Bitcoin. The election of the leader in these systems is usually based on certain criteria such as computing power or the number of stakes. Before its election, its identity remains unknown which prevents it from being targeted by the adversary. Note that in permissioned solutions, this criterion is not taken into account, that's why the nodes able to create blocks are known (PoA) and therefore vulnerable to attacks targeting them.

These models also allow a low communication requirement since only one node is responsible for the creation of a block. However, since the election is done in a probabilistic way, it can happen that more than one leader is elected at the same time, which would imply a consistency problem. This problem is solved by committee-based solutions which generally allow a better consistency. Since the nodes agree on a given state, there are rarely [3] several states at the same time (fork). Nevertheless, it is also this need for communication that often implies a communication overhead and sometimes a strong need for synchronization. It also makes the system more vulnerable to attacks since the adversary has more time to adapt its target.

[3] Forks can happen if the assumption about the proportion of Byzantine in the committee is not satisfied

In both leader-based and committee-based solutions, the election of block creators is verifiable by any node. This is what allows trust and accountability in these systems whether they are permissioned or not.

2.2 Transaction Models

After having seen the different ways to create blocks in the previous section. In this section, let us look at the different transaction models and how they are confirmed.

UTXO vs Account-based models. Bitcoin introduced the fist 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. Note that every 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. Note that in this model, a user can have multiple UTXOs at a time, which can be combined to reach a given amount of money to spend.

In the account-based model each user has one account on which it can receive and spend money within the limits of the available funds. This model is akin to each individual wallet having a ledger of its own. After every transaction, the new balance is computed using basic arithmetics. One of the main advantages of using the account-based model resides in its simplicity since a transaction with an arbitrary amount of money can be performed with one sending account and one receiving account (instead of multiple UTXOs on both sides). This model is used by Ethereum and it is thought to be better suited than UTXO for supporting smart contracts [44] (see section 5.1.2 for further details on smart contracts). As we will see in Chapters 4 and 5, Sycomore^{++} uses UTXOs while Yggdrasil follows the account-based model.

Finalization and Transaction Confirmation. Leader-based permissionless blockchains guarantee weaker properties than the consensus abstraction by offering probabilistic finality [40]. 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., [33, 44, 21]) for the election mechanism.

Committee-based permissionless PoS blockchains are generally grounded on variants of BFT Consensus [45, 46, 34, 22]. In systems like Cosmos and Tezos [45, 46] a verifiable election mechanism chooses a committee that, once elected, runs the Byzantine consensus protocol (Tendermint [17], and Tenderbake [47], respectively)

to append a unique block to the blockchain. These blockchains are said to have deterministic finality, because conditions to determine if a block is finalized are deterministic and verifiable (e.g. in [45], as soon as a block is appended; in [46] as soon as an appended block is followed by another one). Note that in this case finalization is certain, and a finalized block can never be revoked. Ethereum PoS also uses a committee to finalize blocks [48] generated by underlying Nakamoto-style consensus.

When a block is finalized (finalized with high probability in probabilistic models) all the contained transactions are said to be confirmed. As will be shown in Chapter 5, our solution, Yggdrasil adopts a PoS system that guarantees immediate deterministic finality, similar to [45, 46].

2.3 Incentives for honest participation

As the previous section on committee-based consensus showed, for a peer-to-peer system to work, the nodes that compose it must act correctly. Since there is no way to force them to do so, these systems are based on economic incentives. An economic incentive is any specific non-mandatory economic policy measure that seeks to obtain from actors a specific behavior, not necessarily desired by them, in exchange for one or more specific benefits. Incentives are a subject of research in themselves, but in the literature there are some incentives that are well known to the public, such as the ones of Bitcoin or stake-based solutions.

Bitcoin Incentives Bitcoin's incentives are built on two pillars, (i) Bitcoin scarcity and (ii) transaction fees.

1. Bitcoin scarcity: The supply is limited to 21 million BTC, so there can never be more. These Bitcoins are created with each new block created. And this number is designed to decrease over time. It is divided by two, every 210,000 blocks, which is commonly known as halving. As of writing this manuscript, the current reward is 6.25 BTC.
2. Transaction fees: Since the block size is limited, users are competing to get their transactions prioritized. In order to prioritize and to compensate the miner for his work in including these transactions (and to encourage him not to release empty blocks as well), he receives the full fee for the transactions contained in his block. Since the miners have the power to choose the transactions, they usually choose the ones that make the most money.

Unlike transaction fees, the block reward is a temporary incentive initially created to get users interested in Bitcoin. Eventually, Bitcoin will be completely fee-driven, and although this aspect remains little studied except by a few studies that estimate that Bitcoin would be significantly less secure without its block reward, it is not clear that Bitcoin will be able to continue to operate in the future. It seems that the future of Bitcoin lies in fees and the incentive they represent as well as in the rationality of participants.

Stake-based incentives: Slashing Slashing is a feature that provides security by prompting validators to act correctly so as not to endanger the system. When a validator acts in a malicious way, it can have a negative impact on the system. Slashing will therefore consist in punishing this validator and making him lose between 5 and 20% of his stakes, either by redistributing them as rewards to other validators who have acted well, or by destroying them. Double-signing is an example of an attack that can lead to slashing. When a validator signs the same block twice, it is usually an attack to weaken the network. But it can also be a mistake by an honest user. To protect the system, users who are guilty of double-signing, intentional or not, lose their validator privileges and some of their stakes. The goal is to ensure that all users with stakes comply with the rules and maintain the integrity of the system. Unfortunately, if this is not the case, it can be very costly as all or part of the money invested can be lost without any possibility of recovery. Statistically, blockchains that practice slashing tend to be more secure, thanks to the highly punitive dimension of this feature. This is due to the fact that stakes owners have a lot to lose and therefore have an extra motivation to respect the rules of the system and thus participate in its security.

2.4 Classic Blockchains

In this section, we will focus on some of the most interesting "classic" blockchains in terms of properties. By classical, we mean blockchains with a single chain structure not a DAG, as introduced in Chapter 4. We will develop the functioning of Bitcoin, the first cryptocurrency ever created, Algorand a PoS solution allowing the presence of highly adaptive adversaries, as well as Tendermint, a fork-free solution based on the BFT consensus. Finally, we will also present some second layer solutions that can be grafted to many existing solutions in the literature.

2.4.1 Bitcoin

Bitcoin is the pioneer of crypto-currencies, created in 2008 by Satoshi Nakamoto to get rid of the centralization of the monetary system where a trusted third party is needed for each transaction. It is the first protocol to rely on the blockchain technology. In Bitcoin, users sign transactions using their private key before broadcasting them to the network. The whole mechanism is based on cryptography, not only transactions management. In fact, after the transactions have been broadcast, in order to validate it, miners put it in a block that is then appended to the blockchain. Block creation is done under certain conditions. For Bitcoin, the consensus mechanism is the PoW, which means that miners must prove that they have worked enough to create a block. It translates into a cryptographic challenge that the miner has to solve as quickly as possible to be the first to broadcast the block and receive the reward (for more details, see Section 2.1.1.1).

2.4.1.1 Common Bitcoin attacks

51% attack. A 51% attack [36] is a common attack on PoW systems like Bitcoin. It allows a miner or a group of miners holding more than 50% of the network power to take control of it. Since PoW is based on computing power, holding the majority of this power allows anyone to alter the blockchain, to prevent certain transactions from being validated or even to rewrite past history by introducing a fork that it would consider more interesting. Since they hold the majority of the network, malicious miners can have any invalid transaction confirmed and any valid transaction invalidated. This is an extremely powerful attack, but it is also very difficult to carry out, especially in systems with high participation rates because collusion is almost impossible. This is why the decentralization of a system or its popularity increases its security. It is mainly the considerable cost of this attack that prevents it.

Selfish Mining Among the most common problems Bitcoin has to face in the literature is *selfish mining* [49]. It is portrayed as the most significant threat to Bitcoin's security. The idea is that instead of spreading a block as soon as it is created, the malicious miner keeps it for himself and creates his own "private" chain. It is a matter of going faster than the public chain and thus having a head start. When the public chain is almost as long as the private one, the miner broadcasts all or part of his alternative chain, thus creates a fork to destabilize the chain. He takes all the rewards and make sure that the honest miners have wasted their time and energy working on a shorter chain, not taken into account by the majority when resolving the fork. Note that [49] showed that this attack is feasible for any miner that controls more than 33% of the computing power, which is lower than the usually assumed 50% bound.

Importance of block reward in Bitcoin In the literature, there exists other threats to Bitcoin's security such as the importance of static reward in blocks. Indeed, Carlsten et al. [50] argue that the decrease in block reward in Bitcoin can only be bad for Bitcoin's security since the only incentive left for miners would be the transaction fees. With only transaction fees as a reward, the variance of a block reward is very high and forking a block to steal its reward becomes more interesting. The problem is that if the only reward for a miner when creating a block is the transaction fees, any miner would be tempted to fork a block that took the most interesting transactions in terms of fees. We would have a system with more and more forks, and therefore less resistance to 51% attack. We would also have a less "fair" system for the users since the transactions would only accumulate and cause starvation thus increasing latency and transaction fees.

2.4.2 Algorand

Algorand [22] is a proof-of-stake blockchain solution which aims to solve scalability in blockchain to millions of users. Concerning the scalability aspect, their performances are impressive with one block created every 22 seconds in average with 50.000 users and a throughput multiplied by 125 with respect to Bitcoin. In Algorand, consensus is achieved if 2/3 of users are honest, i.e. they make the hypothesis that at least 2/3 of the money is owned by honest users. Essentially, committee members elect themselves randomly based on the users' weights (based on stakes) to avoid Sybil Attacks [51]. Every user in the system can independently determine if they are chosen to be on the committee by computing a VRF (Verifiable Random Function) [52] using their private key and some public information contained in the blockchain. The VRF then returns a proof. Please note that a VRF is a cryptographic function that takes a private key and a public seed as input parameters and outputs a results value with a proof. Only the holder of the private key can compute the VRF, but anyone with public key can verify the correctness of the proof. VRFs are useful to decide the block creator for a certain height without everyone, and especially the adversary, knowing it in advance, which increases the resilience of the election protocol. This cryptographic sortition ensures that a small fraction of users are selected at random weighed by their account balance. It provides each selected user with a priority which can be compared between users and a proof of the chosen user's priority. There may be multiple users who propose blocks and the priority determines which block to adopt. Additionally, committee members speak once to prevent adversaries to target them.

More operationally, each user initializes the Byzantine Agreement with the highest priority block they received, checks if it has been selected, broadcast a message which includes the proof and repeat these steps until enough users reach consensus.

Final and tentative consensus Algorand's Byzantine Agreement protocol (BA) can produce two kinds of consensus : final and tentative [22]. If one user reaches final consensus, this means that any other user that reaches any type of consensus must agree on the same value.

Tentative consensus means that other users may have reached tentative consensus on another value. A tentative block is confirmed only if and when a successor block reaches final consensus, which means that a final block confirms not only its transactions but also its predecessors' transactions.

2.4.3 Tendermint

While Algorand proposes asynchronous probabilistic BFT with the usage of VRFs for leader selection, Tendermint [17] proposes a deterministic synchronous BFT protocol. It manages to work correctly even if up to 1/3 of the machines are Byzantine, i.e. they work in an arbitrary way and do not follow the protocol. The consistency of the replicated application lies in the fact that all non-Byzantine users have the

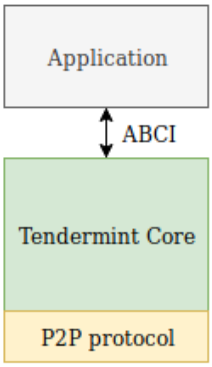


Figure 2.2 – Tendermint Architecture.

same state, and the same copy of the blockchain. In distributed systems, security and consistency are fundamental; they play a key role in fault tolerance for a wide range of applications.

Unlike the previously presented solutions, Tendermint has a modular architecture. It separates the consensus engine from the blockchain application. More operationally, Tendermint [17] relies on two main components: (i) a consensus engine, called Tendermint Core, which ensures that transactions are recorded in the Tendermint Core, which itself is replicated on all the nodes of the network, and (ii) a Tendermint application to connect users, allow them to exchange transactions, but also to validate these transactions in order to propose them to the Tendermint Core. As illustrated in Figure 2.2, the application and the Tendermint Core communicate through the Application BlockChain Interface (ABCI) [17], which allows transactions to be processed in any programming language. Unlike other blockchain solutions that require users to conform to a whole new framework, developers can use Tendermint for the replication of applications written in any language they want.

In order to work properly, the ABCI uses standardized messages:

- **DeliverTx**: a message used to report transactions validated by the blockchain to the application, which involves state changes.
- **CheckTx**: a message used to check the transactions contained in the mempool before moving on to create a new block.
- **Commit**: a message used for the validation of a block, which momentarily blocks the mempool.

Block creation in Tendermint Before being grouped in blocks, transactions are stored in the mempool of each node. This mempool is a local memory cache where transactions are stored after being validated thanks to the CheckTx message. They are then broadcast to other peers in the form of an ordered list from which transactions are drawn to create "blocks". These blocks are then voted on by the

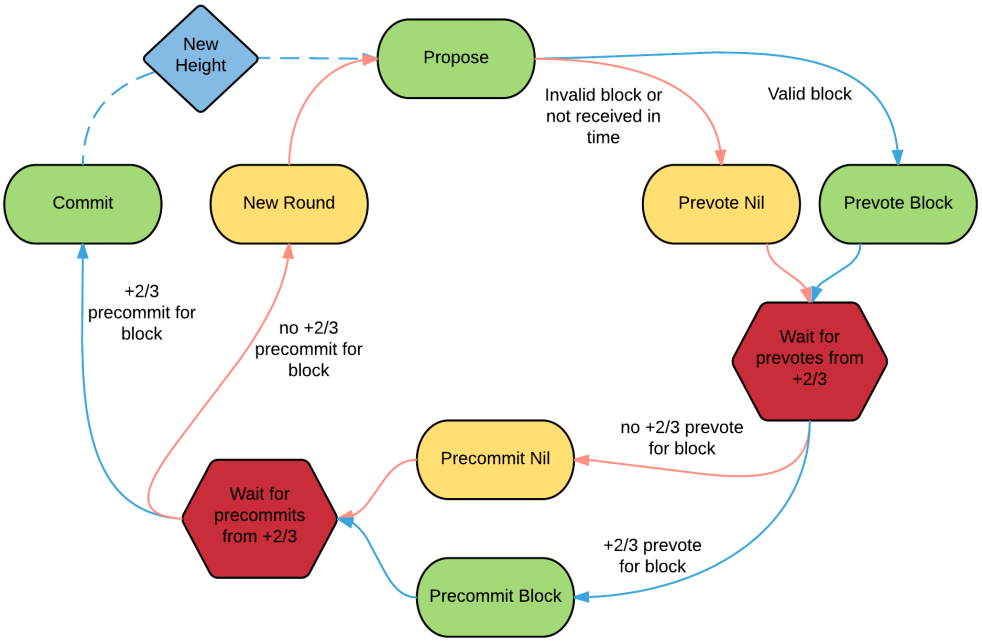


Figure 2.3 – Tendermint Core [17].

committee members. This mechanism takes place as illustrated by Figure 2.3 and is done in rounds.

Each round consists of three phases, (i) a sending phase where the committee is created based on the distribution of stakes, and a node is chosen among the others to be the proposer of the next block. (ii) An intermediate phase where each node reports everything it has received up to time t. Then (iii), a last computation phase where each nodes takes into account all the received data to define its new state.

2.4.4 Second Layer Solutions

So far the solutions we have presented are Layer 1 solutions, which means that they define the base protocol. On the other hand, Layer 2 solutions refer to a secondary protocol that would exist on top of an existing blockchain protocol (of Layer 1). In this section, we will see some Layer 2 solutions that we found interesting in terms of scalability.

2.4.4.1 BitcoinLightning

When it was released in 2008, one of the most obvious problems with the Bitcoin protocol was scalability. Indeed, with one block created every 10 minutes (in average) containing at most about 4.000 transactions, we have an average throughput

of 7 transactions per second. Given that the idea is to compete with the banking market, Bitcoin is far from offering a satisfactory throughput when we know that on average, centralized solutions such as Visa runs on 24.000 txs/sec [4]. In order to address this scalability problem, research has been conducted and solutions have been proposed. For example, increasing the size of a block or reducing complexity [53] would be viable solutions as these are important factors limiting the ability to validate transactions. However, these proposed solutions could also expose the system to more security threats or inconsistency problem such as a higher fork probability provoked by a lower inter-block delay. In any case, there are other problems such as network latency, which makes competition with institutions such as Visa unlikely.

Finally, a brand new approach called Lightning [24] emerges in 2016. It is based on the idea that "If a tree falls in the forest and there's no one around to hear it, did the tree make noise when it fell?". In the world of blockchains, it means that if a transaction only involves two users, not everyone needs to know its details. With this in mind, Lightning [24] proposes to create *lightning channels* between two users. Operationally, pairs of users have to create a *multi-signature wallet* (i.e. a wallet shared between two or more users) and put funds on it in the form of real transactions that appear on the blockchain. Thereafter, transactions will be nothing more than a transfer of funds from one user to another on the same wallet, these transactions do not appear on the blockchain, thus are not limited by its rate. When users want to close this channel, transactions containing what is left of each user's down payment will be sent on the blockchain. In this way, two users could exchange an infinite number of transactions, while on the blockchain only 2 to 4 transactions would appear. In a sense, this solves the scalability problem (since this solution imposes no limit on the transaction throughput) while reducing transactions traceability and creating new security threats targeting lightning channels.

2.4.4.2 Colored Coins

Colored coins [54] are a way to issue and transfer assets on the Bitcoin blockchain. It can be used to represent anything, such as stocks, bonds, smart properties, securities, precious metals, commodities, other traditional currencies (such as dollars, pounds or euros) and even other crypto-currencies. It also allows people to create smart properties. A deed for a house can be represented on the Blockchain as a colored coin. The owner of that coin is then the legal owner of the house. Transferring ownership of the house becomes as simple as making a Bitcoin transaction. Colored coins allow you to transact and hold virtual securities on the Bitcoin blockchain. You could for example have a colored coin that represents your house or car, if/when you sold your house/car you would send the colored coin to the new owner, there would be no need for a physical deed as proof of ownership is in the blockchain. The main advantage of the colored coin/open asset protocol is that it lives on the Bitcoin blockchain and is therefore secured by the massive hardware investment that

[4] Note that this rate is a maximal value and latency in these centralized systems can be quite high and variable.

supports the network.

2.4.4.3 ERC-20

ERC20 [55] is an Ethereum standard. It defines functions and events a token has to manage in order to be qualified as an ERC20 contract. Any code that respects the ERC20 specifications creates an ERC20 token. An ERC (Ethereum Request for Comments) is a process anyone can use to ask the community to comment his proposal. The 20th proposal, submitted on November 19th, 2015 was about standardization of token development on Ethereum (ERC20). An ERC20 token contract in Ethereum is a smart contract containing a mapping between owners and their tokens. It also manages token transfers by updating the mapping. For example, if A wants to send 100 tokens to B. The contract modifies the mapping by reducing the tokens of A by 100 and increasing the tokens of B by 100.

2.5 Graph-based Blockchains

Many graph-based protocols have been explored in the last years. Proposals such as HashGraph [56], ByteBall [57], and Iota [2, 58] do not use blocks, i.e. a graph is formed by transactions pointing to each other. Ghost [3] and Spectre [4] protocols keep blocks, modifying the blockchain data structure from a totally ordered sequence of blocks to a directed graph of blocks. Note that in these approaches, the absence of mechanisms to prevent the presence of conflicting records (i.e., blocks with conflicting transactions) or the presence of cycles in the directed graph (Spectre [4] organises blocks in a directed, but not acyclic, graph of blocks) require that participants execute a complex algorithm to extract from the graph the set of accepted (i.e., valid) transactions [4].

In this section, we focus on both DAGs without blocks (Section 2.5.1) and DAGs with blocks (Section 2.5.2) . Finally, in Section 2.5.3, we provide a thorough comparison of these two models.

2.5.1 DAGs without Blocks

As said before, some graph-based solutions have chosen to work without blocks and use only transactions. The validation paradigms are therefore completely different.

2.5.1.1 The Tangle

Iota [2] is an IoT (Internet of Things) oriented solution with transactions chained to each other, instead of blocks. These transactions form a DAG through confirmation links with other transactions. This is called a Tangle. Iota's operation is based on the work of the users. In order to validate a transaction, the transaction must refer directly or indirectly (see Definition 3) to at least 2 other transactions. In this way, users participate in maintaining the security of the network by choosing which transactions to validate (or not). It is assumed that the majority of users are

honest and will not validate conflicting transactions. These chained transactions all indirectly refer to the genesis transaction which is at the origin of the tangle and which contains the sole creation of Iota tokens. Note that for a node to issue a valid transaction, it must solve a cryptographic puzzle similar to those in the Bitcoin blockchain.

Definition 3 (Indirect reference)**.** *If there is not a directed edge between transaction A and transaction B, but there is a directed path of length at least two from A to B, we say that A indirectly approves B.*

2.5.1.2 ABC : Asynchronous Blockchain without Consensus

ABC [1] is a non-permissive and resilient solution to an asynchronous environment. It is an innovative solution without any need for consensus. It assumes that Bitcoin has vulnerabilities and wants to solve them by replacing PoW with PoS which would rely on randomness and more communication. ABC chooses to confirm transactions instead of grouping them into blocks. To do this, transactions are confirmed via a vote. Each transaction must refer to at least one other transaction. Note that there is a difference between reference and dependency, a transaction can refer to another simply by accepting its presence in the DAG while dependency is a direct link between two transactions. As in IOTA (see Section 2.5.1.1), a transaction is confirmed if enough transactions refer to it directly or indirectly (see definition 3). There may be a conflict between two transactions and it is resolved by the number of references of each of these transactions. This number must be greater than 2/3 stakes to confirm a transaction since at least 2/3 stakes are assumed to be held by honest users who obey the protocol.

This solution has the advantage of using PoS which allows for less energy consumption. The idea of using a DAG of transactions allows it to improve its scalability and makes it applicable to the IoT domain but from this point of view, no contribution compared to other solutions such as Iota [2]. However, creating "fake" transactions only to validate others could lead to a large communication and storage overhead.

2.5.1.3 Avalanche

Avalanche [59] consists of several Snowball [59] instances (i.e. a chain-optimized consensus protocol powered by the Avalanche consensus protocol), instantiated as a protocol that maintains a dynamic Directed Acyclic Graph (DAG) of all known transactions. The DAG has a single starting point called genesis. When a client creates a transaction, it names one or more parent transactions, which are included in the transaction and form the links of the DAG. The parent-child relationships of some transactions in the DAG can, but don't need to, have a direct dependency; e.g. a child transaction must not spend or have any relationship to funds received in the parent transaction. Avalanche uses the DAG structure to check the validity

of a transaction. when a transaction is submitted, it and its ancestors are checked up to the genesis.

2.5.2 DAGs with Blocks

As opposed to the solutions presented above, some other graph-based solutions have chosen to apply DAGs to blockchain while keeping the blocks structure.

2.5.2.1 GHOST

GHOST [3] is not strictly speaking a DAG but a modification of Bitcoin to change the way forks are resolved. This is called a *blocktree*. Indeed, in Bitcoin, the rule is to choose the longest chain. According to GHOST, this is a very insecure and slow way to solve forks. They propose to keep all the forks ever created and to choose each chain according to an associated weight. The weight of a string will be the sum of the weights of each block, and the weight of each block corresponds to the number of forks linked to it. The more a block is at the base of several forks, the more weight it has.

2.5.2.2 SPECTRE

SPECTRE [4] is a solution to improve the scalability of Bitcoin. It believes that Bitcoin's choice to keep only one chain, with a preference for the longest, is a highly unscalable solution. Instead, Spectre opts for a DAG structure of blocks. In this way, miners can create blocks concurrently and thus make the system evolve faster. Each time a new block is created, it must reference all other blocks in the miner's view. It is this referencing that will allow to resolve conflicts between two blocks. Blocks are ordered by pairs to be able to choose between two blocks of this same pair, thanks to the references contained in each block. It is the parallelization of the miners' work that allows this improvement of Bitcoin's scalability. Unlike GHOST, which is only a more advanced fork resolution solution, more than one chain is considered in the state of the SPECTRE system, so the work of the miners is really partitioned.

2.5.2.3 Sycomore

Sycomore [27] has been the first graph-based protocol to be fully distributed. Neither a chain of blocks nor a set of transactions are extracted from the graph to become the valid blockchain or the valid set of transactions. Instead, the full graph is the ledger. Blocks are built so that they commit the state of the directed graph at the time blocks were created, which decreases the opportunity for powerful attackers to create blocks in advance. In contrast to previous approaches, Sycomore allows a better immutability of the transactions thanks to the introduction of the label which allows a better partitioning of the transactions and at the same time of the miners' work. It is this label that identifies each chain of the DAG. Moreover, the

auto-adaptive character of Sycomore allows it to increase or decrease the number of parallel chains and thus the capacity to confirm transactions. This auto-adaptability of the DAG promises an extremely interesting scalability that we wanted to study more in depth in Chapter 4.

2.5.3 DAGs with or without blocks?

Let us provide in this section a quick comparison of the previously presented graph-based solutions.

First of all, transaction-based DAG solutions are not fully decentralized because they leverage the presence of central or trusted nodes. Their use allows a better scalability than classic blockchains. The more transactions there are, the faster they are confirmed, which promises good performance in terms of throughput. This is particularly interesting for domains where scalability is paramount such as the Internet of Things. However, these solutions have not been tested on very high intensity systems, so their performance is not guaranteed. Since transactions are not put in blocks, they are only confirmed if other transactions refer to it, this kind of system also needs a lot of traffic to work well. Similarly, if there is not enough traffic, the system becomes vulnerable and this is what also causes transactions to remain unconfirmed for a while, which would increase the confirmation latency. Additionally, transactions do not need the work of others (miners) to be confirmed and therefore incur little or no cost (fees).

On the contrary, block-based DAGs allow a better decentralization, an interesting scalability but not yet compared to transaction-based DAGs since their applications are so different. Since they use blocks, the need for traffic is not the same as in DAGs without blocks. The security of these systems is not more or less diminished depending on the rate of submission of transactions since it is the blocks that maintain this security and their security impact remains the same when they are broadcast empty (without any transactions). That said, it is for this very reason that the work of the miners is paramount and therefore fees are necessary for the system to work.

To sum up, we would say that both solutions are interesting from a scalability point of view and can afford to coexist by targeting different domains of use, for example IoT for DAGs without blocks and decentralized finance for DAGs with blocks.

2.6 Sharded Blockchains

Sharding, first used in databases, is a method of distributing data across multiple machines with a scaling objective. In blockchains, sharding means to partition 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.

			Elastico [12]	Omniledger [13]	Rapidchain [14]	StakeCube [32]
State-sharding	Support		No	Yes	Yes	No
	Smart-Contract	Support	No	No	No	No
		Atomic-Commit	/	/	/	/
Node-to-Shard Assignment	Model		PoW	PoW/PoX	Offline PoW	UTXO Ownership
	Predictability		No	No	No	No
Adaptability			Time-driven	Time-driven	Time-driven	Event-driven
Type of Protocols	Intra		BFT	BFT	BA	BA
	Inter		/	BFT	/	/
Security Asumptions	Network		Partially synchronous	Synchronous	Synchronous	As required for the BA [5]
	Failure	Adaptive	Weakly	Weakly	Weakly	Weakly
		Threshold	25%	25%	33%	33%
Cross-shard transaction reduction			No	No	No	No

Table 2.1 – Comparison table of blockchain sharding solutions.

			TON [10]	Elrond [11]	Monoxide [15]	BrokerChain [16]
State-sharding	Support		Yes	Yes	Yes	Yes
	Smart-Contract	Support	Yes	Yes	No	No
		Atomic-Commit	No	No	/	/
Node-to-Shard Assignment	Model		PoS	PoS	PoW	PoS
	Predictability		No	No	Yes	Yes
Adaptability			Event-driven	Time-driven	Static	Time-driven
Type of Protocols	Intra		BFT	SPoS	PoW	PBFT
	Inter		BFT	SPoS	PoW	PBFT
Security Asumptions	Network		Synchronous	Synchronous	Partially synchronous	Synchronous
	Failure	Adaptive	N/A	Weakly	N/A	N/A
		Threshold	33%	33%	50%	33%
Cross-shard transaction reduction			No	No	Discussed	Yes

Table 2.2 – Comparison table of blockchain sharding solutions (*Continued*).

2.6.1 Evolution of Sharding in Blockchains

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

RScoin [29] 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[12] 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[13] and Rapidchain[14], showing optimized performances via parallel transaction processing. More recently, Monoxide [15] 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[32] 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 transac-

tions 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 aims at guaranteeing 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[10] and Elrond[11], 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 (i.e. not detailed in the documentation). Brokerchain [16] 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 in charge of sharing out nodes among shards.

Gramoli et. al. [60] present a blockchain-agnostic shard management mechanism applicable to sharded blockchain solutions. For their experimentation, they evaluate their solution on a blockchain called CollaChain [61]. The solution presented a novel idea to reconfigure sharding without disrupting the blockchain service through dedicated smart contract invocations.

2.6.2 Comparison of Existing Solutions

In the remaining of the section, we compare the above discussed solutions along six criteria. We will consider only permissionless solutions, so that RSCoin [29] is not included in the comparison. Tables 2.1 and 2.2 show a summary of this analysis.

State sharding support. Solutions differentiate themselves in the type of supported sharding. Note that the most recent solutions in PoS settings aim at implementing state sharding [13, 14, 10, 11, 15, 16]. All these solutions must provide support to cross-chain 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-chain transactions, like Rapidchain [14], Monoxide [15] and Brokerchain [16], which use *special users* which exist in multiple shards and act as relays between shards. Other approaches [10, 11] use a globally-shared blockchain named masterchain (or metachain) to maintain synchronization between shards and thus confirm cross-shard transactions.

As for smart contract support, only [11, 10] manage smart contracts. The support however is only related to the management of smart contract-to-shard assignment (i.e., the assignment of a given smart-contract to one of the shards of the system) but there is no support for atomicity of smart contracts in the general case.

