UNIVERSITÉ DU QUÉBEC À MONTRÉAL

AN AUTO-SCALING ARCHITECTURE FOR BLOCKCHAIN IN THE CONTEXT OF THE INTERNET OF THINGS

THESIS

PRESENTED

AS PARTIAL REQUIREMENT

TO THE PH.D IN COMPUTER SCIENCE

BY

RIHAM ELSAADANY

MAY 2024

UNIVERSITÉ DU QUÉBEC À MONTRÉAL

UNE ARCHITECTURE POUR LA MISE-À-L'ÉCHELLE DES CHAÎNES DE BLOCS DANS LE CONTEXTE DE L'INTERNET DES OBJETS

THÈSE

PRÉSENTÉE

COMME EXIGENCE PARTIELLE

DU DOCTORAT EN INFORMATIQUE

PAR

RIHAM ELSAADANY

MAI 2024

ACKNOWLEDGEMENTS

I'd like to express my thanks and gratitude to All Mighty God. Your guidance, love, and mercy have sustained me through both joyful and challenging times. Thank You for never forsaking me, for always extending Your hand of mercy, and for the countless ways You enrich my life. I'm always poor to Your kindness.

I'd like to thank Mr. Prof. Eng. Guy Begin, my research director who always gave me hope, specially whenever the road seemed too hard to handle and too dim to see. Thank you for every bit of help, for every moment of encouragement, for every single instruction and for every inspiration that you guided me with, one step at a time, throughout this long research journey, despite the long distance between two different countries and despite the obstacles along the road. I will always appreciate all your help, starting from my first master degree at UQAM and ending by my Ph.D. Thanks for always being there for me, I could never have made it without all your generous help and support.

I also would like to dearly thank my dear husband who supported me all along, who gave me every bit of help when I needed it the most, and who always pushed me forward to achieve my goals. Many thanks go to my dear kids who tolerated my absence while I was always working on my computer, and who always encouraged me to get going and to never give up. Many thanks for my dear brother who encouraged me to publish my research, with every bit of advice. And last but not least, I would like to send many thanks to my dear parents whose prayers were always a guiding star for me. All my prayers go to my dear father who just left our world before my dissertation, may he rest in peace. My dear family, your beautiful love always helped me find my serenity and my determination when things used to go crazy. Thanks for all your love, help and support, I love you back from the bottom of my heart.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AI	Artificial Intelligence.
ACL	Access Control List.
APs	Access Points.
AWS	Amazon Web Services.
BC	BlockChain.
BCHG	Backup CH Group.
BFT	Byzantine Fault Tolerance.
CH	Cluster Head.
CHAC	CH Address Chain.
CHORD	A Scalable Peer-to-peer Lookup Protocol.
CLT	Control Lower Threshold.
CP	Consensus Period.
DHT	Distributed Hash Table.
FIFO	First In First Out.
GN	Guardian Nodes.
IL	Local Immutable Ledger.
IoT	Internet of Things.
IoTDs	IoT Devices.
IP	Internet Protocol.
LSB	Lightweight Scalable BC.

MAC address	Media Access Control Address.
MLT	Management Lower Threshold.
MTUs	Maximum Transmission Units.
MUT	Management Upper Threshold.
ON	Overlay Network.
PBFT	Practical Byzantine Fault Tolerance.
PK	Public Key.
PoL	Proof of Learning.
PoS	Proof of Stake.
PoW	Proof of Work.
P2P	Peer-to-Peer.
QoS	Quality of Service.
RSA	Rivest–Shamir–Adleman.
SLAs	Service Level Agreements.
SSD	Self-Scaling Dynamic Architecture.
Tx	Transaction.
VCP	Virtual Cord Protocol.

RÉSUMÉ

Les dispositifs de l'internet des objets (IoTDs) sont omniprésents, aujourd'hui. Les IoTDs sont caractérisés par le grand nombre de transactions produites et leur susceptibilité aux problèmes de confidentialité et de sécurité. Les chaînes de blocs représentent un registre distribué immutable qui convient à toutes sortes de transactions sans intermédiaire. Les chaînes de blocs sont caractérisées par des long délais reliés au calculs mathématiques et au nombre des sauts (délais de propagations) nécessaires afin que les noeuds participants atteignent un consensus.

Nous proposons une solution d'organisation des nœuds qui pourra permettre aux chaînes de blocs de soutenir les IoTDs en regroupant tous les noeuds incluant les IoTDs. À cette fin, nous introduisons un concept multi-niveaux (forêt) qui sert à augmenter la mise à l'échelle et l'efficacité du consensus ainsi qu'à créer un équilibre entre la décentralisation du consensus et la lourdeur de gestion des noeuds. Au coeur du design, se trouve un algorithme d'accélération de consensus qui sert a limiter le nombre de sauts nécessaire entre les noeuds ainsi que le nombre de noeuds participant au consensus à un nombre borné logarithmiquement comparativement aux noeuds aux niveaux inférieurs. Ainsi, plus la forêt est profonde, plus elle devient évolutive et efficace. Nous proposons trois architectures, indépendantes des applications et des algorithmes de regroupement et de consensus, tirées de notre concept multi-niveaux: i) Self-Scalable Dynamic (SSD) (Auto-Évolutive Dynamique), ii) Partitioned (Partitionnée) et iii) Ring of Rings (Anneau d'anneaux). Nous choisissons d'implementer une des architectures pour valider notre concept multi-niveaux: l'architecture SSD. Dans l'architecture SSD, un seuil minimal est fixé par niveau pour la taille des groupes afin de garantir un consensus équitable conformément aux règles de décentralisation des chaînes de blocs. De même, un seuil maximal par niveau est établi pour faciliter une gestion efficace des nœuds du groupe par les noeuds gérants. Compte tenu de la nature dynamique des nœuds rejoignant et quittant le système, le respect de ces seuils maximums et minimums nécessite des ajustements dynamiques de la taille des groupes. Par conséquent, les groupes sont conçus pour se diviser ou fusionner en cascade. Ce redimensionnement dynamique affecte également l'architecture globale, entraînant son expansion ou sa contraction en termes de nombre de niveaux.

Les IoTDs se trouvent au plus bas niveau de la hiérarchie où les transactions sont générées et propagées vers les plus hauts niveaux de l'architecture. Ce trajet vertical, même s'il peut impliquer un certain nombre de niveaux à traverser, se fait dans un seul saut grâce à l'utilisation d'un mécanisme de repérage de noeuds conçu pour indiquer la position de chaque noeud ainsi que de ses parents aux niveaux supérieurs. La vérification des transactions ainsi que la vérification des blocs se font par un petit nombre de noeuds, ce qui réduit le temps des propagations, spécialement si l'on compare au temps nécessaire à faire les mêmes verifications dans le cas du modèle des chaînes de blocs traditionnel ou même un modèle à deux-niveaux. La colocalité des noeuds participant dans une même transaction dans un même arbre de la forêt a un grand effet sur l'efficacité du modèle car le nombre de sauts nécessaire pour verifier les transactions diminue significativement. Le temps de consensus est réduit à $O(1)$ plus une constante fois l'ordre du délai de propagation maximal vers les noeuds les plus élevés. La constante est égale à trois, sans colocalité au pire cas; tandis qu'avec colocalité, la constante est égale à deux, dans le meilleur cas.

Les simulations effectuées mettent à l'épreuve trois modèles des chaînes de blocs: le modèle traditionnel plat, le modèle à deux-niveaux, ainsi que notre modèle SSD. Les résultats des simulations montrent que bien que le modèle à deux niveaux ait surpassé le modèle traditionnel plat en termes de temps de consensus de 23,7%, le modèle SSD a surpassé le modèle à deux niveaux et le modèle plat (lorsque le modèle plat a 6% des nœuds en tant que nœuds générateurs de blocs) de 198%. L'avantage augmente à mesure que le nombre de noeuds générants les blocs augmente dans le modèle plat, atteignant 2677% lorsque le modèle plat a 88,6% des nœuds en tant que nœuds générateurs de blocs. Nous avons aussi effectué des simulations pour le modèle SSD ayant des différents nombres de noeuds générants les blocs afin de valider le concept de colocalité qui s'est avéré prometteur. Tous nos résultats s'accordent bien avec nos hypothèses et représentent une preuve de concept valide.

Les designs et concepts proposés peuvent servir de solutions à usage général pour la mise à l'échelle, le repérage de nœuds pair-à-pair et le routage de messages pour les systèmes distribués ainsi que l'équilibrage de charge pour les systèmes en grappes.

Mots Clés– chaînes de blocs, IoT, Consensus, Mise à l'Échelle, Équilibrage de Charge, Décentralisation, Sécurité, Lourdeur de Gestion, Architecture Multi-Niveaux, Repérage des Noeuds, Acheminement des Messages.

ABSTRACT

Nowadays, Internet of Things devices (IoTDs) are ubiquitous. IoTDs are characterized by generating a large number of transactions per second and by being vulnerable to security and privacy issues. Blockchain technology represents a distributed immutable ledger designed to support various transactions and achieve consensus without the need for intermediaries. A blockchain ledger typically requires a significant amount of time to reach consensus due to the complexity of computations and the number of hops (propagation delays) needed for transactions (Tx) and blocks to reach all nodes. This presents a challenge for blockchain support in IoTDs.

We propose a node-organizing solution that allows the use of blockchain into the IoT ecosystem by introducing a multi-tiered (forest) design concept that is based on clustering all nodes including the IoTDs. The concept aims to enhance blockchain scalability and consensus efficiency while maintaining a balanced distribution of node management overhead and consensus decentralization. At its core, the design incorporates a consensus-accelerating algorithm that reduces the number of hops among nodes by restricting consensus participation to only those nodes situated at the highest level of the structure, thus to a number of nodes logarithmically bounded compared to the number of nodes located at the levels below. Hence, the deeper the forest, the more scalable and efficient it becomes. Drawing from the multi-tiered concept, we introduce three blockchain architecture designs that are agnostic to applications, as well as to clustering and consensus algorithms: i) Self-Scalable Dynamic (SSD), ii) Partitioned and iii) Ring of Rings architectures. We choose to implement and test one of the architectures in order to validate our multi-tiered concept: the SSD architecture. In SSD architecture, a minimum value is set per level for cluster sizes to ensure fair consensus in accordance with blockchain decentralization rule. Similarly, a maximum value per level is established to facilitate effective cluster node management within the hierarchy. Given the dynamic nature of nodes joining and leaving the system, adhering to these minimum and maximum values necessitates adjustments in cluster sizes. Consequently, clusters are designed to split and/or merge in a cascading manner. This dynamic resizing also affects the overall architecture, leading to its expansion or contraction in terms of the number of levels.

The IoTDs are located at the bottom of the architecture, and transactions are generated and propagated (by the IoTDs) upwards towards the highest level of the architecture. This upward routing is performed in only one hop even though it may involve a number of levels to be travelled, thanks to the use of a node-lookup mechanism specially designed to specify the position of each node and that of its parents at the levels above. Transaction and block verifications are done by a smaller number of nodes compared to a traditional blockchain model or even a two-tiered model. The co-existence of IoTD nodes participating in the same Tx within the same tree of the forest (colocality) significantly enhances the model's efficiency by reducing the number of hops required for Tx verification. Consensus time is reduced to $O(1)$ plus $c \cdot O(maximum\, propagation\, delay\, towards\, topmost\, nodes)$, c is a constant s.t. without colocality, $c \leq 3$ worst case; while with colocality, $c \leq 2$, best case.

Simulations are built to test three blockchain models: the flat traditional model, the two-tiered model as well as the SSD multi-tiered model. The simulation results show that though the two-tiered model outperformed the flat model in terms of the consensus time by 23.7%, the SSD model outperformed the two-tiered model and the flat model (when the flat model has 6% of the nodes as block generating nodes) by 198%. The advantage gets larger as the number of block generating nodes increases in the flat model, reaching 2677% when the flat model has 88.6% of the nodes as block generating nodes. We also performed simulations for the SSD model using different numbers of block-generating nodes in order to validate the colocality concept, which proved to be promising. All our results align well with our hypotheses and represent a valid proof of concept.

The proposed designs and concepts may serve as general-purpose solutions for scalability; peer-to-peer node-lookup and message-routing for distributed systems as well as load-balancing for clustered systems.

Keywords– Blockchain, IoT, Consensus, Scalability, Load balance, Decentralization, Security, Management Overhead, Multi-Tiered Architecture, Node Look-up, Message Routing.

INTRODUCTION

Nowadays, we live in a world of pervasive Internet of Things (IoT). The IoT represents a paradigm that provides users with a wide range of services by integrating interconnected sensors and communication functionalities into various devices. These devices include embedded systems, mobile equipment, wearable computers, and more (Cor, 2018). IoT devices exchange a considerable amount of sensitive data and monetary transactions that may reveal personal behavior and preferences or even lead to a construction of a detailed personal profile. Unfortunately, the IoT devices (IoTDs) have limited resources to deal with such crucial privacy and security issues.

BlockChain (BC) technology, which is a peer-to-peer distributed ledger technology originally developed to support cryptocurrency, can be leveraged to support any type of transaction (Tx), without intermediary. BC nodes achieve consensus to append a new block of Tx into a Merkle tree. In Figure 0.1, there is an example of a binary Merkle Tree where a leaf node contains the hash of the data, while a non-leaf node contains the hash of all its children nodes. Hence the Merkle tree root contains the hash of the whole tree. BC is based on the idea of blocks chained in a Merkle tree as in Figure 0.2: the header of each block contains the hash of the previous blocks as well as the root of the Merkle tree that contains the hash of current block's data. Current block's header is in turn hashed into next block's header to form an immutable chain of blocks. Thus in BC, transactions are stored in blocks, and the blocks are stored in a database. The Tx parties agree over the state of the database through a consensus mechanism. The transactions are produced by non-trusted Tx parties without a centralized authority. The transactions are timestamped for data audit and the blocks are chained in a Merkle tree, which makes them tamper proof. Since an attacker has to control at least 51% of the network computing power (Lunardi *et al.*, 2022) (of Proof of Work-based[1] BC) in order to attempt to compromise the BC network, it is impractical to launch an attack on BC. This immutable nature along with the associated security and auditibility benefits makes BC a potential solution for IoT challenges. However BC has limited scalability (the achievable number of transactions per second is limited[2]), significant bandwidth overhead (too many communications) and large delays (in order to achieve consensus), which renders BC applications for IoTDs to be very challenging. Previous work has been done to address such challenges, however several issues are yet to be addressed regarding BC scalability aspects, namely, achieving a balance between establishing a decentralized control while reducing the node

[1] Proof of Work is a consensus algorithm that requires network members to expend effort in solving a cryptographic puzzle.

[2] IoT requires millions of transactions per second while current BC provides hundreds or thousands of transactions per seconds

management overhead.

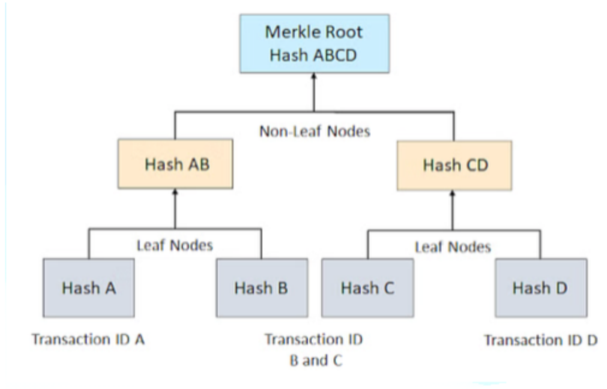

Figure 0.1 A Binary Merkle Tree

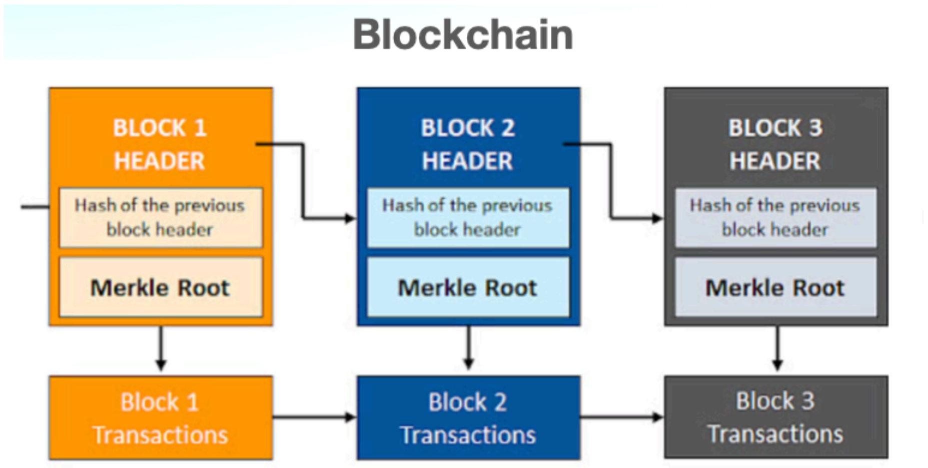

Figure 0.2 Blockchain

The thesis aims to introduce a new node-organizing multi-tiered design concept represented by a number of design architecture options. The concept is to address current BC models' scalability and consensus efficiency challenges. One of the design options (self-scaling distributed (SSD) architecture) is chosen to be further studied and simulated in order to be compared to the literature BC models as one way to showcase the comparative scalability and efficiency of our concept.

Our design concept is inspired by two-tiered node-organizing architectures which aim to improve scalability and efficiency of flat traditional BC chain models. In Flat models, all nodes are located at the same level and they may generate transactions and build blocks. Nonetheless, in flat models, all nodes are visited to achieve consensus regardless of the necessity, which creates considerable amounts of traffic overheads (due to the number of broadcasts involved) as well as scalability limitations not well suited for IoT high transaction rate.

An example of a two-tiered BC model that contributed to our inspiration is the Lightweight Scalable BC (LSB) architecture introduced by Dorri *et al.* (Dorri *et al.*, 2017c; Jurdak *et al.*, 2017; Dorri *et al.*, 2017b; Dorri *et al.*, 2016). In LSB, the smart home, where the IoTDs reside, constitutes the first tier (at a bottom level), and the overlay network nodes located at a higher level constitute the second tier, where the public BC resides. Our design concept is also inspired by BlockSim (Alharby et Moorsel, 2020), incorporating IoTDs located at a lower level and gateways that manage these IoTDs situated at a higher level.

In two-tiered models, we have pinpointed several scalability issues representing significant obstacles that may impede two-tiered models from achieving complete scalability for accommodating real-world applications for today's IoTDs and efficiently achieving consensus.

In our proposed design model, we aim to introduce an elastic design that can seamlessly adjust to accommodate the fluctuating presence of IoTDs within the system, addressing concerns about scalability and flexibility. We also target an enhancement in consensus efficiency by minimizing the number of hops and nodes required, especially as IoTD numbers increase. Our focus remains on managing node overhead efficiently, aligning it with the system's capacity. Ensuring adherence to decentralization principles across all nodes is another key goal. Additionally, we strive to maintain a delicate balance between node management overhead and consensus decentralization throughout our design process.

To the best of our knowledge, the fundamental question of how to design a distributed, hierarchical, dynamically self-scaling, and self-adjusting architecture for IoT-based blockchain has not been explored in the literature so far. Additionally, we are not aware of any consensus-accelerating protocol, positioning mechanism, or self-scaling algorithm that can assist with multi-tiered architectures while meeting the needs of blockchain-based IoT.

0.1 Motivation

The proposed designs and concepts may serve as general-purpose solutions for:

1. scalability, namely the multi-tiered design;
2. peer-to-peer node-lookup and message-routing for distributed systems, namely the positioning mechanism;

3. load-balancing for clustered systems, namely the self-scaling algorithm.

Application scenarios of IoT systems that produce transactions (usually monetary transactions) at a high rate through non-trusted Tx parties, and that may leverage BC are :

1. Supply Chain,
2. Smart vehicles,
3. Healthcare Systems,
4. Smart Energy Grid.

In supply chain, specially in industries like food and pharmaceuticals, IoT sensors can track the movement and temperature of products throughout the supply chain. The sensors do not trust each other. They generate a considerable amount of transactions at a high rate to track all the food or pharmaceutical products. Each Tx (e.g., product movement from one point to another or product purchase) can be recorded in BC for transparency and traceability, ensuring product authenticity and quality control.

In autonomous vehicles, with the rise of autonomous vehicles, there is a massive amount of data generated and transactions occurring between vehicles, infrastructure, and other stakeholders (e.g., toll booths, charging stations). The vehicles and the stockholders do not trust each other and they produce transactions at a high rate. BC can be used to securely record and manage these transactions, such as payments for tolls, parking fees, or energy consumption.

In healthcare systems, IoT devices (IoTDs) such as wearable health monitors can continuously collect and transmit patient data and monetary transactions. The monitors do not trust each other and they produce transactions at a high rate. BC can securely store this data, ensuring privacy and integrity, while also enabling patients to control access to their health information. Transactions related to data access or sharing can be recorded on the BC. Also monetary transactions including payments for health care service providers and insurance companies can be recorded on the BC for privacy and security.

In smart energy grids, IoTDs such as smart meters can monitor energy consumption in real-time. The meters do not trust each other. As numerous transactions occur at a high rate when energy is bought and sold within the grid, BC can be used to securely record and settle these transactions, ensuring transparency and preventing fraud.

Thus BC is important for IoT. With the rise of today's vast use of IoTDs in the aforementioned scenarios as well as in pretty much every aspect of life, there is a need for a BC that helps ensure privacy, transparency, audibility and security of the transactions produced by IoTDs, while being scalable and efficient enough to support the enormous amount of transactions produced at a high rate. Hence, there is a need to ensure BC scalability to fulfill current and expected future amounts of transactions produced by IoTDs at a high Tx rate.

The following chapters are organized as follows. In Chapter 1, we present related studies found in the literature. In Chapter 2, we introduce the research problem. In Chapter 3, we state our research objectives. In Chapter 4, we showcase our multi-tiered concept design (the model), our consensus-accelerating algorithm as well as a number of proposed architecture design options to achieve the multi-tiered design concept. We also introduce one of the proposed architectures in details along with our proposed node look-up positioning mechanism and our proposed self-scaling (load balancing) algorithm that elastically shrinks and expands the structure. In Chapter 5, we present our methodology. In Chapter 6, we present the theoretical analysis of our design, including the theoritical asymptotic analysis of the self-scaling algorithm independently from the simulations. In Chapter 7, we introduce the details of our simulations. In chapter 8, we present the practical asymptotic analysis of the self-scaling algorithm while taking into consideration the operations required to manipulate the complex data structures that holds the node and cluster data as implemented in our simulations. In the same chapter, we compare both asymptotic analysis of the self-scaling algorithm (the theoretical and the practical), and we present our simulations' results along with the results' interpretations. Finally, in Chapter 9, we conclude our study and we state our study's limitations as well as future works.

CHAPTER 1

LITERATURE REVIEW

IoT devices (IoTDs) may range from the size of a grain of sand sensor to the size of a car. IoTDs are usually characterized by having limited CPU capabilities and battery lifetime. IoT operations take place at very high rates and may include personal, financial and other sensitive data that need to be kept private. IoT operations are diverse and vary according to the nature of the IoTD. For example a smart fridge operations might range from interacting with the home owner phone to interacting with other smart devices around the house in order to alleviate the load on resources utilization such as the electric power, gas, water or sewers, during a specific time of day. Such operations might be controlled by a smart phone, tablet or computer, and thus analyzing them might lead to track everyday routine patterns of the home owner.

We start our literature review with a brief background about BC, pinpointing strengths and weaknesses of a number of BC consensus algorithms. Then, we introduce previous BC-based IoT work focusing on scalability. Finally, we introduce BC-based IoT works, classified based on their node organization, specifying how they relate to our proposed design.

1.1 Background

A BC is a distributed database that maintains an immutable public ledger in the form of a continuously growing list of ordered records called blocks (Androulaki *et al.*, 2018; Buterin, 2021; Nakamoto, 2009; Kosba *et al.*, 2016; Valenta et Sandner, 2017; Eyal *et al.*, 2015; Iro, 2018). A BC is usually implemented as a Peer-to-Peer (P2P) overlay network (ON) where the nodes are BC participants. P2P systems and applications are distributed systems without any centralized control or hierarchical organization, where the software running at each node is equivalent in functionality (Stoica *et al.*, 2001). Some of BC nodes act as miners that accumulate and validate transactions (Tx), which are messages between nodes, and generate blocks of such Tx to be added to the chain. The consensus defines the rules that determine how the next block is created and which Tx are to be included. Consensus is established through a consensus protocol, such as Proof of Work (PoW), Proof of Stake (PoS), Proof of Learning (PoL) (Bravo-Marquez *et al.*, 2019), Byzantine Fault Tolerance (BFT) (Clement *et al.*, 2009; Sousa *et al.*, 2018), Practical Byzantine Fault Tolerance (PBFT) (Vukolić, 2016), etc. A PoW-based block of Tx, is created by one network miner who is randomly given a chance to solve a computationally expensive puzzle that can be easily verified by all participating miners in order to solidify a network consensus (Cachin et Vukolić, 2017). The Tx are produced and may be verified by the non-trusted participants without involving any centralized authority (Kalis et Belloum, 2018). Tx are timestamped in order to allow data audit, and blocks are chained in a Merkle tree (Dorri *et al.*, 2018; Bravo-Marquez *et al.*, 2019; Gupta et Sadoghi, 2018; Wilkinson *et al.*, 2016; Bogdanov *et al.*, 2018) in order to prevent data tampering. Nodes might appear close by in the overlay network (ON) whereas in the real, physical network, they might be thousands of miles away, with several intermediary servers and routers in between, causing uncontrollable delays. Due to this reason, BC is not well suited for IoTDs (Singh et Duggal, 2022).

BC's distributed nature means that there is no master computer to manage the ledger, but rather, the participants have a copy (or sub-copy) of the ever-growing chain (usually the BC ledger is public and a copy of it resides on each participating node). And it also means the elimination of single points of failures as well as the presence of P2P messaging, distributed file sharing, and autonomous device coordination (Prabhu et Prabhu, 2017; Atlam *et al.*,).

Thus BC can provide flexibility and auditibility benefits to help secure IoTD interactions. However BC-based IoT implementations lack several aspects required to allow quick response time and/or scalability for the ever-expanding IoT environment (Raza *et al.*, 2017). Other issues, such as the complexity of operations required for reaching consensus, the latency of asynchronous networking, the inability to manage an ever-growing ledger, are challenges for resource-constrained IoTDs. Our research aims to address some of these issues.

The consensus most commonly used in BC are BFT and PoW. In BFT, a group of replicas try to reach a consensus on the execution order of Tx and blocks as well as the results of client (node) requests even though a faulty subset of these replicas may behave arbitrarily. As long as $3f + 1$ replicas behave correctly, the agreement scheme can tolerate up to f faulty nodes. In PoW, participants choose nonces and calculate hashes until the resulting hash is lower than a certain threshold, which is a very expensive computation process. BFT suits permissioned BC where all participants are known in advance, and no new participants are allowed to join in BC (e.g. Tendermint (Buchman, 2016)), which is not the case for BC-based IoTDs. In permissionless BC, BFT does not scale and does not mitigate Sybil attacks where an attacker participates in BC using several identities. Along with good reconfigurability (allowing nodes to join and leave the network at any time), a high throughput and a large consensus group that can tolerate a high number of failures are desirable properties of BFT. However, these properties are often in a direct conflict. Most BFT protocols achieve high throughput but are limited to small groups of participants whose reconfiguration is not easily achievable; while PoW supports a large number of participants with flexible reconfigurability however with very limited throughput. Our study does not aim to introduce a new consensus protocol however it aims to improve what is offered by the literature.

(Rüsch, 2018) proposes improvements for a permissioned BC, the Hyperledger. The design enables transparent access to BFT clusters (leveraging a proxy) without the assumption that the nodes will behave correctly, leaving the computation to more powerful replicas while the peers are kept computation free to match their resource capabilities. This renders the Hyperledger more resilient to byzantine faults with a more efficient communication layer that enables data center machines to access each others' memory without involving the operating system.

1.2 Related Works

1.2.1 Related Works - Classification based on Scalability

Our work focuses on improving BC-based IoT and their scalability issues in terms of the number of IoTDs supported by the system and the Tx throughput. Based on these issues, (Croman *et al.*, 2016) provide a decomposition of BC systems into a set of abstraction layers (planes) which allows for a structured classification of the scalability issues of each plane, pinpointing why and where they emerge along with a number of recommended solution approaches and protocol ideas.

A number of BC-based IoT implementations focus on scalability from different points of view, ranging from the number of IoTDs supported by the system, to Tx throughput produced, to the number of Tx communicated through low power networks. According to (Croman *et al.*, 2016), BC computing complexity to reach a consensus leads to excessive Tx processing time. This problem is addressed in the literature by either building a custom-made BC that implements the consensus in a simplified approach, by deploying a pre-existing scaled-consensus BC platform or by removing the consensus computations off the IoTDs' shoulders.

(Wang *et al.*, 2019) and (Raza *et al.*, 2017) provide surveys on BC for IoT, evaluating its suitability and detailing how adaptations/enhancements should be elaborated on consensus protocols and data structures before they are integrated into IoT.

An example of a custom-made BC focusing on scaling the number of IoTDs supported as well as the throughput obtained is the implementation of (Dorri *et al.*, 2017a; Dorri *et al.*, 2017c; Jurdak *et al.*, 2017; Dorri *et al.*, 2017b; Dorri *et al.*, 2016), which proposes a Lightweight Scalable BC (LSB). LSB is a two tiered distributed architecture. The smart-home network with a Local Immutable Ledger (IL) is separated from the ON that manages the public BC. It is a node-organizing tiered design where IoTDs located at the bottom level are managed by Cluster Heads (CHs) located at the top level and performing consensus. The design aims to address BC scalability issues and to reduce the time to approve both Tx and blocks through clustering IoTDs. The BC ledger contains the transactions created per participant (requestee), and the consensus is simplified by implementing a trust system where some of the Tx can skip verifications in order to accelerate the block creation process. Our work aims to take that implementation to the next level by adding more levels to that two-tiered architecture, thus further enhancing the scalability and the efficiency.

(Heo *et al.*, 2022a) and (Heo *et al.*, 2022b) propose a multi-level of cashing to solve the storage problem introduced by IoTD-based BC, by storage optimisation based on data access patterns. Each node initially stores new blocks broadcast from miners in its local storage. Over time, block headers and transactions that have not been accessed are replaced with their hash values by removing the data itself. These studies aim to reduce BC storage requirements, while maintaining consistency and minimizing query costs.

An implementation of an existing BC platform naturally characterized by a scaled throughput is introduced in (Huh *et al.*, 2017) as a three device implementation on Ethereum (Buterin, 2021), leveraging its smart contracts (Delmolino *et al.*, 2016; Kosba *et al.*, 2016). The design manages keys using RSA public key crypto systems. Public keys are stored in Ethereum and private keys are saved on individual devices.

Another implementation to solve the problems related to the integration between IoT and BC by using smart-contract in the healthcare field is (Chendeb *et al.*, 2020). The architecture introduced is based on publish/subscribe model and an access control and management scheme for a multi-layered system that has the patient layer (the IoT layer), the authority layer (access all data) and the cloud provider layer (gateways that mine using the smart contracts).

(Schiller *et al.*, 2019) introduces various methods based on scaling the number of Tx communicated as Maximum Transmission Units (MTUs), to improve BC-based IoT built over a low power network using a LoRaWAN communication protocol. The system performance, the reliability of transport schemes, and the energy efficiency of the overall system are evaluated. The scheme provides a trade-off between submitted Tx count, packet loss, and energy efficiency. Data integrity is obtained by providing a reliable data transmission scheme from IoTDs to the BC.

LoRaWAN network architecture is deployed in a star-of-stars topology in which gateways relay messages between end-devices and a central network server. The gateways are connected to the network server via standard IP connections and act as a transparent bridge, simply converting RF packets to IP packets and vice versa. The wireless communication takes advantage of the Long Range characteristics of the LoRaWAN physical layer, allowing a single-hop link between the end-device and one or many gateways. In LoRaWAN, there is no need to achieve consensus between the network servers, spanning all of them. Furthermore, the nodes typically have a star-of-stars topology, making it challenging to reorganize them into a more flexible structure like a forest. Additionally, each node has a specific role dictated by its specifications, establishing rigid relationships among the central server, gateways and end-devices. Consequently, nodes may not exchange roles or positions easily. For these reasons, our design may not offer significant benefits to LoRaWAN.

In Ethereum, there are four types of nodes: miners, full nodes, light nodes and archive nodes. Scaling our architecture is based on the flexibility of the nodes to exchange roles and positions, which is not possible for Ethereum nodes since the nodes have different specifications to allow each type of node to perform its specific functionality.

Other approaches to increase the throughput, which is directly affected by the Tx processing time for consensus, follow the trend of assigning the consensus computations to more powerful resources, which falls within the focus of our study. An architecture to decentralize the ledger by assigning the Proof of Work to a number of edge servers acting as miners that validate the produced Tx is proposed in (Angin *et al.*, 2018). The Tx are sent towards a number of powerful servers on the cloud to be appended to the ledger. Keeping the ledger on powerful servers helps maintain an integrity-preserving immutable ledger replicated at multiple sites, and thus overcoming the IoTD resource limitations, and increasing the ledger scalability.

In our work, we focus on simplifying the consensus protocol using clustering as well as studying its suitability for IoTDs. In (Sağırlar *et al.*, 2018), the different consensus characteristics of BFT and PoW are exploited in a hybrid BC-based IoT. As it was observed that PoW BCs containing few hundreds of geographically close

IoTDs have high performance (Tx throughputs) and low block propagation delays, the first step in Hybrid-BC consists of generating multiple PoW BCs. The design uses PoW for a few nodes within small groups, and then applies BFT to connect between the groups.

A number of works consider the suitability of consensus protocols. In (Mackenzie *et al.*, 2018), a number of BC consensus protocols are assessed against an IoT requirement framework in order to identify the strengths and weaknesses as the consensus protocols get implemented into the IoT environment.

Having introduced the first classification of our literature, let us consider some limitations of the introduced designs. (Schiller *et al.*, 2019) only address Tx transmission scalability considering real and random locations of LoRa nodes, while node locality, where nodes that are most frequently participating in common Tx may get logically closer over time to further increase the system performance was not exploited. Our work will study the node locality within our model. (Angin *et al.*, 2018) partially addresses BC storage scalability. Nevertheless cloud servers are potential single points of failure and also due to the high complexity of the computations required to ensure PoW security, this makes it prohibitive (in terms of energy consumption) for the resource contained devices to participate in the block formation process, so AWS t2 micro nodes are unable to compute PoW. In (Huh *et al.*, 2017), the produced throughput is only one Tx per twelve seconds which is too small for today's IoTD needs. Also the lack of a lightweight client is a challenge that cannot be addressed by using an expensive external storage on IoTDs or by using insecure third party proxies.Their approach will be mimicked by one of our proposed model. (Dorri *et al.*, 2017a) and (Chendeb *et al.*, 2020) attempt to address the scalability question in terms of the number of IoTDs supported by replacing the complex computation by a simplified consensus mechanism and by building a multi-tiered architecture based on an access control scheme. Also (Heo *et al.*, 2022a) and (Heo *et al.*, 2022b) attempts to address the storage optimization for BC based on the IoTD on-line availability degree. however scalability in those studies was limited because of traffic overhead due to the almost flat design also due to overlooking the traffic overhead and the node manageability issues that will eventually take place in BC applying such designs.Additionally, we can observe that the consensus plane issues are partially addressed by delegating BC interactions to cloud servers or on faster BC platforms such as Ethereum.

1.2.2 Related Works - Classification based on Node Organization

An alternative categorization for related works may be conducted according to the node-organization adapted in each design. In this subsection, we review designed architectures that are relevant to our multi-tiered concept from a node-organization perspective. The fundamental scalability and efficiency issues are some of the main obstacles to overcome when implementing BC into IoT ecosystems. Scalability is evaluated based on different criteria including the number of supported IoTDs and Tx throughput.

An example two tiered architecture is Lightweight Scalable BC (LSB)(Dorri *et al.*, 2017a) where only IoTDs are clustered and consensus is simplified by putting in place a trust system that allows some Tx to skip verifications, hence, improving consensus time. An implementation for a two-tiered BC architecture where IoTDs located at the bottom level produce transactions while gateways, which are CHs that are located at the top level, interact with BC using a modeled Proof of Work (PoW)(Yu, 2023a; Qi *et al.*, 2023) consensus, is proposed by (Alharby et Moorsel, 2020; Alharby et Moorsel, 2022).

IOTA Tangle(IOT, 2024) is a two-tiered architecture, a cryptocurrency system for IoT based on directed acyclic graph (DAG) structure. IOTA does not use miners to validate transactions, instead, nodes that issue a new Tx on the network must approve two previous transactions. It supports high concurrency, scalability, and zero Tx fees.

A multi-tiered implementation of an existing BC platform naturally characterized by a scaled throughput and low latency is introduced in (Albulayhi et Alsukayti, 2023) as an architecture built on Ethereum (But, 2021), leveraging its smart contracts (Tanwar *et al.*, 2023). A multi-tiered cellular network architecture is proposed by (Alhusayni *et al.*, 2023) and is based on clustering IoTDs and assigning cellular devices at the level above as local BCs to connect to cellular base stations situated at a top level as a global BC. The latter implements decentralized BC mechanisms. A similar architecture is proposed by (Pajooh *et al.*, 2021).

A multi-tiered architecture is proposed by (Oktian *et al.*, 2020) where a core engine, located at a top layer, performs consensus and supervises underlying instances of sub-engines (for payment, compute, and storage) which connect to IoTDs through APIs.

