PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIAS DA COMPUTAÇÃO

HENRY CABRAL NUNES

SMART CONTRACTS FOR APPENDABLE-BLOCKS BLOCKCHAIN

Porto Alegre
2020

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

SMART CONTRACTS FOR APPENDABLE-BLOCKS BLOCKCHAIN

HENRY CABRAL NUNES

Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Avelino Francisco Zorzo

Porto Alegre
2020

Ficha Catalográfica

N972s Nunes, Henry Cabral

Smart contracts for appendable-blocks blockchain / Henry Cabral Nunes . – 2020.
60.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Avelino Francisco Zorzo.

1. Blockchain. 2. Smart Contract. 3. Speedychain. 4. Appendable-blocks blockchain. I. Zorzo, Avelino Francisco. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS
com os dados fornecidos pelo(a) autor(a).
Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

Student's Henry Cabral Nunes

Smart contracts for appendable-blocks blockchain

This Dissertationhas been submitted in partial fulfillment of the requirements for the degree of Master of Computer Science, of the Graduate Program in Computer Science, School of Technology of the Pontifícia Universidade Católica do Rio Grande do Sul.

Sanctioned on March 30th, 2020.

COMMITTEE MEMBERS:

Prof. Dr. Rodrigo da Rosa Righi (Unisinos)

Prof. Dr. Felipe Meneguzzi (PUCRS)

Prof. Dr. Avelino Francisco Zorzo (PPGCC/PUCRS - Advisor)

SMART CONTRACTS FOR APPENDABLE-BLOCKS BLOCKCHAIN

RESUMO

Blockchain possui algumas características únicas, como a possibilidade de descentralização em ambientes não confiáveis, auditabilidade e segurança, citando apenas algumas. Algumas blockchains podem até permitir a execução *smart contracts*, que são programas que podem ser executados de uma maneira distribuída e descentralizada. Eles têm a grande vantagem de permitir estender os benefícios da blockchain para qualquer tipo de aplicação. Contudo, esse conceito não tem sido empregado em todo seu potencial devido a desafios associados a complexidade computacional e latência. Esses desafios estão associados tanto ao uso de *smart contracts*, quanto ao da blockchain. Algumas soluções foram desenvolvidas com o objetivo de mitigar esses problemas. Uma importante solução proposta na área de blockchain é o desenvolvimento da *appendable-block blockchain*. Este tipo de blockchain tem o potencial de reduzir problemas de latência e escalabilidade através da possibilidade de adição de dados de forma paralela na blockchain. Contudo, atualmente, este modelo não possui a possibilidade de executar *smart contracts*. Além disso, modelos tradicionais para a execução de smart contracts não são compatíveis com a *appendable-block blockchain*. Neste trabalho, nós apresentamos uma solução para essa falha. Nós introduzimos um modelo para a execução de *smart contracts*, que nós chamamos de *context-based model*. Este modelo além de permitir a execução de *smart contracts* na *appendable-block blockchain* permite que sejam aproveitados os benefícios de inserções paralelas desse modelo de blockchain. Isto incrementa a escalabilidade, porque permite a execução de *smart contracts* em paralelo. Essa melhora é comprovada por uma prova de conceito implementada neste trabalho, onde uma análise de performance foi efetuada comparando com execução sequencial de *smart contracts*.

Palavras-Chave: Blockchain, Smart Contract, Speedychain, Appendable-blocks blockchain.

SMART CONTRACTS FOR APPENDABLE-BLOCKS BLOCKCHAIN

ABSTRACT

Blockchain has some unique characteristics, such as decentralization in an untrusted environment, auditability, and security, just to cite a few. Some blockchains can even provide smart contracts , which is the ability to execute code in a distributed manner. This has a tremendous potential to extend the blockchain benefits to any type of application. However, this concept has not been fully exercised due to the associated challenges of high computational complexity and latency. Those challenges are both associated with smart contracts and blockchain. Some solutions have been developed to try to mitigate those problems. One important work on the blockchain side is the development of the appendable-block blockchain. This type of blockchain has the potential to reduce latency and scalability problems by allowing the parallel addition of data into the blockchain. However, currently, this model lacks the possibility to execute smart contracts. Furthermore, traditional models for smart contracts are not compatible with the appendable-block blockchain. In this work, we present a solution for this shortfall by introducing a model for smart contract execution, which we call the context-based model. This model not only allows appendable-block blockchain to execute smart contract, but also benefits from the parallel nature of the appendable-block blockchain. This increases scalability, by allowing the execution of smart contracts in parallel. Those claims are backed up by a proof of concept implemented in this work where the performance is compared to the sequential execution of smart contracts.

Keywords: Blockchain, Smart Contract, Speedychain, Appendable-blocks blockchain.

LIST OF FIGURES

CONTENTS

1. INTRODUCTION

In 2008 an unknown author, using the pseudonym of Satoshi Nakamoto, proposed a decentralized digital currency, known as Bitcoin [30]. Until then, despite previous attempts, like Bitgold [37], no such digital currency was viable from a technological perspective. The Bitcoin was the first cryptocurrency to allow untrusting and unknown parties in a peer-to-peer (P2P) network to advance a system state without the need for a trusted third party. In this specific case, the system was a digital currency, where users can make transactions between them, and the creation of new currency was determined in a pre-agreed way. The removal of a trusted third party is a significant development because, during transactions, the other parties involved would be at the mercy of the third party, and frequently there was a payment for their services. This and other such cryptocurrencies possess an important relevance in the financial world. As an example, Bitcoin today accounts for a nearly 140-billion-dollar market [8].

The novelty of Bitcoin came from the use of a blockchain, which works like the technological infrastructure that made it and other cryptocurrencies viable. This way, cryptocurrencies are just one application of the blockchain, and the industry and other players can use this infrastructure in other applications like the Internet of Things (IoT) [17], smart grids [35], resource management [19], healthcare [5] [6], supply chain [4] as some examples.

Another remarkable possibility that was integrated into a blockchain is smart contracts, which consist of Turing-complete programs that can be stored in a blockchain and processed by it. As with any blockchain application, it does not need any centralization. This characteristic means that once a user deploys a smart contract in a blockchain, the blockchain will process the smart contract according to the program logic independent of the creator's will. Additionally, the user or any other member of the blockchain cannot remove the smart contract. This property is called self-enforcing and can be useful in specific applications [17] [40].

However, there are several challenges in the use of blockchains. Those challenges consist predominantly of security issues. Some examples are (i) majority attacks where if a malicious user has enough computing power, it can insert false information in a blockchain; or, (ii) performance issues, as scalability caused by the increase in stored space utilized by the blockchain [12]. Smart contracts have specific problems, like exploitation of smart contracts logic, privacy and scalability [29] [2].

To solve or mitigate those challenges, academia and industry presented a number of proposals. A particular solution relevant to this work is the Speedychain [23] blockchain, which works with a model that we call in this document appendable-block blockchain. This model operates in a different fashion to the traditional blockchain model by allowing the insertion of information in a concurrent way instead of a serial way. This trait can help to

minimize the scalability problem. The focus of Speedychain is to work in IoT settings, but the benefits can expand to other applications.

Although the diverse areas that industry can apply Speedychain to, it does not support yet smart contracts. In this work, we present the Context-based Smart Contracts model, which enables the use of smart contracts on any appendable-block blockchain, such as, Speedychain. This solution is novel because there is no smart contract model capable of working on an appendable-block blockchain. Moreover, the nature of the appendable-block blockchain of concurrent insertion of transactions can bring benefits for smart contracts working in specific scenarios. Additionally, we create a formalization of this model and the appendable-block blockchain using mathematical functions and set theory.

Another contribution of this work is the implementation of the model. For that, an Ethereum Virtual Machine (EVM) [9] was introduced to process the smart contract logic in SpeedyChain. EVM is a virtual machine designed to process smart contracts in the Ethereum blockchain [9], one of the leading blockchains today, which has smart contract capacity [11]. Using the same virtual machine can bring compatibility between the smart contract of both blockchains, which would allow Speedychain to enjoy part of the Ethereum development ecosystem and to facilitate further development.

To evaluate our model, we performed an experiment that consists of a simulation of multiple devices making computation on GPS points using smart contracts stored in Speedychain. The smart contracts keep information about multiple GPS points and calculate the distance between them when requested. The environment was simulated in Core Emulator[1] using ten gateways. Different scenarios with different transaction' loads were created for the experiment. In each one of those scenarios, multiple metrics related to performance were tracked. The results are discussed and show that in the experiment performed the performance for a parallel insertion was significantly better than sequential.

The rest of this document is organized as follows. In Chapter 2 we present the technical background, which includes blockchain, smart contracts, appendable-block blockchain. Chapter 3 we discuss the model, giving an overview, and then presenting a formalization for it. Chapter 4 presents details about the implementation, including the original planning, the realized implementation, and a brief security analysis. In Chapter 5 we discuss our objective with the experiment, tools, metrics utilized, and discuss the results collected from the experiment. In the following chapter, Chapter 6, we present what has been developed in the Smart Contracts area, the works are more general than focused on smart contracts model because of a lack of new proposal in with this specific objective. Finally, Chapter 7 we conclude our work, including further works.