To the best of our knowledge, no academic proposal managing smart contracts in a sharded environment offering a 2-phase commit protocol to assure their atomicity has been proposed so far.

Node-to-Shard Assignment. In permissionless settings, node-to-shard assignment (i.e., the assignment of a given node to one of the shards of the system) must be unpredictable, i.e. must rely on randomness. To this end, the selection and assignment of processes can be based on PoW [12, 14, 13, 15], PoS [10, 11], often coupled with decentralized partitioning mechanisms (identifier-based, DHT, etc.). Some solutions propose to re-assign regularly nodes to shards to cope with an adaptive adversary [32, 10]. 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 [12]. 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 [29, 15], *(ii)* time-driven, i.e. the set of shards changes at specific instants of time [12, 13, 14, 11, 16], or *(iii)* event-driven, the set of shards changes automatically when appropriate conditions are met [32, 10].

Type of protocols. Intra-shard protocols are typically consensus protocols used to create blocks and elect block creators in each shard. Mostly used blockchain consensus protocols in permissionless settings are BFT Consensus protocols [17, 47], Byzantine Agreements (BA) [22] and Nakamoto-style consensus [33, 44, 21]. These protocols are typically re-used with no or small adaptations in sharded blockchains to create blocks (see Table). Election mechanisms, always in place to establish the nodes that have rights to append blocks, are either based on PoW [12, 14, 15] or on PoS [32, 11, 10].

For state-sharding solutions, *inter-shard protocols* can require a BFT protocol [13], an asynchronous communication protocol (e.g. [15]), or synchronous protocols (e.g. BFT synchronous [10], stake-based Nakamoto-style [11]).

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 [13, 14, 10, 11] 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 [12, 13, 14, 32, 11]. Yggrdrasil relies on a partially synchronous network, tolerates 33% threshold of corrupted validators in each shard and provides security against an adaptive adversary, being the sole (at the best of your knowledge) state-sharding PoS solution providing the proper level of security in a permissionless setting.

Cross-shard transaction reduction. Cross-shard transactions are inevitable when state sharding is used. A simple way to reduce the burden of cross-shard

transactions is to let users choose the shards they are interested in, i.e., the ones containing accounts of their sellers and preferred smart contracts. This way transactions do not have to cross different shards. This approach has been mentioned in [15] but without providing any method to implement it. Brokerchain [16] uses a different approach: it proposes a shard formation heuristic to maximize the probability that users interactions will take place inside a single shard. The heuristic, however, is fully public, which does not guarantee unpredictability and the required security level in a permissionless setting.

2.7 Experimental Approach

In the literature, in order to efficiently test a system or a hypothesis, one can usually choose one of two main approaches, experimentation through test-nets or simulation. When it comes to simulation, there are several solutions, some are general and others are dedicated to a specific domain such as the blockchain (e.g. MAX (Multi-Agent eXperimenter) as presented in chapter 3).

2.7.1 Test-Nets

The whole point of test-nets is to provide a realistic implementation of a system before starting the project. Its main use is to allow a considerable number of experiments to be carried out to see if everything is working properly. When it comes to cryptocurrencies, test-nets allow users to exchange fake tokens to test the capacilities of the system. In this way, blockchain providers test their products and users can test new cryptocurrencies for free. For example, Bitcoin [33] has the Bitcoin Faucet test-net, Ethereum [62] has Ropsten, RinkeBy, Kovan, or even Görli test-nets. Finally, Tezos [46] also made the Alphanet network available for its users.

In a more scientific context, it is possible to deploy a private network in a local environment where nodes are connected and running a predefined protocol. For example, one can launch the Tezos or Tendermint protocol and specify the addresses of the network peers [46] to create a private network and test it.

2.7.2 Modeling and Simulation

To shed some light on the dynamics of blockchain systems, recent research focused on suitable simulation models to capture their behavior. The different proposals differ in the level of abstraction of the model, the simulation type and the properties under study. First of all, general simulators, such as network simulators, have an advanced environment as well as a strong community, which makes them a valuable asset when it comes to simulating any network. A second solution would be to use agent-based simulators, their ability to derive global properties from individual behaviors make them particularly applicable to distributed systems such as blockchains. Finally, blockchain-dedicated simulators can be a perfect solution if

they are advanced enough by offering as many basic features as the generic simulators while providing blockchain data structures and primitives for blockchain management.

2.7.2.1 Network Simulators

Network simulators are used in many fields and have the particularity of being able to adapt to the domain of activity with the help of plugins added over time.

Among the most popular ones, OMNeT++ [63] is a framework and a discrete-event simulation library, mainly used in network researches. This simulator is dedicated to networks modeling, it focuses on communication aspects. In OMNeT++, all actions are triggered by messages that can be scheduled and self-sent, which allows great flexibility in terms of experimentation. Topologies are pre-defined by creating the nodes and defining their connections manually. It is possible to add nodes at any time. Thanks to this feature, this simulator makes it possible to create dynamic networks by adding or removing nodes during the simulation. Nodes have access to a unique environment, but it is possible for them to have multiple roles (running different protocols for example) in this same environment. OMNeT++ also allows unit testing and regression testing with the *opp_test* tool.

In addition to OMNeT++, NS-3 [64] is a discrete-event network simulator mainly used for research or educational purposes on network areas. NS-3 only allows nodes and links to be added, not removed, which makes it impossible to simulate dynamic systems where nodes can move in and out. On the other hand, it is possible to turn off certain interfaces of certain nodes in order to "feign" an exit. As with OMNeT++, nodes can only access a single environment and test are made available by a dedicated framework integrated to the simulator [65].

2.7.2.2 Agent-based Simulators

As mentioned above, agent-based simulation is one of the best suited modeling techniques for distributed systems. Among the most used agent-based simulators, JaCaMo [66] is a multi-agent programming framework that combines three different technologies, *Jason* for managing agents, *CArtAgO* for managing the environment and *Moise* for the organization of both agents and environment. The JaCaMo environment is organized in *workspaces*. Each agent can create, join or leave other workspaces on a node (whether local [6] or remote) and has access to the specific information of the workspaces to which it belongs. It has in this way a global perception of the environment in which it evolves, which allows the agent to interact with its environment and neighbours. As Figure 2.4 shows, within a workspace, there are several types of objects: /working-agents, listing the agents linked to the workspace; /hosted-agents, listing the agents that have accessed the workspace via

[6] By allowing to launch several nodes on the same physical machine, JaCaMo allows the creation of private test-nets.

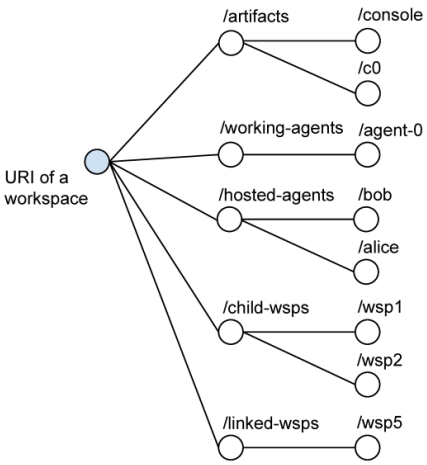


Figure 2.4 – JaCaMo-web Organization.

another workspace; /child-wsps, listing the sub-workspaces; /linked-wsps, listing the workspaces linked to it.

In order to use this environment, JaCaMo has implemented a REST API named JaCaMo-web [67]. It facilitates the use of agents/environments running on a JaCaMo node. In this way, each element of the system is represented by a web resource accessible via URL. The use of JaCaMo is thus essentially done via GET and POST requests.

2.7.2.3 Other Simulators

During our research, we came across solutions that are not simulators per se but are interesting enough as test environments to be considered in experimental work on distributed systems. Among these, one can quote SimEvents [68], a Matlab [69] library (a high level language mainly used for visualization and application development) that is dedicated to event planning and network simulation. It allows the creation of agents, definition of their interaction and the exchange of messages between them. As for the application of such a solution for the study of blockchain systems, a Matlab application that implements a blockchain were implemented [70]. There exist other solutions written in other languages such as Simmer [71] or Agent Based Modelling in R [72]. Both are libraries written in R (which is a programming language used for statistical computing). Simmer is a generic framework that allows discrete event simulation and exploits a new concept: trajectories. It consists of a "path to follow" for agents of the same type, which will define their behavior and therefore their actions. In other words, a trajectory would be a list of actions in sequence that define the behavior of an agent. On the other hand, in ABM-R, models are made of agents that can interact with each other and with their environment, where they can also move. The movements and communications of the agents are

governed by rules that define what an agent can or cannot do.

2.7.2.4 Blockchain-dedicated Simulators

As for blockchain systems, several solutions were proposed in the recent years. As an example, SimBlock [73] is a simulator with event-based progress where each node generates messages and block creation events. Simulation in SimBlock are setup using block-related parameters (size, creation interval), node-related (number, maximum connections, position...), but also network-related (transmission delay). Each node has a defined capacity of creation of block, given that in this simulator, the blocks are created in PoW, that means that creating a block has a certain difficulty (parameter also defined). Even though nodes and network topology are dynamic, SimBlock does not specify whether the system is dynamic, i.e., whether nodes can enter and leave the network. In SimBlock, there is a single environment on which agents can interact, which means all sent messages are received by all nodes, network-partitioning attacks or solutions (e.g. sharding) are not possible. Furthermore, there is no mechanism for orchestrating a scenario, or even for verifying and validating models.

Kaligotla et al. [74] propose an agent-based framework for evaluating distributed ledgers, modelling a blockchain at a very abstract level as a simple append-only queue where a block is added when verified by enough agents. Agents are divided into two categories, users who issue transactions and verify them through atomic actions (no messages are exchanged) with associated costs, fees and energy costs, respectively, and miners who create *blocks* using the PoW consensus model. As in SimBlock, the agents belong to only one environment on which they interact, which can be very limiting in some cases, as we will see in chapter 3.

Memon et al. [75] propose a modeling of a *blockchain* system based on a queue simulation. They assimilate the process of creating *block* in PoW, mining to a queue in a router. They used Bitcoin parameters and simulated it using an M/M/n/L queueing system with JSIMgraph. In this simulation, there is only one queue and a number of miners. The transactions are created randomly. They do not quite model the distributed system that is the *blockchain*, nor do they model any communication protocols of any kind. As for the other simulators we have studied, this one does not offer any verification or orchestration mechanisms, nor does it allow an agent to belong to several environments.

Piriou et al. [76] propose a stochastic simulation model and Monte-Carlo simulations to evaluate the performance of a blockchain when the communication system loses messages. A stochastic model is a tool used for estimating the probabilistic distribution related to the distribution of assets. This being achieved through the random variation of some parameters during execution. Various consensus models are studied and double-spending attacks are modeled by a simplified append-only queue model (which considers instantaneous communication).

Alharby et. al. [77], Rosa et al. [78] and Faria et al. [79] propose discrete-event simulators for distributed ledgers that resemble to Bitcoin and Ethereum, but not

adapted to simulate graph-based distributed ledgers. On the other hand, Bottone et al. [80] present a simulation model for Tangle-like DAG ledgers [2] aiming at studying the grow of the graph of transactions. The simulation model is instrumented on the NetLogo [81] agent-based simulator. Due to the complexity of the computational model, simulated strategies are very simple and network effects are not taken into account.

2.8 Conclusion

In this chapter, we first saw different solutions in a wide spectrum of bricks used in both blockchains and blockchain simulation. We have seen in this chapter that each of the solutions studied, whatever the brick (consensus model, DAGs with or without blocks...) is intended for a specific application domain and has advantages that make it particularly interesting for this domain and disadvantages that make it less interesting in others.

To start with, we have described the consensus models that we have found most interesting. Among these, we have described some leader-based and committee-based solutions. Each solution obviously has its advantages, disadvantages and their particular interest and application domain. We could see for example some interesting consensus models with their own limits. For example, the PoW is a particularly resistant model but very energy consuming and in many cases not very scalable. On the other hand, it can be very resilient in the face of an unsynchronized network model. Faced with this, its first competitor, the PoS, proposes a less energy consuming solution but which poses the same problems of fork that we have also presented during this chapter. In order to solve this problem, the famous BFT protocols have entered the blockchain, allowing the maintenance of a blockchain without fork but which requires a better synchronization than the first two solutions.

Secondly, we have detailed the two main transaction models used in blockchains, namely UTXO and account-based, with their application domains and the interest of using one or the other of these models. Then, we've detailed the basic and most widespread incentives for honest participation.

As these consensus and transaction models can be applied to different solutions, we have then described some classical solutions such as Bitcoin, Algorand and Tendermint while presenting their strong and weak points.

We have then moved on to graph-based blockchains by presenting both block-free and block-based DAG solutions such as GHOST, the precursor, IOTA, the first blockchain without block, and Sycomore, which will be our main subject of study in Chapter 4. In this section, we have compared the use of transactions and blocks in DAGs as well as the benefits of these two solutions in different application domains.

Fifthly, we have focused on sharding and the solutions that propose it. Solutions such as RSCoin[29] and Elastico[12] launched the intuition of sharding while others like StakeCube[32] and Omniledger[13] improved it by pushing it further. More recently, successful industrial solutions not stemming from the scientific environment

such as TON[10] or Elrond[11] appeared. We also had a comparison of these different solutions based on precise criteria defined for this type of system.

At the very end of the chapter, we have also discussed the popular solutions when doing experimental research in blockchain, namely a real implementation with the use of test-nets for the verification of hypotheses or simulation with different techniques and different solutions.

In the next chapter we will develop our penchant for the experimental approach by presenting the tools we have used during this thesis.

Chapter 3

Tools

"It is essential to have good tools, but it is also essential that the tools should be used in the right way"

– Wallace D. Wattles

Contents

"Study of the potential of graph-based approaches in blockchains" is the title of this thesis. In this chapter, we will deepen our understanding of two of the words contained in this title. "Study", which implies a dissection of the strengths and weaknesses of a given subject of study. Then, "Potential", which implies a virtual existence of this subject of study that may, or may not, meet certain criteria. These two words put together imply to push this subject of study to its limit in order to determine whether it meets our criteria or not.

In this manuscript, our subject of study is "graph-based approaches in blockchains" and in this chapter, we detail the method as well as the different tools used to carry out this study.

Indeed, during this thesis, we privilege, when it is possible, the angle of the experimentation with the method of agent-based simulation [82]. Our results and the importance given to them therefore depend on the softwares we used, that's why we dedicate an entire chapter to them. These tools are the very basis of our work and it is necessary, for the good understanding of our results but also to increase the confidence of the reader in them, to understand the tools used to generate them. Which tools have allowed us to (i) model our protocols (ii) execute the different experiments that we have imagined to push these algorithms.

In this chapter, we will present in details our toolbox and each of its tools. First, we define in detail simulation methods and their application to blockchain in Section 3.1. In Section 3.2, we present MAX, the agent-based simulator we used to model our protocols. Then, the experimental environment we used to execute our numerous experimentations in Section 3.3. For the sake of completeness, we will finish this chapter with the different implementations realized for both protocols studied in this manuscript (respectively in Chapters 4 and 5) in Section 3.4.

3.1 Simulation for Blockchains

Usually, the easiest way to test a hypothesis is to test the experiment and observe the result. In many cases the experiment is either not feasible or too expensive. One then has to recourse to simulation [83], a computer tool used to study the results of an action applied on an element without having to carry out the experiment under real settings. Simulations allow to control the environment, whereas "real world" experiments for distributed systems in particular will be done on an environment (e.g. internet) whose conditions are neither observable nor configurable, which can be detrimental to the interpretation of results, especially when comparing different protocols or when we want to evaluate specific scenarios (e.g. malicious behavior).

In the context of system dynamics study, simulation is one of the possible approaches to understand the behavior of a system and to evaluate its performance. It consists in modeling and studying a real system in the form of a computer program called a simulator model. This method is widely used for the study of communicating systems such as distributed systems (e.g. blockchain systems) [84]. Simulation allows to validate hypotheses about the functioning of a given distributed protocol such as blockchain systems while keeping the hand on all the actors of the system, i.e. both the nodes and their environment (which is impossible in reality). In the same way, thanks to this technique, it is possible to define precisely in which context a given system is efficient and in which context it is vulnerable. This is what we will see in chapters 4 and 5 where we will try to validate our hypotheses on the protocols we have designed in both honest and malicious environments (in order to test different attacks or failures that could occur).

We therefore detail in this section some basic principles of simulation before focusing on the application of this experimental method in the blockchain domain.

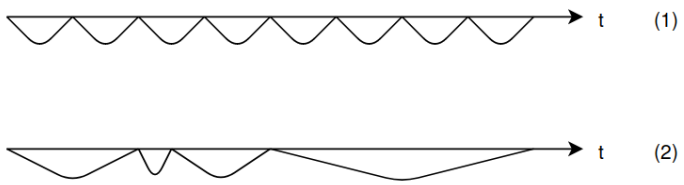


Figure 3.1 – Types of Simulation Execution.

3.1.1 Discrete-event simulation

Essentially, a simulator executes a list of events in a chronological order. Among the techniques used in the context of complex systems simulation is the discrete-event simulation. It is a sequence of discrete events that are executed at a given time, which change the state of the system. This state can only change at distinct temporal instants when there occurs an event, which can be defined as a circumstance that allows the system to change state (i.e. the reception of a message...). Nowadays, this technique is commonly used in order to both design, optimize and validate organizations or to study complex non-linear systems such as networks and distributed systems.

Types of execution The temporal aspect of simulations is managed by a *clock*, used to indicate the current simulation time, counted in *ticks*. The simulation follows a list of events, executes the events one after another and updates the simulation clock according to the executed event's parameters. As illustrated by Figure 3.1, we distinguish two types of simulation execution:

- **(1) Time-driven:** The simulation progresses by incrementing the clock by a predefined value named *simulation step* until it reaches a limit (i.e., the end of the simulation). Events are executed when the clock reaches their execution time.
- **(2) Event-driven:** The simulation progresses from one event to another chronologically by executing them until the simulation ends.

In a discrete-event simulation, all events are not necessarily created at the simulation's beginning. As the simulation proceeds, an event may introduce one or more new events, which are then inserted into the *schedule*, an event list sorted by the execution time of each event. This list is managed by the *scheduler*. It manages the identification and execution of the next event in the schedule, the insertion of a future event in the list or the withdrawal of an event after its execution.

In Table 3.1, is shown an example of a time-driven schedule which associates events to their countdown. At each interval, countdowns are decremented, and the actions that have a countdown of 0 ticks are executed then removed.

Event	Countdown
Action 1	5
Action 2	7
Action 3	10
Action 4	15
Action 5	23
Action 6	29

Table 3.1 – Example of a Schedule.

3.1.2 Agent-based simulation

In order to simulate a distributed system (i.e., a system of distributed nodes), one of the known approaches would be to use self-organizing and autonomous objects, also known as agent-based simulation [82]. It is a subtype of discrete-event simulation [83]. It is a model built from the bottom up which consists in decomposing the global functionality of the system into sub-functionalities, and thus modeling the behavior of each sub-system (node). It is not limited to observed data and can be used to model the counterfactual or experiments that may be impossible to conduct in the real world (e.g. malicious behavior).

As shown by Figure 3.2, the self-organizing objects that make up this system are called *agents*. An agent could be defined as an entity capable of acting or reacting according to its role and what it perceives of the environment in which it evolves. The environment is one of the agents that compose this system. It has a rather special role since it serves as an intermediary between the other agents. It is known as the most important entity of this system since it allows other components' connection. Using agent-based simulation allows to create systems composed of these autonomous entities, which interact with each other in a dedicated environment, leading to the emergence of a global scheme describing the functionality of the system we want to simulate.

3.1.3 Agent-based simulation for blockchains

By definition, the blockchain is a application replicated among nodes of a distributed system. The approach presented in 3.1.2 is therefore quite appropriate in the sense that it allows us to model open, dynamic, distributed and intelligent systems, four of the characteristics of blockchain systems.

In short, a blockchain application is independently executed on several nodes that will interact to maintain or evolve the state of this self-organizing system. Such a system is defined as being able to modify the behavior of its components, without any external involvement, in order to maintain the expected functionality. Agent-based simulation allows us to reduce this system's complexity by focusing on its components. In our case, it allows us to simulate a network of distributed nodes by modeling the nodes by *agents* that will follow a pre-defined protocol (behavior)

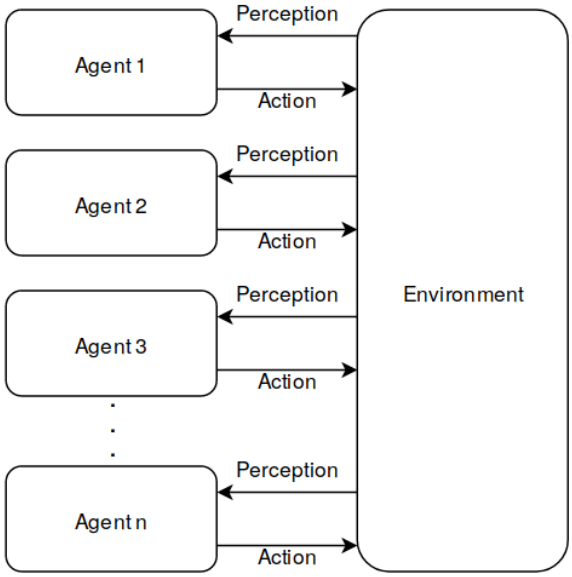


Figure 3.2 – Agent-based Simulation.

to act and/or react in the environment where they evolve. This environment is also one of the main actors of the good functioning of the blockchain, as it involves network parameters such as the transmission delay or its reliability which will be both influencing the state of the blockchain.

3.2 Multi-Agent eXperimenter (MAX)

By simulating the functionality of a system at the agent level, the impact of the behavior of each algorithm is tested and analyzed in detail under different execution scenarios. It helps to focus on the fundamental characteristics that make such a distributed system, namely, the organization of its elements and the ties that bind them.

For the purpose of the various studies presented in this manuscript, we have chosen to use MAX (Multi-Agent eXperimenter) [85, 5], an agent-based simulator based on the MaDKit framework [86]. MAX is a blockchain-dedicated simulator allowing to create models as close as possible to the real algorithm to be modeled, in order to test it and improve it.

Community-Group-Role MAX allows a hierarchical organization of agents thanks to the CGR (Community-Group-Role) concept. According to this concept, two agents can be part of different groups but belong to the same community. In this way, agents can play different roles in the same group (user and miner for example) or in different groups (overlay blockchain, e.g., BitcoinLightning [24]). Note

that this organization model is an abstraction, a community is simply the container of a set of agents divided into subsets called groups in which individuals play roles.

3.2.1 Basic Elements of MAX

In MAX, simulations execute a given scenario, a roadmap where the number of agents, their actions and their timing are strictly specified. The creation of a scenario (example shown by Figure 3.3) in MAX is done in three steps, (i) initialization of the scheduler, (ii) initialization of the environment, (iii) initialization of the agents. These three steps are done by the *experimenter*. It is responsible of creating and setting up the scenario to be simulated. Since it is above all agents, it can observe but also intervene in the course of the simulation. This is what allows the dynamicity of a scenario where agents can be added and removed according to certain conditions. For example, it is possible to add an agent at time t_1 and to remove it at time t_2 in order to study the reaction of the system to a dynamic environment.

For any scenario to run, it needs a *scheduler*. It is responsible of the simulation global clock and for planning and executing the actions of the various agents. Agents which could be defined as an active entity that can play one or multiple roles in one or multiple environments. When an agent takes a role, it gains the right to plan actions that are specific to that role, if it forsakes that role for any reason, the future actions that only that role allows it to perform are cancelled by MAX.

The *environment* is also one of the key pieces when modeling multi-agent systems. It is an abstract element. It can be physical (internet) or logical (social network), and is modeled as an agent that schedules events according to the requests of other active agents. It thus acts as a mediator between agents and allows their interaction. In other words, agents cannot interact directly with each other, they must go through an environment. The environments are modeled as agents belonging to a group and playing the role of environment in this group, the agents which are also part of this group will be able to take the roles of this group and interact via the unique agent which has the role of environment in this group.

As said before, an agent can be part of different groups. In order to keep a separation between the information received in each group, each agent has one context per environment it belongs to. Context is what allows an agent to differentiate the information it has perceived in several environments.

3.2.2 Architecture

MAX is a complex framework, created in such a way that it is generic, easy to use and modular. As illustrated by Figure 3.4, the latter can be decomposed into three subcomponents, which are detailed in the following:

1. **max.core.** It represents the engine of the simulator. It contains the implementation of the basic abstractions presented above, the most general roles, as well as communication structures between agents.

```java
package max.model.ledger.sycomore.exp;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;

import madkit.kernel.AbstractAgent;
import max.core.MAXParameters;
import max.datatype.ledger.Transaction;

/**
 * Defaut Sycomore Experimenter that sets up the scenario with MAXParameters.
 *
 *
 * @author Mohamed Aimen Djari
 * @version $Revision$ $Date$
 */
public class EXSycomoreCPBased extends SycomoreExperimenter<Transaction> {

	@Override
	protected List<AbstractAgent> setupScenario() {
		List<AbstractAgent> agents = super.setupScenario();

		int numberOfUsers = MAXParameters.getIntegerParameter("MAX_MODEL_LEDGER_SYCOMORE_NUMBER_OF_USERS");
		double numberOfHonestMiners = MAXParameters.getDoubleParameter("MAX_MODEL_LEDGER_SYCOMORE_NUMBER_OF_MINERS");
		int numberOfMaliciousChainAttackMiners = MAXParameters
				.getIntegerParameter("MAX_MODEL_LEDGER_SYCOMORE_NUMBER_OF_MALICIOUS_CHAIN_ATTACK_MINERS");
		int numberOfMaliciousLedgerAttackMiners = MAXParameters
				.getIntegerParameter("MAX_MODEL_LEDGER_SYCOMORE_NUMBER_OF_MALICIOUS_LEDGER_ATTACK_MINERS");
		double numberOfAllMaliciousMiners = numberOfMaliciousChainAttackMiners + numberOfMaliciousLedgerAttackMiners;
		int numberOfResilientUsers = MAXParameters
				.getIntegerParameter("MAX_MODEL_LEDGER_SYCOMORE_NUMBER_OF_RESILIENT_USERS");
		int numberOfMoreResilientUsers = MAXParameters
				.getIntegerParameter("MAX_MODEL_LEDGER_SYCOMORE_NUMBER_OF_MORE_RESILIENT_USERS");

		if (numberOfAllMaliciousMiners == 0) {
			MALICIOUSPROPORTION = 0;
		}
		if (numberOfHonestMiners == 0) {
			MALICIOUSPROPORTION = 1;
		}
		double power = 3;

		// Honest part
		List<Double> honestRandoms = new ArrayList<Double>();
		Double sumhonestRandoms = 0.0;
		for (int i = 0; i < numberOfHonestMiners; i++) {
			double weight = Math.random();
			honestRandoms.add(weight);
			sumhonestRandoms += weight;
		}

		// Honest part
		List<Double> xi = new ArrayList<Double>();
		List<Double> honestMinersProportions = new ArrayList<Double>();
		for (int i = 0; i < numberOfHonestMiners; i++) {
			double x = 1 / numberOfHonestMiners * (i + 1);
			xi.add(x);
			double oldx = 0;
			if (i != 0) {
				oldx = Math.pow(xi.get(i - 1), power);
			}

			double y = Math.pow(x, power) - oldx;
			honestMinersProportions.add(y);
		}

		Collections.reverse(honestMinersProportions);

		xi.clear();
		List<Double> maliciousMinersProportions = new ArrayList<Double>();
		for (int i = 0; i < numberOfAllMaliciousMiners; i++) {
			double x = 1 / numberOfAllMaliciousMiners * (i + 1);
			xi.add(x);
			double oldx = 0;
			if (i != 0) {
				oldx = Math.pow(xi.get(i - 1), power);
			}

			double y = Math.pow(x, power) - oldx;
			maliciousMinersProportions.add(y);
		}

		int cpt = 0;
		List<Double> computationalPowers = new ArrayList<Double>();
		for (int i = 0; i < numberOfMaliciousChainAttackMiners; i++) {
			double proportion = maliciousMinersProportions.get(cpt + i);
			double computationalPower = proportion * MALICIOUSPROPORTION;
			computationalPowers.add(computationalPower);

		}
		cpt += numberOfMaliciousChainAttackMiners;
		for (int i = 0; i < numberOfMaliciousLedgerAttackMiners; i++) {
			double proportion = maliciousMinersProportions.get(cpt + i);
			double computationalPower = proportion * MALICIOUSPROPORTION;
			computationalPowers.add(computationalPower);
		}
		cpt += numberOfMaliciousLedgerAttackMiners;
		for (int i = 0; i < numberOfHonestMiners; i++) {
			double proportion = honestMinersProportions.get(i);
			double computationalPower = proportion * (1 - MALICIOUSPROPORTION);
			computationalPowers.add(computationalPower);
		}

		agents = getAgentsListCPBased(agents, numberOfUsers, numberOfHonestMiners, numberOfMaliciousChainAttackMiners,
				numberOfMaliciousLedgerAttackMiners, numberOfResilientUsers, numberOfMoreResilientUsers,
				computationalPowers);

		return agents;
	}

}
```

Figure 3.3 – MAX Scenario Example.

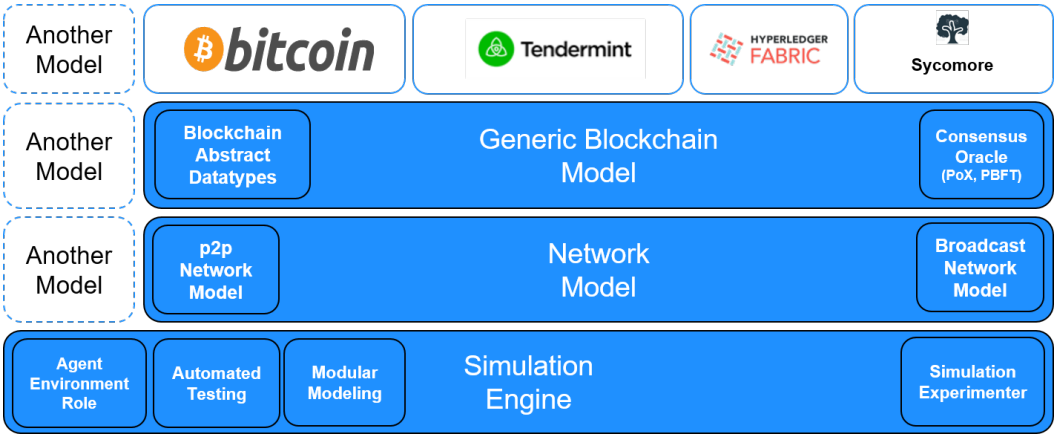


Figure 3.4 – MAX Architecture.