The aforementioned two-tiered designs have the top level prone to face the same scalability issues as in a regular flat model. The designs are rigid s.t. there is no possibility to scale and add more levels or split clusters (to fit more IoTDs) and there is a strict distinction between the IoTDs and CHs. Consequently, they cannot exchange positions or roles, hindering scalability to meet the demands of today's IoT. In these designs, the structure is pre-built before the consensus mechanism is initiated and can not be modified once built, which may not always align with the requirements of a dynamically operating BC that must continually expand on the fly. Decentralization is not achieved in IOTA since it relies on a single coordinator node to achieve consensus. In two-tiered models, we have identified several scalability issues stemming from the rigidity of the design, which lacks the flexibility to accommodate additional levels. We have also identified missed opportunities for potential improvements in consensus efficiency that can be further addressed. We have identified issues regarding node management overhead, where the capacity of managing nodes should not be less than the overhead of the managed nodes. These aspects are necessary key points characterizing a hierarchical design. We have identified issues concerning consensus decentralization, which is a fundamental requirement for correct blockchain functionality. All these issues are significant factors that may hinder two-tiered models from achieving full scalability to accommodate realistic applications for today's IoTDs and swiftly reaching consensus.

In the multi-tiered architectures, tiers correspond to conceptual layers to implement BC rather than a multi-tiered node organization where nodes may span across several levels while the number of levels may be increased and clusters may split to allow for scalability. Thus they face the same issues mentioned earlier. Lastly, all aforementioned architectures overlook creating a balance between node traffic overhead and decentralization, a main requirement for BC security and operability.

The existing BC designs (some of which discussed above) suffer from the following drawbacks:

1. limited scalability,
2. low throughput,
3. consensus inefficiency, and
4. lack of balance between node manageability and decentralization.

In the following chapter, we present the research problems addressed in the literature and those still to be addressed.

CHAPTER 2
RESEARCH PROBLEM

2.1 The Problem in General

IoT devices (IoTDs) are heterogeneous non-trusted Tx parties, typically having limited resources and often experiencing high Tx rates. IoTDs leverage BC as a potential trust and security solution due to its immutability, security and decentralization. In BC, all blocks are verified by all nodes which leads to significant broadcast traffic and processing overheads that increase quadratically with the number of nodes in the system. This leads to long delays, as verifying one Tx may take the order of minutes (10 minutes in the case of Bitcoin) which is not suitable for IoT interactions that are almost instant, according to each application's Quality of Service (QoS) and Service Level Agreements (SLAs). Moreover, most BC have a limited throughput in terms of the number of transactions that may be stored in the ledger per second (in $O(10000)$), which is not suitable for IoT applications that are characterized by producing millions of transactions per second. Thus owing to its heavy bandwidth overhead, latency and limited throughput, classic BC is not well suited for IoTDs. One way to calculate consensus time is to count the number of hops a Tx needs to reach BC within the overlay network. In this research, we assess how BC node organization affects overlay network delays, assuming that these effects will eventually impact the actual physical network.

To pinpoint the problem, let's consider the current BC node-organizing models:

1. In the typical one-tiered (flat) model where all nodes are located at the same level and may generate transactions and build blocks, all nodes must be visited by all transactions and blocks to perform Tx and block verifications. This model is characterized by a significant number of broadcasts, large consensus time and limited throughput due to the fact that all nodes must be visited regardless of the necessity in order to achieve consensus. This creates scalability limitations, increased traffic overhead and increased consensus delays.
2. In the two-tiered model, where block-building nodes (referred to as *Cluster Heads (CHs)*) located on the top level manage the clustered Tx-generating nodes (simply referred to as *IoTDs*), located at a lower level, there is no possibility to expand the design vertically by adding more levels. Expanding the design may only be horizontally by adding more cluster heads at the top level. However adding extra CHs at the top level to accommodate for more IoTDs, will saturate that level at some point in

time. Thus the two-tiered model suffers from the same drawbacks faced by flat model, only reflected at the top level.

In terms of security, it's essential to maintain the blockchain's true characteristic of proper block chaining while ensuring decentralization of control to prevent a small number of nodes from seizing control of the chain. This should be achieved while maintaining efficient processing of transactions, which could number in the millions as generated by IoTDs. Scaling up for such a considerable number of Tx produces a great amount of Traffic and workload overhead that needs to be addressed while preserving the decentralization of control as well as the efficiency, which is a great challenge.

BC ledger can be duplicated on all participating nodes or distributed on other servers such as the cloud. Each such implementation choice has its pros and cons. Though server-based implementations might represent a solution to account for the heterogeneity (in terms of manufacturing, capability, functionality, etc...) as well as the storage, processing and energy limitations of IoTDs, they also represent a reliability problem since servers are considered as single points of failure, also there is a chance to lose some of the data while being collected by the server.

2.2 The Specific Research Problem

Let us first consider the scalability limitations reported in the literature in terms of Croman's (Croman *et al.*, 2016) network, consensus and storage planes.

At the network plane, the existing solutions need an optimization for the routing process and the distance travelled to locate Tx parties in order to timely create and verify Tx. There is a difference between the distance between two nodes in the physical network versus the distance between the same nodes in the overlay network. In the physical network, the distance between two nodes is measured in terms of miles or thousand of miles while in the overlay network (a virtual network), the distance between the same nodes is measured as only one hop, no matter how large the distance is in the physical network. A mechanism is needed for associating a Tx to the Tx-generating and Tx-visited nodes while limiting the use of non-scalable solutions such as lengthy Access Control List-like lists. A mechanism is also needed for locating IoTDs managed by CHs more efficiently, while mapping the logical locations of each node, within the design abstraction layers.

A typical BC is a flat model where all Tx and blocks have to be verified by all nodes in order to append blocks to BC. Though some studies in the literature improved the consensus plane by delegating the verification process to only CHs that manage clustered IoTDs, they propose mostly-flat models, with at most one layer of CHs. As more nodes join in the clusters, the number of such CHs has to increase at some point and so will their workload (packet overhead and end-to-end delays). The workload depends on the number of nodes that exist in a cluster. If a large number of CHs manage such dense clusters, we'll end up in a situation similar to a classic BC, with latency only reflected at the CHs level, creating bottlenecks, and consequently affecting the consensus (block creation and validation) time which will eventually become too long for fast IoT Tx.

The consensus plane issues are partially addressed in the literature (Angin *et al.*, 2018; Huh *et al.*, 2017). Nevertheless what is needed is a mechanism that enhances scalability and accelerates the consensus, that can be both lightweight while liberating most of the IoTDs from heavy computations.

The storage plane issues were addressed in the literature (Oktian *et al.*, 2020; Heo *et al.*, 2022a; Heo *et al.*, 2022b) and may be further explored as BC throughput increases down the road. Since the focus of our research is mainly to provide a proof of concept for our proposed multi-tiered design scalability and efficiency, we choose not to address BC storage plane issues in our study.

None of the BC models from the literature considered node manageability/overhead or the necessity to enforce a balance between consensus decentralization which is at the heart of BC, on one hand, and the ease of cluster node management, on the other hand. The node management overhead increases when one CH manages a number of nodes that exceeds its capacity, preventing the CH from efficiently performing its tasks. Control centralization takes place when only a small number of nodes decide for approving a Tx or a block to be appended to BC, which violates the fundamental laws of BC security.

Thus existing node organizing models suffer from:

1. limited scalability,
2. low throughput,
3. consensus inefficiency,

Additionally, none of the existing models considers the necessity of following aspects:

1. a mechanism to logically locate nodes, specially within a tow-tiered architecture,
2. node proximity considerations: two nodes participating in the same Tx are managed by the same cluster head. The cluster head can then efficiently verify the transactions, thereby accelerating the consensus process. This aspect, though neglected, is one of the main virtues to leverage a two-tiered architecture at the first place.
3. decentralization considerations,
4. node management overhead considerations.

2.2.1 The Solution Hypotheses

In the following, we list our project hypotheses ordered by decreasing priority.

1. Only some CHs may append Tx to blocks while only IoTDs may generate Tx[1].
2. Tx may be simple with one participant node or multiple participants (multi-signature).
3. Node failure may be mitigated by enabling nodes to exchange roles and positions.

We envision that a comprehensive solution, addressing all requirements, could adopt a multi-tiered architecture. In this architecture, if both nodes and CHs would be clustered, then BC interactions would be delegated to only selected CHs (topmost level CHs), drawing inspiration from two-tiered models.

The architecture would aim to introduce self-sufficiency: Instead of relying solely on the highest-ranking CHs to conduct the consensus, one might propose employing resilient machines, similar to the approach used in Ethereum. In case of using robust machines to perform the consensus, the more IoTDs join in the more robust machines would be needed. Limiting the number of those machines while more IoTDs join in would lead to having very dense clusters managed by those machines. Both solutions would lead to

[1] Although our proposed design could initially be implemented under the assumption that all nodes generate transactions, allowing Cluster Heads (CHs) to generate transactions while performing consensus or maintaining the multi-tiered structure, we have chosen to decouple transaction generation from other CH operations. Consequently, we assume that only some CHs are responsible for appending transactions to blocks, while only IoTDs can generate transactions. It's worth noting that alternative implementations of our design may operate under different assumptions.

scalability problems. In our case, if the nodes themselves were promoted as CHs to perform the consensus, assuming they would be robust enough, the number of those CHs would decrease as more levels are added to accommodate more IoTDs, which would enhance both scalability and efficiency.

Additionally, if we assume that node clustering would address the resource constraints of IoTDs and the heterogeneity of IoT networks, this approach would enable the utilization of powerful CHs for BC mining within the network. These powerful CHs would be evenly distributed throughout the structure's levels, facilitating their future promotion to the topmost level based on their capabilities.

If the consensus is designed to be performed bottom-up throughout the structure, our proposed multi-tiered design (forest) could potentially reduce the delays associated with two-tiered models while enhancing scalability. At the core of the design, the implementation of a consensus-accelerating algorithm would significantly reduce consensus time. Consensus would be exclusively conducted by the topmost CHs, whose number would be logarithmically bounded compared to the number of CHs at lower levels. Consequently, the deeper the forest, the more scalable and efficient it would become.

Clusters may be subject to splitting and/or merging to maintain balance in the model by restricting the number of nodes per cluster, ensuring compliance with the decentralization rule of BC. Additionally, this would allow for an easier cluster management based on the capacity of the managing CHs.

2.2.2 The Research Questions

Our hypotheses raise the following questions:

1. Node Management considerations:
   (a) How should we consider the nodes organization in order to achieve the consensus efficiently while ensuring the balance between the consensus decentralization and the node management overhead?
   (b) Accordingly, how should BC traffic flow in the architecture?
2. Consensus verification considerations:
   (a) How and where should the consensus process be achieved in terms of Tx and block verification?

(b) When and at what level should the verifications take place?

(c) How to perform Tx verification, in terms of delegations? Should there be a rigid rule on the hierarchy level at which the verification should start?

In the following chapter, we introduce our research objectives and our contributions to the advancement of knowledge.

CHAPTER 3
RESEARCH OBJECTIVES

The thesis aims to propose an improved BC multi-tiered design concept (a multi-tiered model) with high scalability, based on low cost operations and high efficiency algorithms, a design concept to address current BC flat and two-tiered models' scalability and delay issues and to cope with todays large number of IoTDs, with high Tx rates. The envisioned design concept would be actualized through various architectural options, awaiting comparison and evaluation. Among these options, one could be selected for in-depth studying and simulation, aiming at illustrating the advancements our design concept promises over existing flat and two-tiered models.

Considering BC, there is always a number of broad-line objectives to be considered, concerning security, workload capacity, processing delays, updating the BC using a specific consensus mechanism, etc... We chose to focus on studying node organization and node manageability to provide a decentralized, secure, reliable consensus with less traffic overhead. Also we evaluate the impact of node organization on the efficiency/timely response to IoTD Tx generation. The goal is to achieve a balance between contradicting trade-offs: the decentralization (which ensures BC security in terms of preventing a small number of nodes from taking control of BC) versus the efficiency, while respecting the workload capacity of the nodes.

At the heart of our proposed design lies the potential for a consensus-agnostic consensus-accelerating algorithm, aimed at reducing consensus time. Additionally, a node positioning mechanism could be devised to facilitate node location and transaction routing. Furthermore, a self-scaling algorithm could be developed to maintain a balance between node management overhead and consensus decentralization. Finally, a colocality concept leveraging node proximity within the structure may be employed to further reduce consensus time. Our design is intended to support any consensus algorithm for enhanced efficiency and remains application-agnostic.

We do not attempt to develop new consensus mechanisms or improve existing ones, but we would certainly adopt a consensus mechanism that aligns with our main objective: a node organization that achieves a balance between decentralization and overhead capacity while achieving better consensus efficiency and scalability than the existing models. Our main priority is to find a design that allows for a functional, efficient BC, with minimum consensus delays, suitable for billions of IoTDs. In this chapter we provide the design details of our project.

3.1 General Objectives

In order for BC to work properly within the IoT environment, BC has to accommodate up to billions of IoTDs, hence BC design needs to allow for a number of general scalability aspects, prioritized as follows:

1. Ensuring that all IoTD nodes experience a comparable time to append a Tx to a block, independently of their position in the structure and of their capabilities, which vary from device to device.
2. Reaching a consensus agreement with low delay, to accommodate IoTD's growth and fast interaction needs. Node aggregations along with delegating Tx appends to more powerful nodes might represent a solution. Aggregation, as considered by earlier studies (Heinzelman *et al.*, 2002), may be performed through clustering (Clu, 2012; Clauset *et al.*, 2005; Hwang *et al.*, 2010; Mar, 2017; COD, 2006; Graham-Smith, 2012; Clu, 2021) as a solution towards favoring ON scalability and accelerating the consensus.
3. Reaching a high consensus throughput in terms of the number of Tx processed per second, to account for the volume of IoT transactions which could exceed the order of billions.
4. Avoiding a centralized consensus, so that no small subset of nodes may take control of the BC. Hence, considering that only CHs get to append transactions to blocks, having only one or a small number of such CHs will not achieve decentralization. Thus a design that enforces a minimum number of CH to achieve consensus majority has to be implemented.
5. Ensuring a balance between consensus decentralization and node management overhead.
6. Providing a topology that allows IoTDs to join and leave dynamically, thus elastic enough to expand and contract, as needed.
7. Evaluating the impact of locality: Reducing the consensus time and reducing the amount of traffic (exchanged messages) and the distance travelled between Tx participants for transactions with more than one participating node is needed. Node organization has to be suitable to leverage logical locality between nodes frequently participating in common Tx.

Now that have shaped a proposed design for a BC-based IoT architecture, in the next chapter, we proceed to present our proposed multi-tiered model as well as the design architecture options to achieve the model.

CHAPTER 4
THE MULTI-TIERED CONCEPT MODEL

In our study, we propose an improved BC multi-tiered design concept (a multi-tiered model) that incorporate some scalability concepts referenced in the literature. The design concept is achieved through the development of a number of proposed designed architecture options that are compared and evaluated. One design option is chosen to be further studied and simulated in order to showcase the improvements our design concept provides compared to flat and two-tiered models.

We begin this chapter with an overview of the design concept requirements, considering the limitations of existing BC models from the literature. We then discuss the key features of our design concept and how they can be realized through the development of proposed design options. Additionally, we compare these options, highlighting the requirements addressed by each architecture. Finally, we delve into the details of one chosen design option: the SSD architecture.

4.1 The multi-Tiered Design Concept Model- an Overview

Our proposed multi-tiered design concept does not only address the scalability issues of the literature BC systems, but it also aims to address the auto-organization and node management as well as consensus decentralization issues that emerge while applying BC into IoT world. A set of design requirements logically stem from the need to address such issues, aiming to define architectures that meet those needs.

The approach followed focuses mainly on the importance of producing an architecture that offers short delay and fair consensus while giving room for scaling the number of IoTDs supported.

In this section we proceed by introducing the characteristics of the flat BC model, followed by the two-tiered BC model, in order to compare them to our multi-tiered model, design wise.

4.1.1 A Typical Flat Blockchain Model

A typical flat BC (Nakamoto, 2009) is usually characterized by the following:

1. The nodes achieve a consensus to append a new block of transactions into a Merkle tree where each block contains the hash of the previous blocks to form an immutable chain.
2. The nodes are all unclustered and all nodes can generate Tx and build blocks as well as append blocks into BC. All nodes have the same capabilities of action.
3. We assume that transactions are multi-signature, involving a requester IoTD and a requestee IoTD. We maintain these assumptions in both the two-tiered and our proposed multi-tiered models to facilitate comparison between the flat model, the two-tiered model, and our proposed multi-tiered models. The requester and requestee signatures (along with the Public Keys (PKs) as well as the Tx details) are carried, each, in a message that is broadcast to all the nodes to verify the Tx. When a node receives a message carrying a Tx, it verifies the Tx by checking that the current unique values of the requester and requestee public key hash $(h(PK))$ matches the specified value in previous Tx from the same requester and requestee on the ledger. Also it checks that the requester's and requestee's current signature can be validated with their PKs. Once a Tx is verified by a node, the node broadcasts the Tx once again so that all other nodes get a fair chance to append the Tx into a block. Then nodes may attempt to build a block that contains a certain number of transactions. Once a block is built by one node, the block is broadcast towards all the other nodes to be verified. So most of the nodes are visited four times, once by each Tx party message (the requester and the requestee, assuming that they do not know each other's PK), once by the message carrying the verified Tx and once by the message carrying the generated block.
4. A Tx has to be appended only once into BC. The node with a Tx pool that reaches a certain number of transactions first gets to build the block using the consensus mechanism adopted, and gets to append that block into BC. All Tx included into that block should not be appended again into any future blocks. All the other blocks that are currently being built by other nodes are to be ignored.

A typical BC is an overlay design concept that is built on top of a real physical network, where all nodes exchange messages to verify transactions and blocks. Tx information, blocks as well as a copy of BC are broadcast to all nodes. Hence every node eventually receives three types of information: the message carrying the Tx, the message carrying the block that contains the same Tx as well as a copy of the BC. For the block-building nodes, such information is crucial, and since in a typical BC this information should be always available for all the nodes that may potentially build a block, the number of required messages is quite large. All these required message exchanges might prevent the system from scaling appropriately when the number of nodes and/or the number of transactions increases, as it would be the case for the number of transactions generated by IoTDs. The scalability issue arises because all system nodes have to receive these messages, regardless of whether they are involved in the transaction in question. Ideally, only the nodes involved in the transaction should receive the messages. We call this design “the flat model”. We have no control on the physical network that exists beneath the ON, neither on the route that may be taken by the Tx to reach a certain node. We can only improve the time it takes a Tx to reach BC by improving the organization of the ON, aiming to limit the hops done by the Tx to reach unnecessary intermediate nodes.

4.1.2 LSB Two-Tiered Blockchain Model

In a two-tiered model such as LSB (Dorri *et al.*, 2017a; Alharby et Moorsel, 2020), IoTDs are clustered and decoupled from the more powerful managing CHs that participate in BC operations, as in Figure 4.1.

LSB model is characterized by the following:

1. As in the flat model, nodes achieve a consensus to append a new block of Tx into a Merkle tree where each block contains the hash of previous blocks to form an immutable chain. Unlike the flat model, nodes are organized in two levels.
2. Only bottom level nodes are clustered and only those nodes can generate Tx, while only the top level nodes (referred to as Cluster Heads (CHs)) can build, verify and append blocks into the BC.
3. The Tx are multi-signature with a requester IoTD and a requestee IoTD. When a CH receives a Tx, it verifies it by checking that the current unique value of the requester public key hash $(h(PK))$ matches the specified value in previous Tx from the same requester on the ledger. Then the CH checks that the requester current signature matches its current PK, which changes with every newly generated Tx; then the CH verifies that the requestee belongs to its managed cluster (otherwise the Tx is multicast to all other CHs), and whether the requestee accepts the Tx from that requester (according to an ACL-like entry). The last step helps build a reputation/trust history for the requester in terms of requestee acceptance. All verified Tx are multicast to all CHs. If a Tx is valid, it gets stored in the CH Tx pool. Once the pool reaches a certain threshold number of Tx, the CH may start building a block.
4. A block is appended to BC if all CHs validate the signature of the block generator CH, which uses a key known to all CHs. Then once the block is built, it will be verified by all CHs.

5. So visits in the $O(number\,of\,the\,CHs)$ plus lists of length in $O(number\,of\,the\,structure's\,nodes)$ are read once in case that the requester and the requestee do not belong to the same cluster, then only visits in the $O(number\,of\,the\,CHs)$ will be done once to diffuse the verified Tx and once to verify each block.

6. All transactions have to be appended only once to a block at any node of the structure. The node with a Tx pool that reaches a certain threshold number of Tx first gets to build the block according to the consensus mechanism, and gets to append that block into BC. All Tx included into that block should not be appended again into any future blocks. All the other blocks, containing these Tx, that are currently being built by other nodes are to be ignored.

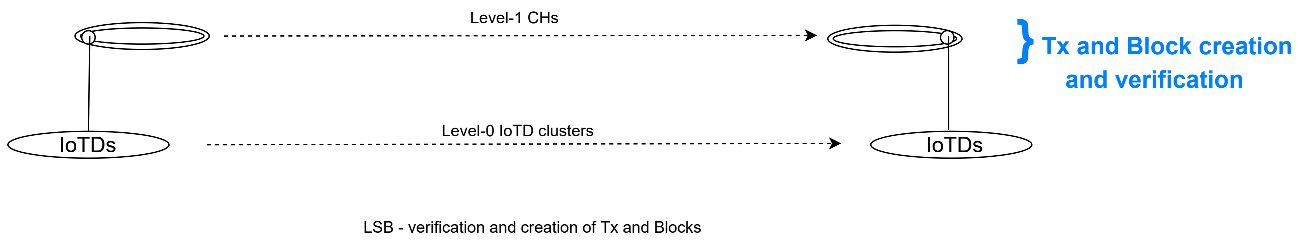
Level-1 CHs
Tx and Block creation and verification
IoTDs
Level-0 IoTD clusters
IoTDs
LSB - verification and creation of Tx and Blocks

Figure 4.1 LSB - Creation and Verification of Tx and Blocks

4.1.3 The Multi-Tiered Blockchain Model

Our multi-tiered model provides a dynamic solution for the scalability limitations of the two-tiered model by leveraging the concept of clustering all nodes according to their capabilities in order to efficiently facilitate the exchange of information among nodes while they reach consensus. We introduce an expanded multi-tiered hierarchical node organization where CHs themselves are grouped into clusters at higher levels. This allows scaling the architecture by increasing the number of levels as well as the number of topmost CHs at any time. Thanks to clustering, each level has a logarithmically bounded number of nodes less than the number of nodes at the levels beneath it. Hence, the deeper the forest, the fewer the topmost nodes that verify transactions and blocks, the shorter the consensus delays using our consensus-accelerating protocol, and the more scalable the model becomes. This is the primary motivation for our proposed multi-tiered clustered design concept versus other horizontally clustered flat and two-tiered designs where clustering serves as an integral part of the consensus mechanism akin to Federated Byzantine Agreement (Tumas *et al.*, 2023), Quorum (Rebello *et al.*, 2022) and context-based consensus (Lunardi *et al.*, 2020).

We propose a design with very dense IoTD clusters at the bottom level compared to the CH cluster sizes at the levels above, which further enhances the model's scalability and efficiency. The number of topmost CHs visited to perform consensus is only a fraction compared to the number of CHs visited by transactions and blocks in the case of two-tiered models while having the same number of IoTD nodes. Thus the multi-tiered design improves BC scalability and efficiency compared to two-tiered architectures.

Also the multi-tiered hierarchical node organization allows the decoupling of Tx verification from block verification process to reach a consensus, as in Figure 4.2. The decoupling is in order to liberate CHs directly managing IoTDs from the block processing overhead and from the packet traffic overhead and to help distribute the overhead among the different management levels above. Hence, it helps the system performance overall. Our design is flexible such that Tx verification may be performed by CHs at the level right above IoTDs if a common CH parent for both Tx parties is located at that level and if Tx tampering is a concern, or it may take place at a higher level if a common CH parent for both Tx parties is located at that level and if efficiency is a priority. Thus there is a trade-off between security and efficiency and careful implementation choices may be needed according to each application needs. For the sake of our study, we prioritize efficiency and hence we choose to perform Tx verification at the topmost level of the hierarchy.Block creation and verification is deferred to CHs at the highest level of the hierarchy.[1] Thus Tx and block verifications involve a smaller number of nodes, and hence, are achieved more efficiently.

The hierarchical relationship among nodes is modeled as a tree-like node organization. At the highest level of the architecture, in order to allow consensus for a number of nodes large enough to achieve decentralization, a single tree is not suitable, thus nodes need to be organized in a forest. To keep consensus time comparable for all IoTs, and to keep node management load-balanced for all CHs, all forest trees have to have comparable height and branching factor, and hence, a comparable cluster size, such that each CH manages only one cluster. Since Tx are generated at the lowest level of the architecture, going upward to be appended into a block, the hierarchy needs to maintain its balance (comparable per-level cluster sizes, branching factor), since this ensures that the total time elapsed between Tx creation and block creation is similar for all nodes.

Since IoT nature dictates a need to dynamically add or remove IoTDs to/from clusters, cluster size can be-

[1] When the only common CH for both Tx parties is at the highest level of the hierarchy, that CH is responsible for the both Tx verification as well as block verification.

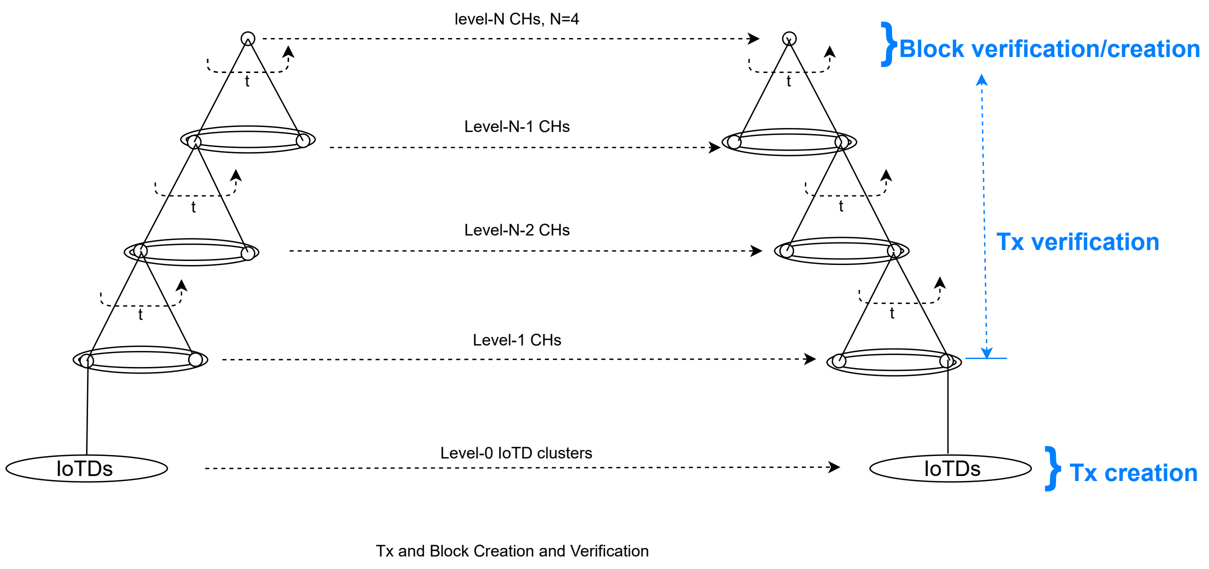

Figure 4.2 Multi-Tiered Design Concept - Creation and Verification of Tx and Blocks

come too large to be managed or too small to perform consensus. Therefore dynamic mechanisms are needed to allow for cluster splits and merges, for the architecture expansion and contraction, as well as for maintaining the architecture's balance. Enforcing some form of lower and upper threshold on the cluster size helps maintain a load-balance between consensus decentralization on one hand, and CH node management burden (to match the managing CH capacity), on the other hand.

Since only topmost level CHs get to build the ledger while IoTDs only generate Tx and since the clusters continuously split and merge as IoTDs join in or leave adhering to the thresholds and to the structure's integrity where each cluster is exactly managed by one CH, in order for the architecture to expand and contract, CHs and IoTDs need to exchange roles and positions within the hierarchy. Also since two distant IoTDs within the logical structure may regularly participate in common Tx, logical node locality will be a sought after feature. Locality implies that nodes most frequently participating in common Tx should be in logical proximity within the structure. This may facilitate the Tx verification at the CH parent common to both nodes, hence this may reduce consensus time.

In SSD, data should be maintained and exchanged among nodes so that each node may determine its position within the hierarchy, the subtree where it belongs and the size of its managed clusters. Since SSD is tiered, performing faster communication among different nodes to exchange such information and to locate nodes with a particular data item is necessary.

In our multi-tiered design concept, the design expand on LSB by creating several clustering levels: levels $[0 \ldots N]$ where level N is always the topmost level. All nodes are IoTDs s.t. a Tx may only be generated by an IoTD located at the bottom level (level 0), simply referred to as *IoTD*, while Tx and block verification as well as block creation are performed solely by the topmost IoTDs (at level N), referred to as *CHs*[2]. The rest of the nodes, located at levels $[1 \ldots N-1]$, are simply referred to as *CHs* and they help maintain the multi-tiered structure. Thus all nodes are distributed throughout the levels according to their capabilities.

A CH at level-i manages level-$(i-1)$ nodes, level-i CHs are managed by level-$(i+1)$ nodes, etc. Each cluster is managed by one CH that holds its managed tree and/or cluster nodes' information, including the Public Key (PK). Clusters' sizes are comparable by level. The nodes are distributed in clusters, such that for level 2 through N, each CH manages one cluster, on the level below, containing a number of nodes equal to a branching factor[3], t. Thus each level has a number of nodes (denoted $\#$[level i]) less than $\#$[level-$(i-1)$] on the level beneath it. A level-1 CH manages one IoTD cluster, each containing m_0 IoTDs. IoTD clusters located at level 0 are dense compared to the CH clusters located at levels $[1 \ldots N]$. Each node is labeled with a Cluster Head Address Chain (CHAC) to specify its logical position in the structure as well as its chain of parents, s.t. CHAC length of a node is equal to the node's depth+1. An example CHAC for the IoTD having $ID = 49$ shown in Fig. 4.3, is $\{0:49,\ 1:A,\ 2:L,\ 3:R,\ 4:Y\}$. CHAC, as a node-lookup mechanism, helps locating nodes, routing Tx and enabling the upward propagation of Tx from IoTDs towards topmost CHs to occur in $O(1)$ no matter how deep the forest is. Hence using CHAC as a node positioning mechanism helps to further reduce consensus delays.

We propose a consensus-accelerating protocol as follows:

1. Tx generation and propagation: A Tx involving two IoTD participants is generated by an IoTD at the bottom of the structure. The Tx propagates upwards towards level N to be verified and appended into a block, as in Fig. 4.3. Propagation occurs in $O(1)$ time, thanks to the use of CHAC.
2. Tx verification: To verify a Tx, both parties' public keys (PK) must be verified against their respective signatures. Each party's hash(PK) is verified against the same value in previous transactions from the same party (IoTD) on the ledger.

[2] Our multi-tiered model may be designed such that CHs are also allowed to generate Tx. The design will still provide promising results compared to flat and two-tiered models.

[3] The branching factor in our model sometimes fluctuates due to splits and merges

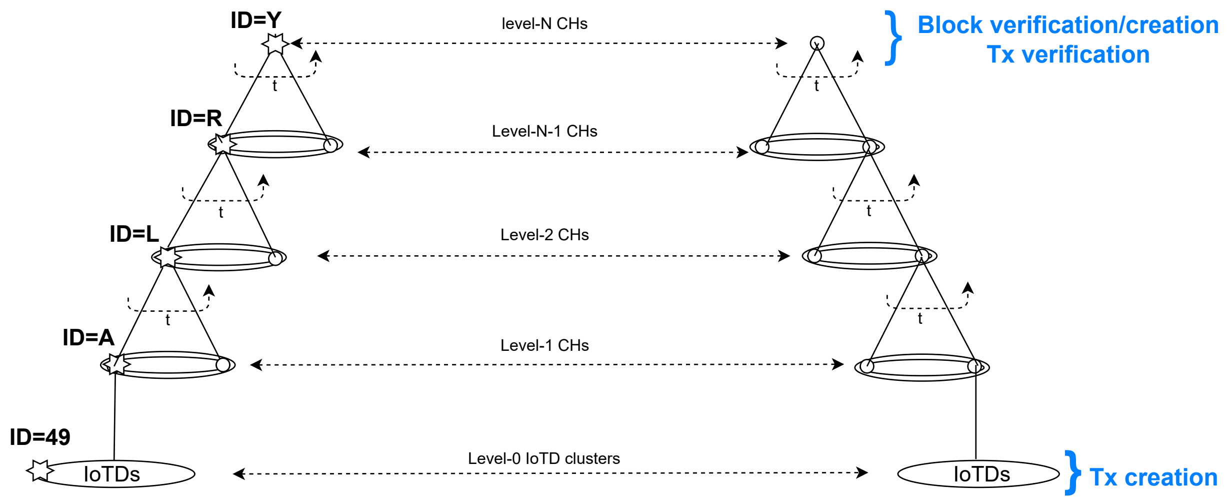

Figure 4.3 Proposed Multi-tiered Design Concept with $N = 4$.

(a) Colocality consideration: If both Tx parties are located in the same tree (referred to as *Colocality*), Tx verification takes place at the tree root (topmost CH) common to both parties, in $O(1)$, since the root holds both parties' PK.

(b) Non-colocal Tx parties: If Tx parties are in different trees, the Tx is non-colocal: after a Tx propagates from an IoTD towards its topmost parent (in $O(1)$ using CHAC), the Tx is partially verified for the IoTD's PK. All topmost CHs are visited by the Tx and searched for the other party's PK. Once the CH containing the other party's PK is found, full Tx verification takes place at that CH, resulting in Tx verification time of $O(maxpropdelay)$ (maximum propagation delay towards topmost nodes).

3. Block creation and verification: Once verified, the Tx propagates in $O(maxpropdelay)$ towards all topmost CHs so that the CHs append the Tx into blocks. When a topmost CH receives a predefined number of verified transactions, it builds a block using any consensus mechanism (Proof of Work (Qi *et al.*, 2023; Yu, 2023b), Proof of Stake (Chen, 2023), ...etc), since our design is consensus-algorithm agnostic. Block verification is performed by all topmost CHs at level N, taking $O(maxpropdelay)$ time.

4. Consensus efficiency: Consensus time is calculated as $O(1)$ plus $c \cdot O(maxpropdelay)$, where c is a constant. For non-colocal Tx, $c \leq 3$; otherwise, $c \leq 2$ in the best case. Consensus is performed by the topmost CHs, whose number is logarithmically bounded compared to the number of CHs at lower levels. Increasing the number of levels decreases the number of topmost CHs, improves node

colocality, and enhances consensus efficiency and overall scalability, as shown in Fig. 4.4.

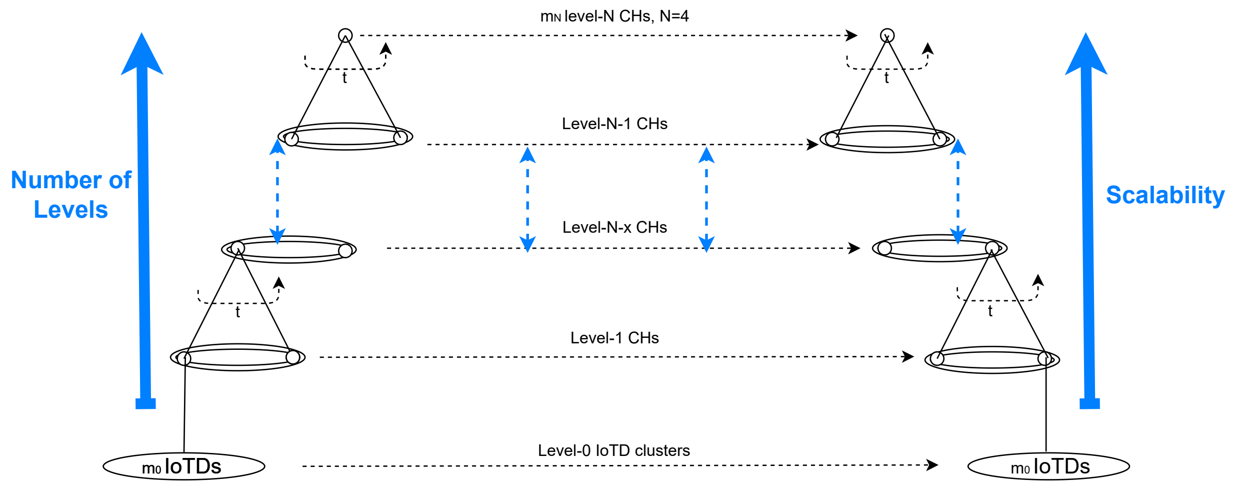

Figure 4.4 Multi-Tiered Concept - The Number of Levels and the Colocality.

Control decentralization is achieved by specifying a per-level minimum number of nodes that must exist in each cluster (control thresholds). Also the compliance with the capacity constraints, associated with the node management, is achieved by specifying a per-level maximum cluster size that suits the management capacity of the managing CH (management thresholds). Cluster splits and merges according to the thresholds help achieve a balance between node management overhead and decentralization. Cascading splits and merges may propagate upwards through the architecture. This can lead to the addition or removal of levels as needed to scale the system. A forest has at least two levels: IoTDs at level 0 and CHs at level 1 (level $N = 1$). A forest may have more levels of CHs till we reach level N where $N > 1$. So the forest has $N + 1$ levels at most: where N is the number of CH levels. This design provides flexibility to expand the system, as needed, in order to support a large number of CHs, with decentralized control, and hence, a large number of IoTDs, overall. IoTDs and CHs may exchange roles and position[4] in the structure to allow for splits and merges and for compensation for node failures.

Our proposed multi-tiered design is clustering algorithm agnostic, consensus algorithm agnostic as well as application agnostic.

[4] Roles and positions are exchanged with the assumption of always choosing more powerful nodes to be promoted as CHs, according to nodes' capacity.