2. BACKGROUND

This chapter presents the concepts and technologies necessary to understand the Context-based smart contracts model for appendable-block blockchain. First, some mathematical notations that will be used posteriorly are presented. The second section is on the traditional blockchain, how it works, the data structure, and other necessary aspects. The next part is on smart contracts, more focused on smart contracts stored on-chain, like the Ethereum blockchain smart contracts. Finally, the third section is about appendable-block blockchain and its implementation SpeedyChain, describing its difference to traditional blockchains.

2.1 Mathematical Notation

The functions that are used throughout this dissertation are the following:

- Function p is used to extract element e from a tuple using a lambda function as presented in equation 2.1:

$$p_e(tuple) = (\lambda(T_1, ..., T_e, ...T_n) \rightarrow T_e) \tag{2.1}$$

  As an example for (2.1), considering $t = (1, 6, 3)$ the operations $p_1(t), p_2(t)$ and $p_3(t)$ will result in $1, 6$ and 3 respectively.

- Function $H(x)$ is a hash function that can receive any sequence of bits x as input and output another sequence of bits [20]. The specific $H(x)$ function used here will be abstracted. Proprieties of a good hash function, as collision-free, pseudo-randomness and unpredictability will just be assumed as true.

- The PK function receives a digital signature as input and returns the public key from an asymmetric cryptography scheme. For this work, a cryptography scheme in which it is possible to recover a public key directly from a digital signature is used [15].

2.2 Blockchain

A blockchain is a distributed ledger that instead of storing just the final state after an operation in a system, it stores all transactions that brought the system to the current state. To assert if a state is correct, the blockchain can review its history [12] [30]. This approach has the burden of storing all transactions, which can build up to take a lot of

storage since it can save transactions from many years of operation. However, this permits to audit the current state of the blockchain and the system at any time. The distributed means that the system will work based on a Peer-to-Peer (p2p) network, in which each node in the blockchain will maintain a local copy of the history or, in other words, a copy of the blockchain.

The system state progresses when a node adds a new transaction. Although nodes do not have to trust or know each other, the system can progress. This progress is possible because of a consensus algorithm, which we will explain better in Section 2.2.2. A consensus algorithm enforces all nodes to obey a pre-agreed logic. Any node, save from specific situations, can trust that all the information added to the history in the network is valid and respects the pre-agreed logic, despite any mistrust among nodes.

2.2.1 Data structure

The data structure used to store the blockchain is a key component of its architecture. The first element is the block. It stores a group of transactions. In Figure 2.1, the basic structure of a block is presented, where it is divided into two parts: the block header containing metadata about the block; and the block data containing a group of transactions. Depending on the blockchain it may have different metadata fields, but normally those fields include:

- Block parent hash.
- Merkle tree root hash, a hash representing a tree of hashes from all the transactions, to guarantee the integrity of all the transactions in that block against tampering.
- Consensus algorithm fields that are used for the consensus (see Section 2.2.2).
- Timestamp field, a date and time representing the block creation time.

The block has a determined number of transactions that it can store, so multiple blocks are necessary to store all the history.

The second structure is a linked list, as shown in Figure 2.2. The nodes of the list are the blocks explained before. Although instead of using a pointer referencing the previous block, the reference uses a hash from the metadata of the previous block. The field that holds this reference is the block parent hash field mentioned before. In the implementation of a blockchain, most of the time, a key-value database to store this structure is used. The reasons are to have a persistent storage of the blockchain and to have a quick way to access the father block. From any block, you can find the father key by consulting the block metadata.

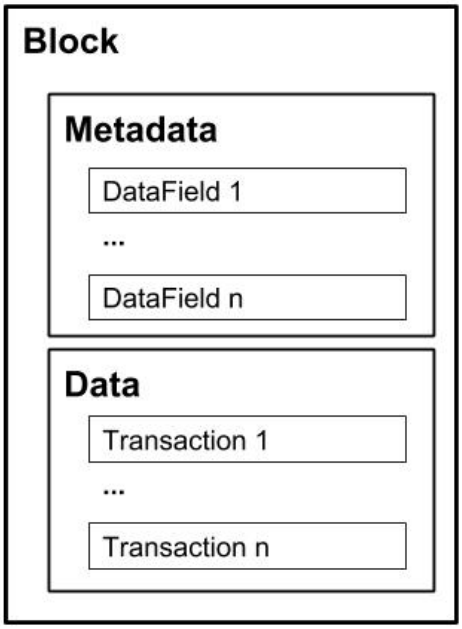

Figure 2.1 – Block structure

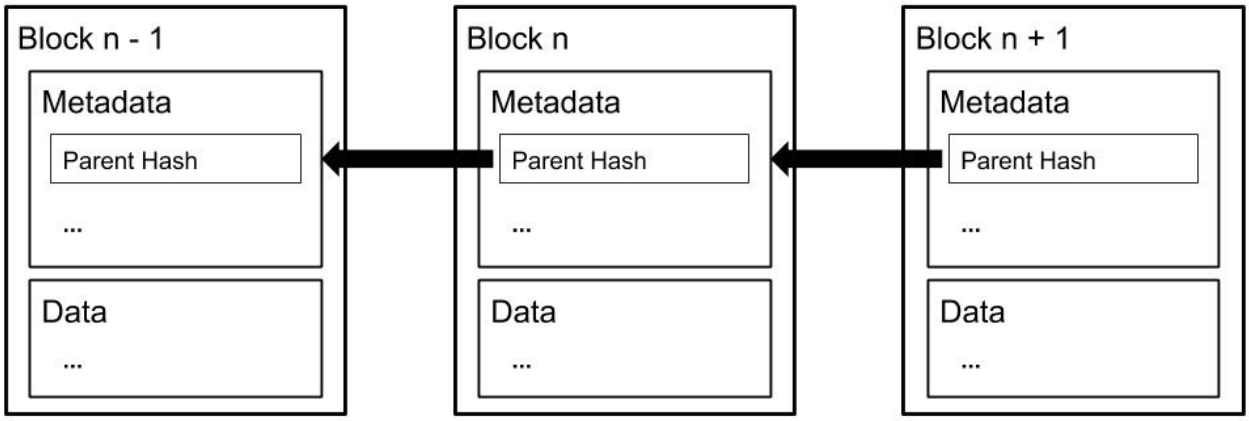

Figure 2.2 – Linked list structure

An essential aspect of the blockchain is the use of hashes to reference the previous block. This feature helps to guarantee the integrity of the chain. If a malicious node tries to change one of the blocks, inside the blockchain, that would alter its header hash, and as a consequence, all the following blocks would lose the reference to it. Since all the transaction hashes are used in the Merkle tree root field, any change to any transaction would bring changes to the blockchain, allowing any honest node to detect tampering.

2.2.2 Consensus algorithm

As the nodes add new data, they will create a new block, add transactions, and using the consensus algorithm, try to add it to the blockchain. Those transactions come from a transaction pool shared between all nodes. They can add new transactions to it representing operations in the blockchain.

The consensus algorithm is vital to the blockchain, as it guarantees that only valid blocks with valid transactions are added to the blockchain, ensuring the existence of one unique, valid state [17]. The first of such algorithms created was **Proof of Work (PoW)**, proposed along with the Bitcoin cryptocurrency. The idea behind is a computational puzzle, which is easy to check if correct but hard to solve. If a node wants to add a block to the blockchain, it needs to solve the PoW puzzle that is influenced by the current blockchain state and the transactions that it intends to add to the blockchain. When resolved, the node can send its block to the other nodes. Each node will then check individually if the puzzle is correctly solved; if not, then it will discard that block; if it is correct, then it checks if the block is correctly built and if the transactions are valid, following the pre-agreed logic; if all is correct, it adds the block to its own copy of the blockchain. Note that since it is easy to check if the puzzle is correctly solved, it is easy to discard invalid blocks without the need to check if the transactions are valid. Additionally, the puzzle difficulty is adjustable, so in the pre-agreement, it can be set how hard it is to solve the problem.

There is a competition between nodes to see which one solves the puzzle first and adds a block because the addition of a new block changes the blockchain state, so all the other nodes will need to start the puzzle again with a different new block. Furthermore, the block each node is trying to insert does not necessarily contain the same transactions like the one the other nodes are trying to add.

The puzzle used in PoW is to find a hash value in a specific range using the proposed block header as input. The difficulty is related to a particular metadata field in the block called **nbits**, which represents an integer number. That number is a number of zeros at the start of the hash that is necessary to accept it as valid. This field value is part of the pre-agreement. As an example, a difficult of 8 accepts a hash as 0x00f51C2A as valid. Since the block header currently has just fixed values, if the header hash is not valid, it would be impossible to insert the block. To solve that, there is another integer field called **nonce**, which is used to discover a valid hash value. The node is free to change the value of this field and use it to try to find a valid hash value, changing it successively until a valid header hash is found. This process is presented in the flow chart in Figure 2.3.