2. **max.datatype.ledger.** It groups data structures specific to the *blockchain* such as *blocks*, *transactions*...

3. **max.model.** It includes created models, thus the protocols implemented, their specific messages or their specific roles.

3.2.3 Core

The MAX core is responsible of the management of the basic functionalities of the simulator, namely action scheduling (with the scheduler), hierarchical organization (CGR), as well as the scenarization (with the experimenter). It is based on a discrete-event management framework named MaDKit (Multi-agent Development Kit) [86], which (hierarchically) organizes agents, manages their addressing, as well as the differentiation between roles.

MaDKit is a tool for developing multi-agent platforms. It is based on an organizational model based on groups containing agents and the roles associated with them. As shown by Figure 3.5, MaDKit follows the AGR model (Agent-Group-Role). This model is used to organize agents in a hierarchical way, and give them different actions according to their position in the hierarchy (i.e. to which group they belong to). This functionality allows the modeling of a heterogeneous system (e.g. a distributed network).

3.2.3.1 Basic Elements of MaDKit

In MaDKit, there is two types of agents. The first one is called *AbstractAgent*. It is defined as a reactive entity which can interact with the other agents belonging to its group by adopting a behavior defined by the role(s) it has in this group. The perception of the AbstractAgent is limited to what the other agents want to show him by sending him a message. In contrast, the second type of agents, called *Watcher* has a supervisory role in a given group. It has access to all the attributes

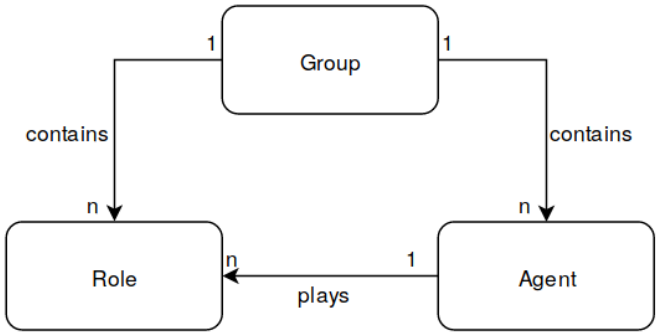


Figure 3.5 – Agent-Group-Role Organization.

and methods of any agent. It is used for scripting (experimenter) or to ensure the smooth running of the simulation. To do so, it uses the "probe" object which allows it to access the information of the agents to "monitor".

In MAX, agents follow the same model. *MAXAgent* extends *AbstractAgent* by adding some specific MAX actions while *ExperimenterAgent* (used for scripting of scenarios) and *Collector* (used for monitoring of simulations) extend *Watcher* agents.

3.2.4 Datatype

In this section, we present the data structures used in MAX and made available to model any blockchain system. These data structures are decomposed into two sub-sets, (i) max.datatype.com, which gathers the necessary structures for communication, thus separated from any notion of blockchain, and (ii) max.datatype.ledger, which will complete max.datatype.com by adding the blockchain-specific data structures necessary for the implementation of any blockchain protocol.

3.2.4.1 max.datatype.com

By definition, a blockchain system is a replicated application. It is based on a network of nodes which communicate between them following a given protocol. Therefore, for simulating properly any blockchain system, communication is necessary. *max.datatype.com* groups all the data structures necessary for communication between agents.

Each agent in each group to which it belongs have an assigned Address. It is an identifier that has a unique representation and is of type UUID (Universally Unique IDentifier). It is a structure of 128 bits used to identify agents. Messages are then used for interactions between agents in a given environment, they constitute a data structure that takes as parameters the sending agent's address, the receiving agent's address and a payload.

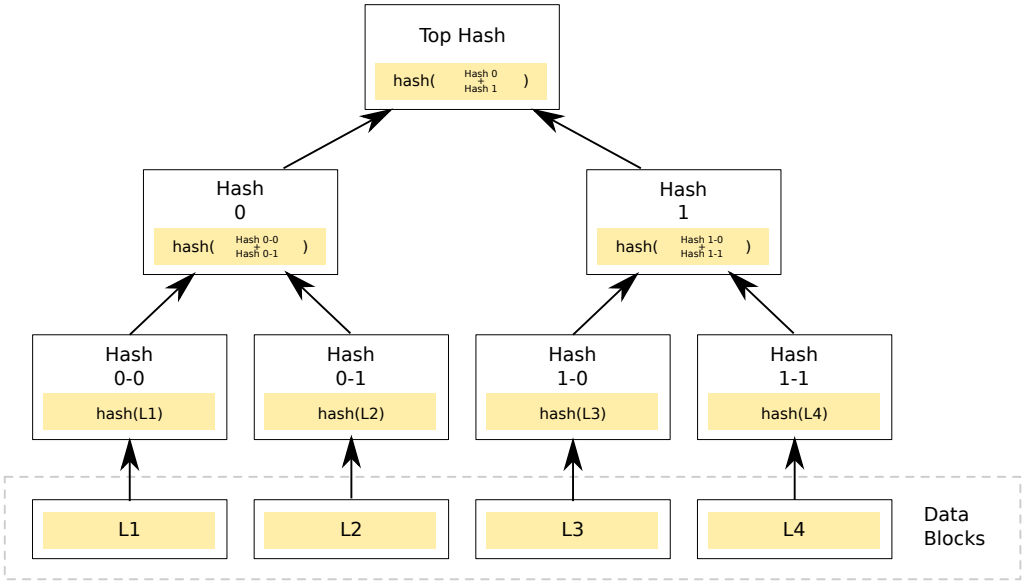


Figure 3.6 – Merkle Tree Construction.

3.2.4.2 max.datatype.ledger

As said before, MAX is a simulator dedicated to blockchain. It must necessarily embark data structures specific to blockchain. For example, in a blockchain, each user has a *Wallet* that contains important information such as his address, his cryptographic key pair and his transaction history. For a user to transfer goods to another, it sends them from his wallet using a *Transaction*. After its creation, this transaction is broadcast to the network to be put into a *Block*. It is modeled as a *Message* where the payload represents the amount transferred. Transactions are then organized in a *MerkleTree* [87]. As shown by Figure 3.6 [87], the Merkle Tree is a tree created from transactions, these are hashed two by two until there is only one hash left, the root of the tree, also called Merkle Root.

The MerkleTree is then put in the header of a Block, which is a data structure containing a set of transactions chosen by the miner in his *Mempool* in addition to an extra transaction which represents his reward, the coinbase transaction. The Block also contains other information such as the version of the protocol, a timestamp, a nonce and the hash of the previous block, which allows to build what we call, a *Blockchain*. In MAX, a Blockchain is a dynamic list of blocks configured as "append-only" where each block contains a cryptographic reference to the previous block. The very first block is called a genesis block and is created when the blockchain is created.

3.2.5 Model

Usually, methods used to study complex protocols are based on mathematical approaches that are long and difficult to implement. Agent-based approach allows us to better understand the algorithm, and modify it in order to improve it. To do so, MAX proposes abstract models which will be used as bases for the various blockchain protocols that could be implemented. Message exchanging and blockchain data structures management are two common aspects of the majority of existing blockchain protocols.

3.2.5.1 Network Model

It is the ability of nodes to communicate that allows blockchain systems to function properly. In this section, we will present the "network" communication module of MAX: *max.model.network.p2p*. It represents the modeling of a classical unstructured peer-to-peer network.

As illustrated by Figure 3.7, P2PAgent sends and receives messages via P2PEnvironment which manages the communication between agents according to parameters such as delay and transmission reliability.

In this model, agents are linked through a *Connection*. It allows them to communicate. However, a connection being a logical link, message sending is done via the *P2PEnvironment* which delivers the message to the recipient(s) specified by the sending agent. Those recipients being contained in its *ConnectionList*, which groups all of its connections.

In order to add a Connection to its ConnectionList, an agent has to request a Connection to another agent. To do so, it sends a "CONNECT" message to an other agent it wants to communicate with. At its reception, the corresponding agent can reply with an "ACCEPT", which implies the establishment of the connection or a "REJECT" message. Note that while waiting for the other agent's reply, the connection request is added to the *waiting list*. A list of connection requests waiting to be replied to. This is done to respect limitations on the number of connections an agent has. In our model, these limitations are represented by connection bounds that define the maximum number of both incoming and outgoing connections.

3.2.5.2 Blockchain Model

As the MAX simulator is dedicated to blockchain, an abstract model representing the blockchain and its agents is provided to contain all the necessary actions for the proper functioning of a blockchain model. In this way, it is made easy to create blockchain protocols that follow this abstract model.

This model links the communication primitives of the Network model (see section 3.2.5.1) with the data structures described in section 3.2.4. In this way, agents in this model (Blockchain agents) are Network agents having access to (i) blockchain data structures and (ii) specific blockchain actions such as the creation of blocks, issuing of transactions... Due to their complexity, some of these actions are abstracted

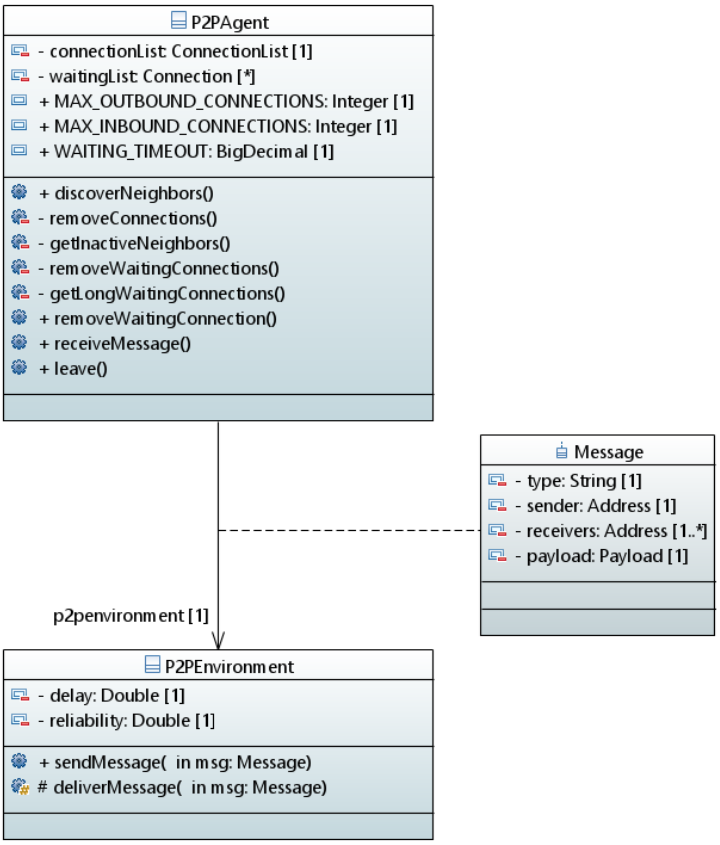


Figure 3.7 – Class Diagram: Network model.

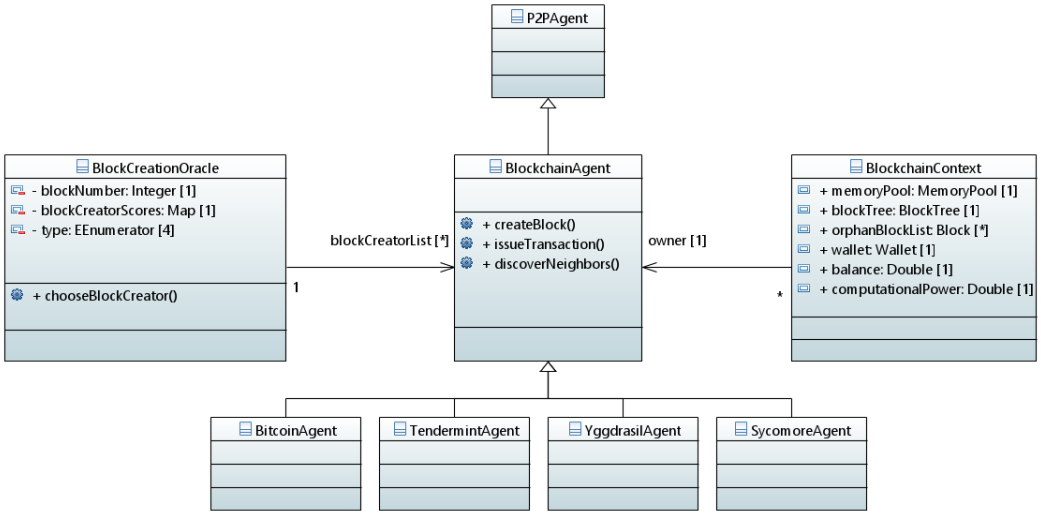


Figure 3.8 – Class Diagram: Blockchain model.

using an oracle [88]. It is a Watcher agent (see section 3.2.3) which can access the properties of all the agents. In this way, it can elect some specific agents to execute some actions such as the creation and broadcast of a block at a given time. Note that this agent is abstract and can be completed in any model inheriting from the blockchain model. However, some block creation models are already made available to facilitate the creation of new models (i.e. PoW, PoS).

As shown by Figure 3.8, BlockchainAgent is a P2PAgent that has access to *blockchain* structures. This agent also has the ability to create *blocks* by being helped by a new type of agent "BlockCreationOracle". The modeling of Blockchain protocols such as Bitcoin, Tendermint, Sycomore and Yggdrasil extends the Blockchain model.

3.3 Experimental Environment

The mono-threaded architecture of MAX made simulations relatively long depending on the complexity of the model and/or the scenario [1]. As we will see later, the implemented protocols are complex and require a lot of resources. The different experiments carried out during this thesis (see chapters 4 and 5) were therefore quite heavy to simulate. In order to carry out these experiments in a timely manner, we have chosen to use a high performance computing grid specialized in distributed computing coupled with a task distribution software between different machines. Note however that we did not distribute the calculation for a simulation (impossible due to the architecture of MAX), we only used this platform to parallelise our

[1] The transition to a multi-threaded architecture that would allow better performance is part of the future work.

simulation scenarios (several thousands) on multiple nodes and thus benefit from a considerable time saving (a few hours rather than several days or weeks), no changes were made on MAX to this extent.

3.3.1 Grid'5000

In 2003, a platform for experimental research on parallel and distributed systems was created. This platform, called Grid'5000 [6], is a test bed for experiments on different types of distributed systems (high performance computing, peer-to-peer systems, etc.). It is currently composed of 40 clusters, 752 nodes and 15788 CPU cores, mainly located in France. The different nodes of this system are cut off from all access to the Internet except for some sites on a regularly maintained white list. This is to avoid any misuse of the Grid'5000 power (DDoS attack for example).

Operationally, in order to execute an experiment, users must (i) choose and reserve resources for a given period of time and (ii) specify the script to be executed. As far as resource reservation is concerned, users can either browse the list of resources on the web interface provided and choose the ones he/she wants or specify his/her needs to the system which will then choose the appropriate resources that will be free the fastest. Resource reservation management is done through the OAR tool [89]. Regardless of the approach used for the two steps described here, access to resources (sites and nodes) is via SSH. Each site has its own NFS server. This is to ensure that the resources of a particular site can be used even when the link to other sites is being maintained. Please note that Grid'5000 is a set of nodes having all exactly the same software environment. Nevertheless, it is possible for any user wishing to experiment on a heterogeneous system to implement specific system images for each node (or group of nodes) he will have reserved, using Kadeploy [90].

3.3.2 GNU parallel

In order to parallelise our simulations on the several nodes provided by Grid'5000, we used the GNU parallel solution. GNU parallel [91] is a shell tool used for running tasks in parallel on multiple threads. A task can be a single command or a small script that has to be executed for each line of the input. The typical input is a file containing a list of parameters (files, hosts, URLs...).

3.4 Implemented Models

After having presented our simulation environment and the softwares we used, we present in this section the implementations of the protocols we study in this manuscript. These implementations include both data structures and actions set up to model the desired behaviors. As all other models in MAX, our newly implemented models meet MAX's requirement for code testing, which is used for analysis of completeness, correctness and requirement fulfillment of our models in various contexts.

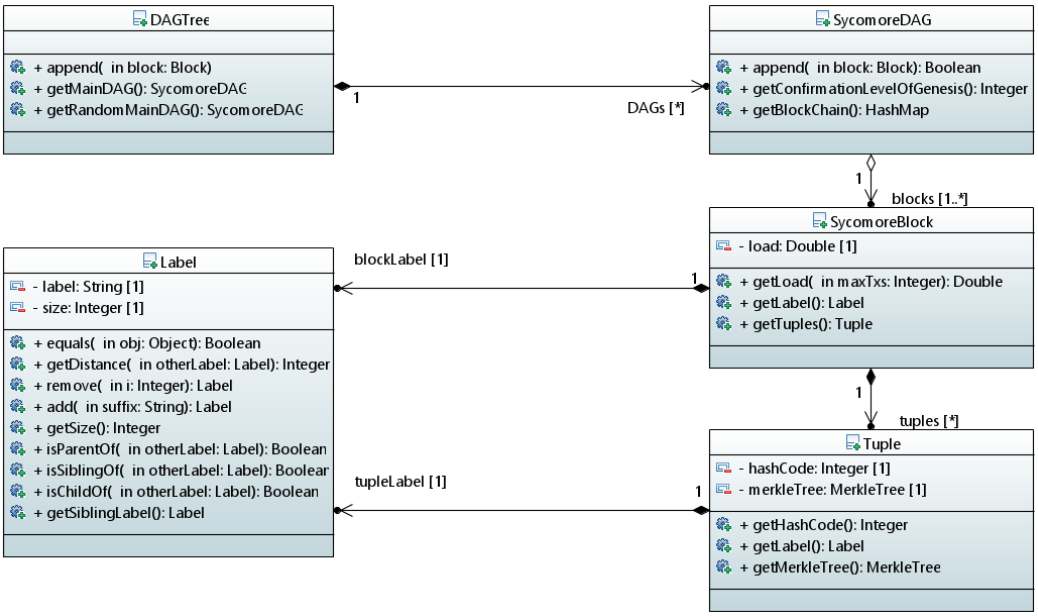


Figure 3.9 – Class Diagram: Sycomore Datatype.

Note that both these implementations are based on the abstract blockchain model presented in Section 3.2.5.2 and the peer-to-peer network model presented in Section 3.2.5.1.

3.4.1 Sycomore

Sycomore [27] has been the first protocol we have studied during this thesis. It consists of a Proof-of-Work blockchain protocol maintaining a Directed Acyclic Graph of blocks. It uses the same basic primitives as Bitcoin with a DAG structure. Thanks to it, Sycomore promises to improve the scalability of Bitcoin by self-adapting its structure to transaction demand. In order to verify its properties, we wanted to study this protocol from different angles presented in Chapter 4.

This section will present the modeling of Sycomore in MAX. Modeling that we will use to carry out the different experiments presented in Chapter 4. Note that in the following, we will use some concepts introduced in Chapter 4. Deep understanding of these concepts is not necessary for the proper reading of this section. However, feel free to refer to Chapter 4 for more details.

Datatype Being based on Bitcoin, Sycomore receives the same basic primitives as the latter already modeled in MAX. However, its graph shape requires the addition of new structures that we had to implement in order to model this protocol correctly. As illustrated by Figure 3.9, the data structures used in Sycomore are the following: *DAGTree*, *SycomoreDAG*, *SycomoreBlock*, *Tuple* and *Label*. This paragraph details these different classes and the links between them.

The DAG structure of Sycomore is implemented using the *SycomoreDAG* class which is nothing else than a list of blocks chained to each other forming a graph. Each of these blocks (*SycomoreBlock*) has two essential attributes, the *Label* and the *Tuple*, they compose the header of the SycomoreBlock. Sycomore uses the Label, a binary string used to differentiate DAG chains, it is thus the first element that we have implemented. This class makes it possible to modify binary elements of the label by removing the n least significant bits (*remove* method) or on the contrary to add some bits (*add* function). These functions were implemented to allow the creation of new labels (e.g. in case of split or merge [27]). It is also possible to check if a label is parent of another, if it is sibling of another, if it is child of another or even to get its distance [27] from another label.

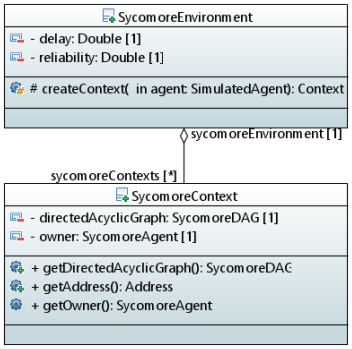
Moreover, as one of the properties of Sycomore states that the predecessor of a block cannot be known before its creation, each miner must integrate his view of the DAG in each block he creates. He then integrates this view in the header of his block in what we call tuples. These tuples are groups of 3 elements, the hashcode of the previous block, the potential label of the block being created and the merkleTree of transactions assigned to this label [27].

Finally, in order to manage the appearance of forks, we have added the *DAGTree* class which is nothing else than a list of *SycomoreDAG*. Indeed, a fork being nothing else than an alternative history of the ledger. It can be seen as an n^{th} DAG different from the other DAGs of the DAGTree in at least one conflicting block. This class allows us to resolve conflicts related to forks. At each moment, it is possible to calculate which DAG will be considered main DAG thanks to getMainDAG(). As a reminder, in Sycomore, the main DAG is the DAG whose genesis block is the most deeply confirmed. If two DAGs satisfy this criterion at the same height, we say that there is no main DAG at that specific moment, so this method returns *null*. In this case, another method allowing to choose one of these two DAGs at random has been developed. Note that this function only exists for the purpose of monitoring the correct execution of the simulation.

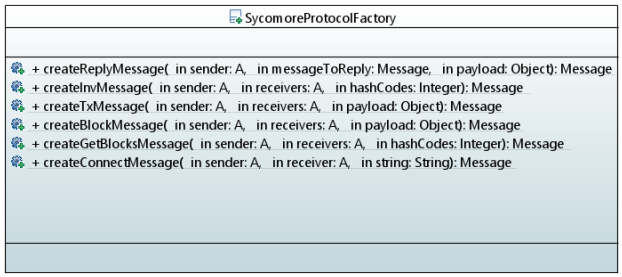
Environment As illustrated by Figure 3.10b, Sycomore uses the same message types as Bitcoin, namely *INV*, *GETDATA*, *BLOCK*, *TX* and *GETBLOCK*. Since these messages have a complex structure, we have created *SycomoreProtocolFactory* in order to facilitate their creation and thus limit development errors. This class makes it possible to easily create these different types of messages with the right arguments in the right places.

Messages are then sent via the environment to be received by the receivers in order to be processed. As the reception of messages in MAX is managed as an event, we have also created handlers (see figure 3.10c) associated to each type of message. This way, when a message of a certain type is received, the right handler is triggered and the message is processed according to the desired behavior (e.g. *MHInv* handles *INV* messages).

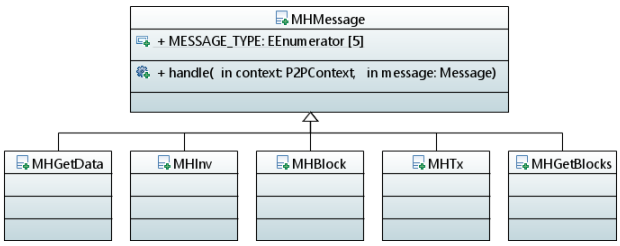
In this model, *SycomoreEnvironment* manages sending and reception of mes-



(a) Class Diagram: Sycomore Environment.



(b) Class Diagram: Sycomore Protocol Factory.



(c) Class Diagram: Sycomore Messages.

Figure 3.10 – Class Diagram: Sycomore Environment and Messaging.

sages with various network parameters, namely transmission delay and reliability. Additionnally, each agent being able to be part of several environments, it must differentiate what it receives from these various environments. We have therefore created *SycomoreContext* which allows to make this link between *SycomoreAgent* and *SycomoreEnvironment*. The context contains the state of the agent in one particular environment (see figure 3.10a).

Experimenter In our Sycomore model, we extend the Experimenter presented in section 3.2 and create *SycomoreExperimenter* (see Figure 3.11 for UML diagram), which allows to launch the agents necessary to the good functioning of the simulation. It also defines the various parameters of the simulation. Among these parameters necessary to the good functioning of the simulation are values linked to the Sycomore protocol such as *GAMMA*, *TAU* (in reality, it is a small gamma in Sycomore), *CMIN* and *HMAX*. Others are broader parameters related to our experiments such as *TXS_CONFIRMATION_DELAY* (i.e. the depth of a block needed to consider it and the transactions it contains as confirmed), *DIFFICULTY* (the difficulty to solve the PoW, which will impact the inter-block delay) and *MALICIOUS-PROPORTION* (proportion of computational power allocated to the malicious ones when the scenario requires it).

An example of configuration file is illustrated by Figure 3.12.

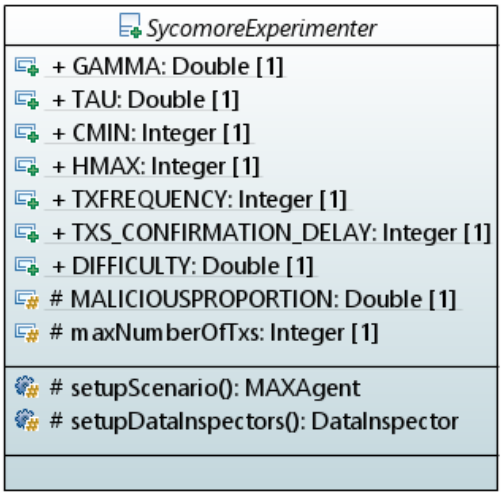


Figure 3.11 – Class Diagram: Sycomore Experimenter.

Agent In MAX, an agent executes actions in a planned way. These actions are executed according to the role the agent can have. In Sycomore, these roles are named *RSycomoreUser* or *RSycomoreMiner*. Globally, the miner (RSycomoreMiner) executes the actions associated to the creation of blocks while the user (RSycomoreUser) can only create transactions. Note however that a miner is also a user and therefore has the same actions as the user.

Figure 3.13 represents the SycomoreAgent and the different actions implemented in our model. These actions can be divided into two categories, the user actions and the block creation actions intended for the miners.

For users, *ACIssueTransaction* allows to broadcast a transaction to the network, while *ACGetBlocks* allows to send a "*GETBLOCK*" message, an action usually executed when an agent connects to the network so that it can synchronize its state with the others. As for the miners actions, we can find *ACDeciteToCreateBlock*, which allows to start the creation of a block following a geometrical or binomial distribution, as desired [2]. *ACDecideToBroadcastBlock*, which allows the agent to check if he has received anything conflicting with the block he is about to broadcast. If yes, the block is dropped, otherwise *ACBroadcastBlock* is executed to broadcast the block.

3.4.2 Yggdrasil

Apart from Sycomore, we have studied another protocol, one that we have designed and built ourselves. It is a state-sharding solution called Yggdrasil (presented in details in Chapter 5) that we modeled and then studied from a simulation point

[2]Note that we have configured these two PoW models to always have an inter-block delay per string of 10 ticks.

```
<?xml version="1.0"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
        <comment>Sycomore Model Properties</comment>

        <!-- Core Properties -->
        <entry key="MAX_CORE_EXPERIMENTER_NAME">max.model.ledger.sycomore.exp.EXSycomoreCPBased</entry>
        <entry key="MAX_CORE_RESULTS_FOLDER_NAME">./results/</entry>
        <entry key="MAX_CORE_EXPORT_PNG">true</entry>
        <entry key="MAX_CORE_SIMULATION_STEP">0.1</entry>
        <entry key="MAX_CORE_UI_MODE">SERVER</entry>

        <!-- Network Properties -->
        <entry key="MAX_MODEL_NETWORK_DELAY">0</entry>
        <entry key="MAX_MODEL_NETWORK_RELIABILITY">1</entry>
        <entry key="MAX_CORE_MAX_OUTBOUND_CONNECTIONS">100</entry>
        <entry key="MAX_CORE_MAX_INBOUND_CONNECTIONS">117</entry>

        <!-- Blockchain Properties -->
        <entry key="MAX_MODEL_LEDGER_FEE">0.0002</entry>
        <entry key="MAX_MODEL_LEDGER_MAX_NUMBER_TXS">100</entry>
        <entry key="MAX_MODEL_LEDGER_REWARD">12.5</entry>

        <!-- Sycomore Properties -->
        <entry key="MAX_MODEL_LEDGER_SYCOMOREPP">false</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_NUMBER_OF_USERS">1</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_NUMBER_OF_MINERS">64</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_NUMBER_OF_MALICIOUS_MINERS">0</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_NUMBER_OF_MALICIOUS_CHAIN_ATTACK_MINERS">0</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_NUMBER_OF_MALICIOUS_LEDGER_ATTACK_MINERS">0</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_NUMBER_OF_RESILIENT_USERS">0</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_NUMBER_OF_MORE_RESILIENT_USERS">0</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_TIMEOUT_BLOCK">500</entry>
        <entry key="MAX_CORE_TIMEOUT_TICK">1000000</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_GAMMA">0</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_TAU">1.0</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_CMIN">1</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_HMAX">1000</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_SPLITMAX">0</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_TXS_CONFIRMATION_DELAY">6</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_TXS_FREQUENCY">160</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_TXS_PERIOD">100</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_MALICIOUS_PROPORTION">0.5</entry>

        <!-- Blockchain Observation / Data Collection Properties -->
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_BCBLOCK_CREATING_ACTIVITY">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_BCCONFIRMATION_TIME_OF_TXS_PER_BLOCK">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_BCCONFIRMATION_TIME_OF_BLOCKS">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_BCPOWER">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_BLOCK_WINDOW">CLI</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_GENVIRONMENT">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_LCBALANCE">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_LCCONFIRMATION_TIME_OF_TXS">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_LCNUMBER_OF_AGENTS">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_LCSYNCHRONIZATION">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_LCPERCENTAGEOFSYNCHRONIZATION">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_LCTOTAL_FEE_BY_BLOCK">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_LCTXS_BY_BLOCK">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_LCUNCONFIRMED_TXS_BY_TIME">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_RGCONFIRMED_TX_ACTIVITY_OF_USERS">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_RGCONNECTIVITY">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_RGNEIGHBOR">OFF</entry>

        <!-- Sycomore Observation / Data Collection Properties -->
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_GBLOCKCHAIN">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_GMAIN_BLOCKCHAIN">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_BCPOTENTIAL_GAIN_WHEN_MALICIOUS">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_LCGAIN_OF_MINERS">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_LCMALICIOUS_GAIN">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_BCPOTENTIAL_GAIN_WHEN_MALICIOUS_PER_BLOCK">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_BCPROPORTION_OF_MALICIOUS_MINERS_CP">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_LCUNCONFIRMED_TXS_BY_TIME">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_LCTXS_THROUGHPUT_BY_TIME">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_LCCUMULATIVE_TRANSACTIONS_CONFIRMATION_BY_TIME">CLI</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_BCLATENCY_OF_TRANSACTIONS">CLI</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_FCOMPUTATIONAL_POWER">CLI</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_FENERGY_CONSUMPTION_PER_BLOCK">CLI</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_BCCONFIRMATION_TIME_OF_TXS_PER_LABEL">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_BCCONFIRMATION_OF_TXS_PER_LABEL">CLI</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_BCUNCONFIRMED_TXS_PER_LABEL">CLI</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_LCNUMBER_OF_LEAF_BLOCKS_BY_TIME">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_LCPROPORTION_OF_MALICIOUS_AGENTS">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_BCGAIN_OF_MINERS">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_LCUTILITY">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_NETWORK_VIEWER">OFF</entry>
        <entry key="MAX_MODEL_LEDGER_SYCOMORE_BCINTERBLOCK_DELAY">CLI</entry>
</properties>
```

Figure 3.12 – Sycomore Configuration File.