4.1.4 Comparison among Models

Let us first compare the models from a scalability perspective. In flat models, there are excessive broadcasts and traffic overhead since all nodes need to be visited to perform Tx and block verification, which creates scalability limitations and increased consensus delays. In contrast, in two-tiered models, since there is no room to add more levels, the only way to scale the design is through increasing the number of CHs (to accommodate for more IoTDs) at the top level[5], which eventually reaches saturation at some point and leads to the same scalability issues faced by the flat model, only reflected at the top level of the architecture. Whereas in our multi-tiered model, increasing the number of CHs at a topmost level to accommodate for more IoTDs is less likely to cause scalability issues or consensus delay since before reaching a saturation point of the topmost level, another level is added to the design (as a new topmost level) with a fewer number of nodes.

From an efficiency perspective, in flat models, all nodes need to be visited to achieve consensus regardless of their participation in the consensus process. Thus consensus time is in the order of the number of the system's nodes. In contrast, in two-tiered models, only the CH located at the top level are visited to achieve consensus. Consequently, consensus time is in $O(1)$ plus $c \cdot O(maximum\, propagation\, delay\, towards\, CHs)$, which is in the order of the number of those CHs, which increases with the increase of the number of the IoTDs joining the system. c is a constant that is always equal to three since node colocality is not considered. In the case of our multi-tiered model, consensus time is $O(1)$ plus $c \cdot O(maxpropdelay)$, which is in the order of the number of topmost CHs, hence logarithmically bounded compared to the number of CHs at the levels below. The number of CHs is small compared to the number of nodes in the system, thanks to the dense IoTD clusters in our design. Thus consensus is achieved by a fraction of the number of nodes performing consensus in two-tiered models. Furthermore, this number tends to further decrease with the increase in the number of IoTDs joining the system. Even better, in the case of our multi-tiered model, Tx verification time is reduced to $O(1)$ when node colocality is leveraged, thus c's value decreases from three to two, thereby further reducing consensus time.

Comparing two-tiered models to our SSD model, consensus also takes $O(1)$ plus $c \cdot O(maxprop)$, $c \leq 2$ only for a fraction of the time. Whereas for most of the time, $c \leq 3$ worst case. In SSD, the worst-case scenario

[5] By increasing the number of CHs, either we create enormous clusters (more traffic overhead and delays) or we limit the number of IoTDs that may be supported (we defeat scalability).

involves visiting only a small number of topmost CHs, which is a fraction of the number of CHs visited in the worst-case scenario of two-tiered models, as we will prove in the probability of colocality section.

From the perspective of balancing node management overhead and consensus decentralization, in flat and two-tiered models, considerations of node management overhead and consensus decentralization are often overlooked, despite being crucial, especially in two-tiered models, since they are among the main virtues of leveraging a hierarchical model in the first place. In contrast, our multi-tiered model dynamically maintains this balance by adhering to thresholds and invoking cascading splits and merges to adjust the overall structure's height as needed to accommodate IoTDs joining and departing.

From our quick comparison, it's evident that the multi-tiered concept should enhance both scalability and efficiency compared to flat and two-tiered models. Additionally, it provides a balance between node management overhead and consensus decentralization, a crucial requirement in BC technology that is often overlooked by flat and two-tiered models.

4.2 The Multi-Tiered Design Concept Model- Details

4.2.1 Selection of a Clustering Algorithm

Our focus is to study the advantages of clustering in general to boost the performance of BC consensus, rather than to specify or test a certain clustering algorithm in order to build the clusters.

By itself, the multi-tiered design concept (achieved through the development of any of the proposed design concepts) is application agnostic and runs with any clustering algorithm without significantly affecting its performance. It is nonetheless pertinent to consider a number of criteria that may help achieve a seamless integration of a clustering algorithm into our multi-tiered design concept.

A suitable BC clustering algorithm is characterized by the following criteria:

1. The construction of clusters should take only one algorithm iteration.
2. The algorithm running time should be fast.
3. All topology nodes should be considered for clustering.

4. Node mobility should be considered.

5. Modularity is a recently introduced quality measure for graph clusterings (Feng *et al.*, 2023). The modularity function[6] used when distributing nodes to merge clusters should be uniform. The modularity function is biased when the algorithm identifies the clusters with lower number of members to join the neighbor cluster having the highest number of members. This produces crowded clusters with a CH overburdened with management overhead and BC control centralization, which is not a desirable feature to have in BC systems.

After considering several algorithms, we believe that K-meansGreedy (Han et Kamber, 2006; Tashman, 2024; Panigrahi et Panigrahi, 2024; Cor, 2018; Kousaridas *et al.*, 2015; Yildirim, 2020; Huang, 1998; Karimi *et al.*, 2015; Eyal *et al.*, 2011) would be preferable due to its short running time and suitability for our multi-tiered design concept as it meets all the criteria mentioned above. The algorithm uses the number of hops as a measurement for distance among nodes.

The random choice for clustering centers may yield different results on different runs. That lack of consistency may not affect its integration within our multi-tiered design concept since we do not need to rerun the algorithm to rebuild the clusters once they are built from scratch. Even in situations where a cluster needs to split or merge, re-clustering is only needed for the one or two cluster(s) to split or merge.

K-meansGreedy algorithm has a linear complexity in our case because it runs in $O(n \cdot k \cdot d \cdot i)$:

1. We cluster a finite number of nodes, n.

2. We have no distance property to take into account. And we have no other features to consider, thus we have only one dimension, d.

3. We need not to update the centroids with every iteration since our centroids are the predetermined CHs to be promoted to form a new level, thus we have only one iteration, i.

4. We build a finite (small number of) k clusters.

[6] Modularity compares the number of edges inside a cluster with the expected number of edges that one would find in the cluster if the network were a random network with the same number of nodes and where each node keeps its degree, but edges are otherwise randomly attached. (Mod, 2023)

The dominant factor in the time complexity remains the number of data points, n. Thus K-meansGreedy is linear or near-linear with respect to the size of the dataset (nodes to cluster), n.

K-meansGreedy considers all nodes for clustering using a uniform modularity function and it takes the number of clusters to be built, k, as input. That aspect makes it work hand-in-hand with our self-scaling algorithm, since the latter has to start with a $number\,of\,nodes \geq CLT$ to build a new level with a minimum cluster size value that may be equal to k.

The clustering algorithm is executed decentrally without a central coordinator since clustering is always invoked through the self-scaling algorithm, as shown in the upcoming sections.

4.2.2 Node Management In SSD

In order for our multi-tiered design structure to maintain its balance, there is a need for clusters to exchange their size information as well as their position within the logical hierarchy, through their CHs as well as through some specific nodes. Every cluster has a small number of its nodes (say 3 or 4 for redundancy) arbitrarily selected as "guardian nodes" (GNs) to hold information about the adjacent clusters within the same level and/or within the same tree. All nodes periodically multicast a heartbeat (in the form of Hello messages) (Haffey *et al.*, 2018) within the same cluster, so that cluster's GNs can evaluate the current cluster size. This facilitates enforcing management thresholds on cluster size.

GNs from different level-i clusters in the same subtree communicate their cluster size to their CH, so that each CH is always aware of the size (node count) of its managed clusters and/or managed subtree. A CH communicates that size to all of its managed GNs so that a GN are always aware of the number and size of its neighboring clusters in the same subtree.

Each level-i node identifies the subtree it belongs to and the CH it reports to by storing a label that specifies its logical address within the hierarchy, a CH Address Chain (CHAC). CHAC is a key-value dictionary that contains the addresses of its level-$i+X$ CHs, where $X = [1 \ldots (N-i)]$. Each level-i CH communicates (through hello messages) its address as well as its CHAC to its managed level-$i-1$ cluster nodes to label them with the same CHAC plus one more digit to identify each node. CHAC allows efficient determination of a destination location in the forest, in $O(1)$ no matter how deep is the forest.

4.2.3 BC Operations in our Multi-Tiered Design

In our multi-tiered design, once a Tx is created by an IoTD, the IoTD sends the Tx to its parent on the topmost level (thanks to the use of CHAC), where all CHs exchange the Tx and update their Tx pools. Once a threshold, T_{max}, is reached for the number of Tx that exist in the Tx pool of a level-N CH, that CH can start building a block. At the same time, that CH transfers the Tx to all topmost level CHs so that they build their own blocks as well. This decoupling between Tx creation and block creation liberates level-1 CHs from some of the packet traffic overhead and helps distribute the overhead among the different management levels above, hence, it helps the system performance overall.

The architecture is characterized by:

1. IoTDs are located on level 0, while CHs are located on level 1 through N.
2. Each level-1 CH has one IoT cluster of m_0 nodes, to manage.
3. Each level 2 through N CH manages one cluster of t CH nodes, where t is the branching factor.
4. Level N has only one cluster, but no CHs above.
5. For all CH clusters, we have:
   (a) A per-level Control Lower Threshold (CLT) defining the minimum cluster size to form a consensus majority;
   (b) A per-level Management Upper Threshold (MUT) defining the maximum cluster size so that cluster management would remain within the capacity of the managing CH;
   (c) A per-level Management Lower Threshold (MLT) defining a minimum cluster size below which cluster management may be too expensive, as CH capacity would be wasted.
   (d) MLT and CLT can be utilized interchangeably.
6. CH Cluster size is t, such that $CLT \leq t \leq MUT.$ For level N, CH Cluster size is m_N, such that $CLT \leq m_N \leq MUT.$
7. All IoTD clusters have CLT_0 and MUT_0 cluster size thresholds, where IoTD cluster size is m_0, such that $CLT_0 \leq m_0 \leq MUT_0$.

4.3 The Proposed Architecture Options

While our multi-tiered design concept, as described, may be achieved through a number of designs (architecture options) based on different conceptual approaches, all our proposed architecture options achieve consensus through the same consensus-accelerating protocol. Searching for a node can be performed at the price of some complexity of the initial construction and overall maintenance of the designs.

In the proposed design options, we have to constantly keep track of four main concerns:

1. the management overhead/consensus control balance;
2. the upkeep of threshold measurement checks across the dynamic structure at all times;
3. the balance of the overall forest;
4. the handling of node joining and leaving.

4.3.1 Architecture Option-1

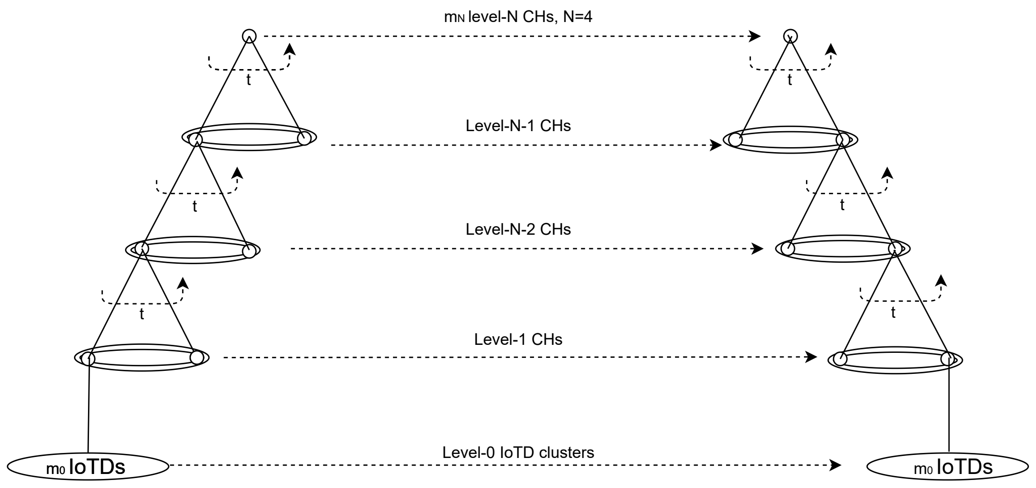

Figure 4.5 SSD Architecture

Architecture option-1 is our proposed Self-Scalable Dynamic (SSD) architecture, as depicted in Figure 4.5. SSD has a flexible elastic[7] self-scaling design where IoTDs and CHs may exchange roles and positions within the structure. The levels are constructed bottom-up through cascading cluster splits/merges, resulting in

[7] Elastic design in the sense that it may shrink or expand.

the addition or elimination of levels at the top of the structure. This process leads to the contraction or expansion of the structure. SSD is a self-organizing design. The architecture integrity, where each cluster is exactly managed by one CH, imposes the necessity of associating a cluster split with every cluster merge and vice-versa. At the heart of the design lies the proposed Self-Scalable Dynamic (SSD) algorithm that strives to maintain a balance between node management overhead and consensus decentralization, while also allowing for the maintenance of the structure's elasticity and balance in terms of cluster size and branching factor. Maintaining the management-decentralization balance requires an orchestration between the GN nodes of each cluster and the parent CHs in order to continuously exchange cluster size information, check whether cluster sizes adhere to the thresholds and accordingly invoke splits and merges. Hence, the SSD algorithm is executed decentrally, without a central coordinator, to avoid single points of failure as well as trust issues. The balance between management overhead and decentralization is dynamic, occurring simultaneously with blockchain operations. With IoTDs joining and leaving, the design is autonomously, decentrally[8] and dynamically readjusted.

In order to construct SSD architecture, two levels are first built based on the two-tiered models found in the literature. Then IoTD clusters at level 0 are overflown beyond MUT threshold with more nodes joining in. Consequently, the SSD algorithm detects the overflow and triggers cluster splits at that level. Nodes are promoted to join the clusters on the level above (level 1). This process, in turn, overflows those clusters and the SSD algorithm invokes the clusters' splits. Nodes are promoted to form a new level. And so forth, as cascading splits and merges occur, new levels are added to the structure.

The cost of building the architecture depends on the clustering algorithm used. In case K-means Greedy algorithm is applied, all IoTDs are clustered to form level-0 clusters, and subsequently, all levels are clustered except for the current topmost level. Consequently, with each newly added topmost level (level N), the construction cost linearly increases with the number of CHs on level N-1. Nevertheless, SSD maintenance cost is high as the system must maintain a number of data structures to store the node-cluster-CH relationships. Consequently, altering the position of a single node results in a substantial number of operations.

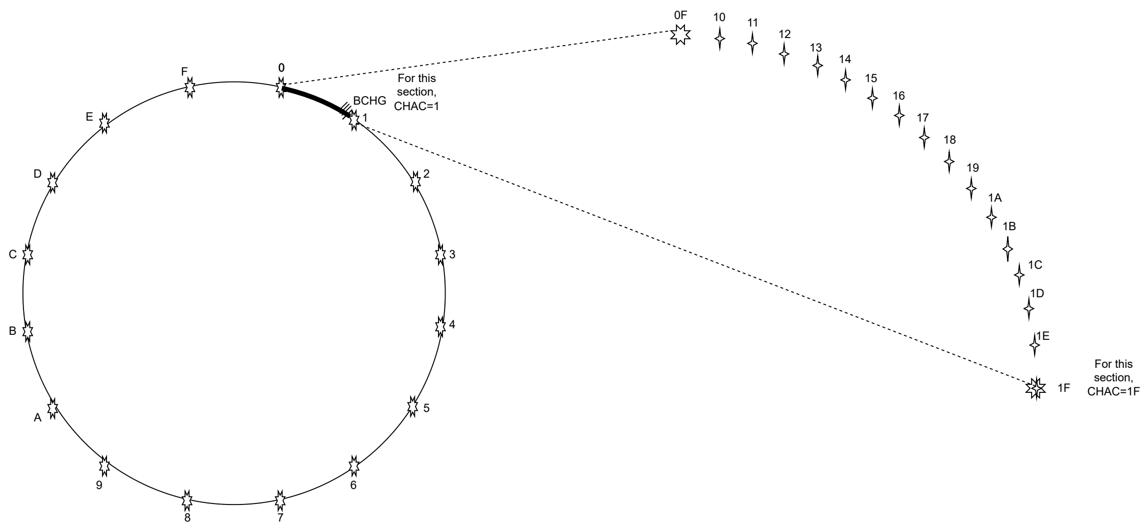

Figure 4.6 Option-2 Architecture

4.3.2 Architecture Option-2

Architecture option-2 is our proposed Partitioned architecture which is based on a uniform distribution of IoTDs on a ring using a DHT-like approach and a rigid incremental assignment of labeled CH to construct a forest containing labeled CHs and IoTD clusters. The design abstracts the forest into a ring-like graph where nodes are organized and logically positioned clockwise by increasing order of the hash of their MAC addresses. The design is based on partitioning the nodes in successive passes and it represents a forest. Partitioned architecture has a rigid design as it enforces a strict differentiation between IoTD and CH roles, thus they may not exchange roles and positions. In this design, CHs are accounted for in advance, the number of levels is pre-determined, the structure's levels are built using a top-down approach and the clusters may not split or merge. Also management-decentralization balance is not dynamic since it may not occur at the same time as blockchain operations takes place. Management-decentralization balance only occurs during the construction phase of the structure.

In order to construct the Partitioned architecture, levels and clusters are built top-down: for level-N, the top level, IoTD nodes are positioned on a hypothetical ring clockwise by increasing order of $hash(MAC\,address)$. Then, the ring is partitioned by applying modulo-m reduction to the hashed addresses, where m is a power of two ($m = 2^k$). For example, with $k = 4$, 16 partitions (i.e. sections) are created, indexed with 1-digit index, for example: from 0 to F. The nodes of each section are associated to a

[8] Decentrally in the sense that no control of power is allowed for any number of nodes.

given tree of the forest. The section node with the highest hash value is the section CH and is associated to the root of its tree. CHs are labeled with a one-digit-hexadecimal-CHAC. To build levels $N-1$ through level 1, we recursively partition the resulting sections using the modulo-m reduction function, assign CHs and label additional CHs, using CHAC, to form additional subtrees with every iteration, such that: For each previous level section, we consider the branching factor, t (assuming $t = 16$ for simplicity, to match $m = 2^k$), and we repeat the partitioning recursively by applying modulo-t reduction to the hash addresses after truncating the $k = 4$ least significant bits. Thus, for each level, we divide each of previous level sections into $t = 16$ smaller sections and for each section, we assign CHs, labeling them with the previous section CHAC after appending a new one-digit-CHAC, such that each forest level is identified by one CHAC digit, as depicted in Figure 4.6. And those sections may be further partitioned, and so on to build all the architecture levels. t can not be randomly chosen as it is recommended that it would be equal to $m = 2^k$. CHORD (Stoica *et al.*, 2001) a Distributed Hash Table lookup system, maps node identifiers and keys onto the same virtual ring. After partitioning and labeling the CHs according to the desired number of forest levels, we obtain a series of unlabeled IoTD sections followed by labeled CHs (with CHAC). The unlabeled IoTDs are the forest leaves at level 0 and the sections are the IoTD clusters. We assign labels to all unlabeled IoTDs in each section using the section's longest CHAC (as in Figure 4.6). We then apply CHORD algorithm on each section such that CHORD node ID is the most significant m-bit of the node's $hash(IP\, address)$, while CHORD Key ID is the most significant m-bit taken from rest of $hash(MAC\, address)$ bits that are still unused after performing the modulo reduction. Locating CHs is in $O(1)$ thanks to the use of CHAC, while locating IoTDs within each section using CHORD is in $O(log\, U)$, where U is the number of the section's IoTDs.

The architecture is built at the outset, since the structure cannot be modified once constructed, thereby incurring no maintenance costs. With every level to be added, we perform a hashing round for all nodes. Consequently, the construction cost increases linearly with the number of levels to be built. In addition, once all levels are constructed, the cost of constructing a CHORD ring for each resulting IoTD cluster, must be considered.

4.3.3 Architecture Option-3

Architecture option-3 is our proposed Ring of Rings (RoR) architecture which achieves node locality as well as CH flexibility with minimized consensus delays. The architecture is based on mapping the node's logical location within the forest to a logical location within a hash space using a DHT-like approach to organize

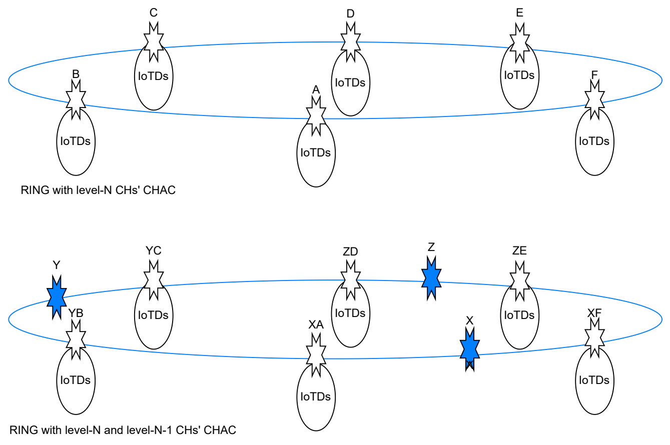
C
D
E
B
A
F
IoTDs
RING with level-N CHs' CHAC
Y
YC
ZD
Z
ZE
YB
XA
X
XF
RING with level-N and level-N-1 CHs' CHAC

Figure 4.7 Ring of Rings

CHs. Some of those nodes will get gradually promoted as CHs in a recursive manner so that the final result leads to a forest-like structure, as depicted in Figure 4.7. RoR has a flexible design where IoTDs and CHs may exchange roles and positions, as in SSD architecture. The levels are built bottom-up and clusters may split according to thresholds similarly to SSD architecture. Also the management-decentralization balance is dynamically adjusted while BC operations take place.

In order to construct RoR architecture, a location independent $GlobalLocalityID$ is assigned to each node, namely the node's IP address. Virtual Cord Protocol (VCP) (Awad *et al.*, 2011; Fersi, 2015) is a virtual location based protocol that computes node relative positions and uses greedy routing. In each building/house, VCP is used to assign a local relative ID ($LocalLocalityID$) to each IoTD. For each cluster, one IoTD is chosen as CH, labeled with a CHAC. The CHAC value is communicated to the cluster's IoTD nodes. Using a locality table, the CH maps an IoTD's $GlobalLocalityID$ to its $LocalLocalityID$. Using a VCP table, the CH routes traffic towards its managed IoTDs in $O(\sqrt{\eta} - 1)$, where η is the number of the cluster's IoTD nodes.

To add one more level : From every two adjacent clusters, one IoTD having the closest $GlobalLocalityID$ (IP address) to any of the CHs managing those clusters, is promoted as a CH on a new level. CHAC of the promoted IoTD is communicated to its managed nodes. Similarly, adjacent IoTD clusters promote nodes upwards to join the new level.

Thresholds may be set for IoTD cluster size and an algorithm similar to SSD Algorithm may be designed to trigger cluster splits according to the thresholds. At the end of the construction process of the architecture, we obtain a ring with CHs having various CHAC lengths to indicate their level within the structure. Each level-1 CH has a locality table, a VCP table and a set of managed IoTDs. A CH is identified by a $GlobalLocalityID$

and a CHAC, while a level-1 CH is additionally identified by a $LocalLocalityID$. Each IoTD has a VCP table and is identified by a $GlobalLocalityID$, a $LocalLocalityID$, as well as the same CHAC as that of its managing CH.

A CH role may change on a timeout round-robin basis to allow a fair chance for all the local IoTDs to act as CHs. If a CH becomes unavailable or proves incapable of managing its assigned IoTD cluster, it can undergo one of two processes: demotion as an IoTD node and replacement by another node from the IoTD nodes, or reassignment with a lower management load. The latter process includes splitting the CH's managed cluster and reapplying VCP. The CH still manages a sub-cluster while the other sub-cluster is assigned to another CH (a promoted IoTD node from that sub-cluster), thereby sharing the load.

The cost of building the architecture includes the cost of assigning $GlobalLocalityID$ to the nodes, calculating the per-cluster VCP, creating the DHT tables, as well as promoting CHs to build the levels.

The cost of maintaining the structure (to accommodate future splits) includes the cost of splitting the clusters, calculating VCP, updating the DHT tables affected, as well as promoting the associated CH. Thresholds may be assigned to cluster sizes, akin to SSD's MUT threshold, s.t. all level-0 clusters may split once a predefined number of clusters exceeds the threshold (to keep all cluster sizes comparable), resulting in adding more levels to the hierarchy.

4.4 Comparison among the Architecture Options and the Chosen Architecture

4.4.1 Comparison among the Architecture Options

All the proposed designs achieve the same multi-tiered design concept, adopt the same consensus-accelerating algorithm with the same complexity. The consensus-accelerating algorithm remains algorithm agnostic and a given architecture option may be chosen in order to better adapt for a specific consensus algorithm. For instance, if we consider Ethereum consensus algorithm, it has four types of nodes: miners, full nodes, light nodes and archive nodes. Scaling our architecture is based on the flexibility of the nodes to exchange roles and positions, which is not possible for Ethereum nodes since the nodes have different specifications to allow each type of node to perform its specific functionality. However this may match our proposed architecture design option 2: the partitioned architecture, if the miner nodes are placed at the upper level (limited in numbers) while the Tx proposing nodes are placed at the bottom level of the architecture. Our design aims to accelerate consensus in general and it may work with any consensus mechanism or system like Ethereum.

While efficiency criteria are crucial, the decision to adopt one architecture over another depends on whether BC construction and consensus functionality timing requires a given level of flexibility. In case where BC must continuously adapt to IoTDs exchanging roles with CHs to keep a continuous load balance between node management overhead and decentralization (as consensus takes place), options such as SSD and RoR architectures may be suitable. Conversely, if the construction phase has to strictly occur before conducting BC operations with the number of CHs accounted for in advance, the Partitioned architecture may be the appropriate choice. However, flexibility comes with associated maintenance costs that should also be taken into account.

The cost of maintenance translates into the cost of adding levels to expand the architectures Since the Partitioned architecture is rigid, it has no maintenance cost. Adding a new IoTD only takes the constant time to hash its MAC address, to perform the modulo reduction function to obtain its CHAC and to match the CHAC with a level-0 cluster's CHAC ($O(number\, of\, level-0\, clusters)$). In case of SSD and RoR architectures, adding a new IoTD only takes $O(the\, number\, of\, levels)$ times the number of data structures containing the node and cluster information to be updated.

The cost of maintenance in case of SSD architecture includes adding a new level of nodes due to bypassing MUT threshold at the topmost level after a number of, possibly cascading, cluster splits at the levels below. This may take $O(the\, number\, of\, levels)$ times the number of data structures containing node and cluster information to be updated as a result of the cluster splits and merges. Additionally, the number of nodes in the clusters to be reclustered (assuming a linear clustering algorithm is used to perform cluster merges and splits) has to be considered in the cost.

In case of RoR architecture, the maintenance cost includes adding a new level of CHs due to splitting the IoTD clusters at level 0. New IoTD nodes are chosen to be promoted upwards as CHs to form the new level. This may take $O(the\, number\, of\, levels)$ times the number of data structures containing node and cluster information to be updated. Additionally, the number of nodes in the clusters to split (assuming a linear clustering algorithm is used to perform cluster splits) has to be considered in the cost. Additionally, the cost of calculating VCP for the clusters resulting after the splits as well updating/creating the corresponding DHT tables are additional costs that have to be taken into consideration.

In case conditions related to node proximity exist for IoTDs, choosing RoR architecture could be appropriate since it is based on clustering the nodes according to their IP address.

Table 4.1 summarizes the advantages of each proposed architecture design concerning the earlier mentioned criteria.

	Consensus Effectiency	**Structure Flexibility**	**Maintenance Cost**	**Node Proximity**
SSD	x	x	x	
Partitioned	x			
RoR	x	x	x	x

Table 4.1 Proposed Designs Comparison Table

4.4.2 Chosen Architecture Option

Since all proposed architecture options adhere to the multi-tiered design concept, selecting any one of these architecture options and providing evidence of its superiority over flat and two-tiered models suffices to establish that the multi-tiered design concept, as embodied by this chosen architecture, outperforms flat and two-tiered models.

Based on the comparison among the architecture options, it is easy to deduce that architecture option-1, the SSD architecture, is characterized by a wide range of features which ensure a high degree of dynamism and flexibility. Features, such as the architecture integrity where each cluster is exactly managed by one CH, the splits and merges as well as the SSD algorithm which dynamically strives to maintain a balance between node management overhead and consensus decentralization, help shape the dynamic characteristics of SSD architecture, showcasing its flexibility and making it an option to be further studied and explored.

For the remainder of the thesis, we refer to our multi-tiered model as the SSD architecture. Thus the SSD architecture will be further studied, analyzed and simulated, in order to be compared to flat and two-tiered models with the goal of validating our multi-tiered model. In next section, we elaborate on the SSD architecture.

4.5 The SSD Architecture

4.5.1 Self-Scaling Dynamic (SSD) Algorithm

In this architecture, adjusting the height of the forest according to the thresholds while keeping the overall forest balance is automatically triggered so that the structure is autonomously adjusting and self-scaling whenever new IoTDs join in or leave the BC system. IoTDs are clustered under a few levels of managing CHs, where the cluster size has to be constantly maintained between the thresholds while IoTDs join and leave. This requirement inspired the creation of a self-scaling algorithm that constantly checks the thresholds (MUT and CLT) and accordingly triggers cluster splits and merges, as illustrated in Figure 4.8 and Figure 4.9.

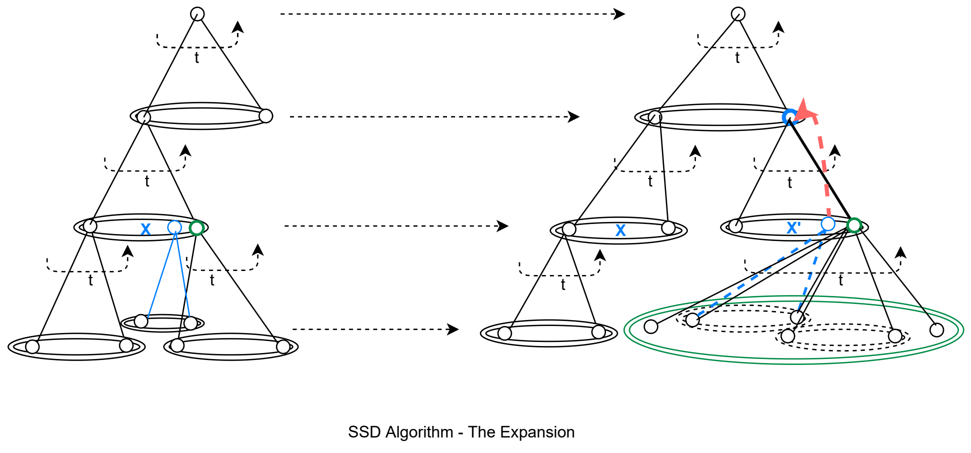

Figure 4.8 SSD - The Expansion of the Architecture

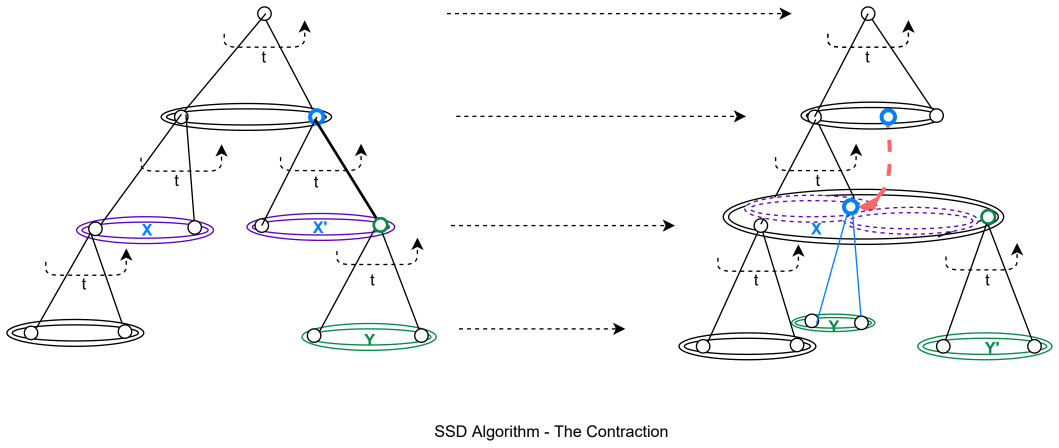

Figure 4.9 SSD - The Contraction of the Architecture

The assumptions of the self-scaling algorithm are as follows:

1. In a structure of height h (with 0 to N levels), a new level can only be added on top of all existing

levels, i.e., above the current level N.

2. Levels can be eliminated as needed, one level at a time, except for level-0 IoTDs and level-1 CHs.

3. Each CH directly manages only one cluster of other CHs, or of IoTDs.

4. A cluster that becomes smaller than CLT needs to merge with the smallest cluster on the same level and within the same subtree to make a bigger cluster. The size of the cluster is communicated to the CHs through GNs so that neighboring clusters on the same level and in the same subtree would be notified to take further actions.

5. For simplicity, we consider $MUT = MUT_0, CLT = CLT_0$ and $MUT = CLT^2$. The final assumption is to ensure that the resulting clusters from a new split are not smaller than CLT and the resulting cluster from a new merge is not larger than MUT.

```
Data: i, CLT, MUT, size(localCluster), h =max(i)
Result: A self-scaling architecture - an expansion
Initialization: starts with clustered IoTDs (cluster size is not less than CLT), managed by CHs, only the
  topmost level has one cluster;
while nodes join in do
    if one IoT joins (at level i=0) then
        //the following is distributed at the level of each cluster;
        if size(localCluster) $\geq MUT$ and localCluster is at level $i \neq h$ then
            split $localCluster$ ;
            //one resulting cluster has $size(resultCluster)$=$\lceil((size(localCluster) - 1) \div 2)\rceil$, and
              the other has $size(resultCluster)$=$\lfloor((size(localCluster) - 1) \div 2)\rfloor$;
            if size(resultCluster) $\leq CLT$ then
                merge with the smallest size cluster from same level subtree using Self-Scaling
                  Algorithm-part2
            else
                assign GNs for each $resultCluster$;
                add one node from the $resultCluster$ without CH, as a CH, to level $i + 1$;
                level-$i - 1$ cluster previously managed by the promoted node merges with the smallest
                  size cluster from same level subtree (if i > 0);
        else if size(localCluster) $\geq MUT$ and localCluster is at level i==h then
            cluster the CHs into CLT clusters;
            add CLT CHs on a newly created level $i + 1$;
            perform CLT merges for level-$i - 1$ clusters previously managed by the promoted CHs, with
              smallest size level clusters;
            $h = i + 1$;
end
```

Algorithm 1: Self-Scaling Algorithm-part1

```
Data: i, CLT, MUT, size(localCluster),h=max(i)
Result: A self-scaling architecture - a contraction
Initialization: starts with clustered IoTs (cluster size is not less than CLT), managed by CHs, only the
  topmost level has one cluster;
while nodes leave do
    if one IoT leaves (at level i=0) then
        //the following is distributed at the level of each cluster;
        if size(localCluster) $\leq CLT$ and localCluster is at level $i \neq h$ then
            merge $localCluster$ with the smallest size cluster on same level and in same subtree ;
            //the resulting cluster has $size(resultCluster)$;
            if size(resultCluster) $\geq MUT$ then
                split it using Self-Scaling Algorithm-part1
            else
                assign GNs for $resultCluster$;
                demote one of level-$i+1$ CHs that used to manage one of the two merged clusters;
                one level-$i-1$ cluster will split into two $resultClusters$, one of which will be
                  managed by the demoted CH (if i > 0);
        else if size(localCluster) $\leq CLT$ and localCluster is at level i==h then
            demote level-$i$ CHs down into their previously managed level-$i-1$ clusters, and uncluster
              those clusters into one cluster;
            eliminate level $i$ // topmostlevel;
            perform $\leq CLT$ splits for level-$i-2$ clusters so that the demoted CHs would manage half
              of the newly formed clusters;
            $h = i - 1$;
end
```

Algorithm 2: Self-Scaling Algorithm-part2

The algorithm starts with two levels: a clustered level i, i=0, with $\#[$level $i]$, cluster size is not less than CLT; and a level i, i=1, with only one cluster with $\#[$level $i]$, cluster size not less than CLT; where $h = max(i) = 1$.

Part-1) As nodes keep joining while SSD expands, if MUT is reached in one of a non-topmost-level cluster, it splits into two. The resulting clusters will check their size for necessary merges using Algorithm-part2, if any is less than CLT. This check is only valid in case MUT is assumed not equal to CLT^2, as in case MUT is equal to CLT^2, there will be no need for any further merges. If none of the resulting clusters is less than CLT, each resulting cluster will assign some GNs to take care of the cluster statistics, and the resulting cluster without a CH will promote one node as a CH on the level above. On the level below, the cluster previously managed by the promoted node has to merge with the smallest size cluster on the same level and in the same subtree (within the shortest logical distance so that both clusters would be under the same direct parent according to their CHAC). In case the resulting cluster size exceeds MUT, it has to be split using Self-Scaling Algorithm-part1. For the case of the topmost level (level N), nodes are considered as clustered into only one cluster. If MUT is reached on the topmost level, the level's nodes need to be clustered into CLT clusters, while creating a new level above. Add CLT CHs are added on the new level. On the level below, the CLT clusters previously managed by the promoted CHs are merged with the smallest size clusters. The height of the structure is adjusted accordingly.

Part-2) As nodes start leaving (SSD contracts), if CLT is reached in one of a non-topmost-level clusters, it merges with the smallest size cluster on the same level and in the same subtree. The resulting cluster checks its own size for necessary splits using Algorithm-part1, if it exceeds MUT. This check is only valid in case that MUT is assumed not equal to CLT^2, as in case MUT is equal to CLT^2, there will be no need for any further splits. If the resulting cluster size is not more than MUT, the resulting cluster assigns some GNs to take care of the cluster statistics, and demotes one of the CHs previously managing one of the merging clusters back into its managed cluster. the demoted CH will manage a newly formed cluster as one split will take place at the level below. In case any of the resulting clusters is smaller than CLT, each of them is merged with the smallest size cluster from the same subtree using Self-Scaling Algorithm-part2. For the case of the topmost level, if CLT is reached, all CHs ($\leq CLT$ CHs) are demoted back to their managed clusters on the level below, and those clusters get unclustered into one cluster. The topmost level is eliminated while a number less than CLT of cluster splits are performed in the next level below, so that the demoted CHs would manage half of the newly created clusters. Then, the height of the tree is adjusted accordingly.