It is important that nodes in PoW use the same hash function; this way, the hash results will be the same in every node. The hash function is used for this because of its pseudorandom nature, which means that the same input will always yield the same output,

but a different input will generate a completely different value, even if only one bit is changed. In our case, the input is the block header, but it is impossible to preview the result of a hash function without executing it. This way, to create a valid block, a node needs to perform the hash function changing the nonce a random number of times until a correct value is found. For a node to validate a block, it just needs to execute the hash function once.

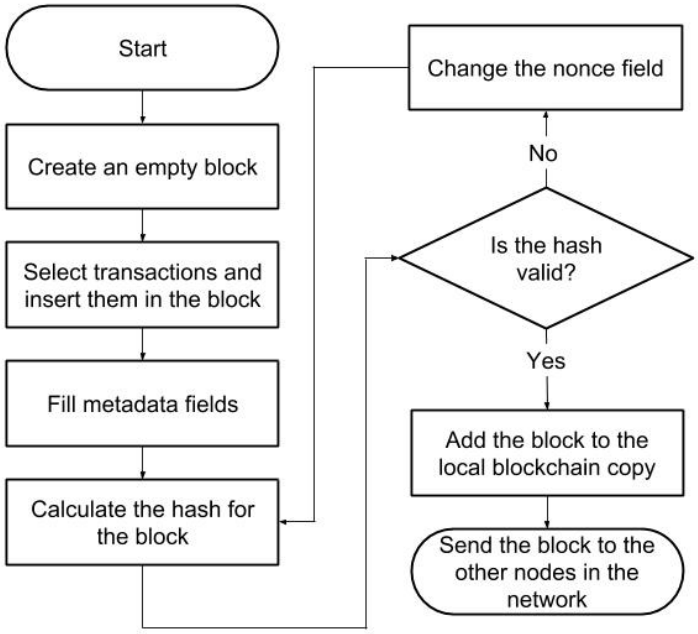

Figure 2.3 – PoW flowchart

After PoW, more consensus algorithms came with different characteristics, such as Proof-of-Stake [32] and Proof-of-Luck [27]. For example, **the Practical Byzantine Fault Tolerance (PBFT)** gives some nodes the right to vote in the consensus algorithm. Those nodes, in this case, know each other, and for the addition of new blocks, at least 2/3 of them must be online and working correctly. When a new node joins the blockchain, it will not receive the voting power automatically, it will need to pass by a particular protocol for the nodes of the network to decide if that node will receive voting power or not. To add a new block, an election will take place, which we will call a round, in it, each node will vote for another node, the one with most votes will be the leader in that round. The leader can propose a new block for the other nodes. They will check if that new block and its transactions are valid; if they are, then the new block will be added to each node blockchain. Otherwise, the block is discarded, and the leader sometimes will receive a penalty. This algorithm works better with a small number of nodes because the election of a leader in each round is communication-intensive, and the latency from multiple nodes can make it impractical.

2.2.3 Blockchain network

The communication between nodes in the blockchain uses a Peer-to-Peer (P2P) network, in which each node has a communication channel with one or more nodes, and often the communication is broadcast to the entire network. There are two types of nodes: (i) full nodes that have a full copy of the stored blockchain. Depending on the consensus algorithm and its permissions, it can participate in the creation of new blocks. This type of node is always present in the network; and, (ii) light nodes that are not present in all blockchains and are not supported in some. They do not download all blockchain, sometimes having just the headers of the blocks or having only the transactions that they have interest in.

Once a block is created, it is broadcast to all P2P network, so nodes can check whether the block is valid before adding it to the blockchain. If a node is just joining the network or was disconnected and needs to update its blockchain for the last created blocks, it can inquiry other full nodes for the current blockchain situation. If required, the inquired node can send the missing blocks for the inquiring node to update its blockchain copy. For this kind of communication, it is not necessary to use broadcast.

New transactions are broadcast across the network. The nodes that receive them will add those transactions to a transactions pool. When a node is creating a new block, it will take transactions from the transaction pool to add to the block. The node can choose what transactions it wants to add to the block it is creating, but generally, it will check if the transaction is valid before trying to create a block. Once a transaction is added to a block, all nodes will remove it from their pools.

In most solutions, an external agent will want to communicate with the blockchain. For example, a cryptocurrency user going to make a transaction or an application wanting to add data to the blockchain. In those cases, it is common to have an interface to a full node where an application can require the node to create a transaction on its behalf. In Ethereum, for example, this operation is performed by the nodes exposing an API known as Web3 [9], which allows them to request the creation of transactions and inquiry info about the current blockchain state.

2.3 Smart contracts

Smart contract as a concept was created by Nick Szabo in 1994 [17]. In his idea, smart contracts were programmable contracts in which its terms are executed automatically without the need for a centralized party. An example proposed by Szabo is where a house can be sold and all the lockers of the house automatically change to attend the new owner,

additionally, all the legal proceedings would be made without the need of a notary office. This notion can be extended to other problems, like a Service Level Agreement [28] between parties, where in case of violation of one of the terms the contract would execute itself and apply a fine to the violator. The important aspect in those cases is that the execution of the contract would be automatic and without a central controller. From a practical perspective, that means that no legal proceedings would be necessary because the contracts would be capable of self-enforcing rules without the need of a central controller blocking or changing its execution. Although, at the time of conception of the term no technology was capable to support smart contracts, with the rise of the blockchain the concept became feasible.

As explained in Section 2.2, a blockchain works based on a P2P network [24], where the execution of the pre-agreed logic is followed by all nodes and made reliable by the consensus algorithm. It is possible to make the blockchain behave like a computer, where the transactions represent a program routine execution. All nodes to do that must know all the commands and behavior expected from that computer. The nodes set this shared knowledge on the pre-agreed logic. Such as in a computer, it has the flexibility to run different programs for different purposes. In this context, those programs are known as smart contracts.

A smart contract is a program as powerful as any other Turing-complete program. However, the blockchain network stores it in its data structure. That allows the blockchain to execute it without the need of a central controller [17] [29]. There are multiple models for smart contracts implementation in blockchains, although a few fundamental properties are common, like the immutability of the appended information, the auditability of the computation and generated information, the need for a consensus logic, and the possibility of self-enforcing terms. Self-enforcing means that once a condition specified in a smart contract is met, the consequences will be processed. As most blockchains have a cryptocurrency, the consequences can involve monetary transactions.

There are multiple models for smart contracts implementation, one of these models is the one used by Ethereum. In it, the blockchain stores smart contracts as special transactions. Those transactions are bytecode that can be processed by the Ethereum Virtual Machine (EVM). Each node in the Ethereum network has an EVM within it. Calls, like a program call, for a smart contract, are appended to the blockchain as a transaction. The transaction contains inside of it the bytecode, representing the program call, to be processed in the context of a specific smart contract. This bytecode is inputted in the virtual machine with the current state of the smart contract. This process will generate a change in the smart contract state, which will be saved by the node. Any new calls for that smart contract are processed using the new state. During the consensus algorithm, like any transaction, all the nodes need to check if the call to the smart contract is valid. This verification is performed using their EVM, if the call is invalid, the block is rejected [40].

Throughout this work, a generic virtual machine, as a function *VM*, is used, which works similarly to the EVM [9], *i.e.*, *VM*(*State*, *Data*) = *State*′. *VM* receives two inputs: *Data*, which is the bytecode with operations for the virtual machine; and, *State*, which is a pointer to a data structure that contains a state for the virtual machine and multiple smart contracts. The output for the *VM* function is a reference to new state *State*′, based on the modifications that the *Data* incurred. Different states are stored in a Merkle Patricia Trie [11]. A new *State* with no modifications is refereed as the constant *newState*.

Despite the idea of smart contracts working like contracts between parties, it is used as a backend for applications. The difference of not needing a central authority gave this kind of application the name of Decentralized Applications (dApps) [9]. The concept includes applications that are not capable of existing using other technologies. An example is a social media network where there is no central authority controlling the influx of information in it.

There are a few important challenges in the use of smart contracts. Those include security aspects, like, all the program logic is known by other parties. In Ethereum, as an example, all bytecode is stored in transactions in the blockchain, which any node can consult. This means that a smart contract with security vulnerabilities is noticeable by any node that can exploit it. It is aggravated by the fact that once deployed, a smart contract cannot be removed from the blockchain. For instance, Atzei *et al.* [2] explore some identified types of exploits that can be used in smart contracts. There are performance issues, compared to a centralized solution, for example, smart contracts will yield less throughput and have higher latency. Another consideration is regarding privacy aspects since all information is available to all nodes, which can be problematic to some applications [17].