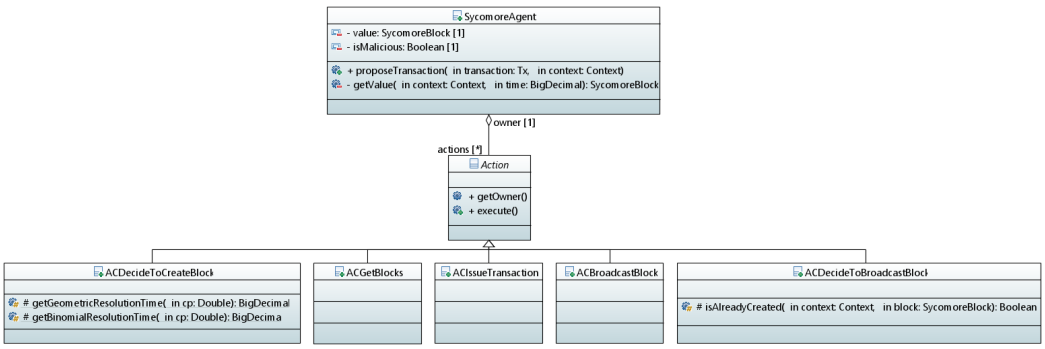


Figure 3.13 – Class Diagram: Sycomore Agent and actions.

of view in order to demonstrate its full potential but also to assess some essential properties. In this section, we will present our modeling of Yggdrasil in MAX. Note that in the following, we will use some concepts introduced in Chapter 5. Deep understanding of these concepts is not necessary for the proper reading of this section. However, feel free to refer to Chapter 5 for more details.

Core Yggdrasil, being completely different than all the other protocols modeled in MAX (since it is a system with several environments that evolve in parallel), some basic functionalities were not compatible with the functioning of a system such as Yggdrasil. Some major modifications had to be made in some cases. For example, the management of event-driven executions in MAX did not take into account the environment in which the action should take place. It implied that the creation of a block in one environment lead to the creation of blocks in all environments at the same time. Moreover, it was impossible for two agents belonging to two different environments to communicate, but in a sharding context, cross-shard communication is essential. We therefore had to modify the lower layers of MAX to (i) model the correct behavior of our system and (ii) make MAX more intuitive when managing agents, environments or actions.

Datatype In Yggdrasil, we differentiate our shards using the Label, a binary element already present in Sycomore (see figure 3.9).

In addition to that, we introduce two new types of transactions, the *ShardUpdate* Transaction, created for the good function of Yggdrasil. It allows communication between the shards and the masterchain, thus the synchronization of all shards states and the maintenance of the global state's coherence. Moreover, for our experimentations' needs, we have also implemented the notion of smart-contract. Indeed, having proposed a 2-phase commit protocol for smart-contracts in Yggdrasil (see Chapter 5 for more details), we wanted to study it more closely by modeling its execution. To do so, we used *SmartContractInvoke* transactions that would invoke one of the methods of a *SmartContract*. Note that no method is executed when the transaction

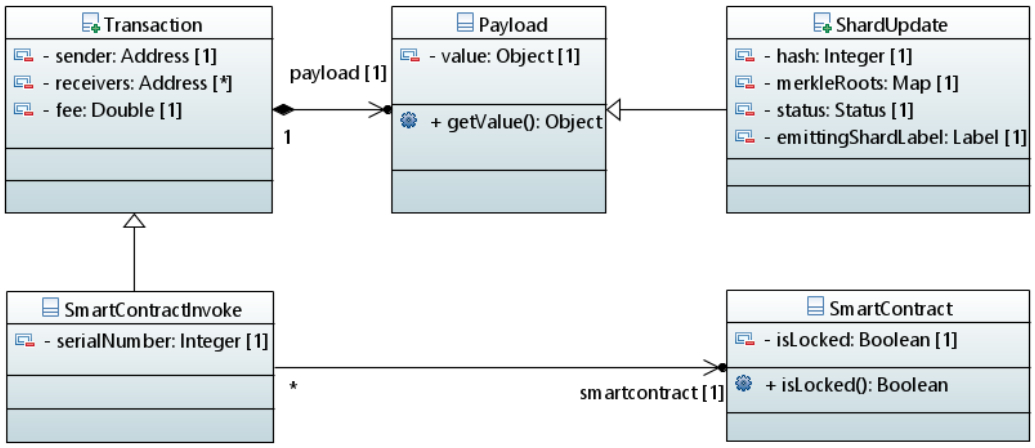


Figure 3.14 – Class Diagram: Yggdrasil Datatype.

is admitted in a block, we only use the smart-contract concept to demonstrate the performances of our 2-phase commit algorithm. Thus, we are only interested in the scheduling of transactions related to the execution of a smart-contract. To do so, we have added a *serialNumber*, which allows us to order transactions, thus confirm them in a precise order.

Environment Figure 3.15 illustrates our modeling of the environment in Yggdrasil. Yggdrasil practices state-sharding, which implies the maintenance of several "shards" that have neither the same state (blockchain), nor the same composition (agents). These shards are therefore different environments that evolve in parallel (*Shard*) where the agents assigned to the Shard can communicate with each other.

As specified in Chapter 5, Yggdrasil uses a masterchain to maintain the global state of the system, this masterchain is managed by all the validators of the system on an environment that we call *MasterShard*. This environment is nothing else than a *Shard* where all agents (of the system) can communicate. These two elements are identified by their label (previously presented in section 3.4.1).

Due to its dynamic sharding property, Yggdrasil create or delete shards thanks to split and merge actions. In order to abstract this feature from our algorithm, we have implemented an Oracle, responsible of managing these mechanisms, thus managing the set of shards.

Each of these shards uses a Tendermint consensus engine [3] which is used here as a black box. This black box is composed of (i) *TBEnvironment*, a blockchain environment on which the elected validators can communicate to allow the creation of new blocks in the shard. (ii) *TBTMOracle*, an oracle which will allow us to manage more efficiently the addition and removal of agents in the TBEnvironment. (iii) *TBContext*, a necessary class in MAX that allows to link an agent to an environment

[3]Note that Tendermint is also implemented in MAX [5]

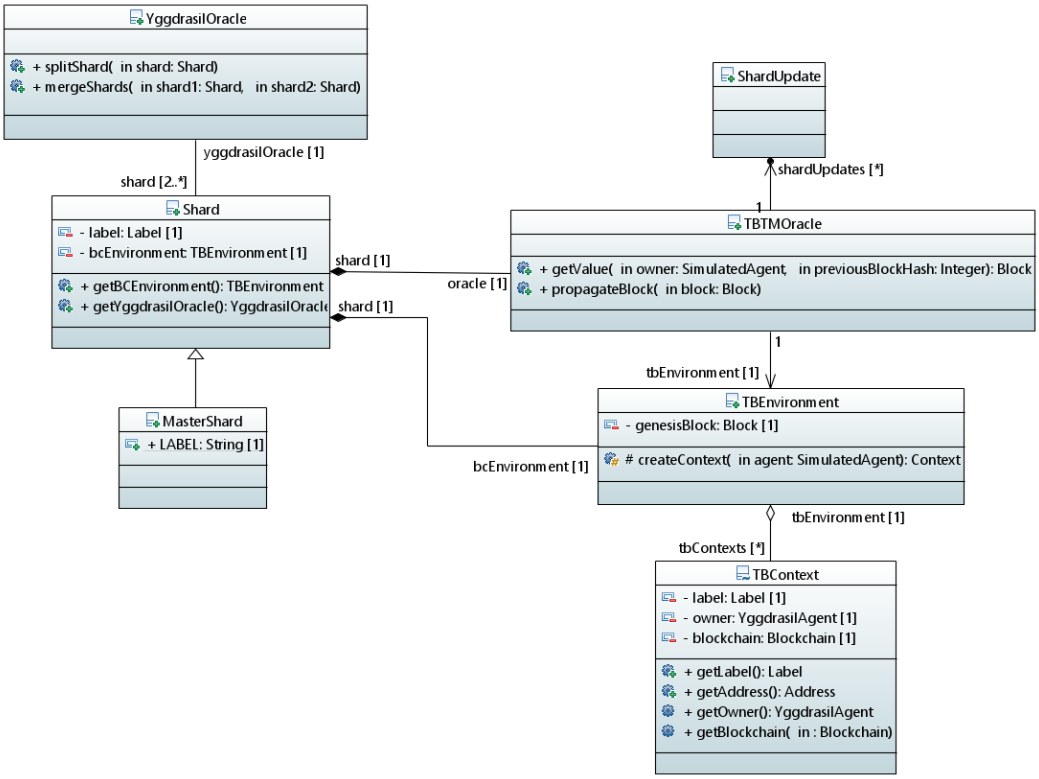
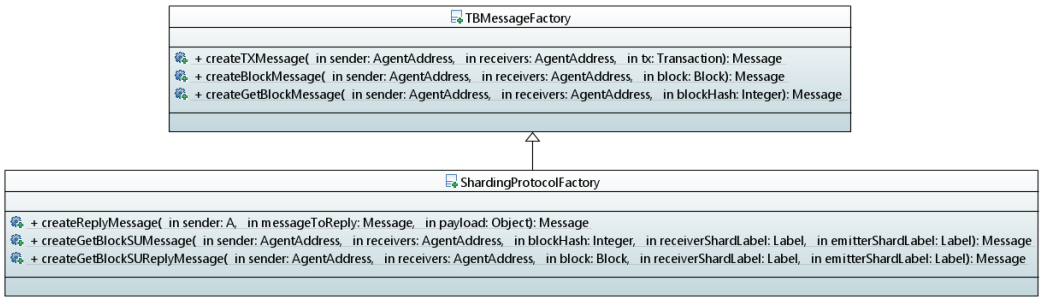
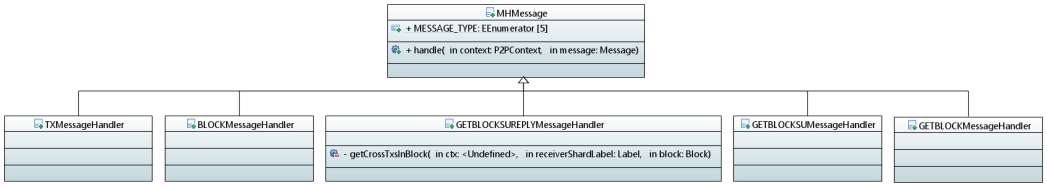


Figure 3.15 – Class Diagram: Yggdrasil Environment.



(a) Class Diagram: Yggdrasil Protocol Factory.



(b) Class Diagram: Yggdrasil Messages.

Figure 3.16 – Class Diagram: Yggdrasil Message Creation and Handling.

in order to maintain an agent-environement specific state (see section 3.2 for more details).

Messaging Each shard in Yggdrasil uses Tendermint as consensus engine. It therefore uses some necessary messages for its correct operation (*Tx*, *Block*, *GetBlock*)[4] but also other types of messages specific to Yggdrasil and which concern especially the management of shard update transactions (*GetBlocksSU* and *GetBlocksSUReply*). These last ones allow respectively to ask for information following the reception of a ShardUpdate and to answer this request for information. In order to facilitate the creation of these messages, we have implemented two classes *TBMessageFactory* for the Tendermint messages and *ShardingProtocolFactory* for the Yggdrasil messages (figure 3.16a).

As explained before, sent messages must be processed by the receivers, that's why we have also implemented classes called "handlers" (see figure 3.16b) associated to each type of message and which will be responsible for processing the message (e.g. *TXMessageHandler* handles *Tx* messages).

Experimenter As for Sycomore, in order to launch the different scenarios of experimentation, only one agent is necessary. YggdrasilExperimenter allows to launch the agents necessary to the good functioning of the simulation. It is used to define the various parameters of the simulation.

[4]The necessary messages for Tendermint consensus are encapsulated in the Tendermint layer, thus not visible in the Yggdrasil model.

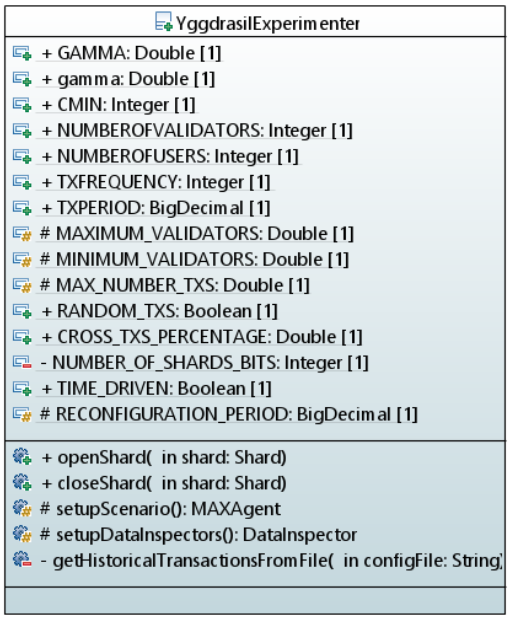


Figure 3.17 – Class Diagram: Yggdrasil Experimenter.

Being the only agent able to create other agents and thus environments, it is also used during split or merge to create new shards. It contains the parameters necessary for the operation of Yggdrasil (*GAMMA*, *CMIN*, *NUMBEROFVALIDATORS*...) but also parameters created for our experiments, such as *CROSS_TXS_PERCENTAGE* (to be able to manipulate the rate of cross-shard transactions), *TIME_DRIVEN* (to activate or not the time-driven solution), *NUMBER_OF_SHARDS_BITS* (to start the simulation with a certain number of shards) For our experiments, we used historical Ethereum transactions downloaded for the occasion. These transactions are loaded at the start of the simulation in the Experimenter so that they can be used by the agents (*getHistoricalTransactions* method).

An example of configuration file is illustrated by Figure 3.18.

Agent As shown by Figure 3.19, Yggdrasil nodes can manage several identities with different roles (*RYggdrasilUser* and *RYggdrasilValidator*). We have implemented two entities, a node (*YggdrasilNode*) that can create agents (*YggdrasilAgent*), which participate in the maintenance of the blockchain either as validators thus participate in the creation of blocks or as users thus simply send transactions. Depending on its role, an agent usually has access to different actions. In Yggdrasil, all agents have access to the only action defined in Yggdrasil, *ACIssueTransaction*, which allows them to create and send one or more transactions to the network. Validators also have access to the actions related to block creation which are contained in the Tendermint level.

```
<?xml version="1.0"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
        <comment>Yggdrasil Model Properties</comment>

        <!-- Core Properties -->
        <entry key="MAX_CORE_EXPERIMENTER_NAME">max.model.ledger.sharding.env.YggdrasilEnvironment</entry>
        <entry key="MAX_CORE_RESULTS_FOLDER_NAME">./results/test/</entry>
        <entry key="MAX_CORE_EXPORT_PNG">true</entry>
        <entry key="MAX_CORE_SIMULATION_STEP">0.1</entry>
        <entry key="MAX_CORE_UI_MODE">OFF</entry>

        <!-- Network Properties -->
        <entry key="MAX_MODEL_NETWORK_DELAY">0</entry>
        <entry key="MAX_MODEL_NETWORK_RELIABILITY">1</entry>
        <entry key="MAX_CORE_MAX_OUTBOUND_CONNECTIONS">100</entry>
        <entry key="MAX_CORE_MAX_INBOUND_CONNECTIONS">117</entry>
        <entry key="MAX_MODEL_NETWORK_NEIGHBOUR_DISCOVERY_FREQUENCY">1</entry>

        <!-- Blockchain Properties -->
        <entry key="MAX_MODEL_LEDGER_FEE">0.0002</entry>
        <entry key="MAX_MODEL_LEDGER_MAX_NUMBER_TXS">100</entry>
        <entry key="MAX_MODEL_LEDGER_REWARD">12.5</entry>
        <entry key="MAX_MODEL_LEDGER_TENDERMINT_SIZE_OF_COMMITTEE">1</entry>
        <entry key="MAX_MODEL_LEDGER_TENDERMINT_DEFAULT_PHASE_TIMEOUT">10</entry>
        <entry key="MAX_MODEL_LEDGER_TENDERMINT_DEFAULT_DELTA_TIMEOUT">5</entry>
        <entry key="MAX_MODEL_LEDGER_TENDERMINT_DELAY_BETWEEN_COMMITTEES">2</entry>
        <entry key="MAX_MODEL_LEDGER_TENDERMINT_PHASE_1">PRE_PROPOSE</entry>
        <entry key="MAX_MODEL_LEDGER_TENDERMINT_PHASE_2">PROPOSE</entry>
        <entry key="MAX_MODEL_LEDGER_TENDERMINT_PHASE_3">VOTE</entry>

        <entry key="MAX_MODEL_LEDGER_BLOCKCHAIN_TIMEOUT_BLOCK">100000</entry>
        <entry key="MAX_CORE_TIMEOUT_TICK">300</entry>

        <entry key="MAX_MODEL_LEDGER_YGGDRASIL_RANDOM_TXS">FALSE</entry>
        <entry key="MAX_MODEL_LEDGER_YGGDRASIL_K_SCALE">FALSE</entry>

        <entry key="MAX_MODEL_LEDGER_YGGDRASIL_CROSS_TXS">2</entry>

        <entry key="MAX_MODEL_LEDGER_YGGDRASIL_NUMBER_OF_SHARDS_BITS">0</entry>


        <!-- Yggdrasil Properties -->
        <entry key="MAX_MODEL_LEDGER_YGGDRASIL_NUMBER_OF_USERS">1</entry>
        <entry key="MAX_MODEL_LEDGER_YGGDRASIL_NUMBER_OF_VALIDATORS">200</entry>
        <entry key="MAX_MODEL_LEDGER_YGGDRASIL_gamma">0</entry>
        <entry key="MAX_MODEL_LEDGER_YGGDRASIL_GAMMA">0.9</entry>
        <entry key="MAX_MODEL_LEDGER_YGGDRASIL_MAXIMUM_VALIDATORS">2</entry>
        <entry key="MAX_MODEL_LEDGER_YGGDRASIL_MINIMUM_VALIDATORS">0</entry>
        <entry key="MAX_MODEL_LEDGER_YGGDRASIL_CMIN">1</entry>
        <entry key="MAX_MODEL_LEDGER_YGGDRASIL_TXS_FREQUENCY">100</entry>
        <entry key="MAX_MODEL_LEDGER_YGGDRASIL_TXS_PERIOD">0</entry>
</properties>
```

Figure 3.18 – Yggdrasil Configuration File.

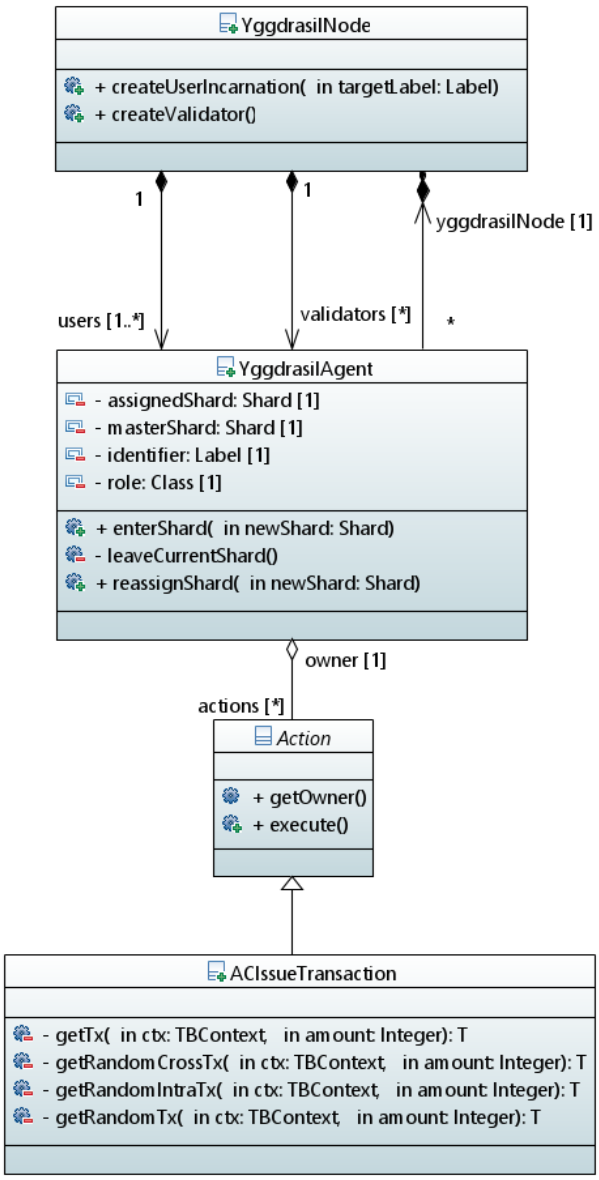


Figure 3.19 – Class Diagram: Yggdrasil Agent and Actions.

3.5 Conclusion

In this chapter we wanted to present the tools we used for our experiments and explain the choices we made in this respect. Many of our results were conducted on this simulation environment and we believe that the good understanding and therefore the confidence given to this environment is essential for the good appreciation of our numerous simulation results.

All the protocols we studied were modeled in MAX, an agent-based simulator

dedicated to blockchains, which allowed us to efficiently model the protocols we wanted to study but also the different scenarios we imagined to stress these protocols and thus show their limits. Unfortunately, such a complexity on a mono-threaded simulator implies excessively long simulations. This is why we requested the use of Grid'5000, a distributed computing grid that allowed us to run several experiments in parallel (using Parallel GNU) and thus benefit from a considerable time saving.

For our own needs, we needed to improve MAX in terms of performance, completeness and usability. These contributions can be counted around thousands of lines of code ($500,000 \sim 600.000$). Let us summarize these in a few lines:

- Implementation of the first graph-based model in MAX (Sycomore): block append mechanism and fork management different from classical systems.
- Creation of a realistic and probabilistic PoW model (before, we had a static model with a block every 10 ticks).
- Implementation of the first model with several parallel environments (Yggdrasil).
- Modification of the MAX core to support event management in parallel environments (for Yggdrasil's implementation).

As a future work, regarding MAX, in view of the lack of performance of the latter, it has been detected some important areas of improvement. Among these:

- Switch to a multi-threaded architecture to avoid blocking due to the use of a single scheduler.
- Implement different types of network models or extend them to allow more flexibility when creating malicious models.
- Allow a more dynamic management of agents and their actions (e.g. entrance/exit of agents).

In addition to the tools used, we also have shown in detail the implementations of the two protocols studied during the chapters 4 and 5. This was done to clarify our models to the reader so that he can better understand our studies. It is important to note that for some models, it was necessary to modify the simulator in depth, which contributed to the improvement of this software and thus to allow it to embark more functionalities like the integration of event-driven executions by environment.

In the next chapter, we will focus on a particular graph-based blockchain protocol, Sycomore.

Chapter 4

Graph-based blockchains

"A single act of kindness throws out roots in all directions, and the roots spring up and make new trees"

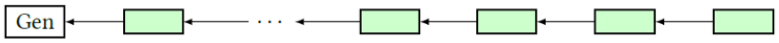
– Amelia Earhart

Contents

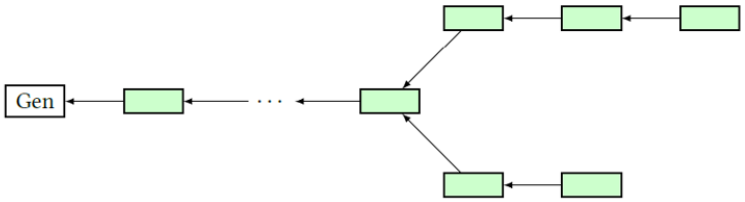
In this chapter, we focus on the potential of graph-based blockchain in terms of scalability. We first analyse Sycomore and propose a new graph-based protocol relying on Sycomore and addressing Sycomore's critical issue: periodic readjustment of the difficulty. Moreover, using the tools presented in Chapter 3, we propose fine-grained simulations to evaluate and compare protocols that have dynamic behavior over complex graph structures.

4.1 Background

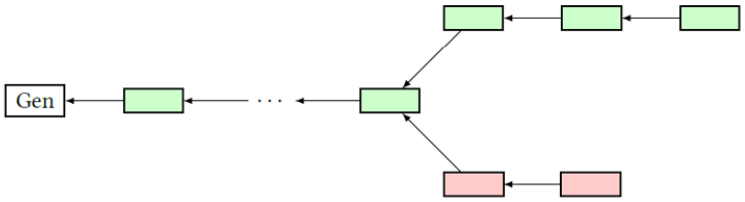
A blockchain constitutes a history that contains all the trades made between its users since its creation (i.e. since the creation of the first block, also called genesis block). This history is secure and distributed: it is shared by its various users, without intermediaries, which allows each one to 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.



(a) Classical Structure.



(b) DAG Structure.



(c) A Forked Blockchain.

Figure 4.1 – Different Blockchain Structures.

Usually, there exists two types of users, those who exchange transactions without any need to keep a copy of the chain, the "light users", and those who contribute more actively to its maintenance by creating blocks, the so-called "block creators". Each user has a "digital wallet" which contains the private key associated with the account, as well as the history of transactions made on the blockchain. Essentially, when a user broadcasts a transaction, it is received by all the other users of the network and stored in their *mempool*, a memory space where transactions awaiting validation are stored. Having the ability to create blocks, the block creators will group these transactions into blocks. Once a block has been created, it is broadcast to the network and appended at the end of the blockchain as shown by Figure

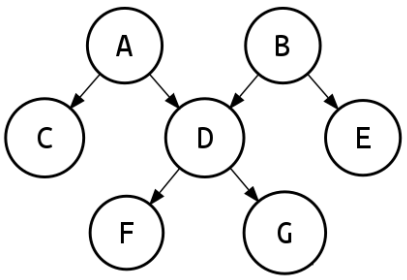


Figure 4.2 – A DAG Example.

4.1a, a mechanism that makes each transaction contained in the blockchain almost impossible to modify or delete. Consequently, the transactions contained in this block are removed from users' mempools.

Directed Acyclic Graph In graph theory, a Directed Acyclic Graph (DAG) is a directed graph that does not have a circuit. This theory, often used in networking, has been applied to the blockchain world since the appearance of [3]. As illustrated by Figure 4.2, graph nodes (A-E) can reference one parent node (e.g. E) or two (e.g. D). They can also reference no node if they are at the graph's root (e.g. A) or have no child node (e.g. G) if they are one of the graph's leaves.

As shown by Figure 4.1b, a DAG is a distributed ledger whose data structure is slightly different than the classical chain form. Instead of a single chain of blocks, we use a graph with vertices and edges. Depending on the solution, these vertices can be transactions [2] or blocks [3].

DAGs are an interesting solution often considered as a replacement to the classical chain form because of its ability to better manage data storage and transaction processing. Since in a graph, each vertex can have several parents, transactions or blocks do not need to wait for a specific parent before being processed, they can be validated simultaneously. However, what remains the same as in classical one-chain blockchains is that each new vertex must refer to the previous vertices as each block must refer to its previous block in classical structures. Also, conflicts are resolved by calculating the confirmation depth of the conflicting vertices. In some solutions, the vertices are associated to a given weight and in this case, the confirmation depth calculation takes it into account.

Fork As illustrated by Figure 4.1c, a fork is more or less a blockchain that splits into two or more chains. This phenomenon is caused by a divergence in a block. In the case of solutions such as PoW-powered blockchains [33], miners are in competition, thus it can happen that two miners find a block almost at the same time, therefore other nodes do not necessarily agree on which one to choose. However, this kind of fork is solved quite quickly when other blocks are added and one chain becomes longer than the other. This chain is then chosen as the main chain and

the blocks of the other chain are dropped. We call these forks *accidental*. However, there are also so-called *intentional* forks that could be triggered to propose modifications to the original protocol. They can also be used to correct a past security flaw as for Ethereum [44] and Ethereum Classic [92] or more recently Luna [93] and Luna Classic [94].

4.2 Overview of Sycomore

Sycomore [27] is a cryptocurrency ledger whose structure is a dedicated balanced directed acyclic graph of blocks called the SYC-DAG. Construction of the SYC-DAG is very close to Bitcoin one, i.e., it is fully distributed, permissionless, and relies on a proof-of-work mechanism. Sycomore enjoys a set of properties that enable it to dynamically adapt the fan-out of the SYC-DAG to the current number of transactions submitted to the system.

Properties of Sycomore Sycomore has been designed to meet the following properties [27]:

P1. Self-adaptation to transaction load. A rise or a drop in the current number of submitted transactions is dynamically handled by the progressive creation or disappearance of sibling leaf chains in the SYC-DAG;

P2. Balanced partitioning of transactions. There does not exist any transaction that belongs to two different blocks.