From the self-scaling algorithm, we can observe that with network expansion, CLT can be automatically enforced at the creation of every layer while MUT triggers either the creation of new layers, or the creation of new clusters on the same layer. With network contraction, CLT triggers either the elimination of a layer or the elimination of one or more cluster(s) on the same layer. Both the expansion and the contraction parts involve splits and merges, and further layers can be added or eliminated down the road while respecting the thresholds, to allow SSD to easily scale up or down as needed. Each threshold check takes place while enforcing the other threshold, dynamically alternating between both parts of the algorithm. Thus the self-scaling algorithm involves a dynamic cyclic behavior that helps enforcing both thresholds at all times, hence, ensuring the balance between node management overhead and consensus decentralization. The SSD algorithm converges according to each application needs. The algorithm operates in a decentralized manner, thanks to the dynamic coordination between cluster GN nodes and CHs, without relying on a central coordinator, hence preventing single points of failure and trust issues. The implementation may be adaptable, allowing the algorithm to persistently run alongside blockchain operations.

4.5.2 Probability of Colocality

Our goal is to minimize the number of node visits (the number of hops between nodes) in order to improve consensus time within the overlay network, assuming that this improvement will be eventually reflected on the real underlying physical network. Once way to decrease the number of hops is through leveraging the concept of node colocality. We refer to the case where nodes most frequently participating in common Tx are in a logical proximity within the structure in terms of their distance on the ON, as " the IoTD logical colocality " . Considering a uniform distribution, let us call the probability of two nodes to be located within the same tree " The Probability of Collision " , according to a hashing terminology. When a collision exists between two nodes (colocality), both nodes are in the same tree of the structure and hence, in order to verify a Tx, the Tx propagates from the IoTD requester that generates the Tx upwards to reach the topmost parent CH common for both Tx parties (the requester and the requestee), in $O(1)$ thanks to the use of the IoTDs CHAC. Then the Tx is verified by that CH in $O(1)$ since the CH holds the information (namely the Public Key (PK)) of both Tx parties. Otherwise, if both Tx parties are not colocal as they are located in different trees, the Tx propagates upwards in $O(1)$ using CHAC towards the requester topmost parent CH which verifies the Tx partially, then the Tx propagates towards all topmost CHs in $O(maximum\, propagation\, delay\, towards\, topmost\, nodes)$ searching for a CH that holds the requestee information (PK) to fully perform Tx verification. Hence Tx verification takes only $O(1)$ best case in case of node colocality, while it takes $O(1)$ plus $O(maximum\, propagation\, delay\, towards\, topmost\, nodes)$, otherwise. Colocality is determined at IoTD level using IoTD CHAC in O(1).

In our SSD design, transaction verification takes place at the topmost level of the architecture since efficiency is prioritized, rather than security. Nevertheless if node tampering is a concern, the multi-tiered design is flexible enough to allow transaction verification to take place anywhere from level $[1 \ldots N]$ only if a verifying node that is a common parent of both IoTD transaction parties is located at that level. In that case, a transaction needs to propagate throughout all the structure's levels looking for a common parent to both transaction parties in order to verify the transaction, sacrificing efficiency for the sake of prioritizing security. Hence node colocality is defined in terms of each levels as follows: Colocality is determined at the IoTD level, where an IoTD's CHAC helps determining the logical distance between nodes in $O(N+1)$. Let us define the distance between two nodes as $dist = 0$ if they are in the same cluster (under the same level-1 parent CH). They then have identical CHAC digits for levels 1 to N (except for the least significant digit), hence they have a direct colocality. When $1 \leq dist \leq N$, they have in indirect colocality. Nodes have $dist = 1$ if they belong to different clusters that share the same level-2 parent CH and so they have identical CHAC digits for levels 2 to N. So nodes have $dist = \beta - 1, \beta \geq 1$, if they share a level-$\beta$ CH parent and so they have the same CHAC for levels β to N and they have an indirect colocality. Also let us refer to these two nodes in the same cluster, by having a logical distance, $dist$ of 0; if they belong to different clusters that share the same level-1 parent CH, then their $dist = 1$; and if their clusters share level-2 parent CH, their $dist = 2$; and so on. So if the shared CH is on level-n, where $n = [0 \ldots N]$, their $dist = n$.

Alternatively, if efficiency is prioritized over security, transaction verification takes place at the topmost level of the structure and transaction upwards propagation takes only one hop, as it is the case in our proposed SSD design.

In order to better explain the colocality concept, let's consider the situation where the nodes are uniformly distributed within the structure. A concept like the bin and ball hashing problem (Motwani et Raghavan, 1995) can be applied for the structure: we can then consider the number of IoTDs, I, as balls and the number of trees of the architecture ($\#$[level N]), E, as bins. The expected number of collisions (with $dist <> 0$) is:

$\binom{I}{2}\ (1/E) = (I(I-1)/2)(1/E) = O(I^2/E)$, which is in $O(E)$ if $I \approx C$, is $O(1)$ if $E = O(I^2)$, and is $O(I^2)$ if $E \approx smallconstant$.

So in case E (the number of trees at the topmost level) is a small constant, the expected number of collisions will dramatically increase. The smaller the value of E, the greater the node colocality achieved. The selection of the value of E should be made carefully, taking into account various constraints imposed by BC application requirements, such as consensus decentralization and the necessity to maintain a certain value for CLT, which represents the minimum number of trees required. Additionally, the choice of E

must adhere to BC MUT, which signifies the maximum allowable value for E. Exceeding this threshold necessitates clustering the topmost level and expanding the structure by adding more levels.

If we expect to leverage/enforce the colocality within the architecture, we need to work towards enforcing node colocality by getting the nodes frequently participating in the same Tx logically closer to each other in the architecture (hence increasing the rate of colocality for the nodes). In order to achieve a colocality, we need two nodes to be within the same tree, so we need to maximize the expected number of collisions as much as possible.

The colocality analysis introduced here is based on the hypothesis of a uniform random distribution of Tx among the IoTDs. In a real-life BC, the distribution might be different yielding better or worse results.

Provided a mechanism for analyzing the frequency of interaction between any two nodes participating in the same transaction (Tx) is available, along with a mechanism for predicting future frequency of interactions between those nodes, then colocality can be leveraged by adjusting the logical distribution of the nodes (node organization) to exploit the advantage of simplified processing. We will not pursue this aspect further and will leave the topic for future research.

4.5.3 A Quick Comparison Among All three BC Design Models in terms of Colocality

In order to achieve colocality, we need two nodes to be within the same tree. So we need to examine the expected number of collisions in all three architectures (the multi-tiered SSD, the two-tiered LSB and the one-tiered Flat architecture) in order to determine the number of node hops (visits) needed to verify transactions and blocks.

Let's compare the three BC models in terms of the worst case scenarios where a Tx needs to be verified by almost all the nodes of the architecture, in the absence of node colocality. This comparison has the goal of illustrating the superiority of our design in terms of efficiency, in worst case scenarios, as well as the importance of the colocality concept.

In the the flat design, in oder to verify a Tx, all nodes are visited, worst case in the absence of colocality, twice: once by messages looking for each Tx party PK, assuming that the requester and requestee do not know each other's PK. And in order to verify a block, all nodes are visited once by messages carrying the block. So all nodes are visited two times the number of Tx and one times the number of blocks generated.

In the two-tiered design, in oder to verify a Tx, all CHs are visited (as well as lists of length in $O(number\, of\, the\, structure's\, nodes)$ are read) once, in case the requester and the requestee do not belong to the same cluster. And in order to verify a block, the CHs are visited once by messages carrying the block.

By the bin and balls hashing analogy, if we denote by I, the number of IoTDs and by E, the number of clusters, the expected number of collisions (i.e. the requester and requestee are in the same cluster) is $\binom{I}{2}(1/E) = O(I^2/E)$. If we assume a two-tiered model having 10 000 IoTDs distributed in 500 clusters performing 20 000 000 Tx, the expected number of collisions is 99 990 collisions. In other words, this is the expected number of times where both nodes interacting in the same Tx exist within the same cluster. It also represents the expected number of Tx that occur between two nodes in the same cluster, which is 0.49% of all Tx. This means that all 500 CHs (each managing one cluster on the IoTD level), are visited 99.5% of the time to verify transactions (as well as lists of length in $O(number\, of\, the\, structure's\, nodes)$). Thus most of the time (99.5% of the time), all 500 CHs are visited while rarely (only 0.49% of the time), only one CH is visited in case of a two-tiered model. So to verify transactions, in case of node colocality, one CH is visited only 0.49% times the number of Tx, while without colocality, all CHs are visited 99.5% times the number of Tx. And in order to verify blocks, all 500 CHs are visited once to verify each block, which is only 5% of the number of nodes that would be visited using the flat architecture.

In our proposed multi-tiered design, to verify transactions, $O(all\, topmost\, CHs)$ are visited, worst case, once, if the requester and requestee are non-colocal (as well as lists of length in $O(number\, of\, the\, structure's\, nodes)$ are read once). Otherwise, if the requester and requestee are colocal, only one CH is visited. To verify a block, all topmost level CHs are visited once. Since our architecture is multi-tiered, we have control over the concentration of nodes in a tree, and our interest is to maximize the number of cases where the requester and the requestee fall within the same tree by increasing the number of the architecture's levels[9]. In our multi-tiered design, the number of CHs on one level is logarithmically bounded compared to the number of nodes on the levels below.

[9] The number of trees must always adhere to CLT and MUT, complying with the decentralization and node management overhead requirements.

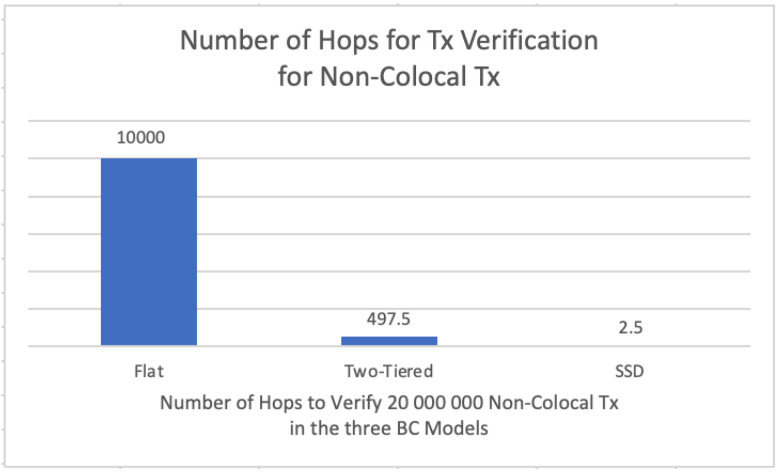

Figure 4.10 SSD - The Number of Hops to Verify Tx in the Three BC models without considering Colocality.

With the same number of IoT devices, increasing the number of levels results in fewer trees, thus the final count of trees depends on the chosen number of levels according to the application's requirements. If we assume that we have the same 10 000 IoTDs distributed into 500 clusters, and these clusters fall under five trees over more than two levels, we can see that the expected number of collisions is 9 999 000 collisions (which is the expected number of Tx that occur between two nodes within the same tree), which is 100 times more than the number of collisions that takes place assuming the two-tiered architecture. Thus, for 20 000 000 Tx, a 49.99% or, in other words, about half of those Tx occur between two nodes in the same tree, leaving 50.01% Tx, or about the other half of the Tx where all five topmost CHs are visited to perform the Tx verification. Thus to verify a Tx within our multi-tiered model, almost half the time, only one CH is visited in case of colocality (best case) while the other half of the time all topmost CHs, whose number is very small, are visited (worst case). Thus in order to verify transactions our SSD design, worst case, without even considering node colocality, only 2.5 CHs are to be visited by each Tx, which is 0.5% of the number of CHs visited assuming a two-tiered architecture, and is 0.025%of the number of CHs visited assuming a flat architecture, as in Figure 4.10.

To verify transactions our SSD design while considering colocality, only one CH is visited about half the time compared to only about 5% of the time in a two-tiered model and compared to 0% of the time in a flat model, as in Figure 4.11.

To verify blocks, the number of topmost CHs are visited (five of them), which is 1% of the number of the CHs

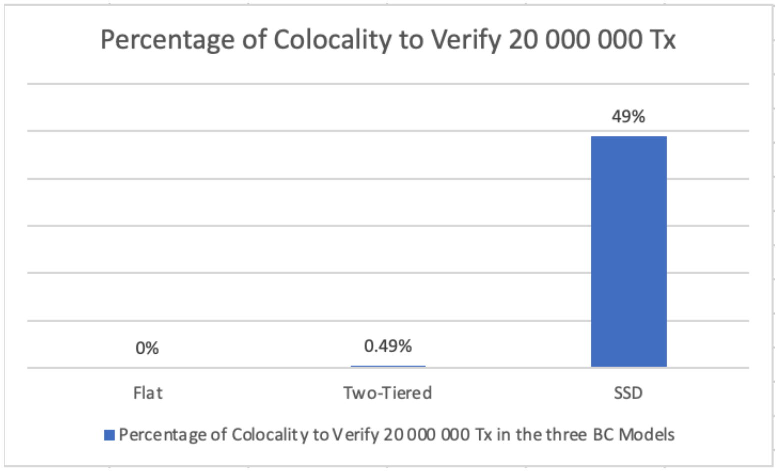

Figure 4.11 SSD - The Percentage of Colocality to Verify 20 000 000 Tx.

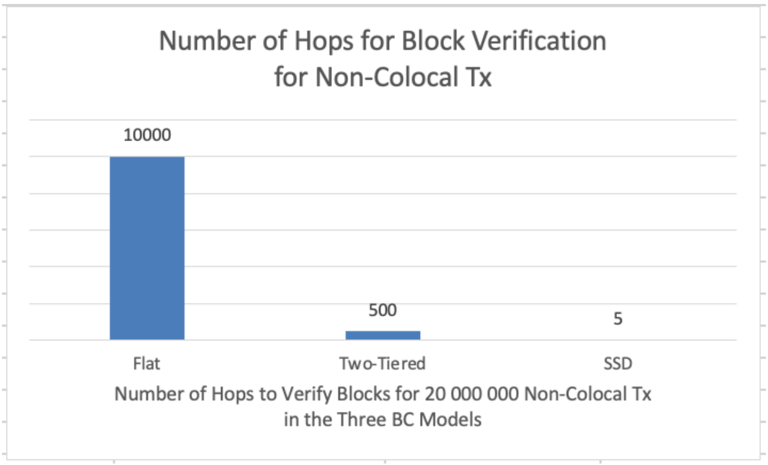

Figure 4.12 SSD - The Number of Hops to Verify Blocks in the Three BC models.

visited assuming a two-tiered architecture and which is 0.025% of the number of nodes visited assuming a flat architecture, as in Figure 4.12.

The comparisons made in this section consider the number of visits made in the case of a flat architecture assuming that the messages propagated are sent directly to the visited nodes, which actually underestimates the number of messages needed to achieve each visit, since the messages in a flat architecture are propagated by broadcasting and hence to achieve one visit, it takes $O(\zeta)$ hops where ζ is the number of nodes in the whole structure. Thus each of the calculation results made above for a flat architecture, let's call it α is actually α^{ζ}. These results put our architecture at an advantage over the flat architecture.

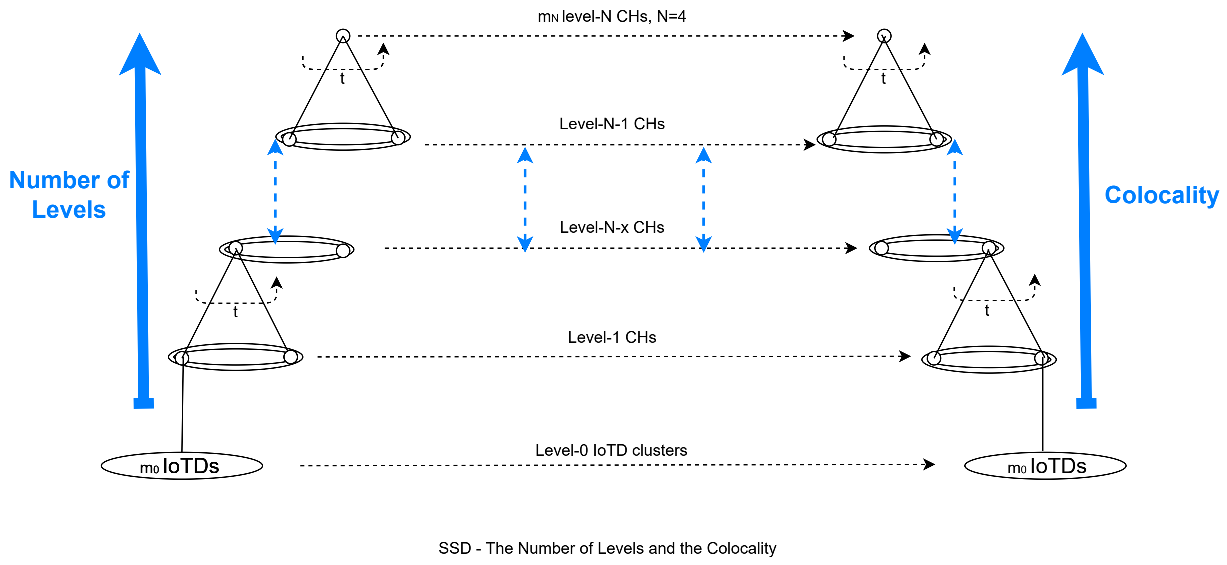

Figure 4.13 SSD - The Number of Levels and the Colocality.

As we can see from the previous examples, the more we increase the number of levels in the SSD architecture and the more we decrease the number of the topmost CHs, we increase the proportion of Tx with node colocality as in Figure 4.13 and hence we decrease the number of hops needed to be taken by the messages verifying the Tx, even when those Tx are non-colocal. We also decrease the number of hops needed to verify blocks. And hence, the total number of hops needed to complete the Tx journey towards BC decreases overall, which puts our architecture at an advantage over the two-tiered architecture.

Also we can see that adopting the colocality aspect may alleviate the latency related to about 50.01% of the Tx as in the case of SSD architecture, about 99.5% of the Tx in the case of the two-tiered model.

From the analysis, we deduce that colocality is achieved nearly half the time in our multi-tiered model, while it occurs in 0.5% of instances with a two-tiered model and never in a flat model. Thus, our model demonstrates greater achievability of colocality compared to both flat and two-tiered models, potentially reducing consensus time.

The ratio of IoTDs to CHs in the two-tiered model may vary depending on the application design, potentially resulting in different colocality outcomes. However, our SSD multi-tiered model outperforms the two-tiered model by offering a dynamic solution to scalability limitations and by making better use of the colocality aspect through increasing the number of topmost CHs and the number of the architecture levels as needed, according to CLT/MUT thresholds.

4.5.4 SSD Operation Under Hostile Environment

In our study, we concentrate on enhancing the scalability and efficiency of blockchain (BC) systems, prompted by several scalability and efficiency drawbacks identified in the literature regarding flat and two-tiered models. Our primary focus is on BC scalability and efficiency, all while ensuring proper functionality adhering to the requirements of consensus decentralization and dynamically maintaining a balance between decentralization and node management overhead.

Ensuring blockchain functionality in hostile environments is beyond the scope of our study, as it represents an additional layer of security that could be incorporated into our solutions and mechanisms to further enhance their resilience. Mechanisms such as Proof of Work (Qi *et al.*, 2023; Yu, 2023b) or LSB's trust algorithm (Dorri *et al.*, 2017a) may be adopted by future research to bolster our system's resilience to hostile environments.

In this chapter, we have introduced the Multi-tiered concept achieved through the development of the SSD model. In the following chapter, we present our methodology through which we will conduct and evaluate our research.

CHAPTER 5
RESEARCH METHODOLOGY

In this chapter, we describe the steps we take in order to proceed with our research and to obtain results for our simulations. In our research, we propose an IoTD-based BC multi-tiered node-organizing concept to enhance BC scalability and efficiency as well as a number of suggested design options to achieve the concept. The design options are evaluated and compared to each other. Each architecture corresponds to a specific set of project hypotheses.

The general approach that we follow starts by identifying the design goals we aim for, and by classifying them according to a priority defined by the design requirements. Inspired by the literature, we consider the architecture option that is closest to our design goals, the SSD architetcure, and we take the design goals as a foundation to improve that architecture in terms of node organization and routing (consensus-accelerating) algorithms applied. We test the improved components in order to evaluate their feasibility to reach the design goals.

The precise design requirements emerge based on the drawbacks that we pinpointed in the systems described by the literature review. We identify the required operations and design features required to maximize the observed strengths and to address the reported limitations. We also consider different data structures than those found in the literature to better depict our new node organization and to keep track of node data for the IoTDs and CHs throughout the levels. We build an event-driven simulation with cascading splits and merges invoked by bypassing cluster size thresholds spanning all levels and working their way bottom-up.

We build a first set of simulations for all three models: the flat, the two-tiered as well as the multi-tiered model achieved through the SSD architecture design option, and we use the same metrics in all three models in order to fairly compare between them. The goal of this set of simulations is to provide a proof of concept for the multi-tiered model compared to the two-tiered and flat models in terms of scalability and efficiency.

In our architectures, wherever tradeoffs between functionality (that favors a decentralized consensus) and complexity arise, we prioritize the choices that allow a fair consensus for all nodes while updating and/or maintaining their logical addressing (within the architecture to keep the relationships between each CH and its managed nodes), giving functionality a higher priority.

We perform theoretical complexity analyses for our SSD design independently from the simulations' data structure considerations, then we build our simulations for the three models to measure the practical complexity analyses of the design as coded in python while considering our complex data structures, in order to compare both complexity analyses.

For BC systems found in the literature, no theoretical results were found, thus we had to reproduce those systems in a number of separate simulations in order to compare their results to the SSD results.

We also perform a second set of simulation by building different variants of SSD architecture having 2, 4 and 8 topmost CHs, all performing the cascading splits and merges. The goal of this set of simulations is to provide a proof of concept of the colocality concept.

Our methodology to evaluate each design is based upon the following:

1. Given the infeasibility of creating a system of millions nodes, we rely on theoretical analysis as a first means for validating our models. This is achieved by evaluating the theoretical complexity and cost of operations involved with constructing the architecture, as well as with the splits and merges.

2. We also conduct performance analyses by simulations for IoTD-based BC mimicking our design in order to validate our theoretical complexity bounds and our proposed model functionalities.

3. We use python3 to recreate and stress test the flat, the two-tiered as well as the multi-tiered models. Since even simulations with millions of IoTDs may not be feasible, we extrapolate in order to extend our results aiming to find general trends resulting from varying the number of Tx and IoTDs as reflected on the amount of traffic, node interactions and required operations.

4. In order to build a simulation for our multi-tiered architecture, we first build a two-tiered architecture, then we overflow the clusters in the lower level to invoke the cascading splits and merges working their way upwards in the structure. The clusters at the highest level end up to be overflown as some nodes from the level below join that level. This causes the split of the clusters at the highest level and the promotion of some of that level's nodes to create a third level on top. All node movement will be reflected in the node data, including the node type (CH or IoTD), the node CHAC value, the node

managed nodes (for CHs), the node cluster value, etc... Once the levels are set, Tx generation process takes place at the IoTD level such that a Tx is transferred to the levels above towards the topmost CHs, the block-generating nodes.

5. Similar simulations are built for the two-tiered model as well as for the flat model (excluding the split and merge functions).
6. All simulations utilize the same parameters to determine the propagation delay functions for both the Tx and the blocks.

In this section, we reconsider the use cases mentioned in the thesis introduction in order to clarify the simulation hypotheses. The process of event generation is as follows in all aforementioned scenarios:

We assume that the Tx emitted and/or captured by the IoT sensors follow a Poisson process throughout the day. In terms of the position within the architecture, the node representing the Tx-generating IoT sensor would be either considered as an IoTD node located at the lower level of hierarchy (level 0) to produce transactions, considered as CHs located on one of the upper levels to maintain the hierarchical structure or considered as CHs located at the topmost level to interact with BC. In terms of the cluster position, we assume that the IoT sensor acting as Tx-generating IoTDs tend to fall randomly into different clusters. We assume that the position within a cluster are taken randomly or according to a realtime geographical distribution. The generated Tx are multi signature with two or more participants.

Finding a generic model for a wide variety of plausible use cases, we can deduce a model where transactions are generated following a Poisson process throughout the day, which is a common model for this type of simulations. The position in the hierarchy is at the lowest level IoTD level 0 for Tx generating nodes. Generally, a Tx is a multi signature and likely has two or three participants, all of which are likely located in different clusters following a uniform distribution model (for IoTD node positioning). This is the main reason why we model our simulation with two-party transactions.

Testing the SSD model is performed on a fixed number of IoTDs in a three-tiered structure with two CHs at the third level, then the number of CHs increases exponentially. As a validation we examine the possibility of achieving a consensus in $O(10 seconds)$ or less.

The following metrics need to be measured, as they have an impact on the quality of service:

1. The maximum number of Tx supported per second in order to be processed.
2. The throughput (number of Tx appended to the ledger per each second of the simulation duration);
3. The consensus time, which is the time elapsed between the Tx generation and its insertion into the ledger;
4. The number of IoTDs supported by the system, assuming adequate throughput and consensus time, was fixed to 256-260 IoTDs.

Metrics like the consensus time yielded by the system may provide points of comparison with other architectures, namely the flat and two-tiered architectures, in order to showcase the validity of our multi-tiered concept. The metrics may also provide points of comparison between the different variations (builds) of the SSD multi-tiered model in order to showcase the validity of the colocality concept.

The detailed criteria stemming from our design objectives enable us to rank the proposed architectures and decide whether a given architecture achieves a specific, prioritized set of goals with respect to BC functionalities, security and reliability requirements. The capacity to fulfill the objectives even as system size grows will be ascertained according to the trends and asymptotic behaviors identified by the analysis.

CHAPTER 6
THEORETICAL ANALYSIS OF THE SSD ARCHITECTURE

In this chapter, we analyze elements of the SSD architecture: the characteristics of the forest trees and the relationships between the height of the structure, the number of nodes and MUT. We also perform a detailed analysis of the SSD algorithm.

6.1 Trees of the Forest

The forest tree described in our architecture is similar to a B-tree (Cormen *et al.*, 1990) in the node insertion and the possible expansion of the tree, as well as the node deletion and the possible contraction of the tree. The advantage of a B-tree lies in the fact that its height grows logarithmically with the number of keys. Let us consider e to be the number of keys in the whole B-tree, t to be the maximum branching factor per node, and $O(\log_t e)$ to be the height of the B-tree. Each node has a number of ordered keys that is at least equal to the $t-1$ and at most equal to twice the $t-1$. All leaves are at the same depth, so the tree is always balanced in height. Starting at the root, each node can be reached in $O(t \log_t e)$. At each level from root down the B-tree, each node has to be searched for one of the $O(t)$ keys.

The expansion of the tree is triggered by an attempt to insert a new key into a full node. The expansion is based on choosing the median among the node keys in order to split the node around the median into two sets of $t-1$ ordered keys each, then to elevate the median upwards to a parent node on the next level above (to a parent with less depth). In case the parent node is full as well, that parent node needs to split before the key insertion takes place. And similarly, the node splitting can propagate all the way up to the root. The median of the node keys can be found linearly in $O(t)$, if a binary search algorithm is used, otherwise, it can be found in order of $O(\log t)$.

Our architecture shares some properties with B-trees, and is similar to a collection of B-trees having their roots located at level N. The number of such roots, m_N, is between MUT and CLT. In our design, clusters are equivalent to B-tree nodes, while nodes are equivalent to B-tree keys. Nodes can be CHs in level 1 to level N, or IoTDs on level 0. Levels $[1 \ldots N]$ contain CH clusters such that each CH manages one CH cluster of t CHs each, where t is an average branching factor, taken over all levels, such that $CLT \leq t \leq MUT$. The bottom level contains IoTDs, such that a CH from the level above manages only one IoTD cluster of m_0

nodes each. Similarly to B-trees, our structure is always balanced in height by design, since a level can be added or eliminated only if it is a topmost level. Reaching a topmost CH starts at the leaf level upwards in $O(1)$ using CHAC, while starting at each topmost CH, each node may be searched by reading a list of the CH's managed nodes.

IoTD node joins or departures lead to the expansion or contraction of the tree once MUT_0 or CLT_0 thresholds are reached within an IoTD cluster or once MUT or CLT thresholds are reached within an CH cluster; similarly to the expansion and contraction of a B-tree. Contrary to the regular B-tree structure, reaching a node starts at the leaf level and goes upwards towards the roots, taking the order of the height of the structure. Thanks to the use of CHAC, traveling the height of the structure will be in $O(1)$. Similarly to a B-tree, splitting a cluster will result into elevating a node upwards to the level above, however the split basically divides the cluster by two and the promoted node is not the median, it is a randomly chosen node, so finding that node is in $O(1)$.

The main difference between both splits is that the node data in our structure changes with every split (i.e. the cluster number that the node belongs to and the node CHAC among other info); whereas in B-trees, the node information does not change. Also in order to maintain the integrity of our structure where every cluster has to be managed by exactly one CH, every split must be always accompanied by a merge and vice-versa; whereas B-trees do not have such a restriction.

6.2 The Height of the Structure and MUT

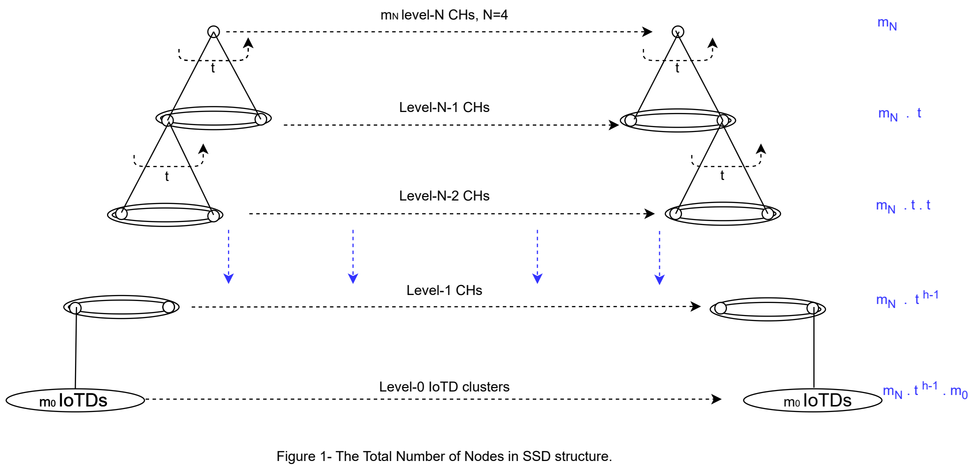

Figure 6.1 SSD - Total Number of Nodes

Let us analyze the number of nodes for a structure of a height h and an average branching factor, taken over all levels, t, such that $CLT \leq t \leq MUT$. At level N (depth $d = 0$), we get m_N CH nodes (such that $CLT \leq m_N \leq MUT$) ; at level $N - 1$ (depth $d = 1$), we get $m_N \cdot t$ CH nodes; at level $N - 2$ (depth $d = 2$), we get $m_N \cdot t \cdot t$ CH nodes and so on till we reach level $N - (h - 1)$ (depth $d = h - 1$), where we get $m_N \cdot t^{h-1}$ CH nodes. Whereas at level 0 (depth $d = h$), we get $m_N \cdot t^{h-1} \cdot m_0$ IoTD nodes, as in Figure 6.1.

So the total number of nodes in the structure is:

$T_{structure} = m_N \cdot (\sum_{j=0}^{h-1} t^j + (t^{(h-1)} \cdot m_0))$ nodes.

To calculate the height of the structure, h, we can first consider the height of the CH portion of the forest, h' (without considering the IoTD level), then we can add one level to that height later on in order to calculate the whole structure while including the IoTD level. Thus with C CHs, spread over the forest levels, and with m_N root nodes, we get:

$C = \sum_{j=0}^{h'} m_N t^j$, so $C = m_N((t^{h'+1} - 1)/(t - 1))$

Thus the height, h', is:

$h' = (\log_t(((C/m_N) \cdot (t - 1)) + 1)) - 1$. Hence the full height, h, including the IoTD level is $h = (\log_t(((C/m_N) \cdot (t - 1)) + 1)) - 1 + 1 = (\log_t(((C/m_N) \cdot (t - 1)) + 1))$. Since the branching factor depends on the per-level values of CLT and MUT, we can consider an average value for t taken over all levels, $CLT \leq t \leq MUT$.

Comparing the height of our forest with a B-tree height assuming that each structure contains the same number of nodes, we can see that the height of a B-tree grows logarithmically with the number of keys, which is equivalent to the number of nodes in our structure. While the height of our structure grows logarithmically with a fraction of that number for the following reasons:

1. We compare a tree to a whole forest thus considering the same number of nodes distributed all over both structures, the height of the forest is much less that the height of a tree.
2. The node distribution in a B-tree is uniform throughout the levels, while the node distribution in our

structure is not uniform since the nodes are more concentrated in the IoTD clusters (which are very dense) at the leaf level while the top levels have smaller size clusters comparably, such that average branching factor t taken over all levels is small and the number of overall CHs C is much smaller than the entire number of nodes.

3. We compare the number of what is equivalent to the number of our nodes in a B-tree to the number of our CHs C, which is too small compared to the entire number of nodes because in our architecture, IoT clusters are much more dense than CHs clusters.

4. In our formula, the number of CHs C is further divided by the number of topmost CHs and is multiplied it by a a comparably small value of t.

All those reasons make the height of our structure much more advantageous than a B-tree height.

The smaller height of our structure does not provide an advantage to the consensus-accelerating protocol since the latter leverages the use of CHAC positioning mechanism which helps propagating nodes upwards throughout the levels of the structure in $O(1)$, no matter the height of the forest. However the small height is significantly useful for the performance of the SSD algorithm since the algorithm has to account for node and cluster data updates all over the levels of the forest. And in the simulations, we will observe that node and cluster data are stored in complex data structures with multiple dimensions due to the intricacy of the design. The number of dimensions in the data structures is equal to the number of levels in the structure, and the complexity of the SSD algorithm grows logarithmically with the number of levels in the structure. Thus the smaller the height of the structure, the more efficient the SSD algorithm becomes.

To calculate the MUT threshold per level, we consider the processing capacity of a CH for each level. In other words, the processing cost for the nodes managed by each CH. We consider the case where there is a small number of Tx and a large number of managed nodes versus the case where there is a small number of managed nodes and a large number of Tx. The MUT provides a guarantee that the CH are capable of performing the processing in terms of the number of managed nodes and the number of Tx generated.

Let us consider the number of nodes managed by each CH. At level 1, a CH manages 1 cluster of m_0 IoTDs or in other words, $t^{1-1} \cdot m_0$ IoTDs; at level 2, a CH manages 1 cluster of t nodes, each of which are managing 1 cluster of m_0 IoTDs, or in other words, $t^{2-1} \cdot m_0$ nodes; and so on till we reach level N where a CH manages $t^{N-1} \cdot m_0$ nodes.

Any node at level $N - d$, where d is the node's depth in SSD, with a branching factor, t, will manage a total number of nodes of $t^{level-1} \cdot m_0$, and this will hence need a maximum per-level processing effort, MUT that is the maximum of all the MUT values calculated per CH situated on the same level, where each such MUT is calculated as: $MUT \geq \gamma \cdot (t^{level-1} \cdot m_0)$, where γ is the maximum per managed node processing cost, as required by the application.

6.3 Theoretical Analysis of the SSD Algorithm

In this section, we analyse the SSD algorithm - Part 1 and Part 2, so we consider the case where the local cluster that we split or merge is at the topmost level of the architecture as well as the case where it is on a non-topmost level of the architecture in each part. The goal of the analysis is to evaluate the complexity of the operations for a comparison with the complexity of the same operations as done through the implementation.

6.3.1 Self-Scaling Algorithm-part1 - The increase of cluster size on level $i, i \neq N$

If size(localCluster) $\geq MUT$ and localCluster is at level $i \neq h$:

a) Splitting one of level-i clusters as it exceeds MUT, will cost $O(M)$, where M is the number of nodes of the to-be-split level-i cluster.

b) One node will leave level i and will join level $i + 1$: The cost for one level-i node to leave level i of Y nodes is $O(Y)$; while the cost for the node to join level $i + 1$ of Z nodes is $O(Z)$. Adding both costs: Since we know $\#[leveli + 1]$ always has a number of nodes that is less than $\#[leveli]$'s, we know that $O(Y)$ is the higher order term.

c) Merging the level-$(i - 1)$ cluster that was previously managed by the promoted node, with the smallest level-$i - 1$ cluster. We consider that merging two clusters of L and W nodes costs the same as clustering both clusters. So the cost is $O(L + W)$.

The total cost of horizontal re-clustering is $O(max(M, Y, (L + W)))$, which is is $O(max(Y, (L + W)))$, where Y is the number of nodes of level i where the split takes place and L and W are the number of nodes in the clusters to merge at level $i - 1$.

6.3.2 Self-Scaling Algorithm-part1 - The increase of cluster size on level $i, i = N$

If size(localCluster) $\geq MUT$ and localCluster is at level $i = h$:

a) Considering i in $[0 \ldots N]$ to identify the levels, clustering the nodes of the current topmost level-$N = max(i)$ with a number of clusters not less than CLT, will cost $O(Y)$, where Y is the number of nodes on the level to be clustered (the current topmost level).

b) Creating a new layer with S nodes, where S is the subset of nodes not less than CLT that have been elevated from the current level N, where $N = max(i)$, to form the newer topmost level N, where $N = max(i+1)$. So the cost for a subset S of nodes to leave level i lower layer of Y nodes will be $O(S \cdot Y)$; while the cost for level-i S nodes to join level $i+1$ of Z nodes will be $O(S \cdot Z)$. Adding both costs: Since we know $\#[leveli+1]$ always has a number of nodes that is less than $\#[leveli]$'s, we know that $O(S \cdot Y)$ is the higher order term.

c) Merging S clusters with the smallest size clusters of the same subtree at level $N-2$ (level $i-1$) will take place. Each one of such S clusters was formerly managed by one of the S nodes that moved from level-$N-1$ to the new level N. We consider merging two clusters of L and W nodes will cost the same as clustering both clusters. So the cost is $O(L+W)$. Repeat that for all clusters formerly managed by the S nodes. So the cost is $O(S \cdot (L+W))$.