Taking those issues into account, it is possible to aggregate value in specific solutions. In Section 6, we discuss a few applications that have been investigated with this technology and proposals to minimize (or solve) the presented problems, which, if solved or mitigated, can expand the area of application of smart contracts.

2.4 Appendable-block blockchain

Appendable-block blockchain, and its implementation SpeedyChain [23], came from the context of IoT, where the devices have low computing power and storage capacity. This scenario limits the ability to use blockchain. The reason is the necessity to store the blockchain in the nodes and the computing power needed for some consensus algorithms such as PoW. Furthermore, high communication latency is a critical factor in some IoT applications, which makes the use of blockchain not feasible. To mitigate those problems, SpeedyChain improves the traditional blockchain in two aspects: one from the network architecture standpoint and other from the blockchain data structure.

To work around the IoT devices' low processing and storage capacity, SpeedyChain uses gateways. Those gateways are special devices with more processing power and storage capacity than IoT devices. They work like full nodes and receive connections from IoT devices, which will input and request data to/from the blockchain. This characteristic unburdens the devices from the necessity to process and to store the blockchain. Furthermore, to improve their performance, it uses the PBFT consensus algorithm, which requires less processing power from the gateways than the PoW consensus algorithm.

The data structure in immutable-block blockchains, as explained in Section 2.2 creates new blocks with transactions, which can be appended only by the introduction of new blocks. In addition, to expand the blockchain, all the nodes need to work on the last block of the blockchain to create a new block to connect to it. SpeedyChain uses a new method for the addition of transactions. Instead of creating new blocks, it expands existing blocks attaching new transactions to it. The blockchain structure, in this case, is as presented in Figure 2.4, where each IoT device connected to a gateway has an exclusive block. In those blocks, the corresponding device can append new transactions by expanding the block at any time. This potentially improves the blockchain performance and latency, because working this way the nodes can work appending transactions, in this case with data, in multiple different blocks in parallel instead of working on the same one.

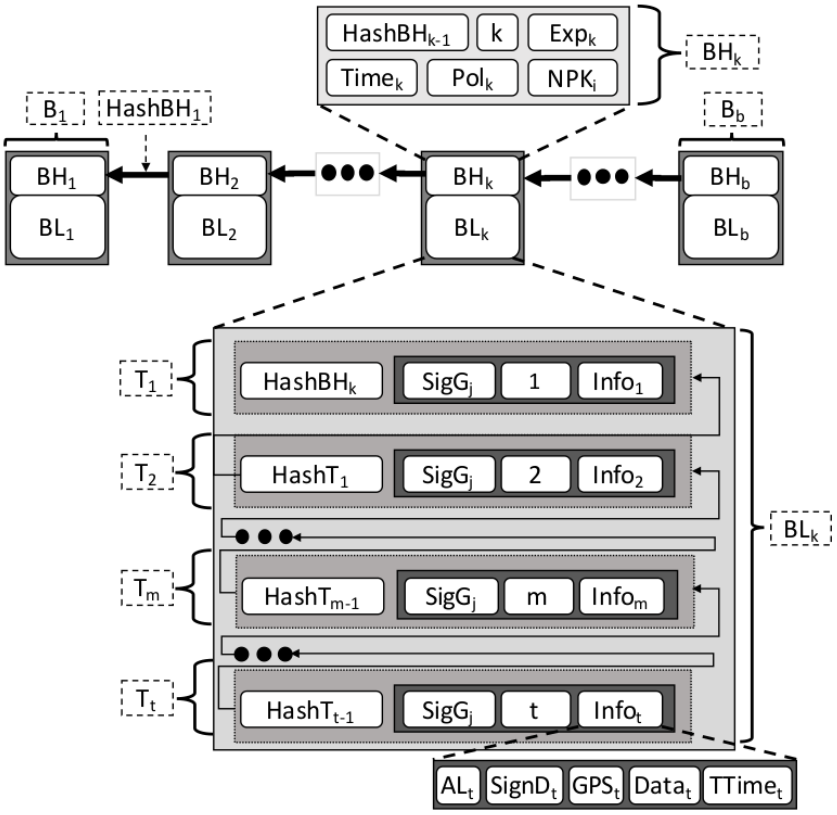

Figure 2.4 – Speedychain block structure *source [23]*

This work also formalizes the appendable-block blockchain model. The appendable-block blockchains has a set of n nodes $N = \{N_1, ..., N_n\}$. Nodes are gateways and IoT

devices that generate data. Each $\{N_i\}$ has a pair of public/secret keys (NPK_i, NSK_i) respectively from an asymmetric cryptography scheme, where the public keys are accessible to every participant of the blockchain[22].

The data structure in an appendable-block blockchain is a non-empty set *BC* of blocks. Each block is a tuple (*BH*, *BL*). *BL*, named block ledger, is a set of transactions that can be incremented as necessary and linked to a *BH*. The *BH*, named block header, is another tuple composed of (*ParentHash*, NPK_i, *T*) with meta-data about the block. Those fields are:

- *ParentHash* is the result of *H*(*BH*) of the block inserted in the blockchain before this one. It works as a pointer to the previous block.
- NPK_i is the public key of a member of the set *N*, only one block can have a specific NPK_i. To enforce that the $\nexists\, x \mid x,\ b \in BC \wedge p_2(p_1(x)) = p_2(p_1(b))$ post-condition must be respected. The node that has the NSK_i to the NPK_i of a block is said to be the block owner, and only that node can append new transactions to that block.
- *T* is the first transaction inserted in a block and the only one to be part of the block header. Furthermore, this transaction is the first transaction signed by a pair of public/secret keys, the public key is the NPK_i value.

The *BC* set will form a hash-linked list of blocks *B* connected by their *ParentHash* in the *BH*. Two post-conditions need to be obeyed to keep the *BC* as a linked list:

- $\nexists.x \mid x,\ y,\ q \in BC \wedge H(p_1(x)) = p_1(p_1(y)) \wedge H(p_1(x)) = p_1(p_1(q))$, which guarantees that each block can be pointed by at most one other block.
- $\nexists.x \mid x,\ y \in BC \wedge\ p_1(p_1(x)) = \emptyset \wedge p_1(p_1(y)) = \emptyset$, which restraints to the existence of just one block that points to no other block with the *ParentHash*, this block is called the genesis block and is always the first block existing in the *BC*.

A transaction withholds data generated by the nodes. The data content depends on the application and context. In the appendable-block blockchain, a transaction is represented as a tuple of (*Data*, *PT*, *Sig*), where: *Data* is specific to the node generating data through the creation of the transaction; *PT* is the hash of the previous transaction inserted in the block, it works as a hash-link connecting the transactions in the block. If it is the first transaction in the block, then it will refer to the hash of the *BH*; and *Sig* is a digital signature from the node originating the *Data* [22].

Before presenting how appendable-block blockchain adds new blocks and appends transactions, we present the auxiliary functions *newBlock* as shown in Equation 2.2, which summarizes the creation of a new block filling the necessary fields, and function *lastBlock* (Equation 2.3) , which returns a block that has no other block with the *ParentHash* in the header pointing to it. In practice, it is the last block created in the blockchain.

$$newBlock(T, BC) = ((H(p_1(lastBlock(BC))), PK(p_3(T)), T), \{\}) \tag{2.2}$$

$$lastBlock(BC) = x | x \in BC \wedge (\nexists y \in BC \wedge p_1(p_1(y)) = H(p_1(x)) \wedge x \neq y) \tag{2.3}$$

To add a new block to the blockchain, function $addBlock(BC, T)$ (Equation 2.4) is used. It creates a new block for a node and appends the new block to the blockchain if there is no other block with the same NPK as the transaction signature public key $PK(p_3(T))$. The predicate $uniqueBlock$ (Equation 2.5) guarantees this requirement.

$$addBlock(BC, T) = \begin{cases} BC \cup newBlock(T, BC), \text{if } uniqueBlock \\ BC, \text{otherwise} \end{cases} \tag{2.4}$$

$$uniqueBlock = \nexists x.x \in BC \wedge p_2(p_1(x)) = PK(p_3(T)) \tag{2.5}$$

New transactions are generated by nodes with new $Data$ to be inserted in the blockchain. This operation is performed only if the node's public key (NPK_i) is present in a block header BH. Function $appendTransaction$ (Equation 2.6) specify the insertion of a new transaction T in a block B that has a public key equal to the public key used in the transaction signature.

$$appendTransaction(BC, T) = \begin{cases} (BC - B) + updateBlock(B, T), \text{if } p_2(T) = PreTHash(B) \\ BC, \text{otherwise} \end{cases} \tag{2.6}$$

$updateBlock$ (Equation 2.7) is an auxiliary function to generate an updated block where transaction T is appended. Function $PreTHash$ (Equation 2.8) returns the hash of a previous transaction appended to the block or the block B header hash. This hash will be used to check if a transaction is pointing to the PT field of the last transaction inserted to the block.

$$updateBlock(B, T) = (p_1(B), p_2(B) \cup T) \tag{2.7}$$

$$PreTHash(B) = \begin{cases} H(p_1(B)), \text{ if } |p_2(B)| = 0 \\ H(lastT(B)), \text{otherwise} \end{cases} \tag{2.8}$$

Algorithm 2.1 shows how the main operation works on this model for the insertion of new transactions in a continuous way. The *mempool* consists of a set of transactions submitted to the blockchain by multiple nodes, but not yet appended to the blockchain, this *mempool* is shared by all nodes. Function *poll* (line 5), on the *mempool*, returns one transaction of the set. Before processing a new transaction, it checks if a block with the public key of the signer exists through function *exists* (line 6); if not, a new block is processed by the consensus algorithm (line 7), and if approved, a new block is inserted and broadcast to the network (lines 9 and 10). Otherwise, the proposed transaction is processed by the consensus algorithm (line 13). If accepted, it is appended to the block owned by the transaction signer (more details about this algorithm is described in [22]).