P3. Unpredictability of the predecessor. The leaf chain to which a new block is appended can neither be chosen nor predicted among all the leaf blocks of the SYC-DAG.

P4. Chain fairness. All the leaf chains of the SYC-DAG grow at the same speed.

P5. Negligible probability of forks. The probability of forks is maximal when the SYC-DAG is reduced to a single chain (i.e, $1,2 \times 10^{-3}$ in the time interval of 30 seconds) and decreases proportionally with the number of leaf blocks.

To make this chapter self-contained, we detail in this section how Sycomore implements those five properties.

Property P1 is implemented by introducing the notion of *splittable* and *mergeable* blocks [27], which are a dynamic response to respectively a rise or a drop in the current submission rate of transactions in the system. Both notions refer to block load, where the load of a block is the ratio between its number of bytes and its maximal load (for instance, 1 MByte in Bitcoin prior to the date of SegWit activation). Hence, a block b appended to the SYC-DAG is called *splittable* if the average load of block b together with the load of its $c_{\min} - 1$ predecessors on the chain exceeds the overload threshold Γ (both $c_{\min}$ and Γ are system parameters).

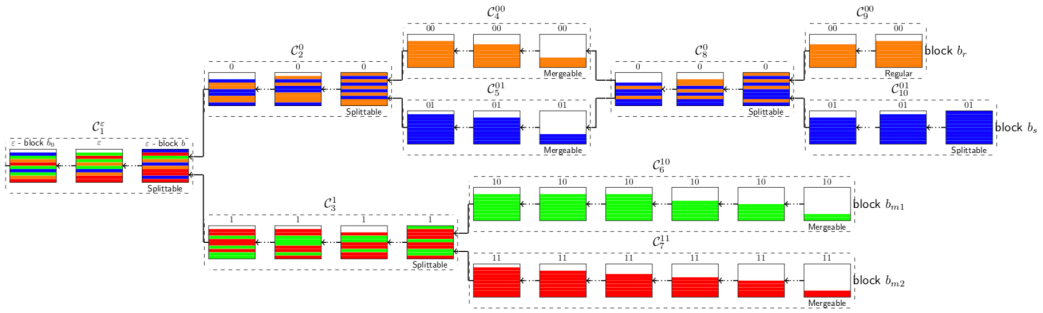


Figure 4.3 – An example of a SYC-DAG built by Sycomore. This figure has been borrowed from [27]. System parameters: overload threshold $\Gamma = 95\%$, underload threshold $\gamma = 15\%$, and $c_{\min} = 1$. The number of bars in each block is representative of block load, and colors of the bars illustrate the prefix of transaction identifiers. This provides an intuitive way to see when chains split or merge, and how transactions are partitioned over the SYC-DAG: When the SYC-DAG is made of a single chain due to a very light load (e.g., chain $\mathcal{C}_1^\varepsilon$), each block contains transactions whose identifiers are prefixed with the empty binary string label (denoted by ε), which explains the multitude of colors of the blocks (which exactly reflects Bitcoin's chain). When the chain must split into two sibling chains because of an increasing transaction load, the new appended blocks partition the transactions into two sets: those whose prefix match label ε concatenated with 0, i.e, label 0, and those whose prefix match label ε concatenated with 1, i.e, label 1. This explains the partitioning of block colors in the upper and lower chains respectively. A similar argument applies when sibling chains merge to a single chain: subsequent blocks of this chain will contain transactions whose identifier is prefixed by the largest common prefix of the labels of these sibling mergeable chains (e.g., label 0 in chain $\mathcal{C}_8^0$). Chains $\mathcal{C}_9^{00}$, $\mathcal{C}_{10}^{01}$, $\mathcal{C}_6^{10}$, and $\mathcal{C}_7^{11}$ are called leaf chains, as blocks b_r, b_s, b_{m1} and b_{m2} are the leaf blocks of the SYC-DAG. Note that this SYC-DAG does not contain any fork.

When a block b is splittable, miners will create subsequent blocks so that they will form two parallel chains of blocks, such that the first block of each of both chains references b. These chains are called sibling chains. Partitioning of the transactions over the chain blocks is explained when Property P2 is explained. Conversely, when the transaction rate drastically drops, the block load decreases accordingly, leading Sycomore to progressively reduce the number of chains in the SYC-DAG to keep blocks sufficiently loaded. Specifically, a block is called *mergeable* if the average load of this block together with the load of its $c_{\min} - 1$ successive predecessors of its chain falls short of some given underload threshold γ (γ is a system parameter). When two blocks belonging to two sibling chains are mergeable, miners will create subsequent blocks so that they will form a single chain (called merged chain). Any block that is neither mergeable nor splittable is said *regular*. As argued in [27], it is clear that everyone, and in particular miners, detect the instant at which a block is splittable or two sibling blocks are mergeable. This is observable and verifiable by

anyone since it only depends on a publicly observable quantity (i.e., block load).

Property P2 aims at fully exploiting the gain brought by sibling chains, i.e, the effective partitioning (in the mathematical sense) of the transactions over the SYC-DAG. This property is implemented by introducing the notion of *label*. A label is a binary string, and characterizes the common prefix of the identifier (i.e. fingerprint) of the set of transactions embedded in a block. Any block when created is tagged with the label of the chain it will belong to (the choice of the chain a block will belong to is explained when Property P3 is discussed). The genesis block b_0 is labelled with the empty binary string ε, and all the blocks from b_0 to the first splittable block b (if any) of the chain, say $\mathcal{C}_1^{\varepsilon}$ in Figure 4.3, are labelled with the empty string ε. Hence, all the blocks of $\mathcal{C}^{\varepsilon}$ contain transactions for which there is no constraint on the prefix of their identifier (this reflects Bitcoin's behavior). On the other hand, all the blocks of two sibling chains, say $\mathcal{C}_2^0$ and $\mathcal{C}_3^1$ in Figure 4.3, appended to the splittable block b inherit b's label extended with 0 and 1 respectively. Hence, all the blocks of $\mathcal{C}_2^0$ (resp. $\mathcal{C}_3^1$) only contain transactions whose identifier is prefixed by 0 (resp. 1). As transactions' identifiers can be considered as random bit strings, transactions are evenly partitioned over sibling chains, and transactions cannot appear in more than one block, which makes the parallelism introduced by the graph structure effective. The same process applies for any splittable block. Conversely, all the blocks that belong to a merged chain inherit the largest common prefix of its predecessor labels. For instance, in Figure 4.3, chains $\mathcal{C}_4^{00}$ and $\mathcal{C}_5^{01}$ give rise to the merged chain $\mathcal{C}_8^0$ whose label is the largest common prefix of both labels 00 and 01, i.e., 0.

Property P3 is implemented by using the unpredictability and randomness of the proof of work (PoW) to assign the *predecessor* of any block b. To make such an assignment immutable, verifiable by anyone and non-ambiguous, the header of any block b contains a set of tuples that *(i)* acknowledges or commit the miner's local view of the SYC-DAG, and *(ii)* characterizes b's predecessor. More precisely, let $\mathcal{L}_u$ be the local view of the SYC-DAG at miner u. Suppose that $\mathcal{L}_u$ contains c leaf blocks $b^{\ell_1}, \ldots, b^{\ell_c}$ at the time u starts b's creation process, and among these c leaf blocks, s of them are splittable, [1] miner u builds b's header as follows: it inserts, among different pieces of information, a set of $(c+s)$ commitment tuples

$$\{\ldots, (H(b^{\ell_j}), \ell'_j, m^{\ell'_j}), \ldots\},$$

where, for $1 \leq j \leq c$, $H(b^{\ell_j})$ is a cryptographic link to leaf block b^{ℓ_j}, ℓ'_j is the label of the block for which b^{ℓ_j} will be the predecessor, and $m^{\ell'_j}$ is the Merkle root of the set of locally pending transactions whose identifier is prefixed by ℓ'_j. If leaf block b^{ℓ_j} is splittable then two tuples $(H(b^{\ell_j}), \ell_j 0, m^{\ell_j 0})$ and $(H(b^{\ell_j}), \ell_j 1, m^{\ell_j 1})$ commit the presence of block b^{ℓ_j} in $\mathcal{L}_u$. If leaf blocks $b^{\ell_j 0}$ and $b^{\ell_j 1}$ are both mergeable and belong to sibling chains then two tuples $(H(b^{\ell_j 0}), \ell_j, m^{\ell_j})$ and $(H(b^{\ell_j 1}), \ell_j, m^{\ell_j})$ commit the presence of those mergeable blocks in $\mathcal{L}_u$. By doing this, block b extends $(c+s)$ commitment paths, one to each leaf block of $\mathcal{L}_u$, and recursively down to

[1] Note that a splittable block b_s is considered a leaf block as long as b_s is not the predecessor of two sibling blocks.

the genesis block. The length of a commitment path (that is the number of blocks on the path) is used to resolve forks if any (see Rule 5). Miner u then engages in finding a nonce ν such that ν is the solution of the PoW applied on b's header (exactly as in Bitcoin). If successful, the predecessor of block b is the leaf block b^{ℓ_i} in $\mathcal{L}_u$ closest to ν. More precisely, for each tuple $(H(b^{\ell_j}), \ell'_j, m^{\ell'_j})$ in b's header, the numerical value of the "exclusive or" (XOR) between ν and ℓ'_j is computed, and the winning tuple is the one that minimizes this distance. Let $(H(b^{\ell_i}), \ell'_i, m^{\ell'_i})$ be that tuple. The *predecessor* of block b is thus the leaf block b^{ℓ_i}. Miner u completes the creation of its block b by embedding the appropriate set of transactions, that is the set of transactions whose identifier is prefixed by ℓ'_i and whose Merkle root is $m^{\ell'_i}$. Extracting b's predecessor from the PoW computed for b makes the choice of block's predecessor an unpredictable and random process. Notice that no specific reference to b's predecessor is added in b's header: b's header is securely sealed with PoW ν, and thus when a node receives block b, it derives b's predecessor by using the information in b's header (i.e. ν and the set of tuples).

Property P4 follows from the assumption that the PoW is modeled by a random oracle, and that transaction identifiers result from the SHA256 cryptographic hash function.

Property P5 directly derives from Properties P3 and P4: since each created block is appended to a random leaf block, the probability that two blocks with the same label share the same predecessor (this is a fork situation) is equal to p/c, where p is the probability of fork in Bitcoin, and c is the current number of leaf blocks in the SYC-DAG.

Based on the above descriptions, a SYC-DAG is defined as follows.

Definition 4 (SYC-DAG [27])**.** *A graph $G = (V, E)$ is a SYC-DAG if G has a unique genesis block b_0 and there exists a partition $\mathcal{P} = \{\mathcal{C}^{\ell_1}, \ldots, \mathcal{C}^{\ell_n}\}$ of V such that $\forall i$ s.t. $1 \leq i \leq n, \mathcal{C}^{\ell_i}$ is a chain with label ℓ_i (note that several chains in $\mathcal{P}$ may be assigned the same label) and the following three properties hold:*

$$\forall \mathcal{C}^{\ell_i} \in \mathcal{P}, \forall k \text{ s.t. } 0 \leq k < |\ell_i|, \mathcal{C}^{\ell_i^{\lfloor k}} \in \mathcal{P} \tag{4.1}$$

$$\forall \mathcal{C}^{\ell_i} \text{ a merged chain } \in \mathcal{P}, \mathcal{C}^{\ell_i.0}, \mathcal{C}^{\ell_i.1} \in \mathcal{P} \tag{4.2}$$

$$\forall \mathcal{C}^{\ell_i}, \mathcal{C}^{\ell_j} \in \mathcal{P}, \ell_i = \ell_j \Rightarrow [\text{pred}(\mathcal{C}^{\ell_i}) \neq \text{pred}(\mathcal{C}^{\ell_j})] \tag{4.3}$$

Similarly to all PoW-based distributed ledgers, the distributed block creation process may lead to forks, that is the presence of at least two concurrent blocks appended to the ledger. In Sycomore two blocks are concurrent if and only if both blocks have the same label and the same block predecessor (which differs from split situations). The presence of forks gives rise to concurrent SYC-DAGs $\mathcal{L}_u$ and $\mathcal{L}'_u$ (both of them being rooted at the genesis block). To resolve forks, that is to locally keep a single SYC-DAG $\mathcal{L}^\star_u$, i.e, $\mathcal{L}^\star_u = \mathcal{L}_u$ or $\mathcal{L}^\star_u = \mathcal{L}'_u$, node u applies the fork rule described below. This rule relies on the confirmation level of a SYC-DAG. By definition, the confirmation level of a SYC-DAG is equal to the number of blocks

that belong to the longest commitment path (as defined earlier in this section) that commit the presence of the genesis block in this SYC-DAG.

Rule 5 (Fork rule [27])**.** *At any time, keep the SYC-DAG $\mathcal{L}^\star$ for which the confirmation level of the genesis block is the largest.*

As for Bitcoin, the fork rule favors the SYC-DAG that has been acknowledged by the largest proportion of miners. Note that two concurrent SYC-DAGs may temporarily have the same confirmation level. By convention, the oldest SYC-DAG is kept as long as it is not superseded. Once a block has been inserted deep enough then by construction of the blocks and by Rule 5, with high probability such a block will remain forever in the local view $\mathcal{L}_u^\star$ of any node u in the system. The notion of "deep enough" relates to the block confirmation level.

4.3 Sycomore's critical issue: difficulty readjustment

By dynamically adapting the width of the graph to the actual transaction load of the network, one might expect that Sycomore would guarantee an almost optimal transaction latency. By transaction latency, it is meant the time that elapses between the instant at which a user submits a transaction to the network and the instant at which this transaction is confirmed, i.e., belongs to a block that is deeply settled down into one of the chains of the graph. Transaction latency deeply depends on the mining difficulty, i.e. the difficulty to create a block. To cope with variations of the network computational power, the mining difficulty is periodically readjusted, to guaranteeing both security and acceptable latency. In Bitcoin, such a readjustment is executed every time the height of the blockchain has been increased by 2016 blocks, i.e. every 14 days. Sycomore has a similar readjusting scheme, to guarantee that the creation time between any two successive blocks of any given chain is constant in average, the mining difficulty D is periodically adjusted based on the current network hashrate (which is reflected by the time it took to mine the last blocks of the SYC-DAG) and the current number $c \geq 1$ of leaf blocks. Specifically, adjustment of the difficulty takes place every time the height of the SYC-DAG has been increased by h blocks with respect to the last time the difficulty was adjusted, that is, when its height h satisfies $h = 0 \bmod H_{\max}$, with $H_{\max} = 2016$. To cope with the fact that some of the leaf chains may grow a little bit slower than others, and thus leaf blocks do not reach height h at the same instant, once a leaf chain has reached height h, miners do not take this leaf chain into account to determine the predecessor of their block, i.e., they only consider all the leaf blocks whose height have not reached height h yet. Once all the leaf chains have reached height h, miners readjust the difficulty, if needed, to fit the actual network hash rate and the width of the graph, i.e., the number of leaf chains in the graph. Note that there is no incentive for an adversary not to follow this rule since its new block will be rejected by the other miners.

Unfortunately, as shown by figure 4.4, it is likely that between any two periodic readjustments of the difficulty, the width of the SYC-DAG drastically increases or

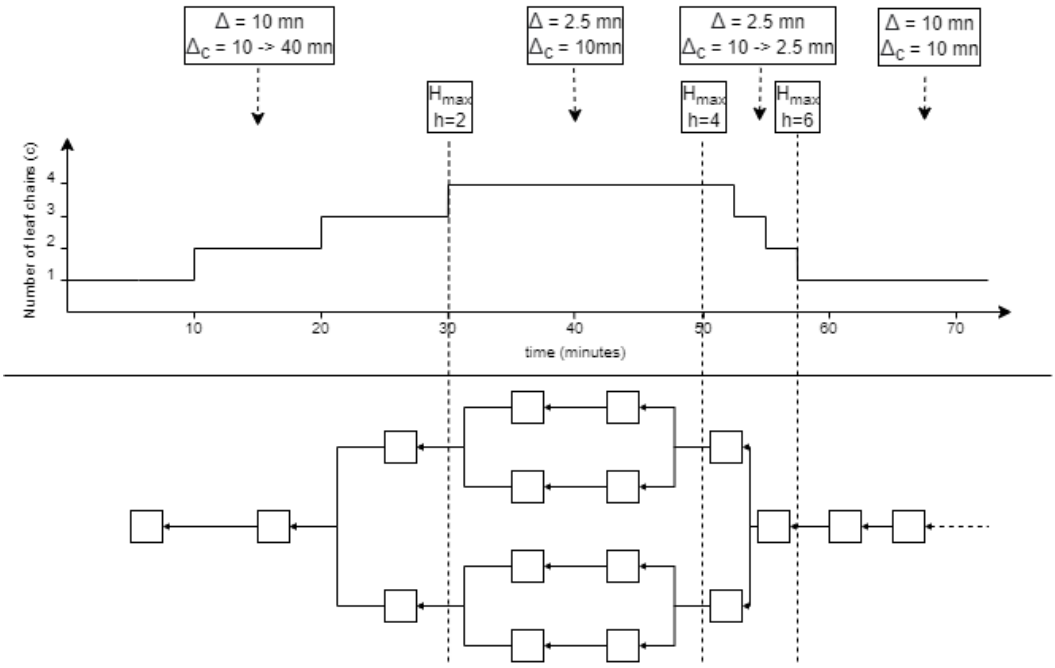


Figure 4.4 – Illustration of the issue caused by periodic readjustment of the difficulty. System parameters: overload threshold $\Gamma = 95\%$, underload threshold $\gamma = 15\%$, $c_{min} = 1$, and H_{max}=2. The figure shows the evolution of a SYC-DAG with a dynamic transaction demand. At the beginning, the graph contains one chain and the difficulty is computed so that the inter-block delay on the SYC-DAG, $\Delta = 10$ mn. A sudden increase in the transaction rate at t=10mn causes several splits. Since the difficulty is not readjusted yet, the inter-block delay on each chain Δ_c increases, and the performances are not improved by the split as they should be. However, when height $h = 2$, happens the first H_{max} (t=30mn), the difficulty is recalculated to fit the new number of branches (4). The performances are improved with $\Delta = 2{,}5$ mn. When the transaction rate decreases at t=45mn, the system starts to merge but as the difficulty is still calibrated for a 4 chains SYC-DAG, the blocks are still created as fast, and Δ_c goes from 10 to 2,5mn, which could cause consistency problems (forks). At the next H_{max} at $h = 6$ (t=57,5mn), the difficulty is re-calibrated and Δ_c returns to a normal value of 10mn.

decreases according to the transaction load demand (a cascade of splits or merges is observed when the submission transaction rate varies as observed in Section 4.5.4). This will lead to an inappropriate difficulty whose impact is twofold: from a security point of view, it may degrade the ledger quality, that is the maximal proportion of blocks contributed by the adversary that belong to the SYC-DAG of any honest node. From a progress point of view, if the difficulty becomes too high, the average creation delay between any two successive blocks will drastically increase, augmenting accordingly transaction latency, and thus transaction confirmation delay. To illustrate this point, let us consider the scenario where, at the time readjustment took place, the graph's width was very large, but subsequently the number of submitted transactions significantly dropped, leading to a progressive diminution of the number of leaf chains, and thus an under-estimated mining difficulty. Such a scenario shows a possible breach of security: adversarial miners can take advantage of a too low mining difficulty to degrade the ledger *quality* [95], that is the maximal proportion of blocks contributed by the adversary in a sufficiently long part of the ledger maintained by a honest node. The opposite scenario may also happen, where a sudden augmentation of the transaction rate will give rise to an over-estimated mining difficulty, and thus a very high power consumption, which will be in the worst case similar to Bitcoin's one. As a consequence, transaction latency will be very high, until the next readjustment of the difficult takes place, thus hindering scalability.

4.4 Sycomore^{++}: a scalable graph-based ledger

To prevent such critical issues, we propose Sycomore^{++}, which aims at guaranteeing that whatever the structure of the SYC-DAG, a constant inter-block creation delay is maintained on any of its chains. The main idea of Sycomore^{++} is to continuously adapt the block creation difficulty to the actual number of leaf chains of the SYC-DAG. Note that this adaptative adjustment does not replace the periodic global computational power readjustment. The former adapts the difficulty to the structure of the SYC-DAG while the latter adapts the difficulty to the hashing power of the system.

Lemma 6. *The expected effort miners must exert in Sycomore^{++} to successfully create a block decreases with the number of leaf blocks of the SYC-DAG.*

Proof. Let $\mathcal{U}$ be the current set of miners that participate to the construction of the SYC-DAG. We suppose that the computational power of the network is uniformly distributed among all the miners in $\mathcal{U}$. Producing a proof of work is a random process with low probability of success so that a lot of trials and errors are required on average before a valid proof of work is generated, the probability of success p_{pow} of each trial being the same. The Geometric distribution models the number of failures before the first success. Thus, if random variable X represents the number of failures before the first success, we have $P(X = q) = (1 - p_{pow})^{q-1} p_{pow}$. Let

W be the total computational power of the system. In Sycomore^{++}, the difficulty is divided by the current number c of leaf chains in the SYC-DAG (Recall that in Sycomore, this does not necessarily hold, in particular in presence of variations of the system workload demand). This comes backs to multiplying the probability of success p_{pow} of the PoW process by c. The probability p of successfully mining a block is thus given by $p = W \times p_{pow} \times c$, and at a miner, this probability is equal to $p_u = p/|\mathcal{U}|$. In Sycomore^{++}, the probability for a miner to work on a given chain is $1/c$, and the average number n_c of miners working on a chain is equal to $|\mathcal{U}|/c$. Thus the probability p_c of successfully mining a block on a given leaf chain p_c given by $p_c = p/c$. Let X_c be the random variable representing the number of trials before the first success on a given leaf chain, we have $\mathbb{E}(X_c) = 1/p_c = 1/(W \times p_{pow})$, and consequently, $\mathbb{E}(X) = 1/p = 1/(W \times p_{pow} \times c)$. □

This lemma shows that in Sycomore^{++} the expected number of unsuccessful trials before creating a block decreases with the number of leaf blocks, which is not the case in Sycomore. This demonstrates the exemplary behavior of Sycomore^{++}: Both its SYC-DAG structure and the mining difficulty self-adapt to the current number of transactions submitted to the system, which allows it to operate in adversarial environments in which the transaction load can change arbitrarily. This is confirmed by the experimental evaluation presented in Section 4.5.

The following Lemma shows that the occurrence of forks decreases exponentially with the number of leaf chains.

Lemma 7. *[27] Given a ledger $\mathcal{L}_v^\star$ with c leaf chains $\mathcal{C}_1, \ldots, \mathcal{C}_c$, each one being selected by the block creation process with probability p_i, with $\sum_{i=1}^{c} p_i = 1$, the probability that two blocks extend the very same chain $\mathcal{C}_i, 1 \leq i \leq c$ during an interval of time $[0, t]$ is $p_i(t) = 1 - e^{-\lambda t/c}(1 + \lambda t/c)$, where λ is the block creation rate.*

Proof. Let us first consider the case where $c = 1$. We model the block creation process as a Poisson process. In the following, an event represents the creation of a block. Let $\{N(t),\ t \geq 0\}$ with rate λ, be the Poisson process representing the number of events in the interval $(0, t)$. We then have, for every $n \geq 0$,

$$\mathrm{P}\{N(t) = k\} = e^{-\lambda t}\frac{(\lambda t)^n}{n!}.$$

For all $t > 0$, we denote by $p(t)$ the probability that at least two events of this process occur in an interval of length t.

$$p(t) = \mathrm{P}\{N(t) \geq 2\} = 1 - e^{-\lambda t}(1 + \lambda t).$$

Let us assume that the SYC-DAG contains $c \geq 1$ leaf blocks, $b_1^{\ell_1}, \ldots, b_c^{\ell_c}$. The probability p_i for a newly created block to have $b_i^{\ell_i}$ as predecessor depends on $b_i^{\ell_i}$'s header. The events produced by the Poisson process can be of c different types. An event of type i represents the creation of a block that matches chain $\mathcal{C}_i^{\ell_i}$. Each

event produced is of type i with probability $p_i = 1/c$, for $i = 1, \ldots, c$. The successive choices for the types are supposed to be independent of each other and also independent of the Poisson process. For every $i = 1, \ldots, c$, let $\{N_i(t),\ t \geq 0\}$ be the number of events of type i produced the Poisson process. It is well-known that $\{N_i(t),\ t \geq 0\}$ is a also Poisson process with rate $\lambda \times p_i$ and that these c Poisson processes are independent. We denote by $p_i(t)$ the probability that at least two events of type i occur in the interval $(0, t)$ (i.e. a fork occurs in the interval $(0, t)$). We then have, for every $i = 1, \ldots, c$,

$$p_i(t) = \mathrm{P}\{N_i(t) \geq 2\} = 1 - e^{-\lambda t/c}(1 + \lambda t/c).$$

It is interesting to remark that this probability holds in Sycomore only at the instants at which readjustments of the difficulty occur. □

Lemma 8. *For any correct node u, $\mathcal{L}_u^\star$ does not contain double-spending transactions*

Proof. The proof is by contradiction. Suppose that $\mathcal{L}_u^\star$ contains two transactions $T_1 = (I_1, O_1)$ and $T_2 = (I_2, O_2)$ such that T_1 and T_2 redeem a common UTXO o, where o belongs to the output set of some transaction $T \in \mathcal{L}_u^\star$. Suppose that T_1 and T_2 respectively belong to blocks b_1 and b_2. Since b_1 and b_2 belong to $\mathcal{L}_u^\star$ both blocks are valid. Two cases must be considered.

- $b_1 = b_2$. This case is impossible since it would mean that block $b_1 = b_2$ is invalid (i.e., it contains conflicting transactions T_1 and T_2).

- $b_1 \neq b_2$. Suppose without loss of generality that node u already appended b_1 to $\mathcal{L}_u^\star$ by the time it wishes to append b_2. Two sub-cases are possible.

  - b_2's header commits the existence of b_1 in $\mathcal{L}_u^\star$, that is b_2's header extends at least one commitment path that acknowledges the presence of b_1 in one of the chains of $\mathcal{L}_u^\star$. This contradicts the assumption that b_2 is valid.

  - b_1 and b_2 have been concurrently mined, that is at the time both blocks were mined their respective miners did not know the existence of the other block. By assumption, $b_1 \in \mathcal{L}_u^\star$ at the time node u wishes to append b_2. Since the presence of $b_1 \in \mathcal{L}_u^\star$ makes block b_2 invalid, node u will reject block b_2. This contradicts the assumptions that $b_2 \in \mathcal{L}_u^\star$. Note that another node v may have first appended b_2 to its ledger $\mathcal{L}_v^\star$, and thus will reject block b_1. Eventually, either $\mathcal{L}_v^\star$ or $\mathcal{L}_u^\star$ will contain the longest commitment path to the genesis block, and thus by Rule 5, the ledger with the longest commitment path to the genesis block will be kept by all the nodes. This completes the proof.

□

4.5 Simulation Study of Sycomore^{++}

This section presents the agent-based simulation study we have conducted on Bitcoin, Sycomore and Sycomore^{++}. The tools we used for this study are detailed in Chapter 3 but in order to make this chapter self-contained, we will give a brief summary of these tools before we develop the rest of our study. Note that the source codes of the protocols we study here: Bitcoin, Sycomore and Sycomore^{++} as well as all the scripts of the experiments are publicly accessible [96].

4.5.1 Simulator and Experimental Environment

We have used an agent-based simulation framework dedicated to blockchain systems, called Multi-Agent eXperimenter (MAX) [5] based on the MaDKit framework [86]. 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. Impact of message losses is left for future works. All the experiments for Sycomore^{++}, Sycomore and Bitcoin have been run on Grid'5000, a large-scale and flexible test-bed for experiment-driven research [6]. Due to the computational complexity of simulation models and experiments involving a representative number of agents, each experiment presented in this chapter takes in average 8 hours. For more details about the tools used for this evaluation, please refer to chapter 3.

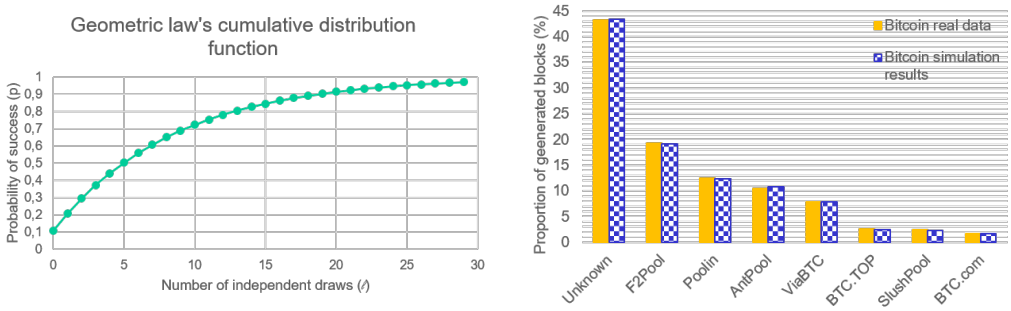
4.5.2 Simulation Model

4.5.2.1 Block creation model

Miners create blocks by following the prescribed protocols, i.e., validation of the set of transactions to be inserted and creation of block header. For straightforward reasons, miners do not solve proof-of-works but follow a simulation model. Before disseminating a block to the network, a miner waits for a time determined by the PoW model described below. Note that for both Sycomore^{++} and Sycomore, the selection of the random predecessor in the model is achieved by computing the distance between the block header (rather than the PoW nonce ν, which is not computed in the model) and each leaf block of the SYC-DAG.

4.5.2.2 Proof-of-Work Model

To simulate the effort needed to find the proof, each miner $u \in \mathcal{U}$ waits for a certain amount of ticks, which depends on its computational power W_u. Specifically, W_u is a fraction of the global computational power distributed among miners according to a power law distribution (with parameter 3) such as $\sum_{u \in \mathcal{U}} W_u = 1$. The probability for miner u to solve the proof-of-work after ℓ successive independent draws is modeled as



(a) Geometric law's cumulative ditribution function.

(b) Proportion of blocks created by each mining pool.