The total cost of vertical re-clustering is $O(max((Y, S \cdot Y, S \cdot (L+W)))$. Thus the total cost is $O(max((S \cdot Y, S \cdot (L+W)))$, where S is the number of nodes at the newly created level N, Y is the number of nodes at level $N-1$, and L and W are the number of nodes in the clusters to merge at level $N-2$.

6.3.3 Self-Scaling Algorithm-part2 - The decrease of cluster size on level $i, i \neq N$

If size(localCluster) $\leq CLT$ and localCluster is at level $i \neq h$:

a) Merging the level-i local cluster that is less than CLT (let us call it "cluster F") of L nodes, with the smallest size cluster (of W nodes) on the same subtree under the same parent, will cost $O(L+W)$.

b) Cluster F's CH will leave level $i+1$ and will join level i. The cost for a node to leave level $i+1$ of Z nodes will be $O(Z)$; while the cost for that node to join level i of Y nodes will be $O(Y)$. Adding both costs: Since we know $\#[levelN = i+1]$ always has a number of nodes that is less than $\#[leveli]$'s, we know that $O(Y)$ is the higher order term.

c) one level-$i-1$ clusters of size P will split into two so that the demoted CH that joined level i would manage one of the resulting clusters. This costs $O(P)$.

Adding the costs, we get a total cost of $O(max((L+W), Y, P))$, where L and W are the number of nodes in the clusters to merge at level i, Y is the number of nodes at level i and P is size of the cluster to split at level $i-1$. So the total cost is total cost of $O(max(Y, P))$.

6.3.4 Self-Scaling Algorithm-part2 - The decrease of cluster size on level i, $i = N$

If size(localCluster) $\leq CLT$ and localCluster is at level $i = h$:

a) Each of level-$N = max(i)$ Z nodes get lowered as a regular level-$N-1$ cluster node, $Z \leq CLT$. The cost for Z nodes to leave level N of Z nodes will be $O(Z^2)$; while the cost for Z nodes to join level $N-1$ of Y nodes will be $O(Z \cdot Y)$. the value of N decreases to eliminate the the topmost level where Z nodes left. Adding both costs: Since $\#[leveli-1]$ is always larger than $\#[leveli]$, then $O(Z \cdot Y)$ is the higher order term.

b) All clustering on the newly called level-N, will be reverted, to form one cluster. This will cost the re-clustering of all Y and the joined Z nodes, so it is $O(Y+Z)$.

c) Z Level-$N-1$ (using a new value of N) clusters of maximum size, T, will split so that half of the resulting clusters would be managed by the demoted Z nodes. Thus the split cost will be $O(Z \cdot T)$.

Thus the total cost of the topmost layer elimination is : $O(max((Z \cdot Y), (Y+Z), (Z \cdot T)))$. Thus the total cost is $Omax((Z \cdot Y), (Z \cdot T))$, where Z is the number of nodes at level $N = max(i)$ to be eliminated, Y is the number of nodes at level $N-1$ and T is the maximum size of the Z clusters to merge at level $N-2$.

6.3.5 Decreasing the cluster size

Decreasing the clustering may involve one of the following level elimination cases. In our design and simulation, we only consider the topmost level elimination.

1. Topmost level elimination:

   If size(localCluster)$\leq CLT$ and localCluster is at level $i = h$:

   The complexity is calculated as shown above.

2. Intermediary level elimination:

   If size(all localCluster)$\leq CLT$ and all localCluster is at level $i \neq h$:

   a) Every level-i node (for all R nodes) gets lowered as a regular level-$i-1$ node of the cluster it used to formerly manage. The cost for R nodes to leave level i of R nodes will be $O(R^2)$; while the cost for R nodes to join level $i-1$ of Q nodes will be $O(R \cdot Q)$. Level $i-1$ ends up having $Q+R$ nodes, where $R = S \cdot t$, and $Q = S \cdot t \cdot t$, S is the number of nodes at level $i+1$ and t is the branching factor. Adding both costs: Since $\#[leveli-1]$ is always larger than $\#[leveli]$, then $O(R \cdot Q)$ is the higher order term.

   b) Decrement i for level-$i+1$ S nodes as well as the levels above. So level $i+1$ becomes level i with a smaller number of nodes (S) than before. This costs $O(1)$.

   c) Level-$i-1$ clusters (which include $Q+R$ nodes) will perform a number of $2 \cdot S \cdot t$ merges in order to match the new $\#[leveli]$ number of managing nodes, S, times t, where $CLT \leq t \leq MUT$. If the size of the average resulting clusters is W, where $CLT \leq W \leq MUT$, then the cost will be $O(2 \cdot S \cdot t \cdot W)$.

   Thus the total cost of a level-i layer/middle CHORD ring elimination is $O(max(R \cdot Q, 1, (2 \cdot S \cdot t \cdot W)))$.

   Thus the total cost is $O(max(R \cdot Q, (2 \cdot S \cdot t \cdot W)))$, where R is the number of nodes at level i to be eliminated, Q is the number of nodes at level $i-1$, S is the number of nodes at level $i+1$, W is the size of each cluster resulting from the merges that will take place at level $i-2$ and t is the branching factor.

The theoretical analysis described above pertains to an SSD structure with three levels. Increasing the number of levels in the SSD structure may yield different results because each CH retains information about its managed cluster (or subtree) nodes, and this information changes with each split and merge. Therefore, the node's information needs to be updated at every parent CH, affecting all parent CHs across all levels. Thus, the complexity increases with the number of levels in the hierarchical structure.

Due to the complexity of the SSD model, which requires accessing many nodes in order to keep track of the node dynamically changing data as well as the clusters node modification during the splits and merges (that involve a number of node repositioning in the structure), a theoretical analysis of the SSD algorithm might be too general and might differ from a real coding simulation complexity of the SSD model, a simulation that includes all the aspects of accessing all the required node and cluster data structures as well as copying all the required data that are involved in the system's splits and merges. Thus we aim to verify the theoretical analysis made thus far with the complexity analysis of our design as it is simulated in real code simulation programmed in Python that will serve both as a proof of concept for our SSD design, as well as a sample simulation example to provide a more feasible and comprehensive complexity analysis according to functional simulated BC requirements.

Therefore, now that we presented the theoretical analysis of the SSD model in general, we can present the simulation as well as the complexity of the SSD model as it is simulated according to our Python coding, in the next couple of chapters, in order to allow for such comparison.

CHAPTER 7
SIMULATION OF BLOCKCHAIN OPERATION

BC can be implemented as a flat one-tiered application as in Bitcoin (Nakamoto, 2009), a two-tiered application as in BlockSim (Alharby et Moorsel, 2022) or a multi-tiered application as in our SSD simulation, presented in this study. The goal of this section is to simulate a BC operation in order to evaluate the time the Tx takes from the time it gets generated by an IoTD until it gets appended into BC (after being inserted in a block). The ultimate goal is to measure that time in the SSD model, in the flat model as well as as in the two-tiered model and to compare them, and also to perform comparisons among different variants of the SSD model (having different number of topmost level CHs). Since BC is an overlay hypothetical design concept that is built on top of a real physical network, measuring the time of a Tx journey is made possible by measuring the number of hops that represent the different interactions that a Tx needs to make and the number of nodes it needs to encounter in order to complete its journey. Measuring that time through simulation does not take into account the actual physical network propagation delays[1] since we have no control over such delays because nodes may be physically far apart, with several routers in between. What we have control over is the node organization and the BC design in order to decrease the number of such hops as much as possible. Hence, our design is only concerned about the overlay network (ON) delays as they occur in a typical BC system. Therefore, we aim to evaluate the impact of node organization and BC design on the ON delays, assuming these impacts will be eventually reflected at the physical level. For the purposes of the simulation, we have modeled the propagation delays between the overlay nodes as drawn from an exponential distribution to represent the delay that might happen during each hop the Tx takes achieving its journey towards BC. It is in our interest to decrease the numbers of such hops in order to achieve a better efficiency for the consensus, and the simulation aims to prove whether reducing the number of hops is achievable using our SSD architecture.

Our SSD multi-tiered architecture has the goal of decreasing the number of hops necessary for a Tx to make in order to reach BC by exploiting two important concepts:

1. The multi-layered architecture benefits from nodes colocality and hence eliminates unnecessary hops for a Tx.

[1] measuring real network propagation delays can not be performed through simulations, hence it is out of the scope of our study.

2. Adopting the CHAC concept eliminates unnecessary hops between levels, no matter how deep the forest trees are.

In our implementation, we use Python version 3 along with pandas, numpy, sklearn and xlsxwriter to simulate a BlockChain based on the Proof-of-Work (PoW) consensus.

In the following, we start by introducing the simulations design hypotheses for a typical flat BC, for a two-tiered BC, and for our SSD BC, followed by the features applied in the simulations and a quick comparison among all three BC design concepts. Then we follow with the design parameters, the simulation design details, then the simulation results.

7.1 Typical Flat BlockChain Simulation Hypotheses

In our flat BC simulation, we assume the following hypotheses:

1. The nodes do not have logical addresses.
2. Flat BC systems are characterized by broadcasting the messages within the P2P ON, so that a message carrying a Tx can be sent to the next node that will, in turn, send it to next node, and so on till the message reaches the block generating node, as one possibility; or the message can take shortcuts being propagated by other nodes that are closer to the block generating node, as another possibility. This broadcast characteristic causes the messages, in order to propagate from one destination to another, to take $O(\lambda)$, worst case, where λ is the number of all the BC system, or to take $O(1)$, best case, if the block generating node is the next node ahead. Such propagation is hard to simulate, and hence we decided to consider only a portion of the nodes as being block-generating nodes, and simulate the messages carrying the Tx to start at a couple of IoTDs, thus, taking the maximum of both propagation delays starting at the IoTDs and reaching the block-generating node, and repeating that process for each of the block-generating nodes; then the Tx will be propagated to all other regular nodes (thus adding the previous delay to the maximum of all propagation delays towards the rest of the non-block-generating IoTD nodes). Thus we assume that the requester and requestee know each other's Public Key (PK) in advance, for simplicity.

3. In the physical network, the IoTD nodes are assumed to be randomly[2] distributed and the resulting delays are modeled by a propagation delay drawn from an exponential distribution for every hop that the Tx carrying-messages needs to take in order to achieve its journey towards BC. We differentiate between the Tx delays as the Tx propagates from the IoTDs towards the block-generating nodes as well as towards all the BC nodes versus the block delays as they propagate between the block-generating nodes.

4. Most of the nodes may attempt to generate a block that contains a certain number of Tx. Once a block gets generated by one node, it gets broadcast towards all the other nodes to get verified.

5. Visits are in $O(4 * number\ of\ the\ structure's\ nodes)$ (worst, without node colocality and best case, with node colocality). These visits occur because most of the nodes are visited two times looking for each Tx party's public key. Then all nodes are visited once a Tx is verified since the Tx needs to propagate towards all nodes to give each node a chance to append the Tx into a block. And finally all nodes are visited once by the message carrying each block to verify the block.

6. A Tx has to be appended only once to a block at any node of the structure. The node with a Tx pool that reaches $Tmax$ Tx first gets to build the block that is currently built, and gets to append that block into the BC. All Tx included into that block should not be appended again into any future blocks. All the other blocks that are currently being built be other nodes are to be ignored.

7. The consensus modeled is PoW, simulated by a simplified consensus mechanism.

8. In order to simulate Tx verification, we simply ensure that a Tx propagates towards the nodes involved in Tx verification, namely all BC nodes.

9. In order to simulate block verification, we simply ensure that a Tx propagates towards the nodes involved in block verification, namely all the block-generating nodes.

7.2 Two-Tiered BlockChain Simulation Hypotheses

In our two-tiered BC simulation, we assume the following hypotheses:

1. The nodes do not have a logical addresses.

[2] In the absence of information about the nodes distribution, we chose to simulate them as being randomly (uniformly) distributed.

2. The IoTDs at the lower level are clustered and can generate Tx, while only the upper level nodes (CHs, having higher capabilities) can build, verify and append blocks into BC.

3. In the physical network, the IoTD nodes are randomly distributed and the resulting delays are modeled by a propagation delay drawn from an exponential distribution for every hop that the Tx carrying-message needs to take in order to achieve its journey towards BC. We differentiate between the Tx delays as a Tx propagates from the IoTDs upwards towards the CH level, versus the Tx delays as a Tx propagates between the CHs. We calculate the latter delays similarly to the calculation of block delays as blocks propagate between the CHs.

4. A Tx gets stored in the Tx pool of each CH. Once the pool reaches a threshold, $Tmax$, the CH may start building a block.

5. Once a block is built, it propagates (in a message) towards all CHs (to simulate block verification).

6. Visits in $O(CHs)$ are performed (and lists of length in $O(number\, of\, the\, structure's\, nodes)$ are read once) in case that the requester and the requestee do not belong to the same cluster, worst case. Otherwise, visits in $O(1)$ are performed if both the requester and requestee are in the same cluster, best case. In the latter case, no need to read the list of the managed cluster nodes thanks to the use of CHAC that helps determine whether a pair of IoTDs are colocal in only $O(1)$. Once a Tx is verified, visits in the $O(CHs)$ are performed as the Tx needs to propagate towards all CHs to give each CH a chance to append the Tx into a block. To verify each block, visits in the $O(CHs)$ are performed once. Thus we assume that the requester and requestee do not know each other's Public Key (PK) in advance.

7. All Tx have to be appended only once to a block at any node of the structure. The node with a Tx pool that reaches $Tmax$ Tx first, gets to build the block that is currently built, and gets to append that block into BC. All Tx included into that block should not be appended again into any future blocks. All the other blocks that are currently being built by other nodes are to be ignored.

8. The consensus modeled is PoW, simulated by a simplified consensus mechanism.

9. In order to simulate Tx verification, we simply ensure that a Tx propagates towards the nodes involved in Tx verification, namely the CH managing the IoTD generating the Tx.

10. In order to simulate block verification, we simply ensure that a Tx propagates towards the nodes involved in block verification, namely all CHs.

11. A random clustering algorithm is used to cluster the IoTDs.

7.3 SSD BlockChain Simulation Hypotheses

In our SSD simulation, we assume the following hypotheses:

1. IoTD nodes are at the lower level of the architecture (at level 0), they are clustered and managed by CHs.

2. IoTD clusters are assumed to have similar size to avoid bottlenecks at the CHs and to ensure a fair consensus time for all nodes, in case where nodes, including CHs, have comparable capabilities. The modularity function of the clustering algorithm is biased when the algorithm identifies the clusters with lower number of members to join a neighboring cluster having the highest number of members, in case of a merge. Whereas a uniform modularity function will allow clusters to merge with a neighboring cluster having a small number of members, which is more desirable. Though in our simulation, we used a random clustering algorithm, applying a clustering algorithm with a non-biased modularity function (K-means clustering algorithm as an example) along with a design that ensures a load-balanced structure is assumed and should be explored.

3. The CHs are located at the upper levels of the architecture (levels 1 to N). Levels-1 through N-1 CHs are clustered and managed by upper level CHs. Level N is the topmost level, the only level where the CHs are in one cluster and are allowed to verify and build blocks.

4. The CHs store the logical addresses (CHAC) as well as the Public Keys (PK) along with the node data of the nodes they manage.

5. There are three types of nodes, according to their functionality and position in the structure. Only the lower level nodes (the IoTDs) can generate Tx, while only the topmost level nodes (topmost CHs, having higher capabilities) can build blocks and append blocks into the BC. The CHs at intermediate levels, possessing greater capabilities, not only assist in maintaining the structure but also facilitate splits, merges, and management of lower nodes.

6. In the physical network, the IoTD nodes are randomly distributed and hence the resulting delays in the ON are modeled by a propagation delay drawn from an exponential distribution for every hop

that the Tx carrying-message needs to take in order to achieve its journey towards BC. We differentiate between the Tx delays as a Tx propagates from the IoTDs upwards towards the topmost level versus the Tx delays as a Tx propagates between the topmost level CHs. We calculate the latter delays similarly to the calculation of block delays as blocks propagate between the topmost level CHs.

7. When nodes join a cluster, the cluster size may exceed the Management Upper Threshold (MUT), which triggers a cluster split into two clusters.

8. When nodes leave a cluster, the cluster size may become less than the Control Lower Threshold (CLT), which triggers a cluster merge with the smallest size cluster on the level.

9. The consensus modeled is PoW, simulated by a simplified consensus mechanism. We modeled Proof of Work [6] consensus (Qi *et al.*, 2023; Yu, 2023b) simply as a means of comparison between models. Since our approach is consensus agnostic, we expect similar results would be obtained with other consensus mechanisms.

10. The different node actors are : the requester, the requestee, the requester parents, the requestee parents, the requester topmost parent (topmost CH), the requestee topmost parent (topmost CH) and the other topmost CHs. The Tx as well as the signatures, the PK and the blocks are carried in messages that propagate among the nodes. The requester is the node that generates the Tx. The requester parents are the nodes that propagate the Tx upwards. The requester topmost parent propagates the Tx towards other topmost CHs. The other topmost CHs add the Tx to their Tx pools in order to build a block. Once $Tmax$ Tx are in a topmost CH pool, a block is built by that CH, then the block gets propagated towards all other topmost level CHs. In the simulation, the block propagation is performed by scheduling the block to be received by other topmost CHs (to simulate the block verification).

11. Each topmost level CH keeps track of its managed nodes data (node ID, cluster ID, CHAC, ..etc) for the clusters it manages throughout all levels and keeps that data in a managed nodes list.

12. Tx are multi-signature with a requester IoTD and a requestee IoTD. When the requester and requestee are in the same tree, the topmost CH common to both Tx parties holds their Public Keys (PKs) and Tx verification takes place at that topmost CH (in $O(1)$) by checking that the value of the public key hash ($h(PK)$) matches the specified value in the previous Tx from the same Tx parties on the ledger and that the parties' current signatures may be validated with their PKs. In the event that no common topmost CH is found (non-colocal requester/requestee), all topmost level CHs are visited by a message

carrying the requester PK and the CHs managed nodes list are searched for the requestee's PK. The Tx is verified by the topmost CH where the requestee PK is found. In that case, $(O(all\, topmost\, CHs))$ are visited. In order to simulate Tx verification, we simply ensure that a Tx propagates towards the nodes involved, as specified in the Tx verification process. Thus we assume that the requester and requestee do not know each other's Public Key (PK) in advance. Once a Tx is verified, it propagates towards all topmost CHs to give each CH a chance to append the Tx into a block. All topmost level CHs attempt to generate a block that contains a certain number of Tx. Once a block is generated by one topmost level CH, the CH propagates the block in a message towards all the other topmost level CHs to verify it.

13. Tx are created and assigned creation times that are drawn from a Poisson process following an exponential distribution, whereas the choice of the requester and the requestee nodes is random as it is based on a uniform distribution.

14. As far as a Tx is concerned, the goal for BC is to propagate a Tx (carried by a message) from the bottom level, where the IoTDs reside, towards topmost CHs, so that such CHs get the same chance to build blocks. When the Tx is generated by colocal nodes, the Tx is transferred to the requester's topmost parent CH first then it gets propagated to all other topmost CHs so that they get the same chance to append that Tx to their Tx pool to build the block, and the first Tx pool that reaches $Tmax$ Tx is the pool where the block will be built. All the Tx used for that pool will be excluded from consideration for future blocks.

15. If both Tx parties (requester and requestee) reside in the same tree, the message carrying the Tx propagates towards the common top-level CH before reaching other top-level CHs. Otherwise, the message carrying the Tx propagates from the bottom level towards the requester's top-level CH while the requestee information is obtained by searching all other top-level CHs managed nodes.

16. Tx have to be appended only once to a block. Each block contains up to $Tmax$ Tx.

17. A blocks is appended into BC, by one of the topmost CHs. This is assumed to take place by using a hash function that includes the hashing of the previous blocks into the current block. In our simulation, the hashing process was simply modeled by including the previous block ID into the current block.

18. The values of MUT and CLT may vary as applied to the different architecture levels according to each SSD application needs. In our simulation, we use the same value for MUT throughout the levels and we use the same CLT value throughout the levels, for simplicity. These values, however, vary from

one simulation to another as we vary the number of the topmost CHs to simulate the different SSD builds.

19. Node reliability is assumed to be in place but in real life BC systems, reliability might present an issue: Node failure may occur at any point of the P2P system, thus a mechanism to verify connectivity among IoTDs (CHORD as an example) may be required and is assumed. A mechanism is assumed to help quickly adapt the architecture to churn caused by frequent new node joins and old node departure or failure; and to efficiently provide a location service for the nodes that store a particular data item, even if the system's organization is continuously changing.

20. Supporting node mobility to accommodate the node movement within the system, is assumed.

7.4 Measured Metrics

In this section, we discuss the metrics to be measured in the simulations as introduced in the methodology chapter, since such metrics have an impact on the quality of service of the BC system as it performs in the context of the case studies mentioned:

1. Metric: The number of levels necessary to ensure a consensus time of $O(10 seconds)$ maximum for a given number of IOTDs.

   Considering only three levels in our simulation was sufficient to measure the latency of SSD architecture. More levels can be added in case the number of level-N CHs becomes larger than MUT, so splits may take place at the topmost level (along with the associated required merges) and another topmost level may be added. Considering the BC functioning according to our multi-tiered design, adding more levels to the architecture does not affect the latency or the complexity of the operations since traveling all levels will be performed in $O(1)$ (one hop only) thanks to the use of CHAC along with the multi-tiered architecture. Thus the number of levels does not affect the consensus time however it has to be be adjusted accordingly to always adhere to MUT bound.

2. Metric: The maximum number of Tx supported per second in order to be processed by the CHs verification for a given number of IOTDs.

   We do not consider Tx verification in particular into our simulation since our main concern is consensus latency in terms of the number of hops performed among the different nodes of the BC system as they are organized in a multi-tiered architecture. In order to measure the number of Tx processed by the BC system, we considered producing 500, 1000 and 2000 Tx in one second to measure the reliability of the simulation. The simulation proved to be reliable processing that large number of Tx per second, however due to the computation limit of the laptop used to run the simulations, we had to stop at 2000 Tx/second. Our architecture ran smoothly with that number of Tx and based on that trend, we can expect the same smooth performance for a very large estimated number of Tx provided more computing power is available, assuming that the trends will continue and that the current SSD architecture will remain applicable.

3. Metric: The growth rate of the incoming Tx that may be supported (interval of Tx arrival). The growth rate can be measured by setting counters at topmost CHs to measure the number of processed Tx at different time intervals.

   In the simulation, Tx at topmost CHs are forwarded in a simple process: once a topmost CH receives a Tx relayed from an IoTD, the CH either 1) verifies the Tx locally if both the requester and the requestee are colocal, or 2) the CH sends the Tx to all other topmost CHs to look for the requestee information and the verification takes place where the requester's information is found. Once the Tx is verified, the verifying CH sends the Tx to all other topmost CHs so that they append the Tx into blocks. Therefore the Tx will be processed by all topmost CHs any way.

   The incoming rate of Tx depends on Tx creation rate which is generated according to an exponential function to represent the probability distribution of the time between events in a Poisson point process, i.e., a process in which events occur continuously and independently at a constant average rate.

4. Metric: The throughput (number of Tx appended to the ledger per second);

   In the simulation, the throughput is measured as the number of Tx per simulation duration (as in BlockSim (Blo, 2020)).

5. The metric: The consensus time, which is the time elapsed between the Tx generation and its insertion into the ledger;

   In the simulation, the consensus time is exactly calculated as mentioned in the metric.

6. Metric: The number of IoTDs supported by the system assuming adequate throughput and consensus time.

For SSD architectures, we have built simulations supporting 30 IoTDs (in a simulation having two CHs), supporting 60 IoTDs (in a simulation having four Chs) and supporting 260 IoTDs (in simulations having two, Four and eight CHs). Following that trend, we can also estimate having a very large estimated number of IoTDs provided more computing power is available, assuming that the trends will continue and that the current SSD architecture will remain applicable.

7.4.1 The Features Applied in the SSD Simulation

In the simulation, we took into consideration all the simulation hypotheses stated above except for the fact that we skipped the verification process of the requester/requestee PKs and signatures as well as the Tx and block verification. This is due to the fact that the time taken by these verification will be the same whether the architecture is flat or tiered and so it will not help answer the research questions. Also since the focus of our research is on improving the BC consensus time through clustering over many levels in order to improve the propagation delay between the nodes, implementing a specific type of consensus will not directly affect our results, and hence we simulated the block appends to the Blockchain to be done so that the nodes would take turns to generate the blocks after waiting for a random time, simulating PoW, by introducing a delay that models the time taken by the fastest CH to create a block. Once a Tx is appended into a block it will get eliminated from the set of the Tx available for the other nodes to generate blocks from.

7.4.2 Simulation Parameters

In order to obtain valid comparison results, we studied and simulated a flat architecture, a two-tiered and the multi-tiered SSD architecture. All three architectures were simulated using the same probabilistic model, which is the same probabilistic model used in other simulations as well ((Blo, 2020), (Eth, 2016), (Dorri *et al.*, 2017a)). All simulation times are measured in seconds. The simulations are modeled based on the following parameters:

1. In order to compare our SSD simulation to BlockSim (Blo, 2020), we considered the same value for $Binterval$[3], which is the average time for creating a block in the blockchain, where $1/Binterval$ is

[3] Changing the values of the parameters will not change the ranking obtained by the simulations for the models or the multi-tiered builds, since we used the same parameters in all our simulations.

the average number of blocks that are created per second. We built a function called "Protocol()" and we applied it per miner CH. In that function, we modeled PoW consensus protocol ($Protocol$) by drawing the time it takes the miner to finish the PoW from an exponential distribution based on the invested hash power (computing power) fraction of the nodes (Blo, 2020). So the miner's hash power is calculated as the average hash power of that miner divided by the sum of the hash power of all miners in the simulation ($TotalHashPower$). Then we use an exponential distribution function of the hash power times $1/Binterval$ (Blo, 2020). The consensus time ($Consensus$), which is modeled as the interval time between the creation time of a new block and the creation time of the previous block will be calculated as the result of the Protocol() function divided by 1000 to decrease the consensus time to get more realistic values for the simulation, as in BlockSim (Blo, 2020). Then we increment the simulation time ($currentTime$) with the consensus time ($Consensus$) so it would get assigned to the new block timestamp ($blockTime$).

$hashPower = miner.hashPower/miner.TotalHashPower$

$Protocol = random.expovariate(hashPower * 1/Binterval)$

$random.expovariate$ produces intervals according to an exponential distribution. So here we use for each miner node, an exponential distribution with an average rate of $hashPower * 1/Binterval$ arrivals per second, to simulate the time intervals between the blocks that can be created by that miner.

$Consensus = Protocol(miner)/1000$

$currentTime + = Consensus$

$blockTime = currentTime$

2. $Bdelay$ is the average block and Tx propogation delay among topmost CHs. In block_prop_delay() function, we modeled the block delay by drawing the time it takes a block to make one hop from one topmost CH towards another topmost CH, using an exponential distribution based on $Bdelay$ value. Thus, $Bdelay$ value is the expected value of the exponential distribution. We considered the same value for $Bdelay$ in our simulations for all three architectures in order to compare our SSD simulation to the flat architecture as well as the two-tiered architecture.

   When a block is created by a topmost CH, messages carrying the block get propagated to all other topmost CHs to validate it. The block delay is simulated for every topmost CH receiving the block by taking the maximum of the propagation delays towards these CHs and adding it to the creation time of the block.

$block\_prop\_delay = random.expovariate(1/Bdelay)$

For Tx where the requester and requestee are non-colocal, we also modeled the Tx delay as the Tx propagates amongst topmost CHs using $Bdelay$, since in that case, their propagation delay mimics a block propagation delay as the distance traveled by the block or the Tx at the topmost level is the same, though the types of the propagated messages are different (a block versus a Tx).

$tx\_prop\_delay_levelN = random.expovariate(1/Bdelay)$

3. In order to compare our SSD simulation to Ethereum (Eth, 2016), we considered the same value for $Tdelay$, which is the average Tx propogation delay between an IoTD and a topmost level CH. In tx_prop_delay, we modeled the Tx delay by drawing the time it takes a Tx to make one hop from an IoTD towards a topmost CH from an exponential distribution based on $Tdelay$ value (Blo, 2020).

   $tx\_prop\_delay\_non\_levelN = random.expovariate(1/Tdelay)$

   For each Tx, there is a Tx propagation delay per non_level-N CH that is simulated as an exponential distribution with an average rate of $1/Tdelay$ arrivals per second (Poisson process), where the expected value of the exponential distribution is $Tdelay$, to simulate the number of Tx that can arrive at the topmost CH per second.

4. In order to compare our SSD simulation to BlockSim (Blo, 2020), we considered the same value for $ProcDelay$, which serves to model the processing delay to append a block into BC (Blo, 2020). Calculating a block "appendTime" into BC, we add the max propagation delay of the block towards level-N CHs which validate that block, as well as another delay to process the block appends to BC, to the block creation time (Block timestamp) in order to calculate the block Append Time to BC. The latter is calculated by drawing the time it takes a block to get appended into BC from an exponential distribution where $ProcDelay$ value is the expected value of the exponential distribution.

   $block\_process\_delay = random.expovariate(1/ProcDelay)$

   For each block, there is a block processing delay per CH that is simulated as an exponential distribution with an average rate of $1/ProcDelay$ arrivals per second, where the mean of the exponential distribution is $ProcDelay$, to simulate the number of blocks that can arrive at the CH per second. The block Append Time to BC is set to the maximum receiving time of the block by each topmost CH plus the processing delay.

5. In order to compare our SSD simulation to LSB (Dorri *et al.*, 2017a), we considered the same value for $Tmax$, which is the maximum number of Tx allowed per block, as implemented by LSB (Dorri *et al.*, 2017a). Once the number of Tx to build a block reaches $Tmax$ at one topmost CH, the block is built and starts to propagate towards all the other topmost CHs.

Table 7.1 states the values used for the delays as implemented in Ethereum (Eth, 2016), LSB (Dorri *et al.*, 2017a) and BlockSim (Blo, 2020), so that our SSD implementation would be comparable to those implementations.

Table 7.1 System Parameters.

Parameter	**Value**	**Unit**
$Binterval$	600	seconds
$Bdelay$	0.15	seconds
$Tdelay$	0.000690847927	seconds
$ProcDelay$	0.000004	seconds
$Tmax$	10	Tx

7.5 Simulation Design

To build the architecture, we build two two-dimensional arrays: one array holds the data representing the nodes ($NodeArray$), while the other array holds the data representing the cluster IDs of the clusters where the nodes belong to ($ClusterArray$). In both arrays, the horizontal dimension corresponds to the architecture levels while the vertical dimension corresponds to the data of each array, such that the structure of both arrays are the same (same number of dimensions and same number of cells in each dimension). The relationship between the arrays is based on the indices of the arrays, such that a node's index in the first array corresponds to the cluster ID at the equivalent index in the second array.

In order to facilitate the navigation and the signature verification throughout the architecture, each CH maintains a list (named "managed_nodes") that contains the nodes that this CH manages along with their CHAC(s). For the same reason, all CHs maintain a list of their managed clusters (named "managedCluster").

7.5.1 The Splits And Merges Phase

In this phase of the simulation, we focus on testing the functionality of SSD splits and merges. In order for the structure to keep the balance between the security constraints of BC by maintaining a minimum number of nodes to perform the consensus per cluster, and between node manageability constraint for each CH. Merges and splits take place in the structure adhering to the MUT and CLT thresholds.

Our simulations follow the same concepts and produce the same results for an architecture of three or more levels, thanks to the use of CHAC which allows travelling the architecture height in $O(1)$ no matter how deep the forest is. Thus we only simulate for an architecture having three levels, since adding more levels will not give different results. Adding more levels could only be required in case the number of IoTDs increases inside the clusters to the point where their management overhead becomes a burden for the current number of CHs; in this case splitting the architecture, and eventually adding more levels, would be necessary. This is programmed in the simulation as we start by an architecture having only two levels, then we overflow the IoTD nodes in the lowest level, which triggers cascading splits of the clusters at the lowest level, followed by the second level, which requires the addition of the third level.

A violation of MUT is simulated where the number of nodes in one cluster is made to exceed the Management Upper Threshold (MUT), so a cluster split is simulated, followed by a merge at the level below for the cluster containing the nodes that become orphans (without a managing CH) due to the split. Similarly, a violation of the Control Lower Threshold (CLT) is simulated where the cluster size is less than the CLT, so a cluster merge is simulated, followed by a split at the level below so that the demoted CH, due to the merge, manages a cluster.

Considering the architecture levels, when a cluster at level i splits into two clusters, the first cluster keeps its cluster ID and its managing CH on level $i+1$, while the second cluster at level i gets a different ID that identifies its level, its number as well as the original cluster it separated from, as described in the next section. One of the second cluster nodes is randomly selected as a new CH for the second cluster and is promoted to level $i+1$. The cluster on level $i-1$ that used to be managed by the recently promoted cluster node merges with the smallest cluster on level $i-1$ to form a larger cluster and is managed by the same CH that manages the cluster it merged with.

When a cluster's size becomes less that CLT at level i, the cluster merges with the smallest size cluster on the level. The CH at level $i+1$ that previously managed one of the clusters is demoted to join the newly formed cluster at level i and is assigned a cluster to manage at the level $i-1$ as a split (of the largest size cluster on the level $i-1$) takes place.

Tracking the smallest size or the largest size cluster is done by traversing the array that contains the cluster numbers at that level and by counting the number of array cells that have the same integers[4]. The number of cells having the same integer (same cluster ID) represent the size of that cluster identified by that cluster ID. In case two or more clusters have the same number of nodes (cells) and are considered the largest clusters, then choosing any of them will be ok.

To implement the MUT and CLT for a level, the cluster number that repeats more than MUT times in the $ClusterArray$ indicates a violating cluster whose number of nodes exceeds MUT. The indices of the violating cluster number will be considered as the node indices of the cluster that needs to split at that level. The nodes in the indices corresponding to that cluster in the $NodeArray$ will split into two halves (groups), where the first group of nodes will keep its values in both arrays, while the second group of nodes will be assigned a different cluster number in the $ClusterArray$ at the corresponding indices to those node indices in the $NodeArray$.

7.5.2 The encoding of the splitting cluster ID and the split details

If the split does take place on the topmost level, two group of nodes form: the first group of nodes is assigned to the original CH that managed that cluster before the split and retains the same cluster number, while the second group of nodes is assigned to a uniformly randomly chosen CH from within that group and is given a new cluster number. In order to differentiate the cluster numbers according to whether they're newly generated or whether they are pre-existing, and in order to determine where the split taks place in the structure, we systemize the generation of the newly generated cluster number.

As an example, let us consider a cluster number like "1010005". The encoding of the number has a meaning in order to identify the splitting clusters and to differentiate among them according to their level and their position in the hierarchy. Thus the digits, starting at the most significant, are as follows:

[4] Those integers are the cluster IDs

(i) 1: the first digit identifies a constant value for the second half (group) of the splitting cluster.

(ii) 0: the second digit identifies the level where the split takes place (here it's level 0)

(iii) 1: the third digit identifies the current split among the number of splits that already took place to the original cluster (here it's only the first split)

(iv) 5: the last digit identifies the original cluster where the split takes place (here it's cluster number 5)

The split function keeps track of the nodes changing position by tracking/changing the cluster number in the Cluster Array at the index of the node that changed position in the structure. So the chosen CH becomes the parent of the second half (group) of the cluster and this is reflected in the cluster nodes CHAC. Then the chosen CH is promoted one level up and its managed_nodes, managedCluster as well as its CHAC are adjusted accordingly. The newly promoted node index is also modified in the Cluster Array to indicate the current cluster number where the node belongs after the promotion. All these changes are also associated with deleting the old position of the promoted node within the Node Array and within the Cluster Array.

Otherwise, if the split takes place at the topmost level which includes only one cluster having a number of nodes equal or greater than MUT (a violating cluster), the number of the resulting clusters is equal to CLT and one node from each of such clusters is randomly chosen as a CH and is promoted to the level above, in order to create a new level having CLT nodes. Since promoting nodes creates orphan nodes a necessity for a merging action arises at one level below the level where the split takes place. The splitting action may propagate/cascade from the lowest level towards the upper levels and additional levels may form adhering to CLT threshold.

7.5.3 Consensus and BC

Once the architectural organization stabilizes after performing the splits/merges and expansion/contraction of the structure, the BC simulation takes place. First, all transactions are generated such that the time intervals between Tx creation times are drawn from an exponential distribution that represents a Poisson process, and transactions are randomly assigned to chosen requester and requestee nodes. Then the Tx is directly sent to the requester topmost parent as indicated by the requester's CHAC (one hop to travel no matter how deep is the forest). In case both the requester and the requestee exist in the same tree, the Tx verification takes place at that topmost parent (since it is the parent common to both the requester and requestee). The common topmost CH verifies the Tx and further propagates the verified Tx to all topmost CHs so that all such CHs get a chance to append the Tx into their Tx pool in order to build blocks. In case the requester and requestee are in different trees, the Tx reaches the topmost CH managing the requester, which propagates messages to all topmost CHs, looking for the requestee information stored in the CHs' managed_nodes list. Once found, the Tx verification takes place at that CH where the requestee information is found (the requestee topmost parent), and that CH verifies the Tx and further propagates the verified Tx to all topmost CHs so that such CHs get a chance to append the Tx in their Tx pool in order to build blocks. Hence, node colocality (where the requester and the requestee share a common parent) is leveraged in order to reduce the number of hops required for Tx verification. CHAC is leveraged in order to reduce the number of hops required for performing the consensus as whole.