Algorithm 2.1 Main operation for Appendable-block Blockchain

```
 1: Input: mempool, BC //Original state
 2: Result: BC //Updated state
 3:
 4: while while(True) do
 5:    T = poll(MemPool)
 6:    if !exists(PK(p3(T))) then
 7:       ConsensusResponse = performConsensus(B)
 8:       if ConsensusResponse then
 9:          broadcast(B)
10:          BC = addBlock(BC, T)
11:       end if
12:    else
13:       ConsensusResponse = performConsensus(T)
14:       if consensusResponse then
15:          broadcast(T)
16:          BC = appendTransaction(BC, T)
17:       end if
18:    end if
19: end while
```

3. PROPOSED MODEL OR SMART CONTRACTS

This chapter, using the concepts discussed in the Background chapter, presents the model for context-based smart contracts used with appendable-block blockchain. First, a high-level overview of the model is introduced in Section 3.1. Then a formalization of the model using *settheory* and mathematical *functions* is introduced in Section 3.2.

3.1 Overview

In several immutable-block blockchains, such as Ethereum, a global state is pointed by each block of the blockchain. This global state works as a snapshot and, carries all smart contracts stored in the blockchain at the global state creation time. Each transaction inside a block contains a bytecode for a virtual machine, as presented in Section 2.3. During the addition of new blocks, the block's transactions through their bytecodes can change or create new smart contracts — thus altering the global state to a new global state. This new global state is saved and a pointer for it in the new block that is generated. To store the global state a data structure, such as the Merkle Patricia Tree, is used. In Figure 3.1, we present a sequence of blocks from an immutable-block blockchain. The original state σ, pointed by the first block, is updated to σ', σ'' and σ''' respectively. Due to the immutability of the already inserted blocks in the blockchain, the previous states, inserted in each previous block, will never change.

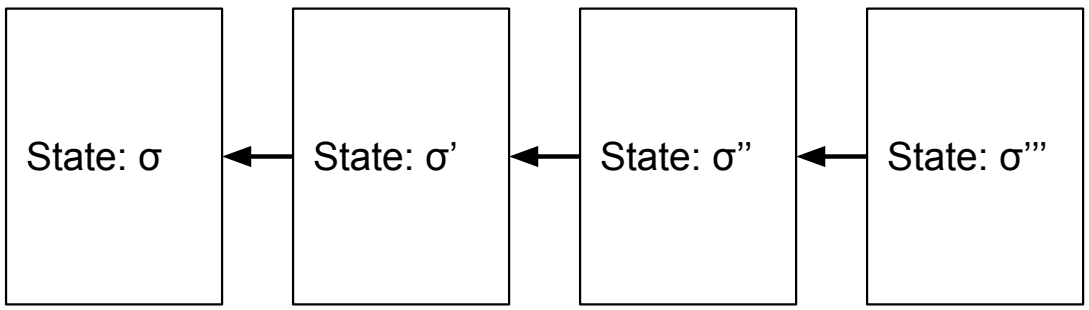

Figure 3.1 – Sequence of the state progress in an Immutable-block blockchain

However, in appendable-block blockchain, this model is no longer feasible because of the mutability of the blocks. Using the same strategy in appendable-block blockchain, an already existing block storing transactions would give its global state for the following block to use and update to a new global state. However, this existing block can be updated in the appendable-block blockchain. If any transaction is appended to this existing block the

global state of the existing block would change. As a consequence, all blocks created after the existing block would need to change to take into account the global state of the existing block. Because of the data structure of the blockchain, it would mean to destroy all the blocks created after the existing blocks and recreate them to consider this state change. This would greatly impact performance. First, because a reprocessing of all blocks created after the changed block may be computational expansive. Second because the blockchain would be unstable, *i.e.* already inserted transactions could be removed at any time that a previously created block had a transaction appended.

From the inability to use this model to improve the appendable-block blockchain with the smart contracts feature, we developed the Context-based model. In Figure 3.2, we show the two directions that new data can be inserted in an appendable-block blockchain, in the horizontal by the addition of new blocks and in the vertical by the addition of new transactions in a previously created block. In this model, each block can hold a state. Although, unlike the immutable-block blockchain, these states are updated as new transactions are inserted. Figure 3.3 shows how this model updates the state. The block header withholds a reference for a starting state, and each subsequent transaction has an updated new state. This updated new state is the result of processing the previous state (which belongs to the previous transaction or the block header) and a bytecode present in a new transaction. A transaction, when inserted in the block, needs to be modified to store the pointer to its state. Therefore, we will call it henceforth a *committedtransaction*. This nomenclature is to differentiate from the original transaction sent by a user.

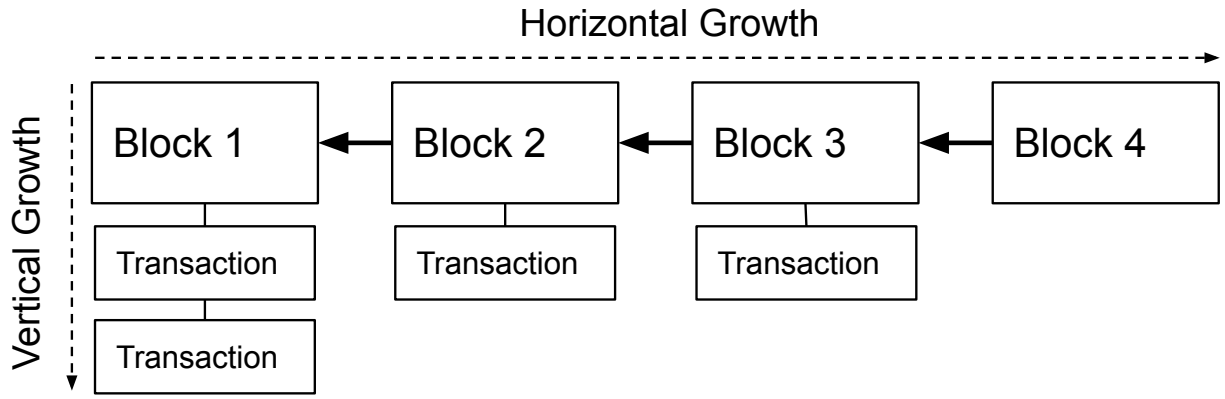

Figure 3.2 – Directions of growth for an appendable-block blockchain

Since each block has its own state that is updated when new transactions are committed to the block, the interaction with the state of other blocks is limited. In other words, a state belonging to a block is isolated from a state stored in another block. Figure 3.4 shows this concept of isolated states, in each block a sequence of states (σ_0, σ'_0, σ''_0, σ'''_0), (σ_1, σ'_1, σ''_1, σ'''_1) and (σ_2, σ'_2, σ''_2, σ'''_2) is stored. We named those isolated states in a block as context. This context can hold the data for multiple smart contracts and their data. Although,

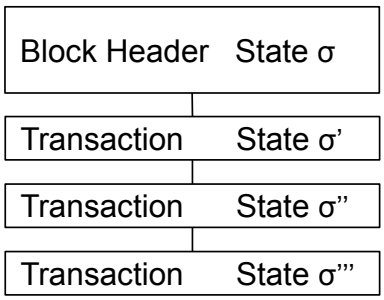

Figure 3.3 – State progress into a block of an appendable block blockchain

as a consequence of the isolation, a smart contract belonging to one context cannot interact with a smart contract in another context. For example, if a smart contract A exists in Context 0 (see Figure 3.4) and a smart contract B exists in Context 1 they cannot interact.