Figure 4.5 – Proof-of-Work Model.

a geometric distribution (figure 4.5a) with parameters ℓ and p_{POW}, where p_{POW} is the probability of successfully solving the Proof-of-Work, i.e., $p_{POW} = D/2^k$, where k the security parameter of the Proof-of-Work and D the difficulty level. Difficulty and security parameters have been set such that for $W = 1$, the time to solve the proof-of-work is 10 ticks in expectation.

Calibration of our model has been set by using Bitcoin real network statistics and by running our model with data extracted from the real network using tools presented in [97]. Figure 4.5b represents the proportion of blocks created (y-axis) for each miner (x-axis) and shows how well our model of proof-of-work (blue bars) fits Bitcoin real data (orange bars).

4.5.2.3 Common parameters of the simulations

For all the experiments presented in this chapter we have fixed some common parameters as follows:

- The block capacity, that is the maximal number of transactions a block can embed, is set to 100 transactions (to avoid the simulator overload). Note that while in Bitcoin the block capacity is approximately equal to $4,000$ transactions [53], reducing the block capacity does not affect the behaviour of the protocols.

- A transaction is confirmed when the block this transaction belongs to has a confirmation level equal to $k = 6$. Recall that the confirmation level of any block b is equal to the number of blocks that confirm the presence of b in the blockchain. Note that differently from Bitcoin, in both Sycomore and Sycomore^{++}, these blocks can belong to different chains of Sycomore, as long as these blocks form a path of commitment down to block b.

- c_{min} is set to 1. Impact of c_{min} on the structure of the SYC-DAG and its performances is left for future works.

- For each experiment, we have run sufficiently many simulations to get a confidence interval equal to $5 \pm \%$.

4.5.3 Scalability Study

This section studies the capability of Sycomore^{++}, Sycomore and Bitcoin to handle high transaction submission rates. Specifically, we evaluate the transaction confirmation rate, 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, and the average number of pending transactions at the end of the simulation (i.e., waiting to be embedded in a block). The energy lost by each protocol is also measured. The lost energy is the sum for each miner u and for each created block b not appended to the distributed ledger of the time spent working on b times the computational power cp_u. In this section, we assume that forks do not occur.

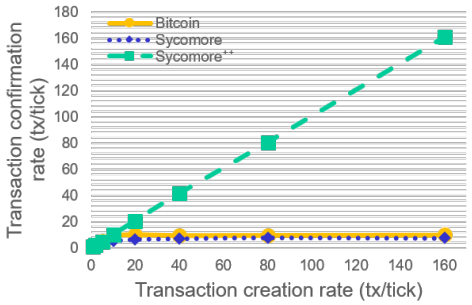
4.5.3.1 Experiment setting

The overload threshold Γ which conditions the SYC-DAG splitting in both Sycomore and Sycomore^{++} varies from 90% to 100%. Note that when $\Gamma = 100\%$, splits never occur and thus both Sycomore and Sycomore^{++} reduce to Bitcoin. The submission rate of transactions f_{req}, which represents the number of transactions submitted per tick of simulation, is set at the beginning of each experiment. f_{req} varies from 1 to 160 txs/tick. Let us remark that we tune the proof-of-work parameters to get in expectation one block mined every 10 ticks. This means that in Bitcoin $f_{\text{req}} = 10$ txs/tick already exhausts the system transaction treatment capacity, as the system mines one block every 10 ticks in expectation and one block contains 100 transactions. From this observation, we might expect that for $f_{\text{req}} > 10$ txs/tick, pending transactions will accumulate over time in, at least, Bitcoin ledger. Note that to avoid the overload of the simulator we were limited to $f_{\text{req}} = 160$ txs/tick. Anyway, setting f_{req} up to 160 txs/tick allows us to severely stress Bitcoin, Sycomore and Sycomore^{++}. Similarly to Bitcoin Core client, miners give priority to old transactions in Bitcoin, Sycomore and Sycomore^{++}.

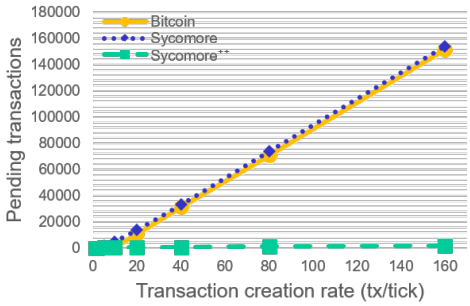
4.5.3.2 Experiment results

The main results of our experiments appear in the graphs of Figure 4.6. Note that in all the graphs, points are linked together with lines. This is only for readability reasons.

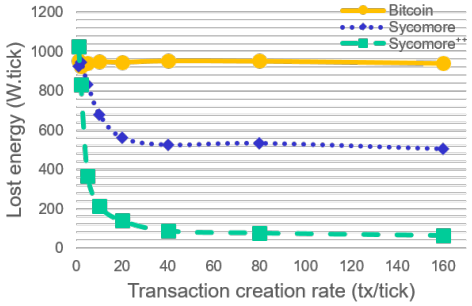
Let us first focus on the confirmation rate of transactions as a function of their creation rate (see Figure 4.6a). The main observation regarding Bitcoin and Sycomore is that whatever the computational power of the network, no more than 10 txs/tick are confirmed, which illustrates the impact of the globally constant inter-block delay (i.e. a block is mined every 10 ticks in average). Sycomore shows slightly worse results than Bitcoin, which is due to the augmentation of the number of chains, in which blocks can be moderately loaded. On the other hand, by continuously adapting the mining difficulty to the number of leaf blocks, and thus to f_{req}, Sycomore^{++} exhibits an optimal behavior regarding confirmed transactions, i.e., $\forall f_{\text{req}}$, the transaction confirmation rate equals the transaction creation rate.



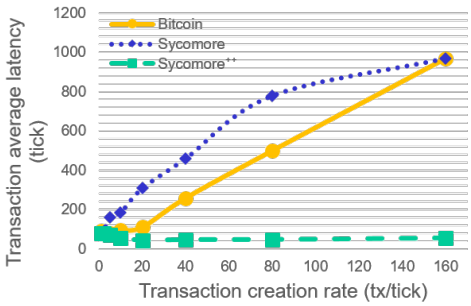
(a) Transaction confirmation rate as a function of the transaction creation rate.



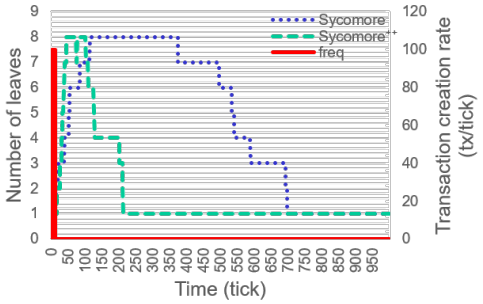
(b) Number of pending transactions as a function of the transaction creation rate.



(c) Lost energy as a function of the transaction creation rate.



(d) Transaction average latency as a function of the transaction creation rate.



(e) Reactivity of both Sycomore and Sycomore^{++} in presence of a peak of load.

Figure 4.6 – Scalability of Bitcoin, Sycomore and Sycomore^{++} (overload threshold $\Gamma = 90\%$, underload threshold $\gamma = 0\%$) and Reactivity of both Sycomore and Sycomore^{++} ($\Gamma = 90\%$, $\gamma = 10\%$).

For the sake of comparison, confirming 160 txs/tick with the simulator (and as previously said we cannot stress more the simulator) means confirming $6,400$ txs/mn in the real life.

Figure 4.6b shows the average number of pending transactions at miners, that is the average number of transactions that accumulate at miners before being embedded in blocks. It clearly shows that for both Bitcoin and Sycomore, this number linearly increases with the transaction submission rate once it exceeds 10txs/tick since this corresponds to the global inter-block creation delay. In contrast, by adapting the number of created blocks to f_{req}, Sycomore^{++} drastically reduces the average number of pending transactions. For example, for $f_{\text{req}} = 10$ txs/tick, this number is equal to 432 transactions, and for $f_{\text{req}} = 160$ txs/tick, it is equal to $1,957$ transactions, compared to $154,000$ ones in both Bitcoin and Sycomore.

Figure 4.6c illustrates the lost energy as a function of f_{req}. In Bitcoin, since the inter-block delay, the number of miners, and the difficulty do not vary during each experiment, the same amount of energy is lost regardless of f_{req}. While this setting also applies to Sycomore, the fact that the SYC-DAG becomes larger with increasing values of f_{req} gives rise to a uniform distribution of the global computational power over the leaf chains, and thus decreases miners' competition. Thus less work is wasted w.r.t Bitcoin. Regarding Sycomore^{++}, for increasing values of f_{req}, the SYC-DAG becomes larger and blocks are created faster (as the mining difficulty adapts to the SYC-DAG structure), which allows Sycomore^{++} to reach an optimal number of leaf chains more quickly than Sycomore does. As a consequence, we get a better parallelism of miners' work, and thus a drastic reduction of energy loss.

Figure 4.6d illustrates the average transaction latency as a function of f_{req}. Recall that the transaction latency measures the time elapsed between the instant at which a transaction is submitted to the network by the user and the time it becomes confirmed in the ledger. In contrast to all the other experiments, transaction latency has been measured as follows: transactions are submitted at f_{req} for a while, then f_{req} is set to 0, and simulations stop once all the submitted transactions have been confirmed. The first observation is that, in Bitcoin, once $f_{\text{req}} \geq 10$ txs/tick, transaction latency linearly increases with f_{req}, which clearly corroborates both Figures 4.6a and 4.6b. Regarding Sycomore, the loss of performance w.r.t Bitcoin is due to the fact that blocks in all the sibling chains are not necessarily fully loaded, which delays accordingly transaction latency. On the other hand, Sycomore^{++} enjoys an average constant latency, which is equal to 50 ticks regardless of f_{req}. Essentially, the more transactions are submitted, the more blocks are filled until the optimal shape of the graph is reached. Results shown in Figures 4.6a, 4.6b combined with these results clearly demonstrate the exemplary behavior of Sycomore^{++}: transactions are confirmed at the rate at which they are submitted by clients to the system, and their latency is constant whatever the submission creation rate, meaning that users can safely predict the time at which their transaction, if valid, will be deeply confirmed in Sycomore^{++}.

4.5.4 Reactivity Study

This section aims at assessing the capacity of both Sycomore and Sycomore^{++} SYC-DAG to react to sudden and abrupt fluctuations in the creation transaction rate. [2]

4.5.4.1 Experiments setting

As briefly presented in Section 4.2, when f_{req} shrinks, the SYC-DAG reacts by progressively decreasing the under loaded sibling chains, and thus the number of created blocks. Thus each merge divides by almost two the number of blocks that will be subsequently created. By the randomness of transaction identifiers, if one chain becomes under loaded, then soon after all, the chains will become under loaded too, and thus merges will occur in cascade. Initially, $f_{\text{req}} = 100$ txs/tick during 10 ticks to mimic a transaction peak load, and then at tick $t = 12$, $f_{req} = 0$ txs/tick.

4.5.4.2 Experiments results

Figure 4.6e illustrates the reactivity of both Sycomore and Sycomore^{++} in presence of a load peak (illustrated by the red constant function from $t = 1$ to $t = 11$ ticks at $f_{\text{req}} = 100$txs/tick). Both Sycomore and Sycomore^{++} initially undergo a series of splits, and then progressively move on to a series of merge up to converging to a single chain of blocks. Sycomore^{++} differs from Sycomore in its rapidity to split and merge: Sycomore^{++} succeeds in coping with the load pick 75% faster than Sycomore does, and 33% faster than Sycomore to cope with the sudden shrink of load. It is worthwhile to observe that those results combined with the one observed in Section 4.5.3, assess the capability of both Sycomore and Sycomore^{++} to meet Properties P1 and P4.

4.5.5 Adversarial environment

This section measures the impact of high transmission delays on the number of forks and the time it takes for Bitcoin, Sycomore and Sycomore^{++} to resolve them. This section supposes that all miners are honest and thus do not design adversarial strategies to create forks (adversarial behaviors are studied in Section 4.5.6). We suppose that simultaneous events are not possible. Thus if transmission delays are null, fork can never occur (once a miner receives a block b that would be appended to the same leaf block as its own currently created block b', it does not broadcast b'). When transmission delays increase, miners will broadcast their blocks before detecting the presence of concurrent ones, giving rise to forks. Let Δ be the constant transmission delay on the network. Let t_b and $t_{b'}$ be the instant at which the two concurrent blocks b and b' are respectively broadcast. Forks can occur only if Δ is greater than the time elapsed between t_b and $t_{b'}$.

[2] We omit Bitcoin from this evaluation since Bitcoin chain does not adapt to transaction demand.

	$\Delta = 0$		$\Delta = 0.1$		$\Delta = 1$		$\Delta = 5$	
	f	t_r	f	t_r	f	t_r	f	t_r
Bitcoin	0	0	0.5	1.4	0.9	10.8	2.3	36.6
Sycomore	0	0	0	0	0	0	0.6	11.7
Sycomore^{++}	0	0	0	0	0	0	1.1	6.1

Table 4.1 – Average number f of forks and average time to resolve one fork (t_r) as a function of the network delay (ticks).

4.5.5.1 Experiment settings

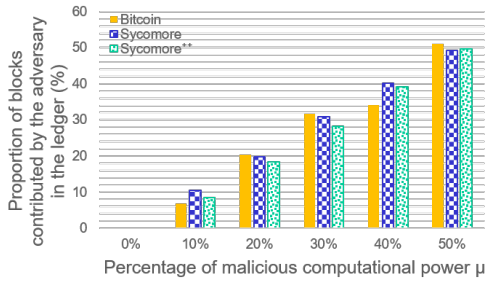
We vary the overload threshold Γ from 90% to 100% and set the underload threshold γ to 0%, so that merge do not happen (for $\Gamma = 100\%$, splits never happen and thus the SYC-DAG reduces to Bitcoin's chain). f_{req} is set to 160txs/tick to provoke splits. Δ ranges from 0 (no fork) to 5 ticks. It is important to observe that $\Delta = 5$ ticks is very large compared to the average time needed to create a block (i.e., 10 ticks). The reason is that we want to stress the system under constant and very high submission rates to provoke splits, and large transmission delays to study their impact on the occurrence and resolution of forks.

4.5.5.2 Experiment results

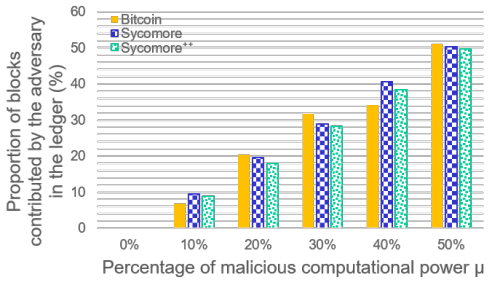
The main results drawn from our experiments appear in Table 4.1, which shows the impact of Δ on the the number of forks f and their resolution time t_r (in ticks). Forks are alternative stories, that is having n forks in a simulation means having $n + 1$ alternative ledgers. The fork resolution time t_r is equal to the time elapsed between the creation of an alternative chain (Bitcoin) or SYC-DAG (Sycomore and Sycomore^{++}) and the instant at which a ledger has the best confirmation level of the genesis block w.r.t the others ledgers (see Rule 1, Section 4.2). As can be observed, the number of forks f increases with Δ. As Sycomore and Sycomore^{++} differ in their capacity to continuously adjust the difficulty to the actual number of leaf blocks, their tolerance to fork occurrence is different: decreasing (resp. increasing) the mining difficulty reduces (resp. enlarges) the standard deviation between any two blocks b and b' creation times, and therefore impacts the probability with which $\Delta > |t_b - t_{b'}|$ holds or not. On the other hand, as blocks are created faster, fork resolution takes less time in Sycomore^{++} than in Sycomore. Hence, if sellers adopt the same rule as in Bitcoin to wait for a given period of time T before sending their goods to buyers, we clearly see that both Sycomore and Sycomore++ drastically reduce T (i.e., $T/3$ for Sycomore and $T/6$ for Sycomore^{++} w.r.t Bitcoin for $\Delta = 5$ ticks).

4.5.6 Adversarial strategies

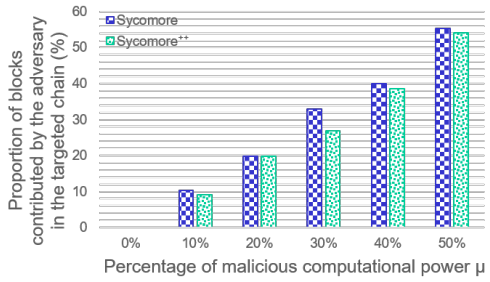
This section studies the resilience of Bitcoin, Sycomore and Sycomore^{++} in presence of adversarial strategies that could hinder the chain quality property as defined



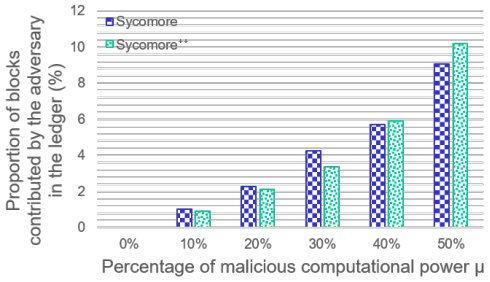
(a) Impact of a ledger attack on the ledger quality. $H_{\max} = \infty$.



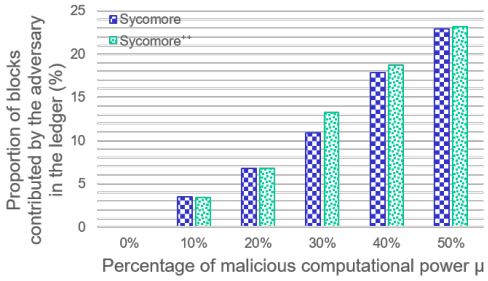
(b) Impact of a ledger attack on the ledger quality. $H_{\max} = 2$.



(c) Impact of a chain attack on the targeted chain quality. $H_{\max} = \infty$.



(d) Impact of a chain attack on the ledger quality. $H_{\max} = \infty$.



(e) Impact of a chain attack on the ledger quality. $H_{\max} = 2$

Figure 4.7 – Impact of the Ledger and Chain Attacks.

in [95]. The chain quality property states that, in Bitcoin, the adversary may control at most a $\mu/(1-\mu)$ percentage of the blocks in the chain, where μ represents the ratio of the network hashing power owned by the adversary. As briefly explained in Sections 4.2 and 4.4, both Sycomore and Sycomore^{++} aim at guaranteeing that miners can neither foresee nor choose the leaf chain to which their block will be appended prior to having irremediably completed the construction of their block's header (Property P3). The reason is to prevent an adversary from devoting all its computational power to the growing of a specific chain. This means that the only way for the adversary to target a specific chain is to repeatedly generate blocks until a valid header for the targeted chain is produced. This process is computationally

intensive and thus, the objective of these experiments is to determine how much mining power the adversary should exert to have an effective impact on targeted chains. This has led us to design and implement the following two attacks. In the first one, called *ledger attack*, the attacker tries to undermine the whole ledger quality, i.e., tries to maximize the proportion of blocks it contributed in Bitcoin chain, and in each chain of the SYC-DAG in both Sycomore and Sycomore^{++}. In the second one, called *chain attack*, the adversary targets a specific chain of the SYC-DAG and tries to maximize the proportion of blocks it contributed within this chain. [3] This requires for the adversary to only keep the blocks they have mined that match the targeted leaf chain. Note that the honest miners feed all leaf chains equally, including the one targeted by the adversary. The impact of the readjustment period $H_{\max}$ is noticeable in the chain attack. Indeed, as discussed in Section 4.2, at each readjustment period, i.e., each time the length of the ledger has been increased by $H_{\max}$ w.r.t to the previous re-adjustment, (*i*) the mining difficulty is recomputed to adapt to the actual computational power of the system and (*ii*) late leaf chains catch up to $H_{\max}$ if necessary. In the latter case this means that if the chain targeted by the adversary is among the first ones to reach $H_{\max}$, then the adversary (and the honest miners) will not be able to contribute blocks on the targeted chain (any block appended to these fast leaf chains will be ignored by the honest miners as long as the other chains have not caught up). So to increase the speed at which all the leaf blocks caught up, the adversary contributes blocks on these late leaf blocks, so that it will be able to contribute on the targeted chain quicker. It is important to note that all the blocks contributed by the adversary are all valid otherwise they would not appear in the ledger. However, they possibly favour particular transactions submitted by the adversary, or in contrast do not contain transactions the adversary wishes to exclude from the ledger.

Experiments setting In addition to the parameters cited in Section 4.5, both attacks share the same experiment settings. We set $f_{\text{req}} = 40$ txs/tick to provoke splits. The proportion of computational power μ owned by the adversary ranges in the interval $[0\% - 50\%]$. Operationally, we divide miners into two groups, the honest and malicious groups, and allocate each group with a proportion of the total computational power W, i.e., $W(1-\mu)$ for the honest group and $W\mu$ for the malicious one. This computational power is then decentralised, i.e., divided among miners of each partition (see Section 4.5.2 for more details on this aspect). Experiments are run with $H_{max} = \infty$ and $H_{max} = 2$.

Ledger attack Figures 4.7a and 4.7b illustrate the impact of the ledger attack in Bitcoin, Sycomore and Sycomore^{++} distributed ledgers. The main observation drawn from both figures is the fact that the ledger quality property [95] holds in Bitcoin, Sycomore and Sycomore^{++}: the adversary cannot control more than $\mu/(1-\mu)$ percent of the blocks in the ledgers. Furthermore the impact of $H_{\max}$ value is

[3] Note that the chain attack does not make sense in Bitcoin.

negligible in both Sycomore and Sycomore^{++} since the adversary has no better strategy than appending the maximum number of blocks on each chain of the SYC-DAG.

Chain attack Figures 4.7c, 4.7d and 4.7e illustrate the impact of the chain attack on the quality of the targeted chain (Figure 4.7c) and on the quality of the ledger (Figures 4.7d and 4.7e). We have implemented the chain attack as follows: the malicious group of miners focuses on the leaf chain with the lowest label (this could be any existing leaf label) and only appends blocks to it, which requires for the adversarial group to discard all the blocks they have mined that do not match the lowest label. The main observation drawn from Figure 4.7c is the fact that in both Sycomore and Sycomore^{++}, the chain quality holds: the adversary cannot control more than $\mu/(1-\mu)$ percent of the blocks in the targeted chain. This figure also illustrates that in both Sycomore and Sycomore^{++}, the computational power is equally distributed on the SYC-DAG leaf chains.

Figures 4.7d and 4.7e show the very low impact of the chain attack on both Sycomore and Sycomore^{++} quality. For instance, if the adversary has 50% of the hashing power, then it will control no more than 10% of the blocks in the honest players's ledger. It is interesting to see the impact of $H_{\max}$ value on the ledger quality: when $H_{\max} = \infty$, the adversary continuously tries to append its contributed blocks to its targeted chain at the expense of throwing away all its blocks that do not fit this chain. On the other hand, when $H_{\max} = 2$, the adversary must periodically feed the other chains when its targeted chain has been increased by 2 before all the other ones (actually this is often the case since both the adversary and the honest miners contribute to this targeted chain). As a consequence, the percentage of blocks contributed by the adversary in the ledger augments in both Sycomore and Sycomore^{++}, while completely satisfying the ledger quality property [95].

4.6 Conclusion

In this chapter, we analysed Sycomore, a graph-based blockchain whose structure self-adapts to fluctuations in transaction submission rates. Unfortunately, between two difficulty readjustment's periods, we have seen that Sycomore had performance and even security concerns. This chapter presents a twofold contribution.

First, we prove that, in Sycomore, between two readjustments of difficulty, the mining difficulty which was computed to fit both the hash rate of the network and number of chains at the last readjustment may currently be either under-estimated or over-estimated, which could cause performance and security concerns.

Given that, we present Sycomore^{++}, a truly scalable proof-of-work based protocol that solves the critical issues mentioned above. Sycomore^{++} inherits the main mechanisms of Sycomore, while adding a new mechanism to adjust difficulty in such a way that at any time a constant inter-block creation delay is maintained on any leaf chains of the graph.

Lastly, we propose fine-grained simulations to evaluate and compare protocols that have dynamic behavior over complex graph structures. More in detail, to finely validate and compare the behavior of Sycomore^{++} with respect to its direct competitors, we have implemented Bitcoin, Sycomore and Sycomore^{++} on an agent-based simulator and have compared these three protocols in presence of adversarial environments, i.e., sophisticated attacks, large communication delays, and sudden and substantial variations of the system workload demand. Lessons learnt from these experiments show that Sycomore^{++} succeeds in providing (*i*) a transaction confirmation rate varying linearly with the transaction submission rate, (*ii*) a very small and almost optimal average transaction latency regardless of the submission transaction rate, (*iii*) a negligible number of transactions pending at miners prior to be embedded in blocks, a drastic reduction of the computational power used to create blocks that will never appear in the ledger w.r.t Bitcoin and Sycomore. Moreover, we have designed sophisticated attacks that an adversary may elaborate to hinder Sycomore^{++}'s quality property [95]. Garay et al. [95] define the ledger quality as the property that ensures that an adversary cannot control more than a $\mu/(1-\mu)$ percentage of the blocks in a sufficiently long part of the ledger, where μ represents the ratio of the network hashing power owned by the adversary.

In this chapter, we have seen that graph-based blockchains are an interesting solution to improve the scalability of blockchains with an interesting concept of sharding the transactions into subsets to be able to process them in parallel. Thus, in the next chapter, we will look at other types of sharding and their application to the blockchain.

Chapter 5

State-sharded blockchains

"Divide Et Impera" ("Divide and conquer")

– Philip II of Macedon

Contents

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.

Recent academic works have addressed this issue by adopting sharding techniques [29, 13, 12, 16, 30, 31, 14]. In these systems a decentralized mechanism assure then the shard formation, i.e., the assignment of validators and transactions to shards. However, because the transaction submission load can vary over time, shards might need to be re-organized at run-time.

Up to now, sharding solutions in permissionless settings have mainly focused on cryptocurrencies or special classes of smart contracts managing payment transactions [13, 16, 30, 15]. Payment transactions need a weak form of atomicity called eventual atomicity [15]. 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 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 chapter, 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 chapter is organized as follows: Section 5.1 presents basic concepts and

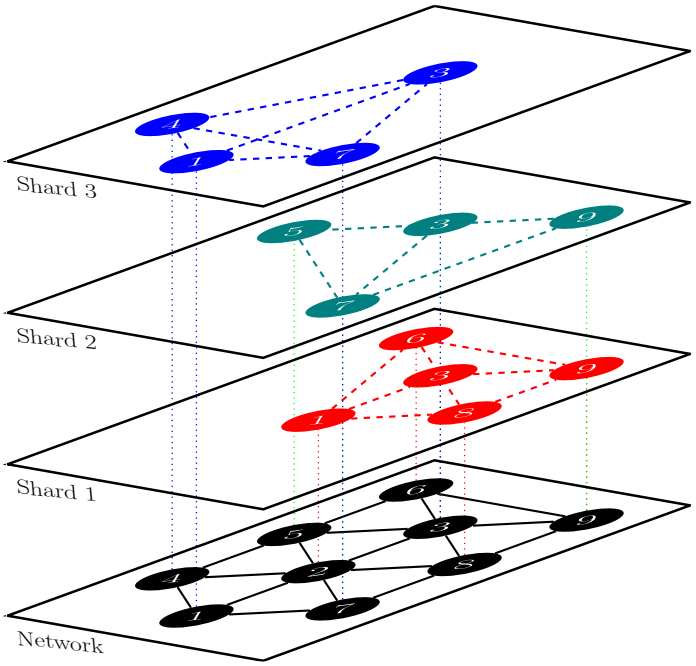


Figure 5.1 – An Example of Network Sharding. Each node belongs to a different shard even though they communicate with the same peer-to-peer network: node 9 belongs to shards 1 and 2, node 7 belongs to shards 2 and 3; and node 3 belongs to all shards.

definitions, Section 5.2 the main building blocks and assumptions Yggdrasil relies on, while Section 5.3 presents Yggdrasil and Section 5.6 an extensive performance evaluation. Section 5.7 concludes the chapter.

5.1 Background

5.1.1 The Many Faces of Sharding

In the ecosystem, sharding (as explained in Section 2.6) exists along three dimensions: network, transaction and state sharding [11].

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.

Let us note that it is not excluded that a node could participate in more than one shard.

Transaction sharding manages the way transactions are assigned to the shards aiming at achieving parallel confirmation of transactions in multiple shards. Transaction sharding aims at both increasing confirmation throughput and reducing la-

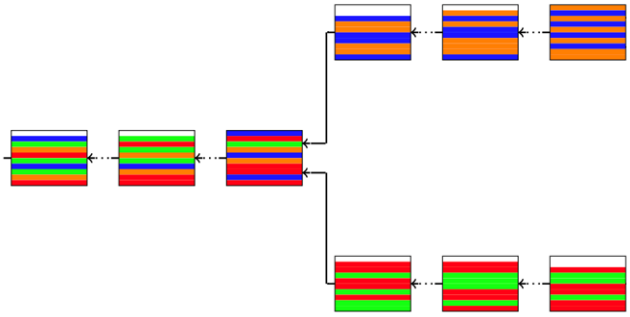


Figure 5.2 – An Example of Transaction Sharding. The colors of the bars in each block illustrate the transaction partition. This provides an intuitive way to see how transactions are partitioned over a DAG is Sycomore [27]: When the DAG is made of a single chain, each block contains transactions of all 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.

tency. 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.

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.

Let us note that transaction sharding distribute transactions among nodes, but this does not imply that the state is sharded as well. State sharding aims at replicating in each shard only the state necessary to validate the given shard of transactions.