Once all the Tx are generated by the simulation, initial blocks which are called "genesis blocks" are created by the topmost CHs. Then those CHs start to create regular blocks (containing Tx) and append them to the global blockchain. Each block contains up to $Tmax$ Tx and is built according to a PoW consensus so that the block creation time is always larger than the maximum creation time of all the Tx contained within. A Tx is appended to a block only once. A block is appended into a global BC, where each block has a ".previous" variable that acts as a hook attaching it to the global BC, we thus skip the usual BC hashing technique for simplicity. Once a block is built at a top-level CH, it is propagated (as carried in a message) to other top-level CHs to be verified, then appended into the global BC.

In the simulation, the block propagation is performed by scheduling the block to be received by other topmost CHs to simulate block verification. In order to simplify the implementation, building a block and appending it to BC takes place once it is received by all topmost CHs, without having to compare the topmost CHs' local BCs to each other. This simplifies our simulation in order to focus on measuring the number of hops required for a Tx to travel within the architecture.

7.5.4 The transactions processing

We followed a probabilistic model to generate Tx, such that the Tx are generated with a random id (using a uniform distribution), at times exponentially distributed with random intervals during the simulation duration (using an exponential distribution to represent a Poisson process). Each Tx is randomly assigned to a randomly chosen requester node as well as a randomly chosen requestee node (using a uniform distribution). The transaction is first created by a requester. Examining the requester's CHAC, the message carrying the Tx shows whether the requestee belongs to the same tree as the requester (whether they are colocal).

For Tx verification, the message carrying the Tx propagates as follows:

(A) Tx propagates upwards in the structure in only one hop (no matter the number of levels of the architecture) to reach the topmost parent.

(B) If the requestee is colocal:

    (i) Tx is verified by the topmost parent.

(C) If the requestee is non-colocal:

    (i) The topmost parent sends messages to all other topmost CHs to looks for the requestee.

    (ii) Once the requestee is found at one of the topmost CHs, the verification of requester and requestee takes place at that CH.

So in case of colocality, the Tx propagates in one hop (upwards) plus in $O(1)$ to be verified by only one topmost CH. While in case of non-colocality, the Tx propagates in one hop (upwards) plus in $O(numberoftopmostCHs)$ looking for a topmost CH that holds the requestee information to verify the Tx.

Once verified, the Tx is sent to all other topmost CHs in $O(numberoftopmostCHs)$ so that they append that Tx to their current or future block.

Propagating in $O(numberoftopmostCHs)$ is simulated by taking the maximum of the propagation towards all topmost CHs. This is done by generating those propagation delays towards all the topmost CHs, while adding them into an array, then getting the maximum value.

The Tx sender is a variable that changes during the simulation in order to identify the current node sending a Tx. Each node has a Tx pool that contains the Tx that have reached the node so far. Once a node becomes a sender and sends those Tx towards another node on the upper level of the architecture, that pool becomes empty. Only the topmost CHs are allowed to preserve their Tx within their pools while being a sender to other topmost CHs. This modeling ensures that by the end of the simulation, all topmost CHs have the same snapshot of all Tx, hence getting the same chance to append Tx into blocks. Also this mechanism prevents any Tx loss while being sent upwards (in only one hop thanks to the use of CHAC) and then amongst the topmost level CHs.

Each Tx has a timestamp list that contains its creation time, its receive time by CHs (which is equal to the Tx creation time at the beginning and then keeps incrementing with the transaction propagation delay with each hop traveled by the Tx) as well as its append time into a block. A Tx receive time is tracked for each Tx received at a specific node and it includes the accumulation of all the propagation delays of the Tx thus far.

In order to simulate Tx colocality verification, we verify the Tx colocality in the same tree of the forest by using two counters counting the nodes according to the colocality of the requester and the requestee. This is in order to define the colocality within the same cluster and the colocality within the same tree, respectively. To verify the Tx colocality in the same tree, we use a function which checks whether the direct parent of the requester and the direct parent of the requestee are the same within the CHAC of the requester and the CHAC of the requestee. And to verify the colocality within the same tree, we apply the same logic to verify the parents on all the levels within the CHAC of the requester and the CHAC of the requestee. This verification can take place at the topmost level CH that is a direct or indirect parent for the requester.

Considering the Tx propagation design, our simulation produce the same results for an architecture of three or more levels thanks to the use of CHAC. Thus we only simulate for an architecture having three levels, since adding more levels will not give different results.

For the Tx verification, due to the time and computation consumption involved in that process, we simulate it only by performing the number of the nodes interactions required for such verification to take place. So what is supposed to be programmed is: we generate private and public keys for a Tx requester, and the Tx is signed by the requester at the creation time in order to be verified later on. verifying the Tx is a two step process: The first step is to compare the hash(requester public key) to the same value in previous Tx on the ledger. The second step is to verify the requester and requestee signatures, comparing them to the requester and requestee public key on the Tx.

The second step verification can take place at two different locations, depending on the trade-off envisioned and the application needs:

(i) The first location is at the closest direct parent in common while traveling up the architecture. If no common parent exists, then the requester gets verified at its topmost parent and the requestee gets verified at its own topmost parent (the CH that holds the information of the requestee).

(ii) The other location is at the topmost direct or indirect parent in common if such parent exists, otherwise the requester is verified at the topmost parent and the requestee is verified at its own topmost parent (the CH that holds the information of the requestee).

If node altering is a concern, the first location is adopted to provide more security. However since reaching that location may involve intermediary hops to be travelled by the message carrying the Tx to reach its destination towards its topmost CH parent, adopting that location is less efficient. If efficiency is a priority, adopting the second location will be key, since it avoids unnecessary intermediate hops while the Tx travels upwards in the architecture.

Since security is not our main concern in our model, we adopt the approach simulating the requester and the requestee signature verification using the second location, by skipping the actual signature verification and by only reflecting the number of hops necessary to perform such verifications within our simulation, which is the approach that favors efficiency.

7.5.5 The blocks processing

A block is created by a topmost CH with a number of $Tmax$ Tx that are randomly assigned from all the Tx generated. The ids of such Tx are appended into a Tx pool at a topmost level CH, and are then stored in a global variable, (used_Tx) list, in order to keep track of the Tx already assigned to blocks, and to avoid their reassignment for future block creations. Each block creation event is assigned a timestamp generated according to PoW consensus.

The block creation time is always larger than the creation time of all the Tx contained within the block.The block creation time is equal to the largest receive time of the Tx contained in the block.

Block creation is based upon using a scheduler of events that are queued and processed according to their time schedule. Event types are mainly creating a block at a topmost level CH, or receiving a block by a topmost level CH. Once such a CH creates a block, the CH propagates the created block to all other topmost

CHs then that creator CH gets a chance to create more blocks using the generate_next_block() function. Once propagated to another topmost CH, the receiver CH gets a chance to validate the block and then creates another new block (using the generate_next_block() function). Creating new blocks is always based on the Tx available in the creator CH Tx pool as long as those Tx are not already included into the used_Tx list.

Once a block is created and validated, it is appended by the creator CH into the BC. The block's appendTime includes the accumulation of all the propagation delays of the block. This time value is then assigned to the corresponding Tx append timestamp. The difference between the Tx append time and the Tx creation time gives us the elapsed time for a Tx from its creation until its inclusion into a block.

The created block is appended to the topmost level CH's local BC, which starts by a genesis block. It is also appended into the global_BC shared by all topmost CHs, where blocks are chained in the order of their creation and validated by topmost CHs. PREVIOUS is a global variable to represent the chaining of global_BC indicating the ID of the previous block on the chain and relating a new block to the previous block. PREVIOUS is initiatized by -1 when the BC has no blocks, then it is incremented to 0 when the BC has one genesis block (it is the first genesis block generated by the topmost CHs) and then it gets the id of the next block to join the global_BC and gets updated every time a new block is appended to the global_BC.

Block appends into the BC are simplified by directly appending the block into the global BC by one miner at a time, skipping the comparison between the local BC of each topmost level CH (miner) as well as forking. This shortcut is due to the fact that the main aspect that we need to evaluate is the number of hops travelled by the Tx and the blocks within the architecture before the append to the BC. This approach adopted by the simulation has been introduced in other BC simulations to achieve simplification as well (Dorri *et al.*, 2017a).

7.5.6 The verification checks

We incorporated a few checks in our simulation, as follows:

(i) check_Block_id_is_UNIQUE() function to assert the block ID non duplicate status.

(ii) check_Tx_id_is_UNIQUE() function to assert the Tx ID non duplicate status.

(iii) check_each_LevelN_CH_has_GenisisBlock() function to assert that each level-N CH has one genesis block at the beginning of its local blockchain.

7.5.7 The simulation details

The simulation is event-driven: The creation of a block takes place before the propagation of the block, then handling the creation event takes place before the propagation which includes the validation of the block by other level-N CHs. The sequence of the block generation is shown as in Figure 7.1, the node events are shown as in Figure 7.2.

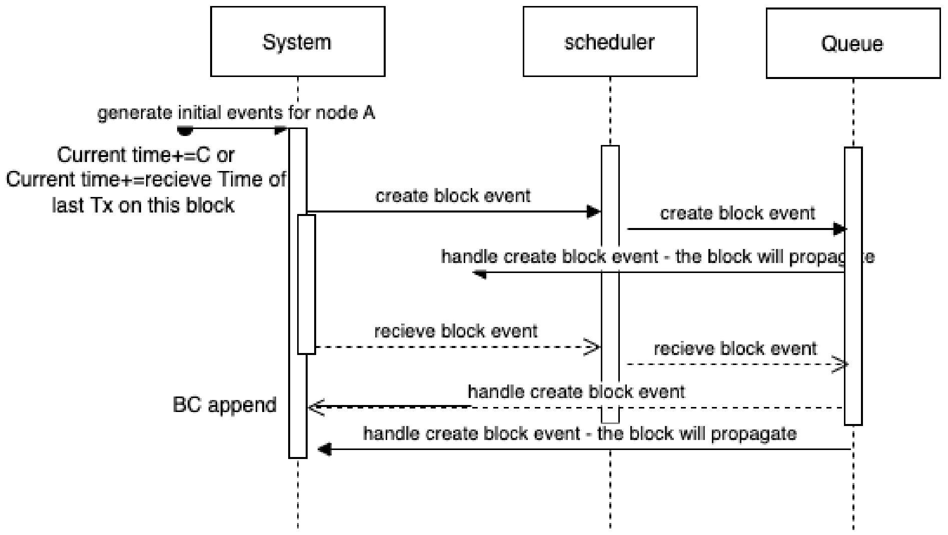
System
scheduler
Queue
generate initial events for node A
Current time+=C or
Current time+=recieve Time of
last Tx on this block
create block event
create block event
handle create block event - the block will propag e
recieve block event
recieve block event
BC append
handle create block event
handle create block event - the block will propagate

Figure 7.1 The current sequence of events

According to the figures, we can see that a block is generated with $Tmax$ Tx drawn from a Tx pool that contains all the Tx available in the system (here the pool started with 30 Tx available for the first block created), then once another block is created, the number of the Tx pool gets smaller to prevent reusing the Tx already appended into a previously created block (the pool then has 20 Tx available for the second block created), and so on until Tx available get all consumed.

Also we can notice that though Block A was created and added to the queue before block B, block B was processed first since the time associated with its addition to the queue of events comes before block A. The block processing is : handling create block, receive block by other topmost CHs, and appending Block to BC. We can notice that the append to BC takes place after the block propagation towards all other topmost level CHs takes place.

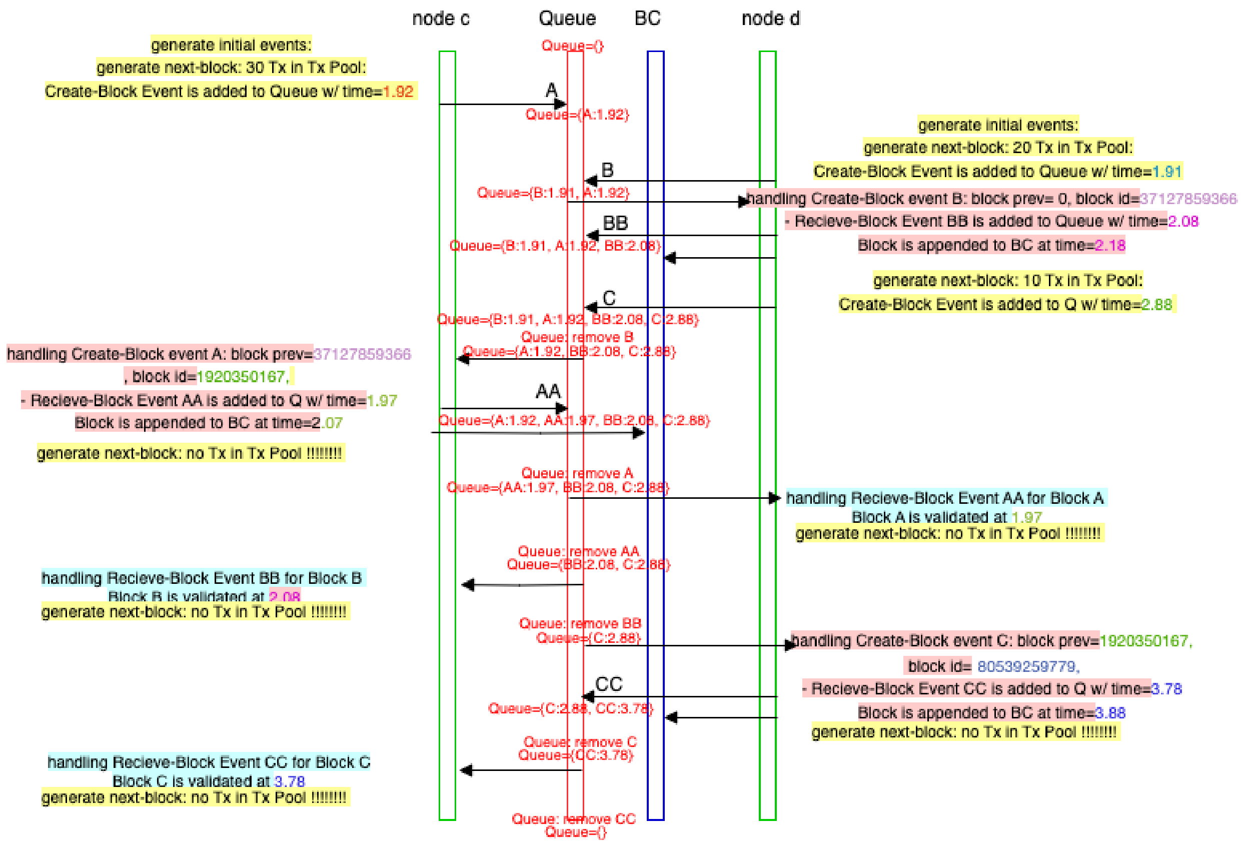
node c
Queue
BC
node d
Queue={}
generate initial events:
generate next-block: 30 Tx in Tx Pool:
Create-Block Event is added to Queue w/ time=1.92
A
Queue={A:1.92}
generate initial events:
generate next-block: 20 Tx in Tx Pool:
Create-Block Event is added to Queue w/ time=1.91
B
Queue={B:1.91, A:1.92}
handling Create-Block event B: block prev= 0, block id=37127859366
- Recieve-Block Event BB is added to Queue w/ time=2.08
BB
Queue={B:1.91, A:1.92, BB:2.08}
Block is appended to BC at time=2.18
generate next-block: 10 Tx in Tx Pool:
Create-Block Event is added to Q w/ time=2.88
C
Queue={B:1.91, A:1.92, BB:2.08, C:2.88}
Queue: remove B
Queue={A:1.92, BB:2.08, C:2.88}
handling Create-Block event A: block prev=37127859366
, block id=1920350167,
- Recieve-Block Event AA is added to Q w/ time=1.97
AA
Block is appended to BC at time=2.07
Queue={A:1.92, AA:1.97, BB:2.08, C:2.88}
generate next-block: no Tx in Tx Pool !!!!!!!!
Queue: remove A
Queue={AA:1.97, BB:2.08, C:2.88}
handling Recieve-Block Event AA for Block A
Block A is validated at 1.97
generate next-block: no Tx in Tx Pool !!!!!!!!
Queue: remove AA
Queue={BB:2.08, C:2.88}
handling Recieve-Block Event BB for Block B
Block B is validated at 2.08
generate next-block: no Tx in Tx Pool !!!!!!!!
Queue: remove BB
Queue={C:2.88}
handling Create-Block event C: block prev=1920350167,
block id= 80539259779,
- Recieve-Block Event CC is added to Q w/ time=3.78
CC
Queue={C:2.88, CC:3.78}
Block is appended to BC at time=3.88
generate next-block: no Tx in Tx Pool !!!!!!!!
Queue: remove C
Queue={CC:3.78}
handling Recieve-Block Event CC for Block C
Block C is validated at 3.78
generate next-block: no Tx in Tx Pool !!!!!!!!
Queue: remove CC
Queue={}

Figure 7.2 Node Events

As far as node colocality is concerned, there are two approaches to choose from in order to verify node locality: the first approach verifies the locality at the first direct parent of the Tx requestee, which consumes one hop (thus one propagation delay). The second approach verifies the locality directly at the requester, through verifying both the requester and the requestee CHAC, which is an approach that may save some delay for the Tx, by eliminating one propagation delay (hop towards the first direct parent). Our design aims for the latter approach as implemented in our simulation.

The next point that we need to emphasize lies in the fact that the requester and requestee signature verification takes place at the topmost level CHs, no matter the number of the architecture levels, and reaching such a level will take only one hop (one propagation delay) in all cases, thanks to the use of CHAC. Thus building an architecture with only three levels is sufficient as a proof of concept. And that's the reason why our implementation is limited to model one tier to represent Bitcoin, two tiers to represent the literature and three tiers to represent our design. The only case where it will be recommended to increase the number of tiers in our design is when the number of the topmost CHs increases to the point that exceeds MUT threshold, which represents the point beyond which managing one cluster of that size becomes too much of a burden or, in the case of the topmost level, the point beyond which the number of the topmost level CHs becomes too large, which renders the block verification process too expensive. Such a threshold is application agnostic and, hence, can be implemented according to each organization needs. Thus, increasing the number of tiers will decrease the number of topmost CHs and will hence decrease the number of hops amongst them and, eventually, the total time needed to verify Tx and blocks and to achieve consensus. Otherwise, the implementation performs the same way whether the number of tiers is three or more and hence, there is no need to test the design with more than three tiers in terms of time performance.

Another hypothesis that does not need to be verified is the effect of including the requester and requestee signature verification in the simulation. We expect that such verification will add a factor of delay for the Tx and block delays. Since the main focus of our study is on decreasing the number of hops necessary for a Tx to get appended into Blockchain, including the Tx verification or not will not directly affect our results, thus it will not be tested.

7.6 Hypotheses to Verify

We aim to verify the following hypotheses:

(i) The multi-level architecture is better than flat architectures in terms of the Tx throughput, the average Tx delay and the average block delay.

(ii) The multi-level architecture is better than two-tiered architectures in terms of the Tx throughput, the average Tx delay and the average block delay.

(iii) The multi-level architecture has a better performance for nodes that are colocal within the same tree and it has an even better performance when the colocality verification takes place directly at the requester.

7.7 Performance Metrics

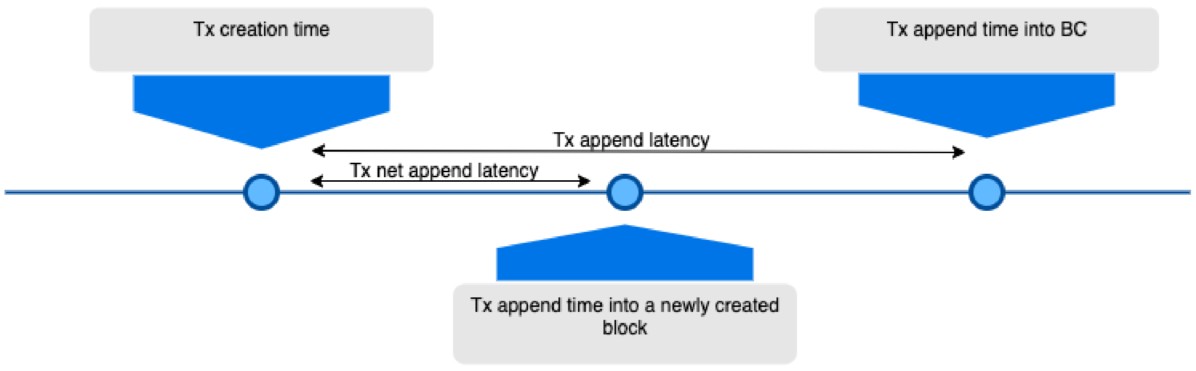

Figure 7.3 Tx Append Latency

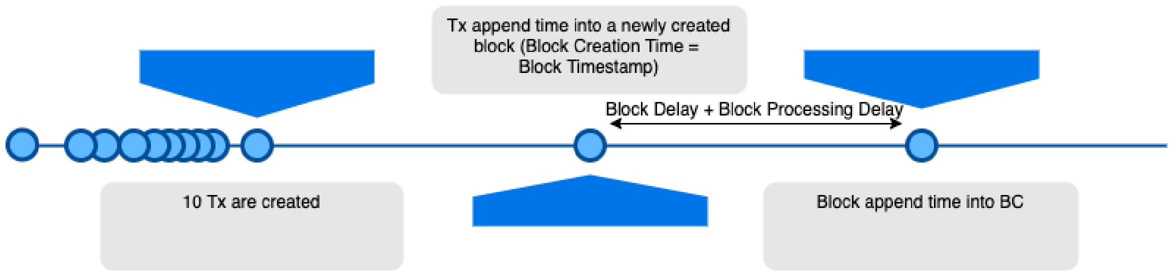
Tx append time into a newly created block (Block Creation Time = Block Timestamp)
Block Delay + Block Processing Delay
10 Tx are created
Block append time into BC

Figure 7.4 BlockCreationTime

In order to verify our hypotheses stated above, we evaluate a number of performance metrics to use when comparing our architecture to a flat one-tier Bitcoin as well as to a two-tier architectures:

(i) The Tx throughput: the number of the appended Tx divided by the simulated duration.

(ii) The simulated duration: the difference between the earliest Tx creation time and the latest block append time.

(iii) The Tx append latency (Tx time elapsed): the difference between the Tx creation time and the Tx append time into BC, as shown in Figure 7.3.

(iv) The Tx net append latency (net time elapsed): the difference between the Tx creation time and the Tx append time into a newly created block, as in Figure 7.3.

(v) The block creation time: the maximum of the following : the creation time of the block according to the consensus; and the maximum creation time of all the block's Tx. The block creation time is equal to the block timestamp.

(vi) The block delay: the delay taken by the block to join all the topmost CHs, which is the maximum propagation delay towards each such CHs.

(vii) The block append time: the time at which the block is appended to the BC. The way this time is calculated in the simulation is by adding the block timestamp, the block delay, as well as a processing delay to append the block into BC.

(viii) The net block delay: the difference between the block creation time (block timestamp) and the block append time, as in Figure 7.4.

Now that we have introduced our SSD simulation details, in the next chapter, we show the SSD asymptotic analysis according to the simulation, the simulation results along with the required interpretations.

CHAPTER 8
SIMULATION ASYMPTOTIC ANALYSIS, RESULTS AND INTERPRETATIONS

In Chapter 6, we introduced a theoretical asymptotic analysis of the SSD algorithm. In this chapter, we produce the detailed practical asymptotic analyses of the SSD algorithm as it was implemented in our simulations while taking into account the data structures used, as well as the comparison between those analyses and the theoretical analyses of BC that were performed in Chapter 6. The chapter also includes the comparison among the simulation results of the three different BC architecture models (the flat, two-tiered and multi-tiered models) as well as the comparison among the simulation results of the three different SSD builds (having two, four and eight level-N CHs).

Thus in this chapter, we introduce two sets of event-driven simulations. All simulations are programmed in Python3.

We run a first set of simulations for the three BC models: the SSD, the two-tiered and the flat architecture. Let's call them "the models". We run the simulations for each model for 10 runs, each processing 100 Tx and each having 260 IoTDs (with 1.5% fluctuations for the number of IoTDs). The goal of this set of simulations is to verify the hypotheses mentioned in the previous chapter and to validate the multi-tiered design concept. Both the SSD architecture and the two-tiered model use 8 CHs in the topmost level.

We also run a second set of simulations for the different SSD builds, each having a different number of topmost level CHs: 2 CHs, 4 CHs and 8 CHs. Let's call them "the SSD builds". So we run the simulations for each SSD build for 10 runs, each processing 500 Tx and each having 260 IoTDs (with 1.5% fluctuations for the number of IoTDs). The goal of this set of simulations is to evaluate the impacts of leveraging node colocality.

8.1 Practical Analysis of the SSD Algorithm

In this section, we introduce the results of the practical analysis of the SSD algorithm according to our simulations, while taking into consideration the data structures used, particularly for the split and merge actions required so that the structure expands and contracts easily. To better express the analysis, we consider the same symbols and definitions used for the number of per-level nodes expressed in section 6.3

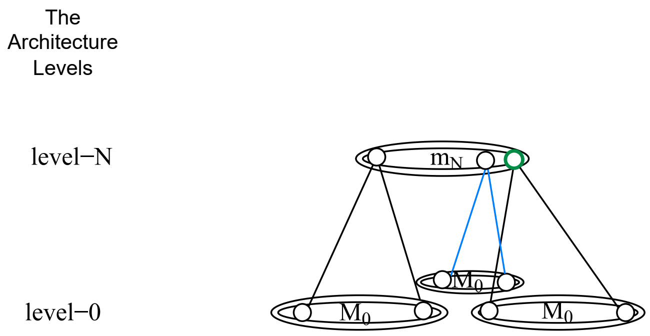

Figure 8.1 Building the Structure

(Theoretical Analysis of the SSD Algorithm). Accordingly, let's consider an architecture of three levels, level 0 having $\#[level0] = Y$ nodes, followed by level 1 having $\#[level1] = Z$ nodes, where the first set of CHs reside, followed by the topmost level of the architecture, level N having $\#[levelN] = S$ nodes.

8.1.1 Asymptotic Analysis of Building The Structure

In this section, let's analyze the cost of building the architecture. The architecture is built by first building two data structures: one data structure to store the node IDs and another to store the cluster numbers of the nodes. Both data structures are identically built so that the index of a node ID in the node structure refers to that node cluster number when used in the cluster structure. Each data structure is a list of lists (two-dimensional arrays), NODES[Level][nodeID], and CLUSTERS[Level][ClusterNumber], where each list represents a level of the architecture. After building the structure, we start the splits and merges. In this section, we analyses the structure building.

To build the cluster structure, we start by building two levels as in Figure 8.1. A nested loop sweeps all $\#[level-N]$ CHs (m_N) and travels each level-0 cluster's nodes (m_0), then the cost will be $O(m_N \cdot m_0)$.

To build the node structure, a loop that sweeps all $\#[level-N]$ CHs (m_N) and then a nested loop sweeps all level-N CHs (m_N) and sweeps each level-0 cluster's nodes ($m_N \cdot m_0$), then the cost is $O(m_N \cdot m_0)$, which is the total number of nodes in the architecture.

Considering the costs of building both structures, since the latter is the higher order term, then we can consider that building the architecture takes $O(m_N \cdot m_0)$.

8.1.2 Asymptotic Analysis of the Split

In order to split one cluster of size M that's managed by a parent node X and that resides on level-i, we have to go over all the nodes in NODES[i] list in order to identify the nodes that belong to the cluster to be split, split them and assign them to two different clusters, renaming the second cluster according to the method explained before in chapter 7. Also going over the nodes is necessary to adjust each node's managed_list by removing the node itself from that list. We go over the nodes of the newly renamed cluster to chose a randomly chosen node. The latter is to be promoted to the level above to manage the newly renamed cluster. This will be reflected in the CHAC of that cluster nodes as well as in the managed_list of the promoted node. The promoted node will be extracted from NODES[i] and CLUSTERS[i] lists and will join NODES[i+1] and CLUSTERS[i+1] lists.

In order to achieve that, we need to do the following:

1. Append one extra node in one or more cluster(s) in NODES[i] at level 0, this takes $O(1)$. The newly appended node(s) will violate the MUT of that level.
2. Identify the MUT violating cluster by calculating the cluster size of each cluster at level Y, which takes $O(Y)$.
3. Divide the violating cluster in two by traveling all that cluster nodes, thus it takes $O(M)$. This step will cost $O(Y)$ in case the split is at the topmost level.
4. Travel the second half of the cluster to remove those nodes reference from the managedNodes of the parent node X, to change their cluster number stored in their nodes in the NODES[i] list of lists which costs $O((1/2) \cdot M)$. Thus it takes $O(M)$.
5. Travel the second half of the cluster to update those nodes cluster number in the CLUSTERS[i+1] list of lists which costs $O((1/2) \cdot M)$. Thus it takes $O(M)$.
6. Travel the second half of the cluster to append those nodes in a temporary list ($O((1/2) \cdot M)$) and to choose a random node (new parent CH) out of that list ($O(1)$). Then update that chosen node's information and insert it as a new CH in the level above ($O(1)$). ($O((1/2) \cdot M)$)

7. Travel the second half of the cluster to change the CHAC of those nodes to reflect the newly chosen parent CH($O((1/2) \cdot M)$), and insert the new CH in the level above in NODES[i+1] ($O(1)$), and in a nested loop, loop through these nodes again to assign their CHAC into the managedNodes of the new CH($O((1/2) \cdot M)$). Thus it takes $O(M^2)$.
8. Remove the new CH from the level below in NODES[i] and remove its reference cluster number in CLUSTERS[i] and insert a new reference cluster number in CLUSTERS[i+1]. This takes $O(1)$.

Thus the final split takes $O(Y) + O(M^2) + O(M)$. So it takes $O(max(Y, M^2))$, where Y is the number of nodes in the level where the split takes place, and M is the number of nodes in the cluster to split. This cost analysis is for the splits whether they are at the topmost level or not.

In case the split takes place on the topmost level, we follow the same process except that, instead of only considering the second half of the cluster to split, we consider both halves, which will give the same complexity.

8.1.3 Asymptotic Analysis of the Merge

There are two cases when we have to merge one cluster of size M that's managed by a parent node X and that resides on level i. The first case is when the cluster size is less that CLT and the second case is when a split happens at NODES[i] where one or more CH needs to be promoted to NODES[i+1]. In the former case, the parent of the merging cluster that exists in NODES[i+1] will rejoin it's own cluster in NODES[i]. In the latter case, the managed nodes of the promoted cluster at NODES[i-1] will need to be merged with another cluster at the same level in order to share the same CH that manages that cluster. And the parent of the merging cluster that exists in NODES[i+1] will be promoted to join NODES[i+2].

For the merge to be successful, clusters that belong to one tree can merge only with other clusters on the same level in the same tree.

In order to realize that, we need to do the following:

1. Identify the CLT violating cluster of all clusters at level i, thus it takes $O(Y)$.
2. Identify the considered clusters to merge with under the same tree at level i, thus it takes $O(Y)$.

3. Identify the specific cluster to merge with under the same tree at level i, thus it takes $O(Y)$.

4. Identify the nodes of the cluster to merge with from all the nodes at this level, thus it takes $O(Y)$.

5. Travel NODES[i+1] to identify the CH parent of the merging cluster and in a nested loop, travel NODES[i] to get the nodes in the cluster to merge and change their CHAC and their cluster number. Thus it takes $O(Z \cdot Y)$.

Thus the final merge takes $O(Y) + O(Z \cdot Y)$. So it takes $O(Z \cdot Y)$ since this is the higher order term, where Y is the number of nodes in the level where the merge takes place, and Z is the number of nodes in the level above.

In case the merge requires a CH to join the level above or to rejoin its own cluster at the level below, this takes $O(1)$ in both cases and so the complexity will remain the same.

8.2 Final Practical Asymptotic Analysis of the Split and Merge

8.2.1 SSD Algorithm - The Increase of Cluster Size on Level $i, i \neq N$

The split algorithm described here is when the cluster to split is at level 0. In case the cluster to split is at level i where i > 0 (having Y nodes), the algorithm will also include a merge at level $i-1$ (having Z nodes) as shown by Figure 8.2.

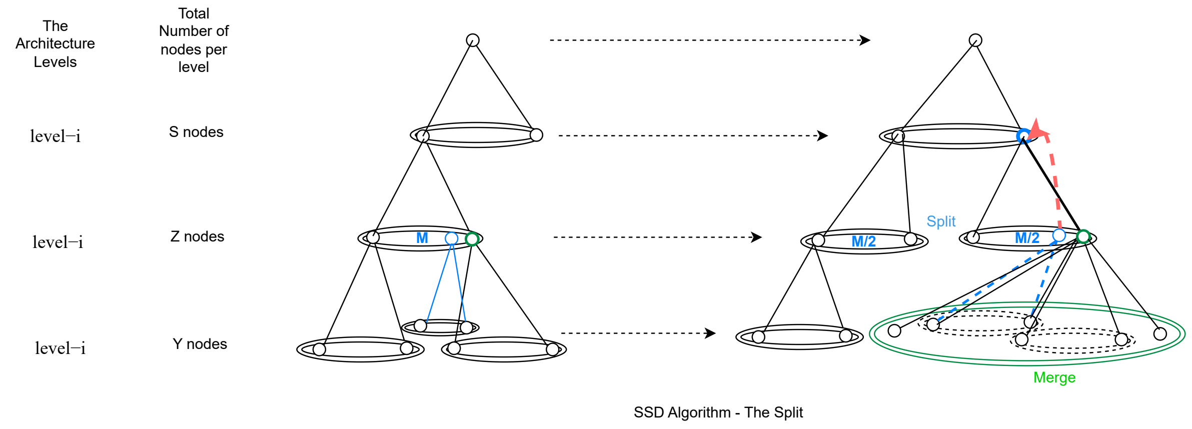

Figure 8.2 The Split

In that case, the complexity of the split will be the maximum of these two values:

1. $O(max(Z, M^2))$, where Z (previously referred to as Y) is the number of nodes at the level where the split takes place (level i), and M is the number of nodes in the cluster to split.
2. $O(Z \cdot Y)$, where Y is the number of nodes at level $i-1$ where the merge takes place, and Z is the number of nodes at the level above (level i).

In that case, the Cost of the split will be in $O(max(Z \cdot Y, M^2))$, where Z is the number of nodes in the level where the split takes place, Y is the number of nodes at level $i-1$ where the merge takes place and M is the number of nodes in the cluster to split.

8.2.2 SSD Algorithm - The Increase of Cluster Size on Level $i, i = N$

The split of the topmost level includes the clustering of the current topmost level, which implies the expansion of the architecture by one level (future topmost level), which is in $O(Z)$ where Z is the number of nodes at the level above (level i) where the split takes place (the current topmost level), whereas adding Chs on the new topmost level costs $O(1)$. Which keeps the cost to $O(max(Z \cdot Y, M^2))$, still.

8.2.3 SSD Algorithm - The Decrease of Cluster Size on Level $i, i \neq N$

For the merge at level i where i > 0 (having Z nodes), the algorithm will also include a split at level $i-1$ (having Y nodes) as shown by Figure 8.3.

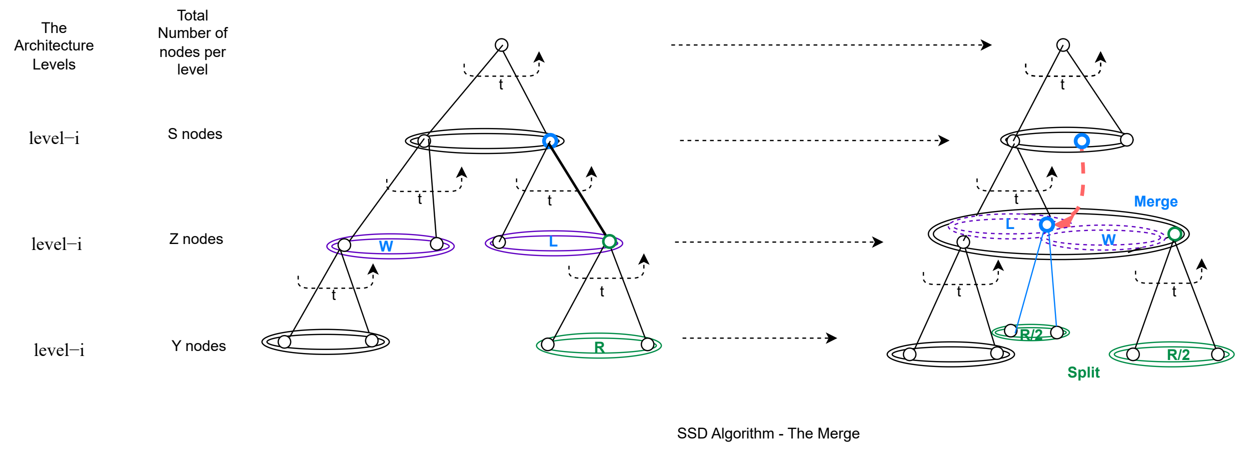

Figure 8.3 The Merge

In that case, the complexity of the merge will be the maximum of these two values:

1. $O(Z \cdot Y)$, where Z is the number of nodes in the level where the merge takes place, and S is the number of nodes in the level above.
2. $O(max(Y, R^2))$, where Y is the number of nodes in the level where the split takes place, and R is the number of nodes in the cluster to split.

8.2.4 SSD Algorithm - The Decrease of Cluster Size on Level $i, i = N$

In that case, the Cost of the split will be in $O(max(Z \cdot S, R^2)$, where Z is the number of nodes in the level where the merge takes place at level i, S is the number of nodes at level $i + 1$ and R is the number of nodes in the cluster to split at level $i - 1$. The merge of the topmost level, which implies the shrinkage of the architecture by one level, was not considered in the simulation, due to its impracticality.