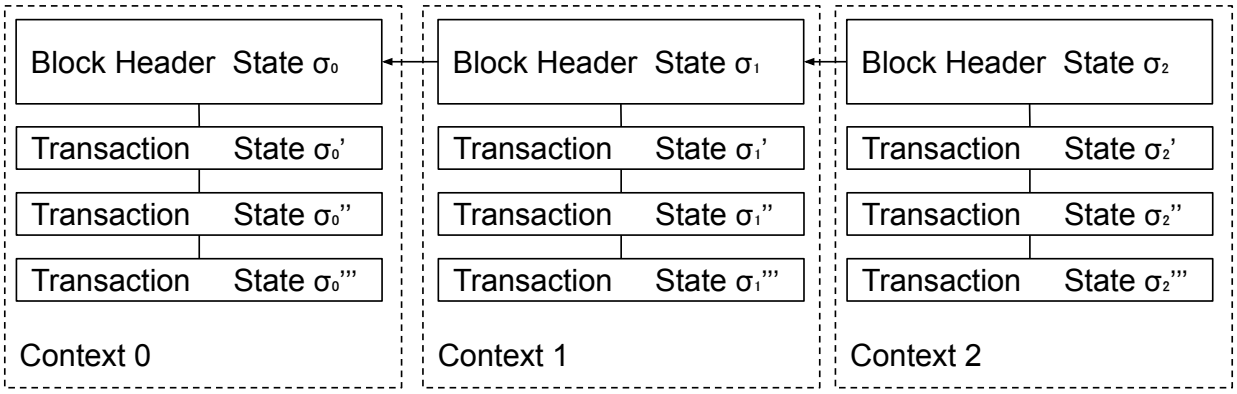

Figure 3.4 – Multiple contexts in the appendable-block blockchain

Finally, this model is designed to still support the previous block from the appendable-block blockchain that just holds data. This decision is to allow fast transactions insertion. This is faster because there is no necessity to process bytecode in a VM and to consult the global state. Hence, we named a block with a state reference as a 'block with context' and the block that just holds data as a 'pure-data block'. Another difference between them is that the block with context can receive transactions from any device in the network, unlike the pure-data block that works as specified in Section 2.4, where each block can have data inserted by just the same device. In Figure 3.5 we summarize the architecture for this model.

3.2 Formalization

All the functions in an appendable-block blockchain work exactly as previously presented in Section 2.4, unless stated otherwise. The elements of the tuple (*BH*, *BL*) representing the blockchain are different. *BH* is a tuple (*ParentHash*, *Index*, *NPK*, *CommittedTransaction*)

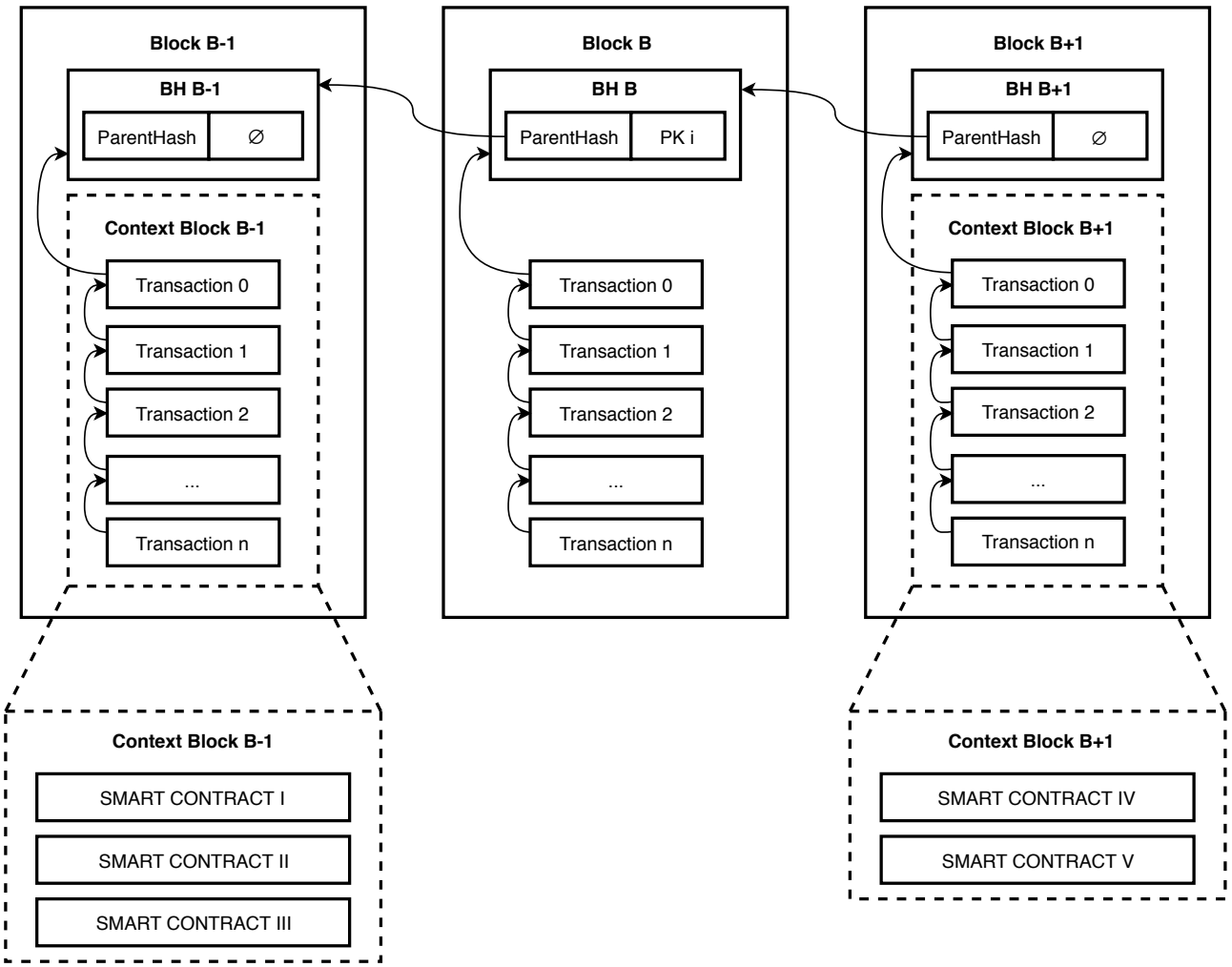

Figure 3.5 – Smart contracts and Block context. Source: Nunes *et al.*[31]

where *ParentHash* and *NPK* work as in the appendable-block architecture, although when the block has a context the value of *NPK* will be equal to $\emptyset$, because there is no block owner or restriction of devices who can operate in a block. The *Index* field is a natural number whose value corresponds to the order of blocks created in the blockchain. The *CommittedTransaction* field stores a **committed transaction**, it is different from a **transaction**. A node can check if a block B has a context by using function *HasContext* (Equation 3.1), and can get a specific block by the index using function *getBlock* (Equation 3.2).

$$HasContext(B) = \begin{cases} true & \text{if } P_3(P_1(B)) = \emptyset \\ false & \text{if } P_3(P_1(B)) \neq \emptyset \end{cases} \tag{3.1}$$

$$getBlock(Index) = x | x \in BC \wedge P_2(x) = Index \tag{3.2}$$

A **committed transaction** is defined as a tuple ($Data$, Sig, PT, $BlockState$), and it is originated from a transaction that was processed by a node. The tuple fields are:

- *Data* is a binary sequence that depending on the block type will be treated differently. If it is a pure-data block, then *Data* represents data generated by a node, which will not be processed as *bytecode*. If it is a block with a block state, then it is a *bytecode* that will be inserted as an input in the *VM* function.
- *Sig* is the digital signature of the transaction that generated this committed transaction.
- *PT* is the hash value of the previous inserted committed transaction or the block header if it is the second inserted committed transaction, then the value will be the hash of *BH*.
- *BlockState* is a pointer to a data structure holding the block state, as the Merkle Patricia Trie. The last committed transaction in the block has the most updated state, its value is generated by *VM*.

When a *Data* transaction includes bytecode to a block with context, it will be processed by function *VM* (Equations 3.3). For that, *Data* and *BlockState* of the last inserted committed transaction in the block are inserted as input in the *VM* function, the resulting pointer to a new state will be attached to a new committed transaction in the *BlockState* field. S' is a pointer to the resulting state, S is the original BlockState and *NewS* is an empty BlockState.

$$VM(S, Data) = \begin{cases} S' & \text{if } S \neq \emptyset \\ VM(NewS, Data) & \text{if } S = \emptyset \end{cases} \tag{3.3}$$

A node that wants to operate on the blockchain will create a transaction for that operation. The transaction is composed of (*Data*, *ToBlock*, *Sig*, *PT*, *OPCode*), where the fields previously described in committed transactions are the same, the *ToBlock* represents the destiny block where this transaction is to be processed. If the *ToBlock* value is equal to $\emptyset$ and *OPCode* is a specific value, then the transaction intention is to create a new block with a context or a pure-data block. *OPCode* is an integer that represents a code for the transaction intention, where 1 means the transaction is to create a new pure-data block, 2 means the transaction is to create a new block with a context, and 3 means it is a transaction to be appended to a block.