5.1.2 Smart Contracts

Popularized by Ethereum, many blockchains today (e.g., [46], [98]) 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

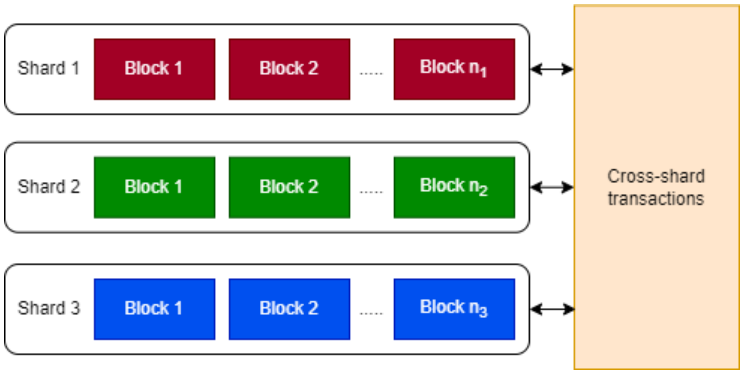


Figure 5.3 – An Example of State Sharding. Each shard keeps its own state and confirms its own transactions. However, when transactions involve more than one shard, cross-shard communication is necessary.

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 [99]. 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.

5.1.2.1 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 [100], 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 [15]. More formally, *eventual atomicity of confirmation* guarantees that for each transaction between a user and a front-end smart contract, if one shard confirms the transaction then other shards will eventually confirm it.

Besides users, smart contracts themselves can call other smart contracts. The case of a smart contract calling smart contracts belonging to the same shard can be treated as in a non-sharded system, or, if the user invoking the smart contract is in another shard, by employing mechanisms to guarantee eventual atomicity as explained above. On the other hand, invocations crossing shards cannot be treated as internal invocations, like in the non-sharded case, but must be represented as cross-shard transactions. Then, we need to guarantee the *atomic commit of the distributed execution* of the front-end smart contract across shards, i.e., either cross-chain transactions in the call graph originated from a given user invocation are all committed or they are all aborted[1] [101]. 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 [102], 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 Chapter 5, our solution, Yggdrasil combines a 2PC protocol with a cross-chain confirmation mechanism to assure *atomic commit* of the distributed execution of smart contracts and *eventual atomicity* of confirmation. Moreover, adaptivity of Yggdrassil allows to dynamically adapt shards to reduce the

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

overload generated by these protocols.

5.2 System Model

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

Adversarial model. We suppose that at any time some processes can fail in any arbitrary manner. These processes are indifferently called *faulty* or *Byzantine* processes. Byzantine processes can "pollute" the computation (e.g., by sending messages with different contents, when they should have sent messages with the same content if they were not faulty). Processes that always follow the protocol are called *honest*. We model the behavior of faulty processes as a weakly adaptive adversary. We characterize the power of the adversary as follows [103]. 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 5.3.8).

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

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

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 [104] that provides deterministic finality [40]. Specifically, we assume that each blockchain is grounded on Tendermint [17], 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 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 ([22, 45, 47, 34]), 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 [46, 45] or Ethereum PoS [34]. 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 [105]. 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 [106], 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 [107]. 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

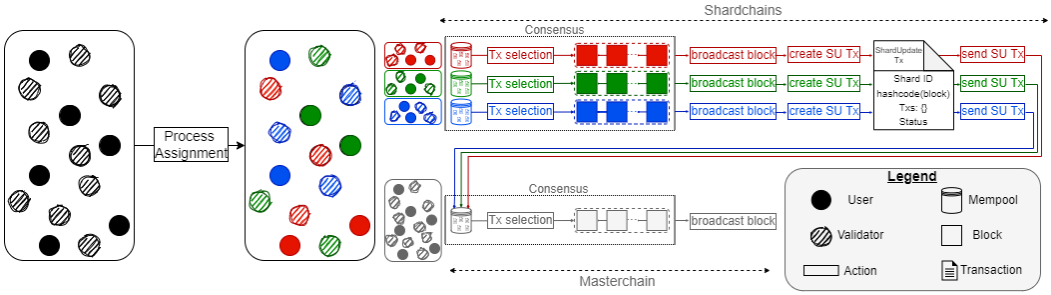


Figure 5.4 – A Simple Overview of Yggdrasil.

by Yggdrasil, messages sent by a shard are never lost and are received by all honest processes.

5.3 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 5.2), 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 5.1.2) 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 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.

5.3.1 Transaction Life-Cycle through Sharding

An Yggdrasil's process with the role of user can transfer coins to another user, deploy smart-contracts, invoke smart contract methods, or deposit coins to become a validator as realized in common PoS-based blockchains. For each of these actions different user transactions are submitted to Yggdrasil, i.e., *payment transactions*, *smart contract deployment transactions*, *smart contract method invocation call transactions*, and *stake transactions*[3], respectively. Yggdrasil manages all these transactions in a unified way as described below.

Transactions and state sharding. As will be detailed in Section 5.3.4, 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 [108], 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 5.2. 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 9 (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 5.4). Transactions are collected by the shard's validators[4], and locally stored in

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

[4]Users can also store blocks and transactions if they want to but since they are not responsible

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 5.4, 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 5.4.3.

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

In the case of cross-shard transactions, the issuer and the recipient entities of the transaction are assigned to two different shards [5]. As mentioned in Section 5.1.2, to avoid inconsistent situations or double spending, it is sufficient to guarantee the eventual atomicity of cross-shard transactions confirmation. This is because (i) the check of the issuer balance, which is done in the issuer's shard, is the only condition to confirm or reject a transaction and (ii) shard's behavior, as a whole, is honest. Yggrdrasil ensures that if the issuer is honest then her transaction is eventually confirmed. For both payment and smart contract invocations, cross-shard transactions are managed by relying on the masterchain. The different steps involved to confirm a cross-shard transaction tx_1 from shard s_1 to shard s_2 are illustrated by Figure 5.5 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

of building blocks, this is not mandatory.

[5]For a payment transaction, the two involved entities are user's accounts. For smart contracts invocations, the two entities are a user account and a smart contract account. Of course, smart contracts can call in their turn smart contracts in another shards. Nested calls generate cross-shard transactions that are managed by the 2PC protocol presented in Section 5.3.2

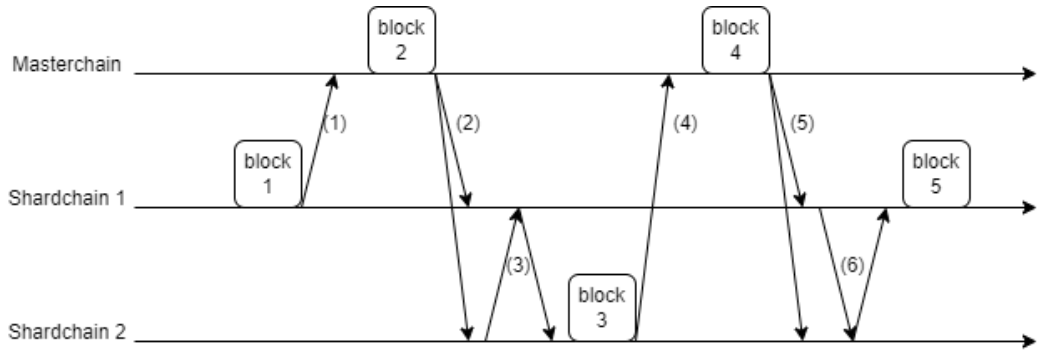


Figure 5.5 – The different steps involved to confirm a cross-shard transaction.

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 5.4.4.

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

Definition 10 (Masterchain transactions confirmation)**.** *Any stake and shard update transactions is confirmed when it appears in a block of the masterchain.*

Definition 11 (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 12 (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.*

5.3.2 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 $tx_{0,i}^{query}$ 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 $tx_{i,0}^{vote}$. 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_0^{unlock}, while the decision is a cross-shard transaction $tx_{i,0}^{decision}$ for each shard s_i. At receiver side, all the shards commit or roll-back accordingly with the decision. Roll-back is implicit, the validator does nothing in this case. In case of commit, the computation must be redone by the new proposer (the proposer might have changed since the last pre-execution). Since the state

of the smart contract did not change from the last prepare-to-commit because of the lock, the result is the same as in the pre-execution phase (smart contracts are deterministic). After the execution, an unlock intra-shard transaction is inserted in the block tx_i^{unlock}. Note that the whole process is recursive to explore the whole call graph. In case of loops in the call graph, to avoid deadlocks a locked smart contract can accept incoming calls when originating by the same root of the call graph that caused the smart contract to be locked (line 5). Let us stress that the call graph is distributed among shards. To cope with that, call paths, which are added at each outgoing invocation in the call graph, allow to trace back the path till the root and find if there is a common root.

As mentioned 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.

5.3.3 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 13. *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 $tx_{0,i}^{decision}$; (ii) $sc_i \neq sc_0$ and delivered a transaction DECISION $tx_{j,i}^{decision}$ 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 $tx_{0,i}^{decision}$ 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. □

Lemma 14. *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

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

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

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

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

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

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

sc_i votes for PREPARE and issues the transaction $tx^{vote}_{j,z}$ 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 15. *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 14. Let us proceed by construction. Let sc_i be the smart contract that votes ABORT. sc_i issues $tx^{vote}_{i,j}$ 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 $tx^{vote}_{j,z}$ 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 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, otherwhise all $sc_j \in \mathcal{G}$ decides for COMMIT.*

Proof. The proof follows from Lemmas 13, 14 and 15. □

Lemma 16. *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.*

Algorithm 4 *insertDecisionTxs(decision, voteTxs)*

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

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 $tx^{query}_{0,child}$ 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 $tx^{query}_{child,child's\ child}$ 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 17. *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 13, having that if sc_0 issues a decision, i.e. a transaction decision $tx^{decision}_{0,j}$, 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 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 16). 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 17 all the smart contracts in $\mathcal{G}$ deliver the decision, apply it and unlock. This concludes the proof. □

5.3.4 Process-to-shard assignment

Shards are uniquely identified by their label l (the computation of shards' label is described in Section 5.3.5). At any time, any process is assigned to the (unique) shard whose label minimizes the distance with the process's identifier.

Definition 18 (Distance function)**.** *[27] 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 19 (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 5.1.*

$$S_\ell = \arg\min_{S \in \mathcal{S}} D(id_i, S) \tag{5.1}$$

By construction of the shard labels mechanism (see Section 5.3.5) 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 5.4.2.

5.3.5 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 20 (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 $\ell = \varepsilon$. 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.

5.3.6 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.

$$SU = (\ell, h(b), \mathcal{T}, \theta), \tag{5.2}$$

where ℓ is the label of shard s, $h(b)$ is the cryptographic hash of the latest block b created in s, $\mathcal{T}$ represents the set of cross-shard transactions contained in b that

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

$$\mathcal{T} = \left\{ (\ell', m'), \ldots, (\ell^j, m^j), \ldots, (\ell^{(r)}, m^{(r)}) \right\} \tag{5.3}$$

5.3.7 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 [6]. Based on its node's public key, the default user incarnation would be identified by $id_{node} = 1001$, therefore, would be assigned to shard 10. However, if the node intends to repeatedly interact with another user or with a smart contract located in shard 0, Yggdrasil enables it to incarnate in s_0, with the identifier $id_{incarnation} = 0.id_{node} = 01001$. Operationally, to create an incarnation, initial funds must be deposited into its account by sending a cross-shard transaction tx_1: $< id_{node}, id_{incarnation}, \_>$. If the node wants to withdraw its funds from its incarnation it must send a transaction tx_2: $<id_{incarnation}, id_{node}, \_>$.

5.3.8 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 [7]. 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

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

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

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 18) between their identifier id_v and id_v^h $D(id_v^h, id_v)$. The validator is allowed to move if its $D(id_v^h, id_v)$ is below a threshold such that the probability for the validator to be shuffled is equal to $1/k$ (line 3 of Algorithm 5). Then, the validators calculate $D(id_v^h, \mathcal{S})$ and assign the validator to the closest shard to id_v^h (line 4 of Algorithm 5). Note that the distance function could return the same shard the validator was in for the last height, thus, it would not move. Hence, the probability of of having a different shard than the former shard the validator was in would be $1 - \frac{1}{|\mathcal{S}|}$. In this way, for each masterchain block, we have a probability $\approx 1/k$ for a validator to be re-assigned and each process in the system could compute its assignment. Yggdrasil's properties and proofs are presented in Section 5.5.

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 18. */

5.4 Implementation Details

5.4.1 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.

5.4.2 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.

5.4.3 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 $\mathcal{C}^\ell(h)$. 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

Algorithm 6 General Routine

```
1:  h_i^m := 0
2:  h_i^ℓ := 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^ℓ) := 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 : 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 ← getClosestShard(p_i.identifier, S)
                  /* getClosestShard() returns the closest shard between p_i label and the
                  shard labels in S using the distance function defined in Definition 18. */
```

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^ℓ
   $\mathcal{S} := \{\varepsilon\}$
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:

5.4.4 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 5.6 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 18) to the one of the sending/receiving shard.

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

```
1:  Initialization : S := {ε}
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 ∈ block)
6:    if (CrossTx.label == p_i.label) then
7:      send <GETBLOCK, CrossTx.label, getBlockHashOf(CrossTx)> to all processes be-
    longing 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 ∈ 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:
```

5.5 Yggdrasil Analysis

5.5.1 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 9, 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

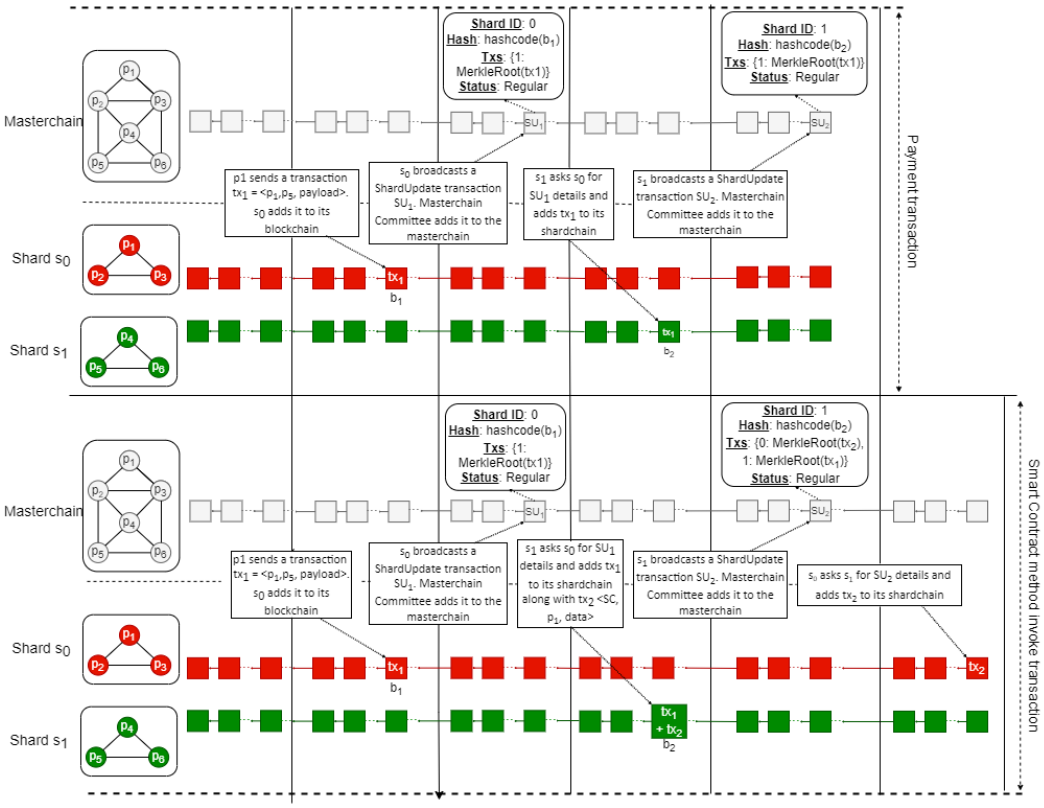


Figure 5.6 – Handling cross-shard Transactions. Yggdrasil relies on the masterchain to handle the confirmation of cross-shard transactions. When the cross-shard transaction tx_1 is broadcast, *(i)* tx_1 is confirmed by the sending shard's committee, *(ii)* a ShardUpdateTx containing the merkle root of 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.

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 $\mathsf{state}' \sqsubseteq \mathsf{state}$, if either state' is the initial state of s, or state' is obtained after applying blocks $b_1^s, \ldots, b_{k'}^s$ to the initial state of s, with $k' \in \{1, \ldots, k\}$.

Definition 21 (State-sharding)**.**

- *The state of a node is a prefix of the state of the shard he is member of:*
  $\forall p_i \in s_l, \mathsf{state}_i \sqsubseteq \mathsf{state}_{s_l}$*, 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* $\mathsf{state}_1 \sqsubseteq \mathsf{state}_2$ *or* $\mathsf{state}_2 \sqsubseteq \mathsf{state}_1$*.*

- *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* $\mathsf{state}_1 \cap \mathsf{state}_2 \sqsubseteq \mathsf{state}_{\text{SharedKnowledge}}$. *When there is no structured shared information between shards,* $\mathsf{state}_{\text{SharedKnowledge}}$ *is always empty.*

Lemma 22. *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 5.1.1 for more details).

To prove that Yggdrasil implements state-sharding, we prove here each point of definition 21:

- 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.

□

5.5.2 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 23. *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 24. *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_i^m (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 5.3.8, 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 25. *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 19. It uses the distance function (see Definition 18) 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 [109], 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.

□

5.5.3 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 26. *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 5.6.3). 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 27. *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 12).

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 26, 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 26, it will be put in the masterchain, which confirms b_1, and by extension tx_1. As defined in equation 5.2, 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 12).

□

5.5.4 Security

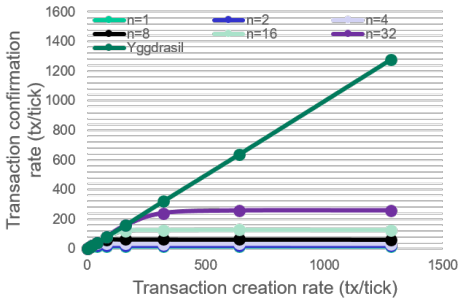
The security properties concern the guarantee Yggdrasil provide against adversaries. Concretely, we have that in Yggdrasil, (i) *Validators (re-)assignment is unpredictable in advance (before a new block is appended to the masterchain)*, (ii) *Validators are dynamically re-assigned*, (iii) *No validator has control on how it is (re-)assigned*. 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.

5.6 Performance Evaluation

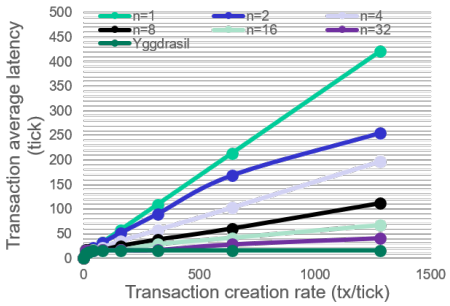
We evaluated the performances of Yggdrasil against the following properties: (i) scalability, i.e. capacity to scale during a peak of transaction load in terms of *block/transaction throughput and latency*, (ii) *reactivity in terms of number of shards in the system*, against a sudden and abrupt transaction 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 [110] 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 [111].

5.6.1 Simulator and experimental environment

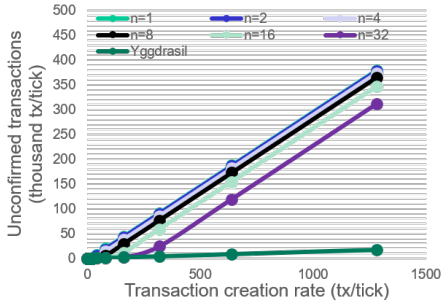
We used an agent-based simulation framework dedicated to blockchain systems, called Multi-Agent eXperimenter (MAX) [5] based on the MaDKit framework [86]. MAX offers generic libraries to easily develop distributed ledger protocols and a large range of simulation scenarios. The simulator is a discrete 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 5.2 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



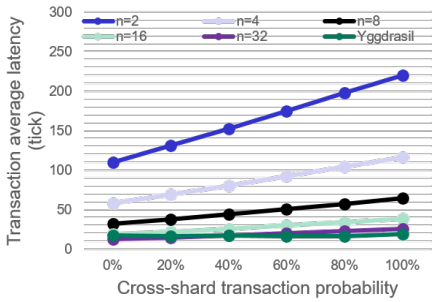
(a) Transaction confirmation rate as a function of the transaction submission rate.



(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.

Figure 5.7 – Performance Evaluation of Yggdrasil.

research [6]. Due to the computational complexity of simulation models and experiments involving a representative number of agents, each experiment presented in this chapter takes in average 24 hours.

5.6.2 Simulation model

The Yggdrasil protocol has been implemented in the simulator on top of an implementation [112] of the Tendermint protocol[17], 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 [110]. For reactivity and the impact of 2PC algorithm, synthetic workloads have been generated.

For all the experiments presented in this chapter, 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 [53], 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 $5 \pm \%$. For each experiment,

5.6.3 Scalability

This section studies the capability of Yggdrasil to handle high transaction submission rates. Specifically, we evaluate the transaction confirmation rate, the number of unconfirmed transactions and the transaction latency, i.e., the average time elapsed between the submission of a transaction in the network and the time at which the transaction is confirmed. We compare the performance of Yggdrasil to solutions with static sharding such as Monoxide[15] with a number of shards n throughout the simulation.

5.6.3.1 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 $f_\mathrm{t} = 10$ 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 $f_\mathrm{t} > 10$ 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 $f_\mathrm{t} = 1280$ 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.

5.6.3.2 Experiment results

The main results of our experiments appear in Figures 5.7a, 5.7b and 5.7c. Note that in all the graphs, points are linked together with lines. This is only for readability reasons.

Figure 5.7a 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 [113], 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

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

Figure 5.8 – 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 [11] and Omniledger [13].

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 5.7b 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 5.7c 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.

5.6.4 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 [11], Omniledger [13]). We thus study the reactivity of solutions that adapt the number of shards at specific reconfiguration periods rp.

5.6.4.1 Experiments setting

As briefly presented in Section 5.3.5, 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, $f_\mathrm{t} = 500$ 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 [13] and Elrond [11] (a day).

5.6.4.2 Experiments results

Figure 5.8 shows the reactivity of Yggdrasil in presence of a load peak (constant function from $t = 1$ to $t = 11$ ticks at $f_{\text{t}} = 500$txs/tick). Yggdrasil initially undergoes a series of splits, it reaches a maximum transaction confirmation rate of 375 txs/tick in order to lower latency to 35 ticks. Then, it progressively moves on to a series of merge up to converging to a single shard. The time-driven solution, on the other hand, performs less well since it does not adapt its number of shards automatically. Indeed, at low values of rp such as 10 or 20 ticks, the system still manages to increase the confirmation rate (190-250 txs/tick) to absorb the increase in throughput thus lower latency (70-85 ticks). At medium values such as 50 or 100 ticks, the system reacts late and many transactions are already passed at a lower confirmation rate (60-120 txs/tick) and therefore with a higher latency (115-125 ticks). For our highest values rp $> 100ticks$, 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.

5.6.5 Cross-shard volume

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

5.6.5.1 Experiment setting

Additionally to the experiments settings defined in section 4.5.3.1, 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.

5.6.5.2 Experiment results

The main results of our experiments appear in the graphs of Figure 5.7d. Note that in all the graphs, points are linked together with lines. This is only for readability reasons. Figure 5.7d 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

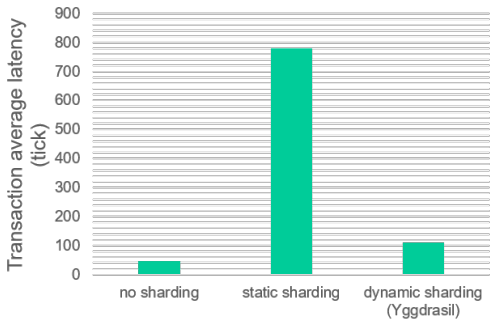


Figure 5.9 – Transaction average latency with 2-Phase Commit Algorithm.

p_c increases. It also decreases as n increases and is extremely low for Yggdrasil (as shown in Section 5.6.3) since the number of shards in this specific scenario depends on the transaction arrival rate.

5.6.6 2PC algorithm

This section studies the performance impact of our newly presented 2PC algorithm for distributed smart-contracts (see Section 5.3.1). 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.

5.6.6.1 Experiment setting

Additionally to the experiments settings defined in section 4.5.3.1, 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.

5.6.6.2 Experiment results

The main results of our experiments appear in the graph of Figure 5.9. 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.

5.7 Conclusion

In this chapter 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.

Chapter 6

Conclusions

"The greater the difficulty, the more glory in surmounting it"

– Epicurus

Contents

6.1 General Conclusion

Blockchain solutions have the ambition to replace traditional infrastructures used for instance by finance, traceability, or medecine applications. In fact, any application relying on a trusted third party could be replaced by blockchain. To get there, the path is long and full of obstacles. These obstacles are often illustrated by the blockchain trilemma conjecture according to which blockchain cannot meet the needs of decentralization, security and scalability at the same time. One of the challenges that we need to face today is the design of blockchains that are very close to «solve» this trilemma (if it is solvable), i.e., find the perfect solution that fulfills all three axes of the trilemma. During this thesis, we were interested in this blockchain trilemma and in particular in one of its axes: scalability, i.e., the capacity of a system to adapt to meet a growing demand. This is a lever of blockchain systems that has interested the community for many years and is still a topic of current interest. It inspired this thesis which raises some questions, namely: "Is it possible to improve the scalability of existing blockchain solutions while maintaining their decentralization and security?" and "Are graph-based blockchains a solution to blockchain's lack of scalability?".

In order to answer these questions, we have been looking at solutions that have been shown to improve scalability while maintaining decentralization and security. Graph-based blockchains reveal to be relevant candidates since their graph structure enables parallelisation of nodes' work while keeping its decentralized aspect.

In this thesis, we studied Sycomore [27], the first ledger protocol, relying on Bitcoin design principles, that addresses Bitcoin's scalability issues by having a

graph-structure design that allows for the “parallel" creation of valid and durably appended chains of blocks. Since the main difference between Sycomore and Bitcoin lies in their structure, it helped us exhibit some benefits of using a graph structure in comparison with a classical structure with a single chain. This study led us to propose Sycomore^{++}, a blockchain protocol based on Sycomore. It consists of an improvement of Sycomore enabling an auto-adaptation of the PoW difficulty to the graph structure. Sycomore^{++} has shown us its true potential in terms of scalability and its particular suitability for applications exhibiting strong and rapid load variations. Moreover, as our analytical and experimental results show, Bitcoin's security has not been sacrificed for scalability either.

Based on this study, we can say that in the specific case of Bitcoin and Sycomore^{++} it is possible to improve one of the axes of the blockchain trilemma without sacrificing the other two, and this thanks to graph-based solutions. That said, this has raised another question, to what extent is it possible to improve one of these axes before having to sacrifice one of the other two?

To answer this newly raised question, we first had to know what these limits represent. Since we were interested in scalability, we decided to exploit this axis by looking at the most innovative solutions improving blockchains' scalability. That's how we got interested in sharding [1], a solution of state compartmentalization, historically used in databases and then adapted to blockchains. It is in this context that we have proposed Yggdrasil, a solution implementing state-sharding (one of the three types of sharding presented in Chapter 5). Thanks to this technique of dividing the state into several shards, we have been able to achieve very promising performances better than those of Sycomore^{++} in terms of scalability. Since state-sharding's main advantage is to reduce communication and storage overhead, solutions implementing it would tend to scale more easily. One of its other advantages being its capacity to keep a strong decentralization by giving more power to more nodes in separate shards. Additionally, our analytic and experimental study of Yggdrasil allowed us to assess its security.

Finally, our work on both DAGs and state-sharded solutions helped us establish that graph-based solutions are indeed a very interesting and promising solution in terms of scalability. Moreover, since this solution does not sacrifice decentralization nor security, it seems to be a good balance between the three blockchain pillars. This work made us understand that scalability as well as each of the two other axes of the trilemma could be pushed back. However, since we do not know the limits of each of these axes given the current state of art in blockchains, we consider that it is not possible to attain these limits. The best that any solution can do is to find the right balance between these pillars, which is what graph-based blockchains have the potential to achieve, as we have seen in this thesis.

[1] Given the graph structure of the shards, we consider this type of solution as a more advanced graph-based solution.

6.2 Future Work

In this section, we will discuss potential future work arising from this thesis. These perspectives are extensions of the work presented in this manuscript.

We have structured this section in the same way as our main contributions. Namely two main subsections, one for graph-based solutions (Sycomore^{++}, see Chapter 4 for more details) and another for solutions implementing sharding (Yggdrasil, see Chapter 5 for more details).

6.2.1 Graph-based blockchains

In this section, we study the perspective and future work derived from our work on graph-based blockchains such as Sycomore^{++}.

Sycomore^{++} and its performances. During this thesis, we have proposed Sycomore^{++}, a graph-based blockchain protocol (see Chapter 4 for more details). This protocol is a modification of Sycomore [27], which itself is an adaptation of Bitcoin in graph structure. Its main feature is to dynamically self-adapt the number of created blocks to the current number of submitted transactions. We have also presented an advanced experimental study to assess the properties of three permissionless PoW-based distributed ledgers under high or chaotic submission transaction rate, and adversarial environments. Experimental results show the nice behavior of Sycomore^{++} compared to both Bitcoin and Sycomore in terms of scalability, energy loss, reactivity, resilience, and quality of the distributed ledger. Our experimental study allowed us to show the potential of our solution. However, we believe that implementing Sycomore^{++} and testing it in a real environment would get us past the resource limitations. In this way, we would be able to show how Sycomore^{++} behaves when exposed to the same conditions as Bitcoin or others today.

We believe that in this way, we could realistically compare Sycomore^{++} to other graph-based blockchains [3, 4, 2] or any layer-1 blockchain solutions that would aim to improve the scalability of Bitcoin [114, 115, 116] and thus give a more accurate answer to the question "are graph-based blockchains a solution to blockchain's lack of scalability?".

Deepen the study of Sycomore's properties. As said before, Sycomore^{++} is a modification of Sycomore. It is the correction of a flaw that we noticed while studying Sycomore. Nevertheless, Sycomore is a complex protocol which properties are numerous and intricate. In order to optimize the performances of Sycomore^{++}, we think that it would be appropriate to deepen the study of Sycomore. More specifically, we would focus on the different system parameters (Γ, γ, c_{min}, H_{max}...) that could impact the performances as well as the security of both Sycomore and Sycomore^{++}.