8.3 Comparison between the Theoretical Analysis and Practical Analysis of the SSD Algorithm

8.3.1 SSD Algorithm - The Increase of Cluster Size on Level $i, i \neq N$

If size(localCluster) $\geq MUT$ and local cluster is at level $i \neq h$:

The cost of splitting a cluster with a size violating MUT at level i , followed by a merge at level $i - 1$ is calculated as follows :

1. Theoretically, the total cost is $O(max(Z, (L+W)))$, where Z is the number of nodes of level i where the split takes place and L and W are the number of nodes in the clusters to merge at level $i - 1$.
2. Practically, as per the simulations, the total cost is $O(max(Z \cdot Y, M^2))$, where Z is the number of nodes in the level where the split takes place, Y is the number of nodes at level $i - 1$ where the merge takes place and M is the number of nodes in the cluster to split.

We can see that the simulation complexity costs more than the theoretical analysis due to the need to travel all the nodes in one level to search for a specific node/set of nodes.

8.3.2 SSD Algorithm - The Increase of Cluster Size on Level $i, i = N$

If size(localCluster) $\geq MUT$ and localCluster is at level $i = h$:

The cost of splitting a cluster with a size violating MUT at level i , followed by a merge at level $i-1$ is calculated as follows :

1. Theoretically, the total cost is $O(max((S \cdot Y, S \cdot (L+W)))$, where S is the number of nodes at the newly created level N, Y is the number of nodes at level $N-1$, and L and W are the number of nodes in the clusters to merge at level $N-2$.
2. Practically, as per the simulations, the total cost is still the same as in the the case when the merge $O(max(Z \cdot Y, M^2))$, where Z is the number of nodes in the level where the split takes place, Y is the number of nodes at level $i-1$ where the merge takes place and M is the number of nodes in the cluster to split.

Since the higher the level the less the number of nodes it contains, we can consider that $(Z \cdot Y) \geq (Z \cdot Y)$, however since level $N-2$ might be the IoTD level, with clusters containing a very large number of nodes, we can not be certain wether $(S \cdot (L+W)) \geq (M^2))$

8.3.3 SSD Algorithm - The Decrease of Cluster Size on Level $i, i \neq N$

If size(localCluster) $\leq CLT$ and localCluster is at level $i \neq h$:

1. Theoretically, the total cost is $O(max(Z, R))$, where Z is the number of nodes at level i where the merge takes place and R is size of the cluster to split at level $i-1$.
2. Practically, as per the simulations, the total cost is $O(max(Z \cdot S, R^2)$, where Z is the number of nodes in the level where the merge takes place at level i, S is the number of nodes in the level above and R is the number of nodes in the cluster to split at level $i-1$.

We can clearly see that the simulation costs more than the theoretical analysis due to the fact that we need to travel all the nodes in one level to search for a specific node/set of nodes.

8.3.4 SSD Algorithm - The Decrease of Cluster Size on Level $i, i = N$

In theory, the BC architecture may decrease in size through one or more level eliminations. However in practice BC size is mainly expected to increase as more BC nodes will join in with time. That's why we did not test clustering decreases in our simulations.

8.3.5 Comparison Results - Theoretical Analysis versus Practical Analysis of the SSD Algorithm

Both the theoretical and the practical analysis of the SSD algorithm are performed for only three levels of the architecture. Increasing the number of levels yields different results due to the fact that each CH holds information about its managed nodes, and while node information changes with every split and merge, it needs to be updated at every parent CH, affecting all parent CHs across all levels. Thus, the complexity increases with the increase of the depth of the structure.

The practical cost analysis of the SSD algorithm (for three levels) in simulations exceeds its theoretical cost. This disparity is attributable to the necessity to account for node and cluster data updates across all forest levels (such as CHAC, list of managed nodes, etc.). These data are stored in complex, multi-dimensional structures to reflect the intricacies of the multi-tiered design and the number of levels involved. Consequently, the updates of these data structures increase exponentially with the number of dimensions or forest levels.

We can see that in most cases, the practical costs observed by simulation are larger than what was expected given the theoretical analysis. This is due to the fact that, in the simulation, we need to travel the data structures sequentially in order to search for one node or a set of nodes. The search becomes linear in the case of searching within one dimension of the data structure; and becomes quadratic, worst case, when it involves a second dimension, specially if the numbers of nodes on both levels are comparable. Since the splits and merges take place in pairs on different levels, as we explained above, those costs can be quadratic. Adding a third dimension makes the cost cubic and so forth. Hence the cost exponentially increases with the number of dimensions. We believe that this will be the case for all possible simulations implemented for the SSD architecture, due to the complexity of the node organization involved and due to the need of keeping track of the relationship between the nodes, the clusters, and the CH managed nodes as well as the CHAC, no matter how dynamically the nodes change their positions, their levels, their clusters, their managed nodes and the CHs that they belong to, within the architecture.

The simulation was not considered for the case of the architecture shrinkage, which involves the elimination of one or more levels off the architecture. This is due to the fact that BC architectures in general tend to increase in size with time. Hence the BC decreasing case was not considered for our simulation. Nevertheless, the architecture shrinkage can be expected to yield a comparable cost to the architecture expansion.

We expect that a real SSD architecture implementation would give results closer to the asymptotic analysis results of our simulation rather than our SSD theoretical analysis results due to the fact that a real implementation is expected to eventually face similar programming restrictions as our simulation, as described above, unless faster search algorithms are applied in order to search the data structures more efficiently and to improve the results, which will be mostly logarithmically reduced, in that case.

8.4 Comparison of the Simulation Results for the Three BC models

In this section, we present the first set of simulations. This set of simulations is to compare the efficiency of flat, two-tiered and multi-tiered models. All simulations ran for 10 runs while processing 100 transactions with about 260 IoTDs (with 1.5% fluctuations). SSD and two-tiered models have 8 CHs at the topmost level. SSD has three levels since adding more levels barely affects the calculations of the consensus-accelerating protocol due to the use of CHAC.

We present the global results for the runs performed for each of the three BC architecture models in order to verify the hypotheses stated in chapter 7 and to validate the efficiency of the SSD model. The global results for all three architectures are presented in Table 8.1, Table 8.2 and Table 8.3. In each table, each row corresponds to the data of one run.

In Table 8.1, the flat architecture has one level of 256 total number of nodes, generating a hundred Tx in ten blocks. In Table 8.2, the two-tiered architecture has two levels of 256 total number of nodes: eight topmost level CHs (each, managing a cluster of 32 IoTDs), generating a hundred Tx in ten blocks. In Table 8.3, the SSD architecture has three levels of 260 total number of nodes: eight topmost level CHs (each, managing a cluster of three to four other CHs on the second level, and each of the managed CHs managing a cluster of IoTDs), generating a hundred Tx in ten blocks.

Considering the average Tx latency in the three tables, we can see a better latency in the two-tiered model compared to the flat model by 23.7%. Whereas the SSD model outperformed the flat model by 198% and

	Txthroughput	**AverageTxLatency**	**Simulation Duration**	**Average Block Delay**
	2.05	23.67	48.86	14.76
	2.00	25.12	50.04	11.78
	2.27	24.75	44.10	14.62
	2.20	23.40	45.39	13.37
	1.98	27.13	50.44	14.31
	2.46	23.07	40.60	14.32
	2.29	26.80	43.64	14.86
	2.56	22.60	39.12	13.64
	2.47	23.93	40.52	14.32
	1.71	23.13	58.35	13.30
Average	**2.20**	**24.36**	**46.11**	**13.93**

Table 8.1 Global Statistics for the Flat Architecture

.

	Txthroughput	**AverageTxLatency**	**AverageTxLatencyColocal**	**AverageTxLatencyNonColocal**	**Number of Colocal Tx**	**Number of Non-Colocal Tx**	**Simulation Duration**	**Average Block Delay**
	2.50	17.78	20.70	17.69	3.00	97.00	39.93	4.48
	2.22	17.12	32.41	16.97	1.00	99.00	44.95	3.12
	1.95	26.95	29.48	26.85	4.00	96.00	51.27	3.33
	2.59	18.14	15.08	18.27	4.00	96.00	38.58	3.83
	2.05	18.08	16.68	18.09	1.00	99.00	48.75	3.42
	2.26	16.81	16.16	16.82	1.00	99.00	44.21	3.73
	2.86	16.93	14.78	17.00	3.00	97.00	34.93	3.59
	2.12	23.12	10.75	23.24	1.00	99.00	47.07	3.16
	1.89	21.43	14.85	21.56	2.00	98.00	53.02	4.64
	1.96	19.54	25.42	19.29	4.00	96.00	50.93	2.90
	2.09	**20.28**	**24.30**	**20.11**	4.00	96.00	47.79	3.49
	2.16	20.02	33.25	19.32	5.00	95.00	46.39	2.68
Average	**2.22**	**19.68**	**21.15**	**19.60**	**2.75**	**97.25**	**45.65**	**3.53**

Table 8.2 Global Statistics for the Two-Tiered Architecture with 8 CHs

	Txthroughput	**AverageTxLatency**	**AverageTxLatencyColocal**	**AverageTxLatencyNonColocal**	**Number of Colocal Tx**	**Number of Non-Colocal Tx**	**Simulation Duration**	**Average Block Delay**
	6.61	8.18	8.04	8.20	16.00	84.00	15.14	2.12
	6.09	8.61	9.04	8.56	10.00	90.00	16.43	2.65
	4.93	8.94	7.73	9.00	6.00	94.00	20.30	2.17
	6.26	8.24	8.68	8.19	11.00	89.00	15.98	2.44
	6.78	7.34	6.80	7.38	6.00	94.00	14.76	2.90
	5.73	7.95	8.40	7.89	14.00	86.00	17.44	2.37
	6.11	8.35	8.33	8.35	12.00	88.00	16.36	2.99
	6.50	8.36	7.33	8.47	10.00	90.00	15.39	2.46
	6.40	7.94	7.60	7.98	14.00	86.00	15.62	2.02
	5.97	7.84	8.06	7.81	12.00	88.00	16.74	2.59
Average	**6.14**	**8.17**	**8.00**	**8.18**	**11.10**	**88.90**	**16.41**	**2.47**

Table 8.3 Global Statistics for the SSD Architecture with 8 CHs

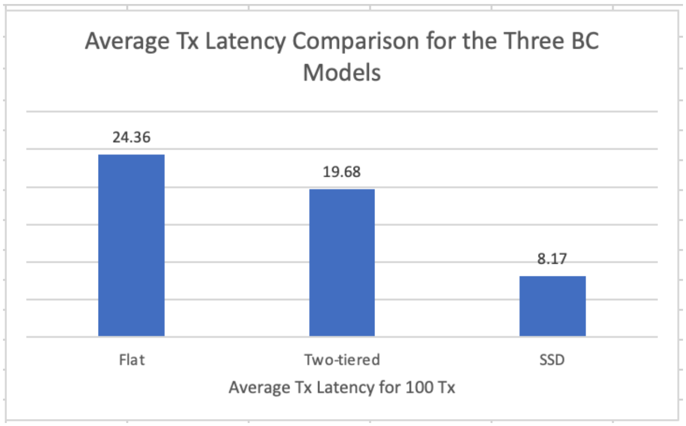

Figure 8.4 Average Tx Latency for 100Tx in the Three BC Models.

outperformed the two-tiered model by 143%, as in Figure 8.4.

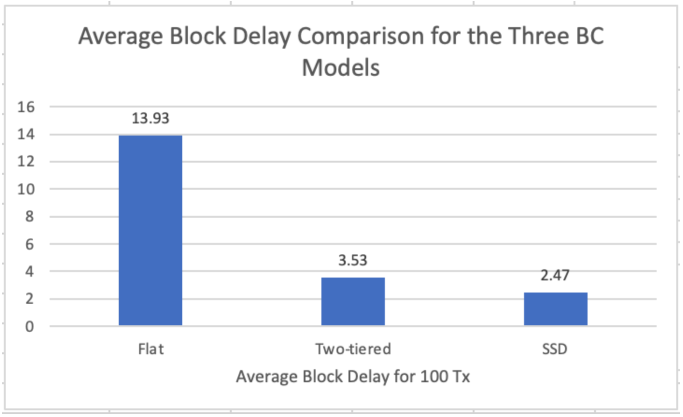

Figure 8.5 Average Block Delay for 100Tx in the Three BC Models.

Considering the average block delay, we can see that the two-tiered model produced shorter delays than the flat model by 294% , whereas the SSD model produced even shorter delays compared to the flat model with a better latency of 463.8% and compared to the two-tiered model with a better latency of 42.9%, as in Figure 8.5.

Considering the throughput, we can see that the two-tiered model produced a higher throughput than the flat model with a better latency of 1%, whereas the SSD model produced an even better throughput compared to the flat model with a better latency of 179% and compared to the two-tiered model with a better latency of 176%, as in Figure 8.6.

What's interesting is the fact that those results are based on a flat architecture having only 6% of the nodes as block generating nodes. Considering the results shown in Table 8.4 (each row corresponds to one run),

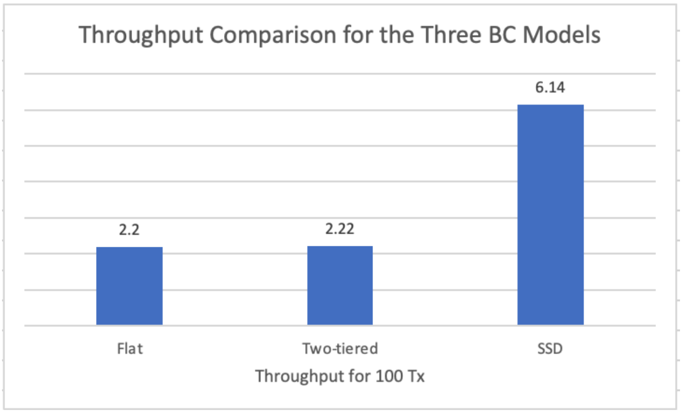

Figure 8.6 Throughput for 100Tx in the Three BC Models.

Txthroughput	AverageTxLatency	Simulation Duration	Number of Blocks	Average Block Delay	Block Generating Nodes
2.20	24.36	46.11	10	13.93	16
0.89	69.07	89.55	8	53.89	60
0.77	72.24	103.73	8	53.49	100
0.16	119.49	122.15	2	114.75	130
0.10	185.60	209.75	2	161.23	200
0.12	226.96	257.05	3	192.57	220

Table 8.4 The Flat Architecture with varied numbers of block-generating nodes (256 total nodes)

where we varied the number of generating nodes, SSD outperformed the flat model by 745%, 783%, 1362%, 2170.9% and 2677% for the average Tx latency in the case of a flat model having 23%, 39%, 50%, 78% and 85% of the nodes as as block generating nodes. SSD outperformed the flat model by 88.6% for the average Tx latency in the case of a flat model having 61% of the nodes as as block generating nodes. We can clearly see that the latency of the flat model exponentially increases with the growth of the number of block generating nodes.

8.5 Comparison of the Simulation Results of the SSD Architecture Built with Different Numbers of CHs

In this section, we present the second set of simulations. This set of simulations is to compare the SSD builds. Each built has a different number of topmost level CHs and approximately 260 IoTD nodes (with a fluctuation of 1.5% for the number of IoTDs, among all three builds) while processing 500 Tx, and building 50 blocks. Each simulation runs for ten runs. The splits and merges take place through running the SSD algorithm so that the end result lead to three three-level structures (since adding more levels lead to same results using CHAC) having a number of level-N nodes equal to $2CHs$, $4CHs$ and $8CH$. The goal is to study how the number of topmost level CHs influences the latency of the SSD BC model in order to validate the concept of colocality.

Table 8.5, Table 8.6, Table 8.7 and Table 8.8 as well as in Figure 8.7, where each row corresponds to one run, will allow us to compare the three SSD builds in terms of a number of criteria, such as the efficiency of the node colocality, the average Tx delay as well as the average block delay (among others).

	Txthroughput	AverageTxLatency	AverageTxLatencyColocal	AverageTxLatencyNonColocal	Number of Colocal Tx	Number of Non-Colocal Tx	Simulation Duration	Average Block Delay
	38.99	6.73	6.68	6.77	258.00	242.00	12.82	0.82
	38.39	7.99	7.97	8.00	244.00	256.00	13.02	0.90
	28.08	8.56	8.53	8.58	227.00	273.00	17.80	0.89
	31.90	7.99	7.98	8.00	258.00	242.00	15.68	0.83
	35.72	9.17	9.14	9.20	233.00	267.00	14.00	1.07
	34.73	7.52	7.43	7.60	242.00	258.00	14.40	0.89
	39.59	6.55	6.52	6.58	261.00	239.00	12.63	0.83
	36.96	7.68	7.67	7.70	271.00	229.00	13.53	0.81
	29.43	8.05	8.01	8.09	250.00	250.00	16.99	0.94
	34.25	10.15	10.14	10.17	259.00	241.00	14.60	0.71
Average	**34.80**	**8.04**	**8.01**	**8.07**	**250.30**	**249.70**	**14.55**	**0.87**

Table 8.5 SSD Global STATS 500Tx 2CHs

	Txthroughput	AverageTxLatency	AverageTxLatencyColocal	AverageTxLatencyNonColocal	Number of Colocal Tx	Number of Non-Colocal Tx	Simulation Duration	Average Block Delay
	24.00	10.61	10.55	10.63	112	388	20.84	1.82
	25.49	9.40	9.18	9.47	120	380	19.62	1.59
	21.25	10.00	9.89	10.03	118	382	23.53	1.85
	21.48	10.02	9.93	10.05	126	374	23.28	1.95
	27.35	9.49	9.31	9.56	133	367	18.28	1.80
	24.88	11.29	11.23	11.31	112	388	20.10	1.91
	31.99	9.53	9.52	9.53	147	353	15.63	1.80
	29.03	8.99	8.69	9.11	134	366	17.22	1.67
	20.31	10.18	9.87	10.27	112	388	24.62	1.70
	27.51	9.68	9.55	9.73	121	379	18.18	1.48
Average	**25.33**	**9.92**	**9.77**	**9.97**	**123.5**	**376.5**	**20.13**	**1.76**

Table 8.6 SSD Global STATS 500Tx 4CHs

	Txthroughput	AverageTxLatency	averageTxLatencyColocal	averageTxLatencyNonColocal	Number of Colocal Tx	Number of Non-Colocal Tx	Simulation Duration	Average Block Delay
	16.79	13.86	12.99	14.02	79	421	29.78	2.40
	13.17	14.00	12.93	14.16	66	434	37.98	2.31
	14.45	14.28	13.89	14.33	56	444	34.61	2.33
	14.85	13.34	12.58	13.44	59	441	33.67	2.42
	18.97	12.78	12.00	12.92	76	424	26.36	2.23
	15.65	14.21	13.16	14.35	59	441	31.96	2.27
	14.73	14.01	13.84	14.04	71	429	33.94	2.40
	20.89	12.99	12.02	13.14	69	431	23.93	2.38
	18.94	11.99	11.76	12.02	58	442	26.40	2.39
	21.52	12.46	12.15	12.49	49	451	23.23	2.30
Average	**17.00**	**13.39**	**12.73**	**13.49**	**64.2**	**435.8**	**30.19**	**2.34**

Table 8.7 SSD Global STATS 500Tx 8CHs

	Txthroughput	AverageTxLatency	averageTxLatencyColocal	averageTxLatencyNonColocal	Number of Colocal Tx	Number of Non-Colocal Tx	Simulation Duration	Average Block Delay
SSD- 2CHs	34.80	8.04	8.01	8.07	250.3	249.7	14.55	0.87
SSD- 4CHs	25.33	9.92	9.77	9.97	123.5	376.5	20.13	1.76
SSD- 8CHs	17.00	13.39	12.73	13.49	64.2	435.8	30.19	2.34

Table 8.8 SSD Global Statistics 500Tx 2CHs VS 4CHs VS 8CHs

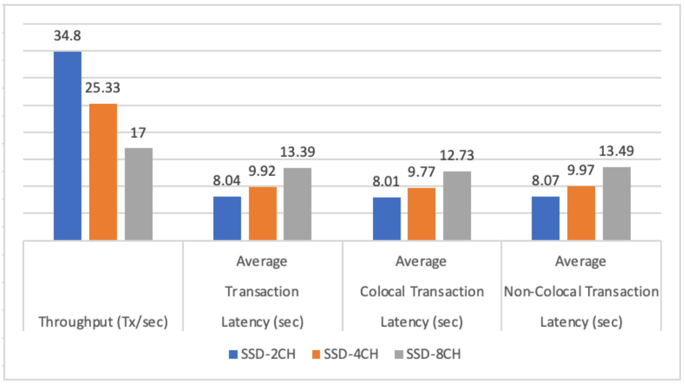

Figure 8.7 Comparison of the SSD BC builds for 500 Tx

Considering the average Tx latency, we can see a better latency of 102% in the 2CH SSD build (8.04 seconds) compared to the 4CH SSD build (9.92 seconds), which outperforms the 8CH SSD build (13.39 seconds) by 35%.

Considering the average block delay, we can see that the 2CH SSD build produced shorter delays (0.87 seconds), which are 51% shorter that the 4CH SSD build which took 1.76 seconds, whereas the 8CH SSD build took 2.34 seconds (33% higher that the 4CH SSD build), which is the longest delay in all three models.

Considering the colocality in SSD, we can see that the average Tx latency for the colocal Tx is less than the average Tx latency for the non-colocal Tx in all three builds. That difference is 0.6% in the case of the 2CH SSD build, whereas it is 2% in the case of the 4CH SSD build, and it is 5.9% in the case of the 8CH SSD build.

Considering the throughput, we can see that the 2CH SSD build outperformed the 4CH SSD build by 37%, whereas the 4CH SSD build outperformed the 8CH SSD build by 49%. The simulation duration increases with the number of the topmost level CHs, so it increases by 38% going from two CHs to four CHs, whereas it increases by 50% going from four CHs to eight CHs.

8.6 Interpretation of the Results

8.6.1 Comparison of the Three Architectures Types

Considering the comparison among the SSD, the two-tiered and the flat architectural models, we can see that the SSD model outperformed both the flat and the two-tiered models in terms of the average Tx latency and the average block latency. This is due to the fact that the SSD model has a smaller number of $level\_N$

CHs to handle the Tx and the blocks and thus less propagation delays thanks to the multi-tiered design and the use of CHAC. Since the efficiency of the flat model deteriorates exponentially with the number of the block generating nodes, SSD outperformance for the flat model would also increase exponentially with the growth of the number of the block generating nodes in the flat model.

Considering the comparison of the three models in terms of the throughput, SSD also outperformed both models due to the fact that SSD simulation duration is 179% less than the two-tiered model's and is 180% less than the flat model. So we can notice that there is an inverse relationship between the throughput and the duration of the simulation, because the throughput is the number of the appended Tx divided by the simulation duration. So since we process a fixed number of 100Tx during each simulation where a 100% of such Tx get appended into blocks on every run, the numbers produced make sense.

8.6.2 Comparison of the SSD Architecture Built with Different Numbers Of CHs

Comparing the three different builds of the SSD model, we can see, by observing the average Tx latency and the average block delay values, That a smaller number of $level-N$ CHs is more efficient in terms of Tx delay and block delay. This is due to the fact that the more the number of $level-N$ CHs that exist in the build, the more the number of hops needed to propagate the Tx as well as the blocks among those $level-N$ CHs, which affects the Tx and block latency.

In the architecture, we leverage the fact that each node has a CHAC to eliminate the extra hop required for colocalitity verification that may be performed at the direct parent CH. Instead, the colocality can be directly verified at the requester node that generates the Tx by checking the CHAC of the requester and the requestee , and if there is a match, hence they are colocal, the Tx is sent directly to the parent topmost level CH in $O(onehop)$. Otherwise, the Tx is sent to the requester parent topmost level CH, where it propagates among all topmost level CHs to look for the topmost level CH that is parent for the requestee. In all cases, the Tx signature verification takes place at the topmost level parent. Verifying the Tx at the topmost level instead of performing that verification at the direct parent eliminates one extra hop (one propagation delay) while the Tx travels up the architecture. This improvement saves $O(t)$ hops, each costing one Tx propagation delay, where t is the number of Tx generated.

Considering the improvement of the average Tx latency for the colocal nodes participating in the same Tx versus the non-colocal nodes, we can see that the latency improvement achieved by the node colocality becomes more obvious the more we increase the number of level-N CHs within an SSD build. Actually with a small number of level-N CHs, the probability of colocality for the nodes participating in the same Tx increases since that probability is proportional to the inverse of the number of the level-N CHs. In that case, if the nodes are non-colocal, the number of hops needed to be performed by a Tx forwarded towards level-N CHs in a search for the requestee information will be small, and hence the maximum of the propagation delays associated with such hops will be small. Thus whether the Tx is colocal or not will not make that big of a difference. Whereas with a larger number of level-N CHs, the probability of colocality for the nodes participating in the same Tx decreases, and the number of hops needed to be performed by a Tx forwarded towards level-N CHs in a search for the requestee information becomes significant, and hence the maximum of the propagation delays associated with such hops will be significant and this is where leveraging the node colocality comes into play, as it will save that extra "significant" propagation delay associated with the search for the requestee, which will have a great effect on the overall latency.

From Table 8.5, we can see that with a small number of level-N CHs, the probability of colocality increases, as shown by the ratio of the number of colocal versus the number of non-colocal Tx in the case of the 2CH build. This fifty fifty ratio might make us think that this will translate into a large difference between the average Tx latency for the colocal and the non-colocal Tx. However that difference gets smaller (0.7% in the case of the 2CH build) the smaller the number of level-N CHs within the SSD build due to the smaller number of hops needed to propagate the Tx in that case. Whereas if we look at the case of the 8CH build, we see that the ratio of colocal versus non-colocal Tx is almost 25% versus 75% and the difference between the average Tx latency for the colocal and the non-colocal Tx is 5.9%.

We can also consider the Tx Time Elapsed, which is the duration starting from the Tx creation until it gets appended into BC, versus the Tx Net Time Elapsed, which is the duration starting from the Tx Creation until it reaches a block as shown in Figure 8.8 and Figure 8.9, as well as in Figure 8.10 and Figure 8.11. We can see that time elapsed and the net time elapsed for colocal Tx are eleven and nine, approximately, respectively, whereas for the non colocal Tx, those values are thirteen and more than eleven, approximately, respectively. These figures represent one run of the simulation, and those values varies from run to run, however they show the clear efficiency benefit achieved by leveraging the node colocality, as it improves the value of the time elapsed (which is the whole journey the Tx needs to travel till it reaches BC), in case of colocal Tx, to be equivalent to the value of net time elapsed (which is the journey the Tx needs to travel till it reaches a block, only) in case of non-colocal Tx. Which is a great improvement, equivalent to the red portion shown in Figure 8.9.

Though this improvement is only represented for this specific run, this improvement has been the case of 90% of the runs and hence, it proves the feasibility of leveraging the node colocality as a main factor to improve BC consensus duration in our multi-tiered architecture.

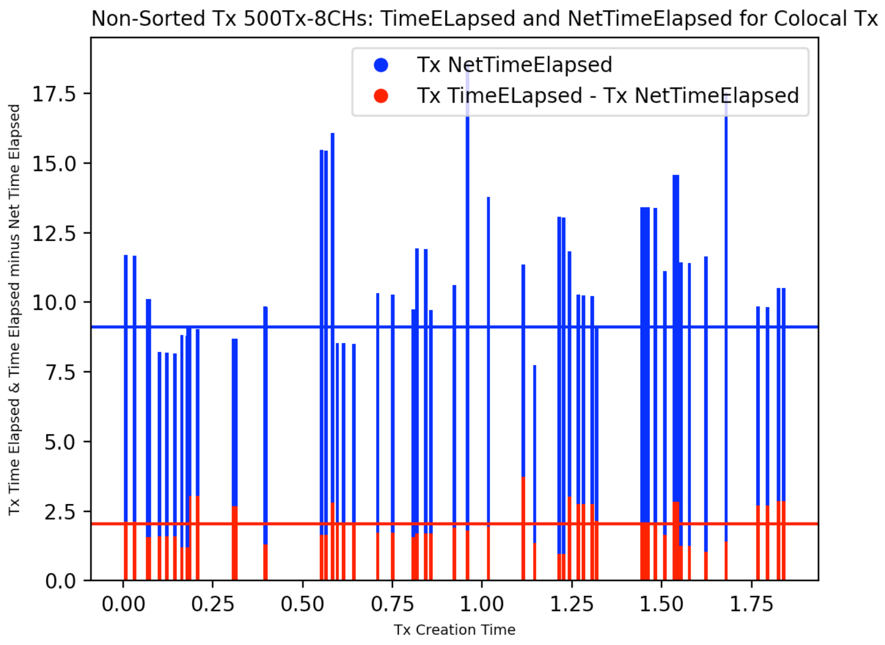

Figure 8.8 SSD 500Tx 8CHs - Time Elapsed and Net Time Elapsed Colocal

The Tx propagation delay base value is a system parameter used to calculate Tx propagation delays as a Tx (colocal or non-colocal) propagates from an IoTD towards a topmost CH. In Table 8.5, Table 8.6, Table 8.7 and Table 8.8, the increase of the difference between colocal and non-colocal average Tx latency values are by 0.75% for the 2CH build, by 2.1% for the 4CH build and by 6% for the 8CH build. Increasing the Tx propagation delay base value by multiplying it by 1000 for 10 runs for each of the 2CH, the 4CH and the 8CH builds results in an increase of the difference between colocal and non-colocal average Tx latency values by 1.5% for the 2CH build, by 3.25% for the 4CH build and by 6.8% for the 8CH build, as shown in Table 8.9, Table 8.10 and Table 8.11, respectively, where each table row corresponds to one run.

As we can see, though the system parameter modified does not have a relationship with node colocality, since it is the same for all Tx, its modification does have a direct effect over the difference between colocal and non-colocal average Tx latency values. This gives us an insight that changing any of the system

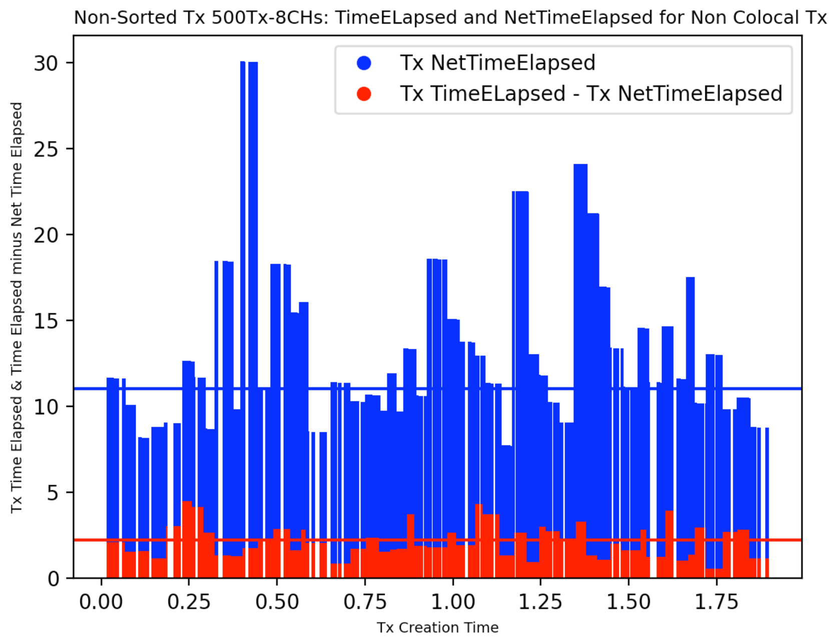

Figure 8.9 SSD 500Tx 8CHs - Time Elapsed and Net Time Elapsed NonColocal

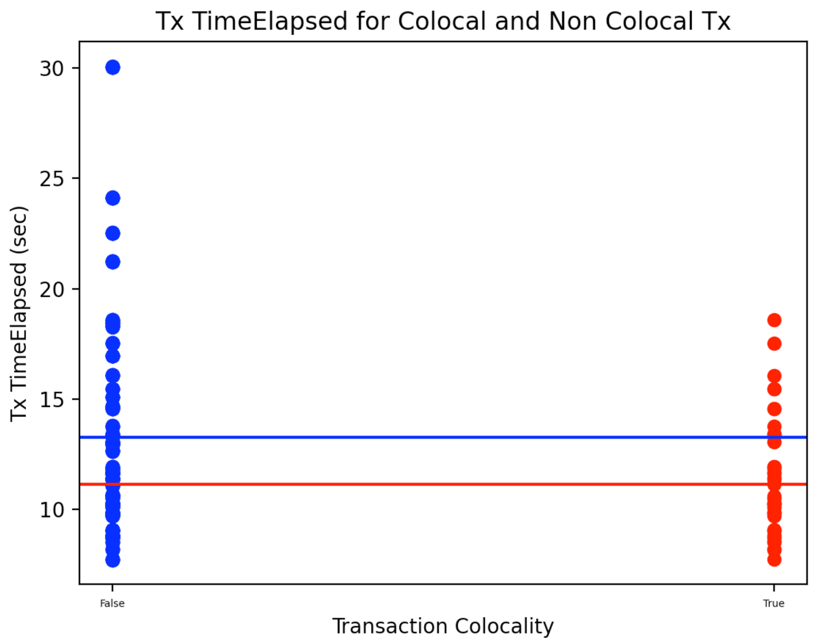

Figure 8.10 SSD 500Tx 8CHs - Time Elapsed Colocal VS NonColocal

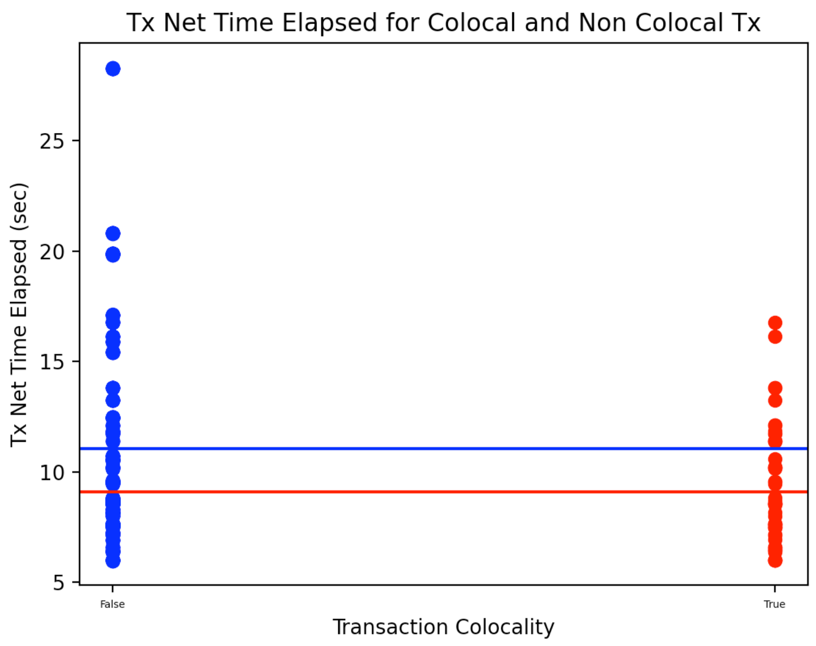

Figure 8.11 SSD 500Tx 8CHs - Net Time Elapsed Colocal VS NonColocal

parameters, even the ones non related to node colocality, may affect the difference between colocal and non-colocal average Tx latency values.

	Txthroughput	AverageTxLatency	averageTxLatencyColocal	averageTxLatencyNonColocal	Number of Colocal Tx	Number of Non-Colocal Tx	Simulation Duration	Number of Blocks	Average Block Delay
	31.06	9.07	9.06	9.08	255	245	16.10	50	0.83
	38.01	8.27	8.25	8.30	272	228	13.15	50	0.77
	30.25	8.25	8.08	8.44	262	238	16.53	50	1.03
	30.72	9.00	8.98	9.03	257	243	16.28	50	1.00
	34.52	9.39	9.33	9.44	253	247	14.48	50	1.05
	31.00	8.60	8.39	8.79	237	263	16.13	50	1.15
	32.63	9.52	9.48	9.57	239	261	15.32	50	0.83
	34.61	10.04	9.99	10.09	258	242	14.45	50	0.94
	35.08	6.94	6.91	6.97	242	258	14.25	50	0.74
	28.22	8.18	8.14	8.22	238	262	17.72	50	1.06
	34.16	8.52	8.47	8.56	242	258	14.64	50	0.82
Average	**32.75**	**8.71**	**8.64**	**8.77**	**250.45**	**249.55**	**15.37**	**50**	**0.93**

Table 8.9 SSD Global STATS 500Tx 2CHs with a modified Tx Latency

	Txthroughput	AverageTxLatency	averageTxLatencyColocal	averageTxLatencyNonColocal	Number of Colocal Tx	Number of Non-Colocal Tx	Simulation Duration	Number of Blocks	Average Block Delay
	20.93	10.83	10.63	10.90	130	370	23.89	50	1.60
	32.42	10.45	10.16	10.56	140	360	15.42	50	1.58
	26.62	10.41	10.20	10.48	128	372	18.79	50	1.64
	25.05	11.66	11.51	11.71	108	392	19.96	50	1.60
	26.77	10.30	10.05	10.39	133	367	18.68	50	1.56
	20.74	11.90	11.68	11.95	98	402	24.10	50	1.63
	22.62	11.72	11.52	11.79	125	375	22.11	50	1.50
	27.45	10.79	10.40	10.92	121	379	18.22	50	1.41
	26.08	10.70	10.28	10.86	137	363	19.17	50	1.79
	24.84	11.54	11.32	11.61	120	380	20.13	50	1.48
Average	**25.35**	**11.03**	**10.77**	**11.12**	**124**	**376**	**20.05**	**50**	**1.58**

Table 8.10 SSD Global STATS 500Tx 4CHs with a modified Tx Latency

	Txthroughput	AverageTxLatency	averageTxLatencyColocal	averageTxLatencyNonColocal	Number of Colocal Tx	Number of Non-Colocal Tx	Simulation Duration	Number of Blocks	Average Block Delay
	20.28	12.40	11.81	12.49	68	432	24.65	50	2.51
	15.52	12.46	11.99	12.53	68	432	32.22	50	2.25
	14.71	13.69	12.73	13.82	60	440	33.98	50	2.35
	10.70	13.75	12.97	13.86	63	437	46.74	50	2.29
	14.67	13.88	12.77	14.02	56	444	34.09	50	2.31
	16.66	13.48	12.57	13.60	58	442	30.01	50	2.48
	17.02	13.23	12.43	13.34	58	442	29.38	50	2.26
	21.24	13.04	12.47	13.12	61	439	23.54	50	2.23
	16.02	13.92	13.43	13.98	57	443	31.21	50	2.44
	14.27	12.76	11.90	12.85	49	451	35.05	50	2.29
Average	**16.11**	**13.26**	**12.51**	**13.36**	**59.8**	**440.2**	**32.09**	**50**	**2.34**

Table 8.11 SSD Global STATS 500Tx 8CHs with a modified Tx Latency

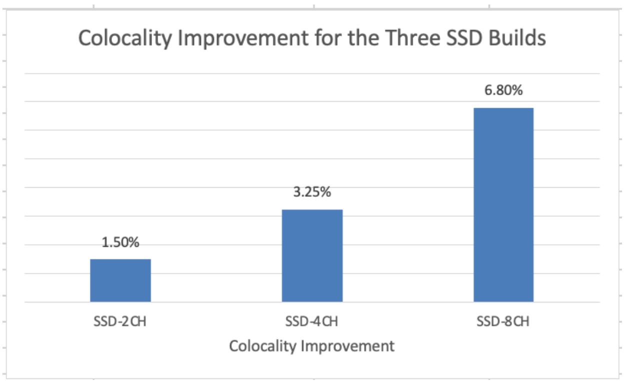

Figure 8.12 Colocality Effect for the Three SSD Builds

According to the observations, BC will be more efficient with more colocal Tx (less number of topmost CHs). However this leads to the risk of having too few level-N CHs, and thus, to a concentration of power into BC. This creates a conflict between performance and security and hence, a tradeoff to be carefully taken care of, when the design of the architecture gets implemented. Also here, it is important to note that doubling the architecture's performance does not always mean halving the security guarantees (or in other words, doubling the security risks), both factors affect each other by different ratios. So careful decisions must be made towards creating a balance between those two tradeoffs.