Two functions will be used to summarize the block creation, when *OPCode* is 1 or 2: Function *NewCBlock* (Equation 3.4), which creates a new block with a context starting from a *newS* state; and, function *NewPDBlock* (Equation 3.6), which creates a new pure-data block.

$$newCBlock(T, BC) = ((H(p_1(IB(BC))), p_2(p_1(IB(BC))) + 1, \emptyset, NCT_C), \{\}) \tag{3.4}$$

$$NCT_C = (P_1(T), p_3(T), p_4(T), VM(\emptyset, P_1(T))) \tag{3.5}$$

$$newPDBlock(T, BC) = ((H(p_1(lB(BC))), p_2(p_1(lB(BC))) + 1,\ PK(p_3(T)), NCT_{PD}), \{\}) \tag{3.6}$$

$$NCT_{PD} = (P_1(T), p_3(T), p_4(T), \emptyset) \tag{3.7}$$

A transaction that has *OPCode* 3 will be appended to a block. However, the appended transaction is treated differently if the intention is to append a transaction in a pure-data block or a block with context. The CT_C function (Equation 3.8) creates a committed transaction to be appended in a block with a context. On the other hand, function CT_{PD} (Equation 3.9) will create a transaction to a pure-data block.

$$CT_C(T, B) = \begin{cases} (P_1(T), p_3(T), p_4(T), VM(p_4(lastCT(B)))), \\ \text{if } p_4(T) = preCTH(B) \\ \\ B, \text{otherwise} \end{cases} \tag{3.8}$$

$$CT_{PD}(T, B) = \begin{cases} (P_1(T), p_3(T), p_4(T), \emptyset, P_1(T))) \\ \text{if } p_4(T) = preCTH(B) \\ \\ B, \text{otherwise} \end{cases} \tag{3.9}$$

$$preCTH(B) = \begin{cases} H(p_1(B)), \text{ if } |p_2(B)| = 0 \\ H(lastCT(B)), \text{otherwise} \end{cases} \tag{3.10}$$

$$lastCT(B) = x \mid x \in p_2(B) \wedge (\nexists y | y \in p_3(B) \wedge p_3(y) = H(x)) \tag{3.11}$$

The functions in Equations 3.8 and 3.9 are used in function *AppendT* (Equation 3.12), which directs a transaction to the correct function type by their *OPCode* and updates *BC*.

$$AppendT(BC, T) = \begin{cases} (BC - B) \cup CT_C(T, B) \\ \text{if } HasContext(B) = true \\ \\ (BC - B) \cup CT_{PD}(T, B) \\ \text{if } p_3(B) = PK(p_3(T)) \\ \\ BC, \text{ otherwise} \end{cases} \tag{3.12}$$

$$B = getBlock(P_2(T)) \tag{3.13}$$

The algorithm for the main operation for this model is presented in Algorithm 3.1. In the algorithm, *memPool* works exactly like presented in Section 2.4. Line 6 checks if the transaction being processed wants to append a new transaction to the blockchain (*OPCode* 3) and if the destination block exists. If both are true, then the transaction is processed by the consensus algorithm and appended using the *appendT* function (line 10). When the destination block does not exist, then a transaction creates a new block with a context (line 11). It checks whether the *OPCode* value is equal to 2 and if the destination block is equal to *emptyset*; if both conditions are true, then a new block will be created (line 15) after begin processed by the consensus algorithm. Finally (line 16), the algorithm checks if the transaction is to create a new pure-data block (*OPCode* 1). If so, then if there is no other block with the same public key as the signature, then it proceeds to create a new pure-data block (line 20).

Algorithm 3.1 Main operation for appendable-block blockchain with context-based model

```
 1: Input: memPool, BC //Original state
 2: Result: BC //Updated state
 3:
 4: while True do
 5:    T = pull(memPool)
 6:    if exists(p2(T)) AND p4(T) = 3 then
 7:       ConsensusResponse = performConsensus(T)
 8:       if consensusResponse then
 9:          broadcast(T)
10:          BC = appendT(BC, T)
11:       else if (p2(T) = ∅ AND p4(T) = 2 ) then
12:          ConsensusResponse = performConsensus(B)
13:          if ConsensusResponse then
14:             broadcast(B)
15:             BC = newCBlock(T, BC)
16:          end if
17:       else if !exists(PK(p3(T))) AND p4(T) = 1 then
18:          ConsensusResponse = performConsensus(B)
19:          if ConsensusResponse then
20:             broadcast(B)
21:             BC = newPDBlock(T, BC)
22:          end if
23:       end if
24:    end if
25: end while
```

3.3 Application of the context-based model in immutable-block blockchain

In this chapter, we discussed the application of the context-based model for the appendable-block blockchain. The model was designed as an improvement for the appendable-block blockchain. However, it is possible to apply it in the immutable-block blockchain if the blockchain provides support to execute smart contracts.

For that, the immutable-block blockchain should use its smart contract feature to emulate the appendable-block blockchain. This is feasible because a blockchain with the smart contract feature is a Turing complete machine. Therefore, it can run any program, including the appendable-block blockchain itself. The context-based model can be then applied in the emulated blockchain. It is important to note that the emulation of a blockchain on top of another blockchain would generate a great overhead hindering the performance and probably nullifying any gain in performance from using the context-based model for appendable-block blockchain.

4. PROOF OF CONCEPT

In this chapter, we present the designed and developed implementation of the model presented in Section 3. First, we discuss the planned implementation, which includes objectives and decisions. After that, we present the realized implementation. Lastly, we make a brief analysis of the security concerns and attack vectors that our implementation is susceptible to.

4.1 Planned implementation

For the implementation of the model, three main objectives were set:

- Maintainability: Changes to the model or the implementations should be easy to achieve and modify.
- Simplicity: The implementation should be as simply as possible, nonetheless assuring that the model is correctly implemented.
- Reuse of code and standards: Prioritize the use of already existing solutions, technologies, and code. On top of that, to use standards already provided by other similar solutions.

Using those objectives as a guide, we came with the following decisions. The use of SpeedyChain, discussed in Section 2.4, as the blockchain platform. As a consequence, our goal implementation should consist of extending the SpeedyChain capabilities by adding smart contracts based on the described model. It is a natural decision because SpeedyChain already uses the appendable-block blockchain model. Moreover, using it would avoid the necessity to develop a new blockchain.

The choice of which virtual machine to process the smart contracts was decided from two possibilities: i) to develop a new virtual machine from scratch or, ii) to use an already existing virtual machine used in other blockchain solutions. Using an already existing virtual machine would allow to reuse a standard of an already established blockchain, for example. Additionally, that would allow us to utilize the code of the virtual machine already implemented. Thus, we decided to use an already existing virtual machine.

The selected virtual machine implementation chosen was the Ethereum Virtual Machine (EVM) [11] used in the Geth client. This client is the official client of the Ethereum Foundation [9], which is the primary maintainer of the Ethereum blockchain. Using this as the virtual machine in our project would allow to use part of the Ethereum tools and ecosystem. For example, the Solidity language used for the development of smart contracts could

be used in our project. Additionally, using the EVM from the Geth client would make it easy to maintain and follow the established standard for the EVM. This is because the Ethereum Foundation would implement any alteration in the EVM specification to their client. That modification would reflect in our solution, *i.e.* ou solution would also be updated. Moreover, the EVM implementation includes a Merkle Patricia Trie, that could be used to store states and smart contracts in our project.

The major drawback was a conflict in the programming languages that is used in the Geth client and SpeedyChain. SpeedyChain is implemented in Python, while the Geth client is implemented in Go Lang. Because of that, it was not possible to integrate the EVM in the SpeedyChain directly. To solve that, it was necessary to separate them in two different processes and use inter-process communication between them. One process running SpeedyChain, and another running the EVM. This design decision exposed our solution to some attacks that will be discussed in Section 4.3. Even though this decision seems to go against one of our objectives (simplicity), we decided that the benefits of using the Geth client EVM outweighs this conflict. We also considered another implementation of the Ethereum EVM, which was developed in Python and if used would allow to integrate everything in a single process. However, the Python implementation was badly documented, incomplete or not supported anymore.

4.2 Actual implementation

The structure of the resulting implementation is presented in Figure 4.1. Three new components are inserted in the SpeedyChain framework: Interface EVM (1), which is an interface in SpeedyChain used for communication; an inter-process communication protocol (2); and, an internal EVM (3).