Analyze the computational cost of adversarial strategies in the presence of transient network partitions. During our study of Sycomore^{++}, we did not focus on the impact of the network on our system. We make the hypothesis that the transmission delay is null and that the transmission reliability is 100% (i.e. all the sent messages are received instantaneously). In order to deepen our study, we find interesting to vary these parameters to make the network less predictable and more realistic. This modification would also open the way to potential network attacks. For isntance, an adversary could use the dynamism of the network topologies to his advantage by creating partitions and thus take the ascendancy on the other nodes of the system (e.g. he could isolate a part of the network to create a fork).

Study the impact of fork resolution on Sycomore^{++} performances. As our study of Sycomore^{++} was mainly about scalability, our goal was to show the performances that Sycomore^{++} could offer in terms of throughput. Thus, during our study on the resolution of forks in Bitcoin, Sycomore and Sycomore^{++}, we did not dwell on the impact of these forks on the system's performance. Forks can lead to the loss of an important number of blocks thus to the loss of miners' work which could harm the performance of the system. Hence, we think that it would be interesting to compare the proportion of discarded blocks in Bitcoin, Sycomore and Sycomore^{++} and the impact of this energy loss on the performance of the system, under different network conditions.

Incentives for honest participation. During our study of Sycomore^{++}, we did not focus on the incentives for a miner to act correctly. As shown by [50], the security of a system is endangered without a static block reward but what about the disappearance of the block fees? In Sycomore^{++}, transactions are uniformly partitioned and since it adapts its graph structure to transactions arrival rate, transactions are less in competition with each other compared to non-adaptive blockchain systems such as Bitcoin.

The question we could ask ourselves is how much lower could fees go? Isn't their presence an incentive for the miner to act properly? And if the fees fluctuate according to the graph structure, wouldn't it be in the miner's interest to prevent the graph from splitting and therefore underfill his blocks? We think that it would be interesting to answer these questions in order to complete our study on Sycomore^{++} with the tokenomics aspect.

6.2.2 State-sharded blockchains

In this section, we study the perspective and future work derived from our work on state-sharded blockchains such as Yggdrasil.

Yggdrasil and its performances. As our second contribution, we have 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. An experimental study confirms the capability of Yggdrasil to scale and to adapt to transaction load.

As for Sycomore^{++}, we believe that the next step would be to implement Yggdrasil and test it in a real environment. The fact is that the arrival of sharding in blockchains is recent. It is an innovative solution with a strong potential, especially in terms of scalability. Comparing Yggdrasil to other non-sharding solutions [17, 22, 115, 114, 116, 62] would give us a strong insight on the interest of using sharding in blockchains. On the other hand, comparing it to other sharding-based industrial [11, 10] or academic [12, 13, 32, 15] solutions in real conditions would show the benefits of the proposed algorithms.

Smart Contracts in Sharding. Smart-contracts represent a large part of the universal daily transactions volume. In state-sharded blockchains, they represent a new challenge. Indeed, a smart-contract is a "generator" of traffic in its shard, which could lead to a workload imbalance among shards.

In the literature, one of the most popular solutions to reduce shards workload imbalance would be to make the smart-contracts "splittable" so that their state would be divisible and that the two resulting smart-contracts would be able to live each in a shard. Even if this is true for some types of smart-contracts such as ERC-20 contracts [102], it is far from being applicable to most of them.

Optimization of cross-shard protocols. The benefits of state-sharding lies in the division of the blockchain state into several sub-states assigned to each shard. The more independent these shards are, the faster they can advance. The dependency between these shards is represented by the cross-shard communication. Cross-shard mechanisms represent a major issue in state-sharding blockchains. They rely on the communication between shards and can considerably decrease the performance of the system since they would increase the dependency between shards and would remove any interest in doing state-sharding. As a reminder, a transaction is said to be cross-shard when it involves two entities assigned to two different shards.

According to our study, the volume of cross-shard transactions is raised by the way users and transactions are partioned. In our system, transactions are assigned to the issuing shard because it is the most likely to be able to verify the validity of the transaction. If the transaction is intended for a user assigned to another shard, it is considered cross-shard. In our work, we have proposed a possibility for users to clone themselves in another shard and thus choose their assignment. This would reduce the number of cross-shard transactions but would be at the discretion of the user. Another more centralized solution has been proposed by [16] which would allow to define user assignments according to their past exchanges. Unfortunately, this solution is centralized and does not offer any guarantee about the future exchanges of users. Moreover, it requires a synchronous environment to be applicable. Thus,

we believe that future work on other more interesting solutions could be done in an asynchronous environment.

Incentives in a sharded system. As future work, it would be interesting to study how incentives and fees could be re-designed in a system where transactions are routed over multiple shards. In this kind of systems, as more than one shard, and therefore more than one committee can be involved in the confirmation of a transaction (i.e. in the case of cross-shard transactions), the fees should be shared between these shards. Since fees cannot be configured from the beginning as a cross-shard transaction because of the system's dynamicity, the state of the shards and therefore the condition of the transaction (whether it is cross-shard or not) can change. So both shards have to settle for a classic fee, but in this case, how to share it? Since the issuing shard is the one that performs the validity check, should it be the one that collects all the fee? But in this case, what interest would the second shard have to include (and thus confirm) the transaction in its shardchain. The addition of smart-contracts only deepens these questions. If a user calls a smart-contract, he has to put in an amount of gas to pay back the committee for doing the calculations. In the sharded case, there are several committees that are responsible of multiple calculations, especially in the case of a cascade of smart-contract calls. How do we share the reward between these committees?

Bibliography

[1] J. Sliwinski and R. Wattenhofer, "ABC: asynchronous blockchain without consensus," *CoRR*, vol. abs/1909.10926, 2019. (Cited on pages iii and 27.)

[2] S. Popov, "The tangle. iota white paper," 2015. (Cited on pages iii, 26, 27, 37, 69 and 131.)

[3] Y. Sompolinsky and A. Zohar, "Accelerating bitcoin's transaction processing. fast money grows on trees, not chains," *IACR Cryptology ePrint Archive*, vol. 2013, 2013. (Cited on pages iii, 26, 28, 69 and 131.)

[4] Y. Sompolinsky, Y. Lewenberg, and A. Zohar, "Spectre: A fast and scalable cryptocurrency protocol," *IACR Cryptol. ePrint Arch.*, p. 1159, 2016. (Cited on pages iii, 26, 28 and 131.)

[5] MAX, "Source code." `https://gitlab.com/cea-licia/max/`. (Cited on pages iv, 43, 59, 79 and 122.)

[6] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science* (I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, eds.), vol. 367 of *Communications in Computer and Information Science*, pp. 3–20, Springer International Publishing, 2013. (Cited on pages iv, 52, 79 and 123.)

[7] A. Djari, E. Anceaume, and S. Tucci-Piergiovanni, "Simulation study of sycomore++, a self-adapting graph-based permissionless distributed ledger," in *2022 4th Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pp. 103–110, 2022. (Cited on pages iv and ix.)

[8] A. Djari, E. Anceaume, and S. Tucci-Piergiovanni, "An extensive agent-based simulation study of sycomore++, a dag-based permissionless ledger," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, (New York, NY, USA), p. 334–336, Association for Computing Machinery, 2022. (Cited on pages iv and ix.)

[9] A. Djari, E. Anceaume, and S. Tucci-Piergiovanni, "Sycomore ++ , un registre distribué orienté graphe auto-adaptatif," in *AlgoTel 2022 - 24èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*, (Saint-Rémy-Lès-Chevreuse, France), pp. 1–4, May 2022. (Cited on pages v and ix.)

[10] N. Durov, "Telegram Open Network," tech. rep., 03 2019. (Cited on pages v, 30, 31, 32, 38 and 133.)

[11] T. E. Team, "Elrond - A Highly Scalable Public Blockchain via Adaptive State Sharding and Secure Proof of Stake," tech. rep., 06 2019. (Cited on pages v, vi, 30, 31, 32, 38, 93, 125, 126 and 133.)

[12] 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*, CCS '16, p. 17–30, Association for Computing Machinery, 2016. (Cited on pages v, 7, 30, 32, 37, 92 and 133.)

[13] 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. (Cited on pages v, vi, 7, 30, 31, 32, 37, 92, 125, 126 and 133.)

[14] 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*, CCS '18, p. 931–948, Association for Computing Machinery, 2018. (Cited on pages v, 7, 30, 31, 32 and 92.)

[15] 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), pp. 95–112, USENIX Association, Feb. 2019. (Cited on pages v, 30, 31, 32, 33, 92, 96, 124 and 133.)

[16] 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. (Cited on pages v, 7, 30, 31, 32, 33, 92 and 133.)

[17] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," 2016. (Cited on pages vi, xiii, 6, 18, 22, 23, 24, 32, 98, 123 and 133.)

[18] A. Djari, Y. Amoussou-Guenou, E. Anceaume, S. Tucci-Piergiovanni, and A. Del Pozzo, "Yggdrasil: Secure state sharding of transactions and smart contracts that self-adapts to transaction load," Available on HAL, 9 2022. (Cited on pages vii and ix.)

[19] L. Chen, L. Xu, N. Shah, Z. Gao, Y. Lu, and W. Shi, "On security analysis of proof-of-elapsed-time (poet)," in *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium, SSS 2017, Proceedings* (P. Tsigas and P. Spirakis, eds.), Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pp. 282–297, Springer-Verlag, 2017. (Cited on pages xiii, 13 and 14.)

[20] D. Bayer, S. Haber, and W. S. Stornetta, "Improving the efficiency and reliability of digital time-stamping," in *Sequences II: Methods in Communication, Security and Computer Science*, pp. 329–334, Springer-Verlag, 1993. (Cited on page 2.)

[21] B. M. David, P. Gazi, A. Kiayias, and A. Russell, "Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain," in *EUROCRYPT*, 2018. (Cited on pages 6, 10, 18 and 32.)

[22] J. Chen and S. Micali, "Algorand: A secure and efficient distributed ledger," *Theor. Comput. Sci.*, vol. 777, pp. 155–183, 2019. (Cited on pages 6, 10, 18, 22, 32, 98 and 133.)

[23] L. Aştefănoaei, P. Chambart, A. Del Pozzo, T. Rieutord, S. Tucci-Piergiovanni, and E. Zălinescu, "Tenderbake - A Solution to Dynamic Repeated Consensus for Blockchains," in *4th International Symposium on Foundations and Applications of Blockchain 2021 (FAB 2021)*. (Cited on page 6.)

[24] J. Poon and T. Dryja, *The bitcoin lightning network*. 2016. (Cited on pages 6, 25 and 43.)

[25] C. Burchert, C. Decker, and R. Wattenhofer, "Scalable funding of bitcoin micropayment channel networks," in *SSS*, 2017. (Cited on page 6.)

[26] A. Ranchal-Pedrosa, M. G. Potop-Butucaru, and S. T. Piergiovanni, "Scalable lightning factories for bitcoin," *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019. (Cited on page 6.)

[27] 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. (Cited on pages 6, 28, 53, 54, 70, 71, 73, 74, 77, 94, 109, 129 and 131.)

[28] 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*, pp. 3–7, Springer, 2020. (Cited on page 7.)

[29] 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), p. 288–302, Springer-Verlag, 2019. (Cited on pages 7, 30, 31, 32, 37 and 92.)

[30] 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)*, pp. 1357–1368, 2020. (Cited on pages 7 and 92.)

[31] 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)*, pp. 1337–1347, 2019. (Cited on pages 7 and 92.)

[32] 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), p. 148–165, Springer-Verlag, 2019. (Cited on pages 7, 30, 32, 37 and 133.)

[33] S. Nakamoto, "Bitcoin : A peer-to-peer electronic cash system," 2009. (Cited on pages 10, 11, 18, 32, 33 and 69.)

[34] "Ethereum proof-of-stake consensus specifications." https://github.com/ethereum/consensus-specs/tree/52a741f7c6d3bec98e04df3441bc8e7681480877/specs/altair. (Cited on pages 10, 18 and 98.)

[35] 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*, vol. 7417 of *Lecture Notes in Computer Science*, pp. 1–13, Springer, 2012. (Cited on page 10.)

[36] S. Sayeed and H. Marco-Gisbert, "Assessing blockchain consensus and security mechanisms against the 512019. (Cited on pages 11 and 21.)

[37] S. N. Sunny King, "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake," 2012. (Cited on page 12.)

[38] V. Buterin, "Slasher: A punitive proof-of-stake algorithm," *Ethereum Blog URL: https://blog. ethereum. org/2014/01/15/slasher-a-punitive-proof-of-stake-algorithm*, p. 85, 2014. (Cited on page 13.)

[39] K. Karantias, A. Kiayias, and D. Zindros, "Proof-of-burn." Cryptology ePrint Archive, Paper 2019/1096, 2019. https://eprint.iacr.org/2019/1096. (Cited on page 15.)

[40] E. Anceaume, A. D. Pozzo, T. Rieutord, and S. Tucci Piergiovanni, "On finality in blockchains," *CoRR*, vol. abs/2012.10172, 2020. (Cited on pages 15, 18 and 98.)

[41] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982. (Cited on page 16.)

[42] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, (USA), p. 173–186, USENIX Association, 1999. (Cited on page 16.)

[43] "Proof of authority." https://github.com/paritytech/parity/wiki/Proof-of-Authority-Chains. (Cited on page 17.)

[44] V. Buterin, "Ethereum white paper: A next generation smart contract & decentralized application platform," 2013. (Cited on pages 18, 32 and 70.)

[45] E. B. Jae Kwon, "Cosmos : A network of distributed ledgers." (Cited on pages 18, 19 and 98.)

[46] M. Bourgoin, "An overview of the tezos blockchain." (Cited on pages 18, 19, 33, 94 and 98.)

[47] 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)* (V. Gramoli and M. Sadoghi, eds.), vol. 92 of *OASIcs*, pp. 1:1–1:23, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. (Cited on pages 18, 32 and 98.)

[48] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *CoRR*, vol. abs/1710.09437, 2017. (Cited on page 19.)

[49] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," *Commun. ACM*, vol. 61, p. 95–102, jun 2018. (Cited on page 21.)

[50] M. Carlsten, H. Kalodner, S. M. Weinberg, and A. Narayanan, "On the instability of bitcoin without the block reward," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), p. 154–167, Association for Computing Machinery, 2016. (Cited on pages 21 and 132.)

[51] J. R. Douceur, "The sybil attack," 2002. (Cited on page 22.)

[52] S. Micali, S. Vadhan, and M. Rabin, "Verifiable random functions," in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, (USA), p. 120, IEEE Computer Society, 1999. (Cited on page 22.)

[53] J. Göbel and A. Krzesinski, "Increased block size and bitcoin blockchain dynamics," in *2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*, pp. 1–6, 2017. (Cited on pages 25, 80 and 123.)

[54] Y. Assia, V. Buterin, L. Hakim, M. Rosenfeld, and R. Lev, "Colored coins whitepaper." http://www.ma.senac.br/wp-content/uploads/2018/05/ColoredCoinswhitepaper-DigitalAssets.pdf. (Cited on page 25.)

[55] V. B. Fabian Vogelsteller, "Eip-20: Token standard." https://eips.ethereum.org/EIPS/eip-20, 2015. (Cited on page 26.)

[56] L. Baird, "The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance," tech. rep., 2016. (Cited on page 26.)

[57] A. Churyumov, "ByteBall : A decentralized system for storage and transfer of value," 2017. (Cited on page 26.)

[58] G. Bu, Ö. Gürcan, and M. Potop-Butucaru, "G-IOTA: fair and confidence aware tangle," *CoRR*, vol. abs/1902.09472, 2019. (Cited on page 26.)

[59] T. Rocket, M. Yin, K. Sekniqi, R. van Renesse, and E. G. Sirer, "Scalable and probabilistic leaderless BFT consensus through metastability," *CoRR*, vol. abs/1906.08936, 2019. (Cited on page 27.)

[60] D. Tennakoon and V. Gramoli, "Dynamic Blockchain Sharding," in *5th International Symposium on Foundations and Applications of Blockchain 2022 (FAB 2022)* (S. Tucci-Piergiovanni and N. Crooks, eds.), vol. 101 of *Open Access Series in Informatics (OASIcs)*, (Dagstuhl, Germany), pp. 6:1–6:17, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. (Cited on page 31.)

[61] D. Tennakoon, Y. Hua, and V. Gramoli, "Collachain: A bft collaborative middleware for decentralized applications," 2022. (Cited on page 31.)

[62] V. Buterin, "Ethereum: A next-generation smart contract and decentralized application platform," 2014. (Cited on pages 33 and 133.)

[63] A. Varga and R. Hornig, "An overview of the omnet++ simulation environment," in *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, 2008. (Cited on page 34.)

[64] G. F. Riley and T. R. Henderson, "The ns-3 network simulator.," in *Modeling and Tools for Network Simulation*, 2010. (Cited on page 34.)

[65] "Ns3 testing framework." https://www.nsnam.org/docs/release/3.9/testing.html#TestingFramework. (Cited on page 34.)

[66] O. Boissier, R. Bordini, J. F. Hübner, A. Ricci, and A. Santi, "Multi-agent oriented programming with JaCaMo," 2011. (Cited on page 34.)

[67] A. Ricci, A. Ciortea, J. F. Hubner, R. H. Bordini, O. Boissier, and S. Mayer, "Engineering scalable distributed environments and organizations for mas," in *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*, 2019. (Cited on page 35.)

[68] "Simevents." https://www.mathworks.com/products/simevents.html. (Cited on page 35.)

[69] MATLAB, *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010. (Cited on page 35.)

[70] "Matlab blockchain example." https://fr.mathworks.com/matlabcentral/fileexchange/65419-matlab-blockchain-example. (Cited on page 35.)

[71] "simmer." https://r-simmer.org/articles/simmer-02-jss.pdf. (Cited on page 35.)

[72] M. Smolla, "An introduction to agent-based modelling in r," 2015. (Cited on page 35.)

[73] Y. Aoki, K. Otsuki, T. Kaneko, R. Banno, and K. Shudo, "Simblock: A blockchain network simulator," 2019. (Cited on page 36.)

[74] C. Kaligotla and C. M. Macal, "A generalized agent based framework for modeling a blockchain system," in *Proceedings of the Winter Simulation Conference (WSC)*, 2018. (Cited on page 36.)

[75] R. Memon, J. Li, J. Ahmed, A. Khan, M. Irshad Nazir, and M. I. Mangrio, "Modeling of blockchain based systems using queuing theory simulation," 2018. (Cited on page 36.)

[76] P.-Y. Piriou and J.-F. Dumas, "Simulation of stochastic blockchain models," in *Workshop on Blockchain Dependability organized with the 14th European Dependable Computing Conference*, 2018. (Cited on page 36.)

[77] M. Alharby and A. van Moorsel, "Blocksim: A simulation framework for blockchain systems," *ACM SIGMETRICS Performance Evaluation Review*, vol. 46, pp. 135–138, 01 2019. (Cited on page 36.)

[78] E. Rosa, G. D'Angelo, and S. Ferretti, "Agent-based simulation of blockchains," *ArXiv*, vol. abs/1908.11811, 2019. (Cited on page 36.)

[79] C. Faria and M. Correia, "Blocksim: Blockchain simulator," in *IEEE International Conference on Blockchain (Blockchain)*, 2019. (Cited on page 36.)

[80] M. Bottone, F. Raimondi, and G. Primiero, "Multi-agent based simulations of block-free distributed ledgers," 2018. (Cited on page 37.)

[81] U. Wilensky, "Netlogo," Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL., 1999. (Cited on page 37.)

[82] C. M. Macal and M. J. North, "Tutorial on agent-based modeling and simulation part 2: How to model with agents," in *Proceedings of the 38th Conference on Winter Simulation*, WSC '06, p. 73–83, Winter Simulation Conference, 2006. (Cited on pages 40 and 42.)

[83] S. Mehta, N. Sultana, and K. Kwak, *Network and System Simulation Tools for Next Generation Networks: a Case Study*. 08 2010. (Cited on pages 40 and 42.)

[84] U. Hatnik and S. Altmann, "Using modelsim, matlab/simulink and ns for simulation of distributed systems," in *Parallel Computing in Electrical Engineering, 2004. International Conference on*, pp. 114–119, 2004. (Cited on page 40.)

[85] N. Lagaillardie, M. A. Djari, and O. Gurcan, "A computational study on fairness of the tendermint blockchain protocol," *Information*, vol. 10, no. 12, 2019. (Cited on page 43.)

[86] O. Gutknecht and J. Ferber, "The madkit agent platform architecture," in *Proceedings of the International Workshop on Infrastructure for Multi-Agent Systems: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems*, 2000. (Cited on pages 43, 46, 79 and 122.)

[87] B. Curran, "What is a merkle tree? beginner's guide to this blockchain component," 2018. (Cited on page 48.)

[88] G. Caldarelli, "Overview of blockchain oracle research," *Future Internet*, vol. 14, no. 6, 2022. (Cited on page 51.)

[89] OAR, "The oar project." `http://oar.imag.fr/`. (Cited on page 52.)

[90] E. Jeanvoine, L. Sarzyniec, and L. Nussbaum, "Kadeploy3: Efficient and Scalable Operating System Provisioning," *USENIX ;login:*, vol. 38, pp. 38–44, Feb. 2013. (Cited on page 52.)

[91] O. Tange, "Gnu parallel - the command-line power tool," *;login: The USENIX Magazine*, vol. 36, pp. 42–47, Feb 2011. (Cited on page 52.)

[92] W. Tang, "Ecip-1029: Include uncles in total difficulty calculation." `https://github.com/ethereumproject/ECIPs/pull/71`, 2017. (Cited on page 70.)

[93] "Luna." `https://docs.terra.money/learn/protocol/`. (Cited on page 70.)

[94] E. Kim, T. Andersen, Marventus, A. E., P. Borges, D. Schmidt, and M. Western, "Emergency management and recovery of luna classic," 2022. (Cited on page 70.)

[95] J. A. Garay, A. Kiayias, and N. Leonardos, "The Bitcoin Backbone Protocol: Analysis and Applications," in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques - Advances in Cryptology (EUROCRYPT)*, 2015. (Cited on pages 76, 86, 87, 88 and 89.)

[96] Sycomore^{++}, "Source code." https://anonymous.4open.science/r/Sycomorepp-412D. (Cited on page 79.)

[97] B. H. Distribution 2020. (Cited on page 80.)

[98] "Cardano." https://github.com/input-output-hk/cardano-node. (Cited on page 94.)

[99] "Pyethereum." https://github.com/ethereum/pyethereum/blob/782842758e219e40739531a5e56fff6e63ca567b/ethereum/utils.py. (Cited on page 95.)

[100] D. Skeen, "Nonblocking commit protocols," in *In Proceedings of the 1981 ACM SIGMOD international Conference on Management of Data (SIGMOD)*, pp. 133–142, 1981. (Cited on page 96.)

[101] P. Robinson and R. Ramesh, "General purpose atomic crosschain transactions," in *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*, pp. 61–68, IEEE, 2021. (Cited on page 96.)

[102] 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*, PLDI 2021, (New York, NY, USA), p. 1327–1341, Association for Computing Machinery, 2021. (Cited on pages 96 and 133.)

[103] I. Abraham and D. Malkhi, "The blockchain consensus layer and BFT," *Bulletin of the EATCS*, vol. 3, no. 123, pp. 1–23, 2017. (Cited on page 97.)

[104] L. Lamport, R. Shostak, and M. Pease, *The Byzantine Generals Problem*, p. 203–226. New York, NY, USA: Association for Computing Machinery, 2019. (Cited on page 98.)

[105] "Whisk: A practical shuffle-based ssle protocol for ethereum." https://ethresear.ch/t/whisk-a-practical-shuffle-based-ssle-protocol-for-ethereum/11763. (Cited on page 98.)

[106] 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.), pp. 151–158, IEEE, 2013. (Cited on page 98.)

[107] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, p. 288–323, apr 1988. (Cited on page 98.)

[108] 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), pp. 3–36, Springer Berlin Heidelberg, 2021. (Cited on page 100.)

[109] 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.), pp. 15–24, IEEE Computer Society, 2008. (Cited on page 120.)

[110] E. API, 2022. (Cited on pages 122 and 123.)

[111] Yggdrasil, "Source code." `https://anonymous.4open.science/r/Yggdrasil-11E5`. (Cited on page 122.)

[112] MAX, "Source code." `https://gitlab.com/cea-licia/max/models/ledgers/max.model.ledger.tendermint_v2`. (Cited on page 123.)

[113] 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)*, pp. 23–33, 2021. (Cited on page 124.)

[114] "Bitcoin xt." `https://github.com/bitcoinxt/bitcoinxt`. (Cited on pages 131 and 133.)

[115] "Bitcoin classic." `https://github.com/bitcoinclassic/bitcoinclassic`. (Cited on pages 131 and 133.)

[116] "Bitcoin unlimited." `https://github.com/BitcoinUnlimited/BitcoinUnlimited`. (Cited on pages 131 and 133.)

Titre : Etude du potentiel des approches à base de graphes dans les blockchains

Mot clés : Blockchain, Graphe, Scalabilité, Simulation basée agents, Sharding, Performance.

Résumé : Les chaînes de blocs sont des systèmes de pair à pair dans lesquels les utilisateurs peuvent échanger des actifs numériques sans autorité de validation centrale. Il s'agit d'un grand livre distribué maintenu par la communication entre les nœuds du réseau. C'est un grand livre sur lequel toutes les opérations sont enregistrées, ce qui contribue à sa transparence puisque chaque ajout dans la blockchain peut être lu par tous et pour toujours (tant que le réseau existe). Selon l'application souhaitée, le bon fonctionnement de la blockchain repose sur trois piliers communs : (i) la décentralisation, (ii) la sécurité et (iii) l'évolutivité. Une solution qui permettrait de réunir ces trois caractéristiques est actuellement considérée comme une utopie que l'on appelle le trilemme de la blockchain, une croyance selon laquelle une crypto-monnaie doit nécessairement sacrifier l'un de ces trois piliers. Au cours de cette thèse, les enjeux ont rapidement été ceux de la recherche de performance, notamment en termes de scalabilité sans pour autant négliger les deux autres aspects du trilemme. Nous avons alors commencé par étudier Sycomore, une blockchain PoW non permissionnée, immuable et sécurisée, dont la structure est basée sur des graphes. C'est au cours de l'étude de Sycomore que nous avons proposé Sycomore^{++}, un protocole de blockchain basé sur Sycomore dont la principale caractéristique est d'auto-adapter dynamiquement le nombre de blocs créés au nombre actuel de transactions soumises. Les résultats de cette étude ont été publiés dans les actes de conférences à comité de lecture. Dans un second temps, après avoir montré l'apport d'une solution classique à base de graphe dans le trilemme de la blockchain, nous nous sommes intéressés aux solutions de sharding qui, compte tenu de leur structure en graphe, nous ont semblé être les solutions à base de graphe les plus avancées et les plus prometteuses en termes de scalabilité. C'est dans ce contexte que nous proposons Yggdrasil, une solution de sharding d'état pour les blockchains sans permissions qui supporte à la fois les transactions de paiement et les smart contracts. Yggdrasil permet de diviser et de fusionner dynamiquement les shards en s'appuyant sur des mécanismes décentralisés pour affecter les nœuds aux shards de manière sécurisée. Dans ce travail, nous proposons également un nouveau protocole 2-Phase-Commit pour garantir l'exécution de smart contracts distribués sur différents shards, même lorsque les shards se réorganisent dynamiquement. Une étude expérimentale confirme la capacité d'Yggdrasil à évoluer et à s'adapter à la charge de transactions avec des performances très prometteuses, meilleures que celles de Sycomore^{++} en termes de scalabilité. Le principal avantage du state-sharding étant de réduire les coûts de communication et de stockage, les solutions qui l'implémentent ont tendance à évoluer plus facilement. Un autre avantage est sa capacité à maintenir une forte décentralisation en habilitant plus de nœuds dans des shards séparés sans entraver sa sécurité. Au moment de la rédaction de ce manuscrit, les résultats de cette étude sur Yggdrasil ont été soumis pour publication à VLDB 2023 et un rapport technique présentant les résultats est disponible.

Title: Study of the potential of graph-based approaches in blockchains

Keywords: Blockchain, Graph, Scalability, Agent-based Simulation, Sharding, Performance.

Abstract: Blockchains are peer-to-peer systems in which users can exchange digital assets without a central validation authority. It is a distributed ledger maintained through communication between the nodes of the network. It is a ledger on which all operations are recorded, which contributes to its transparency since every addition in the blockchain can be read by everyone and forever (as long as the network exists). Depending on the desired application, the proper functioning of the blockchain relies on three common pillars: (i) decentralization, (ii) security and (iii) scalability. A solution that would bring these three characteristics together is currently considered a utopia that is known as the blockchain trilemma, a belief that a crypto-currency must necessarily sacrifice one of these three pillars. During the course of this thesis, the issues at stake were quickly those of the quest for performance, particularly in terms of scalability without neglecting the other two aspects of the trilemma. We then started by studying Sycomore, an immutable and secure permisionless PoW blockchain with a graph-based structure. It is during the study of Sycomore that we propose Sycomore^{++}, a blockchain protocol based on Sycomore whose main feature is to dynamically self-adapt the number of blocks created to the current number of transactions submitted. The results of this study have been published in the proceedings of peer-reviewed conferences. In a second step, after having shown the contribution of a classical graph-based solution in the blockchain trilemma, we looked at sharding solutions which, given their graph structure, seemed to us to be the most advanced graph-based solutions and the most promising in terms of scalability. It is in this context that we propose Yggdrasil, a state sharding solution for permisionless blockchains that supports both payment transactions and smart contracts. Yggdrasil allows for dynamic splitting and merging of shards by relying on decentralized mechanisms to assign nodes to shards in a secure manner. In this work, we also propose a new 2-Phase-Commit protocol to guarantee the execution of distributed smart contracts on different shards, even when shards dynamically reorganize. An experimental study confirms the ability of Yggdrasil to evolve and adapt to the transaction load with very promising performance, better than Sycomore^{++} in terms of scalability. Since the main benefit of state-sharding is to reduce communication and storage costs, solutions that implement it tend to scale more easily. Another advantage is its ability to maintain strong decentralization by empowering more nodes in separate shards without hindering its security. At the time of writing this manuscript, the results of this study on Yggdrasil have been submitted for publication to VLDB 2023 and a technical report presenting the results is available.