8.7 Simulation General Analysis

Considering the overall results, we can deduce that the multi-level architecture outperformed both the flat and two-tiered architectures. This superiority was evident in terms of average Tx latency, block delays, and Tx throughput. The hierarchical design reduced the overall number of hops required for a Tx to make, as well as the number of nodes it needs to encounter from the time it gets generated by an IoTD until it is appended into the BC. The hierarchical multi-tiered design reduces the number of CHs needed to build blocks to a small number of forest trees. Thus to achieve Tx verification with the absence of colocality, a

Tx travels towards a very small number of nodes in case of our multi-tiered model, contrasting with the two-tiered model where a Tx travels towards a large number of nodes at the top level of the structure, and with the flat model where a Tx traverses all the structure's nodes. Additionally, the colocality of nodes participating in the same Tx within the same tree of the SSD model further improves consensus efficiency (latency).

The detailed criteria stemming from our results enable us to provide a proof of concept for the SSD design along with the SSD algorithm and to rank SSD architecture as a BC model that surpasses both flat and two-tiered models. Furthermore, the multi-tiered model successfully achieves its objectives regarding adhering to node management overhead bounds and consensus decentralization bounds, while also complying with blockchain functionality, security, and reliability requirements. The system's capacity to fulfill SSD objectives, even as the system size grows, is well ascertained based on the trends and asymptotic behaviors identified in the results.

CHAPTER 9
CONCLUSION AND FUTURE WORK

9.1 Conclusion

In this study, we introduce a multi-tiered blockchain (BC) node-organizing design concept, along with three distinct multi-tiered architecture designs (SSD, Partitioned and RoR) constructed based on various conceptual approaches, all aiming to realize the proposed design model. The multi-tiered design model aims to address the issues related to flat and two-tiered models found in the literature. Thus our model aims to improve BC scalability and efficiency in the context of the Internet of Things (IoT) by improving the node organization of the overlay network while providing a balance between node management overhead and consensus decentralization—a critical security consideration for BC technology.

At the heart of the model, we propose a consensus-agnostic consensus-accelerating protocol such that consensus time is reduced to $O(1)$ plus $c \cdot O(maximum\, propagation\, delay\, towards\, topmost\, CHs)$, where c is a constant s.t. without colocality, $c \leq 3$ worst case; while with colocality, $c \leq 2$, best case. We also propose a node look-up positioning mechanism, named CHAC, for locating nodes in $O(1)$ time complexity. Additionally, we introduce the concept of node colocality, which, when leveraged, allows consensus time to be further improved. Both CHAC and the colocality aspects are incorporated into our model to overcome some of the design's complexity challenges and to make BC efficiency goals more achievable. The design concept as well as the architecture designs are clustering algorithm agnostic, consensus algorithm agnostic as well as application agnostic.

All the proposed designs are crafted based on the same multi-tiered design concept, adopt the same consensus-accelerating algorithm with equivalent complexity and improve consensus time when compared to flat and two-tiered architectures. This improvement is attributable to the multi-tiered design concept, which increases scalability and reduces the number of topmost CHs involved in the consensus process to a logarithmically bounded reduction of the CHs located at the levels below. The asymptotic analysis of the consensus-accelerating protocol is similar for all the architecture designs, thus the designs may be considered as equally efficient in that respect.

The Self-Scaling Dynamic (SSD) architecture has been selected for further study, implementation, and testing. At the core of the SSD architecture is a dynamic SSD algorithm that, through the use of MUT and CLT thresholds, strives to maintain a dynamic load balance between node management overhead and control decentralization thanks to the cyclic aspect of the algorithm. The algorithm helps the architecture dynamically expand and shrink, through cascading cluster splits and merges, in order to adequately accommodate node joins and departures.

SSD's overall efficiency has an inverse relationship with the number of the topmost CHs. The deeper the forest, the fewer the topmost CHs that verify transactions and blocks, the shorter the consensus delays, and the more scalable and efficient the model becomes. Nevertheless, having too few topmost CHs may lead to a concentration of power, which is a security concern. Thus there is a conflict between performance, security and workload balance while expanding and shrinking the architecture and those factors affect each other by different ratios, hence, a trade-off is to be carefully considered. SSD is application agnostic, however careful decisions must be made towards choosing appropriate values for MUT and CLT for the different levels of the architecture and for the number of minimum levels required. Such decisions, along with the trade-offs mentioned, have to be carefully considered when SSD model gets implemented in a real-word application.

The simulation provides a proof of concept for our dynamic SSD design and algorithm as it showed the plausibility of performing the dynamic architectural modifications without sacrificing the integrity[1] of the node and cluster data and location, while achieving all the sought after goals. Also the simulation results rank the SSD architecture along with SSD algorithm as a BC model that outperforms both the flat and the two-tiered model, while also achieving the load balance, scalability and efficiency goals with respect to BC functionality and security requirements. The simulation results demonstrate assertive trends regarding the system's ability to continue to meet its objectives as it continues to grow. Additionally, the simulations prove the validity of the colocality concept, as well as its promising impact on reducing consensus time.

9.2 Contribution to the Advancement of Knowledge

This research has presented solutions to help BC-based IoTD achieve adequate scalability in terms of consensus throughput as well as in terms of block and Tx latency. We have obtained trends through evaluating the behavior of BC system during our simulations. In this effort, we have not aimed to design a new form of

[1] The integrity is dictated by the rule that each cluster must be managed by exactly one CH. The integrity of the node data might be sacrificed as the nodes change positions during the cluster splits and merges in such a dynamic architecture.

a custom-made consensus. We have rather aimed to improve the node organization, and the overall architectural aspects, starting from recommending the choice of the clustering algorithm, passing by the routing mechanisms that aim to map the logical locations of each node within the design abstraction layers and ending by evaluating the impact of node locality to reduce the number of hops made by messages carrying Tx common for nodes that are in logical proximity, hence, decreasing consensus time. In our research, we have presented a number of varied architectures for BC-based IoTDs, built based on different designs. The reason for adopting such approach is to evaluate the rising needs that each architecture presented, once built, which have lead us to produce better methods and more efficient designs and algorithms in order to meet those needs. Our approach was an attempt to find a solution to many BC problems, particularly the triangular tradeoff between security (in terms of consensus decentralization), traffic overhead and scalability, as we have decoupled all three aspects in order to reach better architectures that better meet todays BC-based IoTD scalability needs.

Our proposed design includes: a multi-tiered model to enhance flat and two-tiered models' scalability and efficiency algorithms, a number of architecture options to implement the model, a consensus-agnostic consensus-accelerating algorithm, a node positioning mechanism, a self-scaling algorithm to balance consensus decentralization versus node management overhead as well as a colocality concept to be leveraged to further reduce consensus time.The design is also application-agnostic and clustering algorithm-agnostic.

The proposed designs and concepts may serve as general-purpose solutions for:

1. scalability, namely the multi-tiered design;
2. peer-to-peer node-lookup and message-routing for distributed systems, namely the node positioning mechanism (CHAC);
3. load-balancing for clustered systems, namely the self-scaling algorithm that load-balances consensus decentralization versus node management overhead.

9.3 Published Contributions

Our work has led to the following publications:

1. Published conference papers:

   (a) **“A Blockchain Node Organizing and Auto-Scaling Architecture for the Internet of Things”** (Elsaadany et Bégin, 2023).

   (b) **“A Self-Scaling Dynamic Blockchain Model for IoT”** (Elsaadany et Bégin, 2024d).

2. Submitted and in-progress papers:

   (a) **“A Blockchain Auto-Scaling Node Organizing Design Model for the Internet of Things”** (Elsaadany et Bégin, 2024b).

   (b) **“A Number of Conceptual Scalable Peer-to-Peer Message-Routing and Node-Organizing Multi-Tiered Blockchain Architectures for IoT”** (Elsaadany et Bégin, 2024c).

   (c) **“A Blockchain Auto-Scaling Node Organizing Design for the Internet of Things”** (Elsaadany et Bégin, 2024a).

9.4 Simulation Limitations

We performed our simulation on an macAir with 1.1 GHz Dual-Core Intel processor and 8 GB memory. This limited computing power forced us to restrict the size of our simulation parameters, in terms of the number of Tx generated as well as the number of IoTDs simulated. Our architecture ran smoothly with the number of Tx and IoTDs simulated in Chapter 7 and based on the resulting trends, we can expect smooth performance for larger numbers of transactions and IoTDs in the future provided more computing power is available, assuming that the trends will continue and that the current SSD architecture will remain applicable.

9.5 Future Work

The goal of our study was to overcome the main issues causing delays in a flat BC as well as in two-tiered BC, and to address them with a newer design. Nevertheless, there is room for more improvement, through applying features that are yet to improve the efficiency of the architecture as follows:

9.5.0.1 Combining CHORD with the Self-Scaling Algorithm

In SSD, there is a need for a mechanism that provides resilience for the BC architecture. Resilience allows communication between nodes located at the same level of the forest. In our design, the topmost CHs holds their managed nodes' data and CHAC helps locating the managed nodes in $O(1)$. Nevertheless, if same-level communication is necessary, it would be beneficial to have a mechanism that efficiently provides a location service for nodes storing a particular data item, even amidst continuous changes in the system's organization. According to the self-scaling algorithm, there exist an amount of data that should be always maintained and quickly exchanged amongst all nodes to allow each such node to determine its position in the hierarchy, its corresponding subtree where it belongs, and the size of its managed cluster(s). Since SSD is tiered, creating faster communication amongst different tiered nodes to exchange such information and to locate nodes with a particular data item (i.e. CHAC) located at a certain level, is necessary, which is another challenge faced. Our architecture aims to provide a practical BC-based IoT scalability, however such issues might slow our architecture down if they remain untacked.

Such issues can be addressed by incorporating CHORD protocol into our architecture. Verifying the relevance of employing CHORD at each level of the architecture to enhance node communication could prove beneficial. CHORD, a distributed lookup protocol to efficiently locate node(s) storing a particular data item.CHORD considers the nodes as being organized on a circle (ring). Given a key, CHORD maps it onto a node using Consistent Hashing and takes $O(\log X)$ running time, where X is the number of CHORD nodes. CHORD maintains its routing information as nodes join and leave the system in no more than $O(\log^2 X)$ messages (scaling logarithmically with the number of CHORD nodes). CHORD specifies how to find the locations of keys, how new nodes join the system, and how to recover from node failure (or planned departure). In particular, it can help to load balance the keys on each node, avoiding single points of failure or control while maintaining scalability. In CHORD, each node only needs to be aware of its successor node on a CHORD circle. Queries for a given identifier can be passed around the circle via successor pointers until they first encounter a node that succeeds the identifier; this is the node the query maps to. Its ability to associate versatile values with keys and to handle node joining and leaving with only $O(1/X)$ fraction of keys changing locations (in order to maintain load balance), are the reasons why we recommend leveraging CHORD to support SSD, as a future work.

CHORD is to be used differently in our architecture options in order to complement our designs. As another logical layer, CHORD can be applied on each SSD layer independently to accelerate message routing, to locate nodes, to connect layers, as depicted in Figure 9.1, which provides reliability for the SSD system. It can be applied on level-0 of D nodes, where $O(\log D)$ nodes will service CHORD routing on that ring. Then it can be applied on level-1 CHs as a second ring built inside the first ring, where $O(\log level1{-}CHs)$ nodes will service CHORD routing. And so on. SSD will be built as one CHORD circle inside another where the innermost circle corresponds to level-N, and the outermost circle corresponds to level-0 IoTDs. We

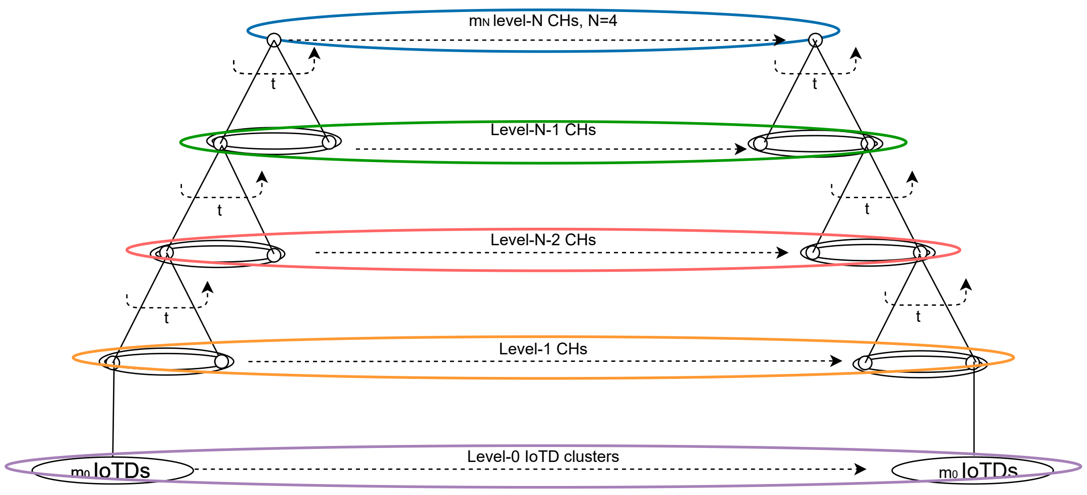

Figure 9.1 SSD architecture incorporating CHORD

recommend integrating CHORD into SSD architecture as a future work since the preliminary cost analysis for integrating CHORD into SSD seem promising.

9.5.0.2 The Probability of Colocality

As a future work, we can find a mechanism to analyze the frequency of interaction between any two nodes that happen to be parties of the same Tx, and apply an algorithm to predict future frequency of interactions between those two nodes (e.g. an AI mechanism). Depending on the analysis and prediction results, we can decide to leverage the concept of colocality for those two nodes while applying a different node organization (reorganization). In this section, we suggest a mechanism for node reorganization that may help leveraging the colocality between frequently interacting nodes within the same Tx.

In order to achieve a direct colocality, we need two nodes to be within the same tree (collisions) as much as possible. If we get no collision, the whole height of the structure will get travelled once (starting from the IoTD generating the Tx, the requester, towards its topmost parent) then all topmost nodes will be visited as well as their managed nodes (until the IoTD participating in the Tx, the requestee, is found) in order to find the parties of a multi-signature Tx. Which will be in $O(n)$, where n is the number of nodes in the whole structure.

As a solution, a mechanism that aims to enforce a logical shorter distance (colocality) between nodes with history of frequent transaction interactions could be applied. The mechanism would aim to logically migrate

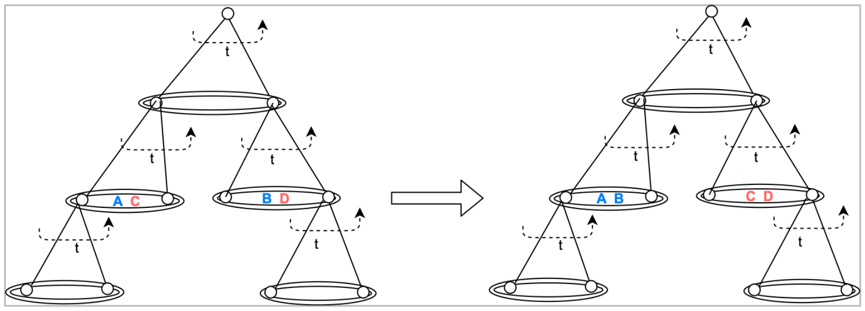

Figure 9.2 SSD - The Colocality through Critical Bonding

one IoTD from one cluster towards another, or even to let two IoTDs of different clusters to exchange their positions (to maintain load balancing) in order to get closer to the IoTDs with which they make Tx more frequently. This would take place by getting the parent CH to register the frequency with which two nodes interact in common Tx and once a threshold is reached, one node migration takes place towards the cluster with the least size. For the frequency of interaction, the parent CH (or one of level-N CHs, if no common parent CH between both nodes) registers the number of common Tx interactions between nodes A and B within a time frame. If the interaction frequency reaches a predefined threshold, the CH marks both nodes as being in a "critical bond" state, such that if two other nodes, C and D, happen to be in a critical bond state as well, such that nodes C and A are at dist=0, while nodes D and B are at dist=0, and both pairs (A, C), and (B, D) are not in a critical bond state, then the pairs can be considered as a potential breakable pairs. The CH can then invoke a node exchange between breakable pairs, exchanging one node belonging to the first critical bond with another from the second critical bond, while their $dist \neq 0$, as in Figure 9.2. Clusters will end up with critical bonds nodes and hence, stronger node colocality for better Tx performance.

9.5.0.3 Other issues to improve

The following mechanisms can be added to the SSD architecture in order to further improve it:

1. Introducing variability in cluster sizes at each level may be an interesting aspect to test.
2. Researching the impact of IoTD mobility and churn[2] may be an interesting aspect to further study.
3. A mechanism that takes into account mobile IoTDs, necessitating changes in routing and CHAC ac-

[2] A mechanism is needed to facilitate the rapid adaptation of the architecture to churn resulting from frequent new node joins and the departure or failure of old nodes

cording to the location of each new IoTD, could be considered.

4. Since we have addressed all the research questions, the question concerning workload considerations still remains. Providing a mechanism to assess the managed node workload (in terms of the number of nodes within the managed clusters (cluster size) as well as the node workload, as well as the of managed clusters) versus the CH workload capacity, which may differ according to the level where the CH resides, an issue that may be considered as design agnostic. The CH capacity should be considered as an important constraint to determine MUT for each of the per-level CHs. Also a mechanism is required for determining a per-level CLT that ensures the decentralization of power within the node's as well as the CH's clusters according to each BC application constraints. Determining the MUT to be limited by/match the CHs capacity as well as the CLT to assure the decentralization of power will help a better design for the splits and merges in order to scale the architecture up or down accordingly. Also determining at which frequency will the workload be compared to the CH capacity (the frequency according to which MUT will change per level), and at which frequency the CLT will be modified according to the change of each application demands, is required. Determining an accurate MUT and CLT are application agnostic however further future work might take place to set some general guidelines in order to facilitate the decision for easily configuring them.

5. Assuming that only CHs get to append Tx to blocks while only managed IoTDs get to produce Tx, all nodes need to have a fair chance to append Tx to blocks and to produce Tx. A mechanism for nodes to exchange roles of IoTDs and CHs has been applied in order to allow the nodes to change position within the hierarchy's levels, however such a mechanism did not address the the exchange of positions and the possibility for the nodes to exchange roles of delegations in order to allow a fair chance to build blocks, a technique to address that latter issue would make SSD more comprehensive.

6. Researching node resilience mechanisms to face hostile scenarios is recommended for future works.

BIBLIOGRAPHY

(2006). Computer Science Department, Duke University cod cluster-on-demand. https://www2.cs.duke.edu/nicl/cod/, Accessed: 2019-03-22. Récupéré de https://www2.cs.duke.edu/nicl/cod/

(2012). StackOverflow-what's the diffference between cloud, grid and cluster. Accessed: 2019-03-22. Récupéré de https://stackoverflow.com/questions/9723040/what-is-the-difference-between-cloud-grid-and-cluster

(2016). Modularity (networks). Accessed: 2022-01-07. Récupéré de https://en.wikipedia.org/wiki/Modularity_(networks)#:~:text=Modularity%20compares%20the%20number%20of,edges%20are%20otherwise%20randomly%20attached.

(2017). Marklogic server-scalability, availability, and failover guide (cluster configuration). Accessed: 2019-03-22. Récupéré de https://docs.marklogic.com/guide/cluster.pdf

(2018). Cs1114: Study guide 3 (clustering algorithms). Récupéré de http://www.cs.cornell.edu/courses/cs1114/2012sp/lectures/cs1114-studyguide3.pdf

(2018). Hyperledger iroha documentation. Accessed: 2021-01-21. Récupéré de https://iroha.readthedocs.io/en/master/

(2020). Blocksim: An extensible simulation tool for blockchain systems. Accessed: 2021-12-16. Récupéré de https://www.frontiersin.org/articles/10.3389/fbloc.2020.00028/full

(2021). A next-generation smart contract and decentralized application platform, ethereum_whitepaper. Accessed: 2024-02-21. Récupéré de https://scholar.google.co.kr/citations?view_op=view_citation&hl=en&user=DLP9gTAAAAAJ&citation_for_view=DLP9gTAAAAAJ:IjCSPb-OGe4C

(2021). Wikipedia weekend project build your own supercomputer (pc and tech authority). Accessed: 2019-03-22. Récupéré de https://en.wikipedia.org/wiki/Computercluster

(2023). Uncle mining, an ethereum consensus protocol flaw. Accessed: 2023-06-07. Récupéré de https://bitslog.wordpress.com/2016/04/28/uncle-mining-an-ethereum-consensus-protocol-flaw/

(2024). Iota. Accessed: 2024-02-22. Récupéré de www.IOTA.org

Albulayhi, A. et Alsukayti, I. (2023). A blockchain-centric iot architecture for effective smart contract-based management of iot data communications. *Electronics*, *12*, 2564.

Alharby, M. et Moorsel, A. V. (2020).

Alharby, M. et Moorsel, A. V. (2022).

Alhusayni, A., Thayananthan, V., Albeshri, A. et Alghamdi, S. A. (2023). Decentralized multi-layered architecture to strengthen the security in the internet of things environment using blockchain technology. *Electronics*, *12*, 4314. `http://dx.doi.org/10.3390/electronics12204314`

Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M. et Yellick, J. (2018). Hyperledger fabric: A distributed operating system for permissioned blockchains.

Angin, P., Mert, M., Mete, O., Ramazanli, A., Sarica, K. et Gungoren, B. (2018). *A Blockchain-Based Decentralized Security Architecture for IoT*, (p. 3–18).

Atlam, H. F., Alenezi, A., Alassafi, M. O. et Wills, G. B. Blockchain with internet of things: Benefits, challenges, and future directions. volume 10. `http://dx.doi.org/10.5815/ijisa.2018.06.05`

Awad, A., German, R. et Dressler, F. (2011). Exploiting virtual coordinates for improved routing performance in sensor networks. *Mobile Computing, IEEE Transactions on*, *10*, 1214 – 1226. `http://dx.doi.org/10.1109/TMC.2010.218`

Bogdanov, A., Degtyarev, A., Korkhov, V., Kamande, M., Iakushkin, O. et Khvatov, V. (2018). About some of the blockchain problems.

Bravo-Marquez, F., Reeves, S. et Ugarte, M. (2019). Proof-of-learning: A blockchain consensus mechanism based on machine learning competitions. `http://dx.doi.org/10.1109/DAPPCON.2019.00023`

Buchman, E. (2016). Tendermint: Byzantine fault tolerance in the age of blockchains.

Buterin, V. (2021). Ethereum white paper: A next generation smart contract decentralized application platform. Accessed: 2021-01-25. Récupéré de `ethereum.org`

Cachin, C. et Vukolić, M. (2017). Blockchains consensus protocols in the wild.

Chen, M. (2023). Comparison on proof of work versus proof of stake and analysis on why ethereum converted to proof of stake. *Advances in Economics, Management and Political Sciences*, *12*, 200–204. `http://dx.doi.org/10.54254/2754-1169/12/20230624`

Chendeb, N., Khaled, N. et Agoulmine, N. (2020). Integrating blockchain with iot for a secure healthcare digital system. 8th International Workshop on ADVANCEs in ICT Infrastructures and Services (ADVANCE 2020), hal-02495262f.

Clauset, A., E J Newman, M. et Moore, C. (2005). Finding community structure in very large networks. *Physical review. E, Statistical, nonlinear, and soft matter physics*, *70*, 066111. `http://dx.doi.org/10.1103/PhysRevE.70.066111`

Clement, A., Wong, E., Alvisi, L., Dahlin, M. et Marchetti, M. (2009). Making byzantine fault tolerant systems tolerate byzantine faults. 153–168.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. et Stein, C. (1990). *INTRODUCTION TO ALGORITHMS*. MIT Press, McGraw-Hill.

Croman, K., Decker, C., Eyal, I., Gencer, A. E., Juels, A., Kosba, A., Miller, A., Saxena, P., Shi, E., Gün Sirer, E., Song, D. et Wattenhofer, R. (2016). On scaling decentralized blockchains (a position paper).

Delmolino, K., Arnett, M., Kosba, A., Miller, A. et Shi, E. (2016). Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. volume 9604, 79–94. http://dx.doi.org/10.1007/978-3-662-53357-4_6

Dorri, A., Kanhere, S. et Jurdak, R. (2018). Mof-bc: A memory optimized and flexible blockchain for large scale networks. *Future Generation Computer Systems*, *92*. http://dx.doi.org/10.1016/j.future.2018.10.002

Dorri, A., Kanhere, S., Jurdak, R. et Gauravaram, P. (2017a). Lsb: A lightweight scalable blockchain for iot security and privacy. *Journal of Parallel and Distributed Computing*, *134*. http://dx.doi.org/10.1016/j.jpdc.2019.08.005

Dorri, A., Kanhere, S. S., Jurdak, R. et Gauravaram, P. (2017b). Blockchain for iot security and privacy: The case study of a smart home. *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 618–623.

Dorri, A., S. Kanhere, S. et Jurdak, R. (2016). Blockchain in internet of things: Challenges and solutions. *CoRR*, *abs/1608.05187*. Récupéré de http://arxiv.org/abs/1608.05187

Dorri, A., S. Kanhere, S. et Jurdak, R. (2017c). Towards an optimized blockchain for iot. *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, 173–178.

Elsaadany, R. et Bégin, G. (2023). A blockchain node organizing and auto-scaling architecture for the internet of things. Dans *2023 Fifth International Conference on Blockchain Computing and Applications (BCCA)*, 36–43. http://dx.doi.org/10.1109/BCCA58897.2023.10338882

Elsaadany, R. et Bégin, G. (2024a). A blockchain auto-scaling node organizing design for the internet of things. *Special Issue on Innovation In Computing, Engineering Science & Technology organized by Advances in Science, Technology and Engineering Systems Journal (ASTESJ)*. Manuscript in preparation, due December 2024.

Elsaadany, R. et Bégin, G. (2024b). A blockchain auto-scaling node organizing design model for the internet of things. *Cluster Computing Journal by Springer*. Manuscript submitted for review.

Elsaadany, R. et Bégin, G. (2024c). A number of conceptual scalable peer-to-peer message-routing and node-organizing multi-tiered blockchain architectures for iot. Dans *2024 Seventh IEEE International Conference on Blockchain (Blockchain 2024)*. Manuscript submitted for review.

Elsaadany, R. et Bégin, G. (2024d). A self-scaling dynamic blockchain model for iot. Dans *2024 3rd IEEE International Conference on Computing and Machine Intelligence (ICMI)*. Manuscript accepted to be published.

Eyal, I., Gencer, A. E., Sirer, E. et Van Renesse, R. (2015). Bitcoin-ng: A scalable blockchain protocol.

Eyal, I., Keidar, I. et Rom, R. (2011). Distributed data clustering in sensor networks. *Distributed Computing*, *24*, 207–222. http://dx.doi.org/10.1007/s00446-011-0143-7

Feng, Z., Qiao, M. et Cheng, H. (2023). Modularity-based hypergraph clustering: Random hypergraph model, hyperedge-cluster relation, and computation. *Proceedings of the ACM on Management of Data*, *1*, 1–25. http://dx.doi.org/10.1145/3617335

Fersi, G. (2015). A distributed and flexible architecture for internet of things. *Elsevier B.V, Procedia Computer Science 73*, 130–137.

Graham-Smith, D. (2012). Weekend project-build your own supercomputer (pc and tech authority). Accessed: 2019-06-06. Récupéré de https://www.alphr.com/features/374980/build-your-own-supercomputer

Gupta, S. et Sadoghi, M. (2018). Blockchain transaction processing. http://dx.doi.org/10.1007/978-3-319-63962-8_333-1

Haffey, M., Arlitt, M. et Williamson, C. (2018). Department of Computer Science, University of Calgary-network heartbeat traffic characterization. Récupéré de https://pages.cpsc.ucalgary.ca/~carey/talks/Heartbeats-CLW.pdf

Han, J. et Kamber, M. (2006). *Data Mining Concept and Techniques*.

Heinzelman, W., P. Chandrakasan, A. et Balakrishnan, H. (2002). An application-specific protocol architecture for wireless micro-sensor networks. *Wireless Communications, IEEE Transactions on*, *1*, 660 – 670. http://dx.doi.org/10.1109/TWC.2002.804190

Heo, J. W., Dorri, A. et Jurdak, R. (2022a). Multi-level distributed caching on the blockchain for storage optimisation. In Proceedings of the 2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC).

Heo, J. W., Ramachandran, G., Dorri, A. et Jurdak, R. (2022b). Blockchain storage optimisation with multi-level distributed caching. IEEE Transactions on Network and Service Management, 19(4).

Huang, Z. (1998). Extensions to the k-means algorithm for clustering large data sets with categorical values. *Data Min. Knowl. Discov.*, *2*, 283–304. http://dx.doi.org/10.1023/A:1009769707641

Huh, S., Cho, S. et Kim, S. (2017). Managing iot devices using blockchain platform. 464–467. http://dx.doi.org/10.23919/ICACT.2017.7890132

Hwang, K., Fox, G. et Dongarra, J. (2010). *Distributed Computing: Cluster, Grids and Clouds (Chapter 3: Clustered Systems for Massive Parallelism)*. Accessed: 2019-03-22. Récupéré de https://pdfs.semanticscholar.org/bc62/5258e06df87eb96b76ce2279997fbedb17c2.pdf

Jurdak, R., Kanhere, S., Dorri, A., Malik, S., Steger, M., Oham, C., Dedeoglu, V., Gupta, P., Hill, A., Dharma Putra, G. et Jha, S. (2017). CSIRO- blockchain for iot security and privacy. Accessed: 2018-10-30. Récupéré de https://research.csiro.au/dss/blockchain-iot-security-privacy

Kalis, R. et Belloum, A. (2018). Validating data integrity with blockchain. 272–277.

http://dx.doi.org/10.1109/CloudCom2018.2018.00060

Karimi, H., Medhati, O., Zabolzadeh, H., Eftekhari, A., Rezaei, F., Dehno, S. et jamalpoor, A. (2015). Implementing a reliable, fault tolerance and secure framework in the wireless sensor-actuator networks for events reporting. *Procedia Computer Science*, *73*. http://dx.doi.org/10.1016/j.procs.2015.12.007

Kosba, A., Miller, A., Shi, E., Wen, Z. et Papamanthou, C. (2016). Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. 839–858. http://dx.doi.org/10.1109/SP.2016.55

Kousaridas, A., Falangitis, S., Magdalinos, P., Alonistioti, N. et Dillinger, M. (2015). Systas: Density-based algorithm for clusters discovery in wireless networks. 2126–2131. http://dx.doi.org/10.1109/PIMRC.2015.7343649

Lunardi, R., Michelin, R., Nunes, H., Neu, C., Zorzo, A. et Kanhere, S. (2022). Consensus algorithms on appendable-block blockchains: Impact and security analysis. *Mobile Networks and Applications*, *27*. http://dx.doi.org/10.1007/s11036-022-02015-4

Lunardi, R. C., Alharby, M., Nunes, H. C., A. F. Zorzo, C. D. et v. Moorsel, A. (2020). Context-based consensus for appendable-block blockchains. 401–408. http://dx.doi.org/10.1109/Blockchain50366.2020.00058

Mackenzie, B., Bellekens, X. et Ferguson, I. (2018). An assessment of blockchain consensus protocols for the internet of things. http://dx.doi.org/10.1109/IINTEC.2018.8695298

Motwani, R. et Raghavan, P. (1995). *Randomized Algorithms*. Accessed: 2022-05-25. Récupéré de https://rajsain.files.wordpress.com/2013/11/randomized-algorithms-motwani-and-raghavan.pdf

Nakamoto, S. (2009). Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*.

Oktian, Y., Lee, S. et Lee, H. (2020). Hierarchical multi-blockchain architecture for scalable internet of things environment. *Electronics*, *9*, 1050. http://dx.doi.org/10.3390/electronics9061050

Pajooh, H. H., Rashid, M., Alam, F. et Demidenko, S. (2021). Multi-layer blockchain-based security architecture for internet of things. *Sensors*, *21*, 772. http://dx.doi.org/10.3390/s21030772

Panigrahi, L. et Panigrahi, R. (2024). *An Enhancement in K-means Algorithm for Automatic Ultrasound Image Segmentation*, (p. 1–8).

Prabhu, K. et Prabhu, K. (2017). Converging blockchain technology with the internet of things. *International Education and Research Journal*, *[S.l.], v. 3, n. 2*.

Qi, L., Tian, J., Chai, M. et Cai, H. (2023). A cooperative pow and incentive mechanism for blockchain in edge computing. *IEEE Internet of Things Journal*, *PP*, 1–1. http://dx.doi.org/10.1109/JIOT.2023.3278314

Raza, S., Abdelraheem, M. A. et Sedrati, A. (2017). Blockchain and iot: Mind the gap.

Rebello, G. A. F., Camilo, G. F., Guimarães, L. C. B., de Souza, L. A. C. et Duarte, O. C. M. B. (2022). Security and performance analysis of quorum-based blockchain consensus protocols. 1–7. `http://dx.doi.org/10.1109/CSNet56116.2022.9955597`

Rüsch, S. (2018). High-performance consensus mechanisms for blockchains. Accessed: 2021-01-21. Récupéré de `https://www.ibr.cs.tu-bs.de/users/ruesch/papers/eurodw18-ruesch-final.pdf`

Sağırlar, G., Carminati, B., Ferrari, E., Sheehan, J. et Ragnoli, E. (2018). Hybrid-iot: Hybrid blockchain architecture for internet of things - pow sub-blockchains. 1007–1016. `http://dx.doi.org/10.1109/Cybermatics_2018.2018.00189`

Schiller, E., Rafati niya, S., Surbeck, T. et Stiller, B. (2019). Scalable transport mechanisms for blockchain iot applications. `http://dx.doi.org/10.1109/LCNSymposium47956.2019.9000673`

Singh, S. et Duggal, S. (2022).

Sousa, J., Bessani, A. et Vukolić, M. (2018). A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. 51–58. `http://dx.doi.org/10.1109/DSN.2018.00018`

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F. et Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. Dans *SIGCOMM'01*, 149–160.

Tanwar, S., Gupta, N., Tomar, J., Kumar, K., Treľová, S. et Ivanochko, I. (2023). A review on blockchain smart contract applications. 175–187.

Tashman, A. (2024). *Clustering with K-Means*, (p. 289–306).

Tumas, V., Rivera, S., Magoni, D. et State, R. (2023). Federated byzantine agreement protocol robustness to targeted network attacks. 443–449. `http://dx.doi.org/10.1109/ISCC58397.2023.10217935`

Valenta, M. et Sandner, P. (2017). Comparison of ethereum, hyperledger fabric and corda. Accessed: 2020-10-30. Récupéré de `http://explore-ip.com/2017_Comparison-of-Ethereum-Hyperledger-Corda.pdf`

Vukolić, M. (2016). The quest for scalable blockchain fabric: Proof-of-work vs. bft replication. 112–125. `http://dx.doi.org/10.1007/978-3-319-39028-4_9`

Wang, X., Zha, X., Ni, W., Liu, R., Guo, Y., Niu, X. et Zheng, K. (2019). Survey on blockchain for internet of things. *Computer Communications*, *136*. `http://dx.doi.org/10.1016/j.comcom.2019.01.006`

Wilkinson, S., Boshevski, T., Brandoff, J., Prestwich, J., Hall, G., Gerbes, P., Hutchins, P. et Pollard, C. (2016). Storj- a peer-to-peer cloud storage network.

Yildirim, S. (2020). Top machine learning algorithms for clustering. Accessed: 2020-05-17. Récupéré de https://towardsdatascience.com/top-machine-learning-algorithms-for-clustering-a09c6771805

Yu, Y. (2023a). Analysis of pow in bitcoin and pos in peercoin. *Highlights in Science, Engineering and Technology*, *39*, 784–788. http://dx.doi.org/10.54097/hset.v39i.6645

Yu, Y. (2023b). Analysis of pow in bitcoin and pos in peercoin. *Highlights in Science, Engineering and Technology*, *39*, 784–788. http://dx.doi.org/10.54097/hset.v39i.6645