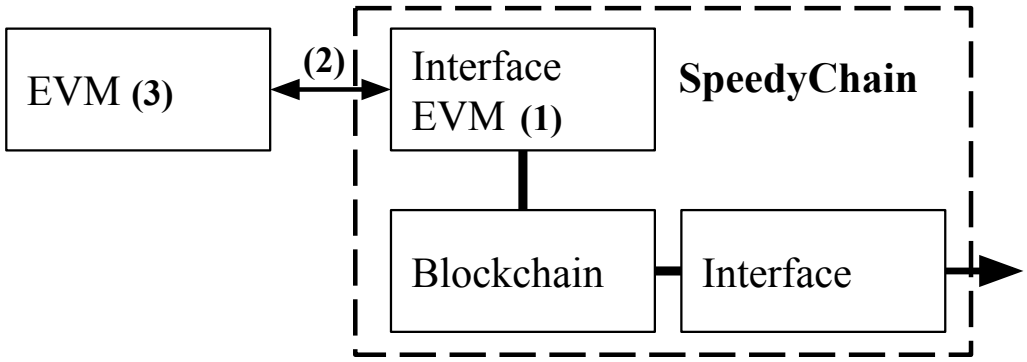

Figure 4.1 – EVM and SpeedyChain

The process to run a smart contract starts with a new proposed transaction, which for this explanation contains a bytecode. We assume that all fields of the transaction and the

blockchain are correct to receive this generic transaction. As presented in the model, see Chapter 3, this transaction targets an existing block. From this block, the last state pointer is extracted and together with the bytecode it is passed to the Interface EVM (1) inside the SpeedyChain.

Interface EVM wraps both datas in a JavaScript Object Notation (JSON) format, which is then sent through a socket to the EVM (3). After sending the JSON, it awaits a return from the EVM with a result. The EVM receives the JSON with the state and bytecode. It changes the virtual machine state to the state received and using the bytecode as input processes the request, which yields a resulting bytecode and an updated state. Both are wrapped in a new JSON object and sent as a response to Interface EVM. An error message is sent, if there is any problem when running the smart contract.

The results are unwrapped by the Interface EVM and handed to the blockchain, which will use the updated state to create a committed transaction as described in Chapter 3. If an error is returned, then the transaction is discarded, and no alteration is applied to the blockchain.

4.3 Security analysis

Although several vulnerabilities might be exploited our implementation, this was not the major concern since we intended to provide a proof of concept for the context based smart contracts ideas. Despite of that we briefly discuss these vulnerabilities in this section.

First, our solution uses the EVM, which has many known vulnerabilities. Those vulnerabilities are cited in Chapter 2 when we discuss Atzei *et al.* work [2] on smart contracts vulnerabilities. Our implementation is vulnerable to all the presented problems in their work. It is hard to mitigate those vulnerabilities in our solution. The reason is that they are interconnected to the use of the EVM.

Other vulnerabilities are related to the blockchain. For example, the SpeedyChain can work with different consensus algorithms; most of those have intrinsic vulnerabilities. The proof of work, for instance, is vulnerable to the 51% attack. Thus our implementation keeps the same vulnerabilities. Additionally, our implementation is vulnerable to a replay attack. A replay attack is when a malicious party copies a transaction and resend it to the blockchain to have it processed again. Since the first transaction is already signed, the transaction owner does not have to sign the duplicated transaction. As this implementation is not for a production environment, we did not solve the problem.

Finally, another vulnerability is in the use of the socket in the connection between the EVM and the Interface EVM, step (2) in Figure 4.1. This open the possibility for exploitation and information leakage. A few examples are: a malicious person can intercept

information about the blockchain state and processing, this can be used to facilitate a front-run attack [2]. The connection can be redirected from the original EVM to an impostor EVM.

5. EXPERIMENT

To evaluate our model we developed an experiment described in this chapter. We start by giving an overview of the experiment, followed by a detailed explanation of smart contracts. Next we describe the five different scenarios used in the experiment, the metrics utilized and finally the hardware that was simulated.

5.1 Overview

In this experiment, we simulate a real situation where data from GPS sensors of multiple IoT devices are stored and processed in the blockchain. The main objective of the experiment is to answer our research question, which is **"How appendable-block blockchain with smart contracts can impact the performance and scalability of applications?"**. In the experiment, each device will have one smart contract stored in a single block in the blockchain. This smart contract can receive GPS positions and store them in a list, to simulate a route. The traveled distance will be calculated by the smart contract automatically as new positions are inserted. Operations on the smart contract, as the distance between two positions or the list of positions, are available too.

The experiment is performed in different scenarios using the following parameters:

- Smart contract operations: the number of operations performed in the smart contract. We consider an operation when a device sends a transaction to add a new GPS position in the smart contract.

- Pure-data block operations: device operations in which a transaction with a random input is sent to the blockchain, the input is stored as data in the device pure-data block in the blockchain.

- Operation mode: if the smart contract operations are performed in sequential or parallel order.

All scenarios were executed using a virtualized environment using a network simulator. During the experiment multiple metrics were collected. Afterwards, we present the results and analyze the obtained results.

5.2 Smart contracts

The smart contract for the GPS tracker was developed in Solidity [10]. It is a Javascript like programming language, specific for smart contracts development that transforms the high-level program in EVM bytecode. In this work, we present just the signature of exposed public methods from the smart contracts, although the full code is available at the Github repository [1].

A typical method signature in Solidity have the following format:

function functionName(*type var*1, *type*2 *var*2) *access returns*(*type var*3, *type var*4)

Where *functionName* is the name of the method, it is followed in parenthesis by the variables that are used as parameters in the method call. The *type* is an established variable type, such as *int*256 or *char*, and *var* is the name of the variable. The *access* specifies the type of access to the method. In this work, we present just methods with the access established as *public*. Finally, if there is any return value, they are specified after the *returns* keyword. In Solidity it is possible to have multiple *return* values with different types, for example, *type var*3 and *type var*4.

The method *addPosition* can be called to insert a new GPS position in the smart contract. The position will be stored in a list inside the smart contract.

function addPosition(*int*256 *latitude*, *int*256 *longitude*)*public*

The methods *getTotalPositions*, *getDistanceFromStart*, and *getTravelledDistance* are used to return values from the smart contracts. The values are respectively the number of positions inserted in the contract, the distance from the last position (or actual position) to the starting position (the first position inserted in the smart contract), and the total traveled distance between all positions inserted in the smart contract.

function getTotalPositions() *public view returns*(*uint*256)

function getDistanceFromStart() *public view returns*(*uint*256)

function getTravelledDistance() *public view returns*(*uint*256)

The last function presented is the *getPosition*, which allows requesting to return an inserted GPS position by specifying the index of the position.

[1] https://github.com/conseg/GPSTracker

function getPosition(*uint*$_p$*os*) *public view returns*(*int*256 $_l$*atitude*, *int*256 $_l$*ongitude*)

The smart contract is straightforward. However, there are limitations to the EVM that complicated its development. The main one that impacted our experiment is the absence of floating-point numbers. Since GPS position distance calculations involve decimal numbers some adjustments were necessary. The first one is the multiplication of all numbers inserted by 10,000, any value bellow 10,000 after the multiplication were treated as the decimals of a number. Other formulas and algorithms were adapted to consider this. The second adaptation is: in the absence of operation available to perform square root in the EVM, we used of the Babylonian square-root algorithm [16]. This method allows to perfomr the square root necessary to calculate the distance between two GPS points. The third adaptation is the use of an existing smart contract [2] that allows to use bits to calculate the cosine and sine of a degree when only integer operations are available. This is done by using a set of predefined values.

To calculate the distance between two GPS positions we used the Equirectangular approximation equation (see Equation 5.1). This is an approximation, as result can yield significant differences to the real distance. Other formulas more precise were available, such as the Haversine formula [34]. However, the use of the Haversine formula was not feasible. The main reason is the limitations caused by the absence of the floating-point system. That limitation made us unable to use some trigonometry operations, in special for this case the Arc-tangent. In the equation 5.1, φ and λ are latitude and longitude respectively, while φ_m is the average of the latitudes, and R is the earth Radius (6,371 km).

$$\begin{aligned} x &= \Delta\lambda \,.cos\, \varphi_m \\ y &= \Delta\varphi \\ d &= R \,.\sqrt{x^2 + y^2} \end{aligned} \tag{5.1}$$

The Equirectangular approximation is less precise near the earth poles and more precise near the bearing (0°, 0°). For this reason, in our experiment, we used coordinates near to bearing.

[2] https://github.com/Sikorkaio/sikorka/blob/master/contracts/trigonometry.sol

5.3 Evaluation

In order to evaluate context-based smart contracts in appendable-block blockchains, we created a virtual environment using a VM in the Virtual Box hypervisor. The VM configuration was a 4-core processor, 16GB of memory and 64MB of graphics memory running Ubuntu18.04 operating system. The hardware where the hypervisor was running on is a Macbook Pro with 2.3 GHz 8-Core Intel Core i9 processor, 32GB DDR4 memory. To create a container-based network and emulate network equipment, gateways, and devices, we used the Core Emulator [1]. The network, shown in Figure 5.1, consists of a central router (n6) connected to five other routers (n1, n2, n3, n4, n5). Each router, except the central one, had two gateways connected to them. Each gateway had multiple IoT devices to simulate the load. For ten devices, one block with context was created. In the block, an instance of the smart contract specified in Section 5.2 is stored. The device stores its generated GPS positions in it. In some scenarios we created pure-data blocks for the IoT devices, and we generated random data to insert in those blocks to simulate parallel operations on the blockchain. This evaluation is the same as the one presented in Nunes *et al.* [31] and similar to the one presented in Michelin *et al.* [26].

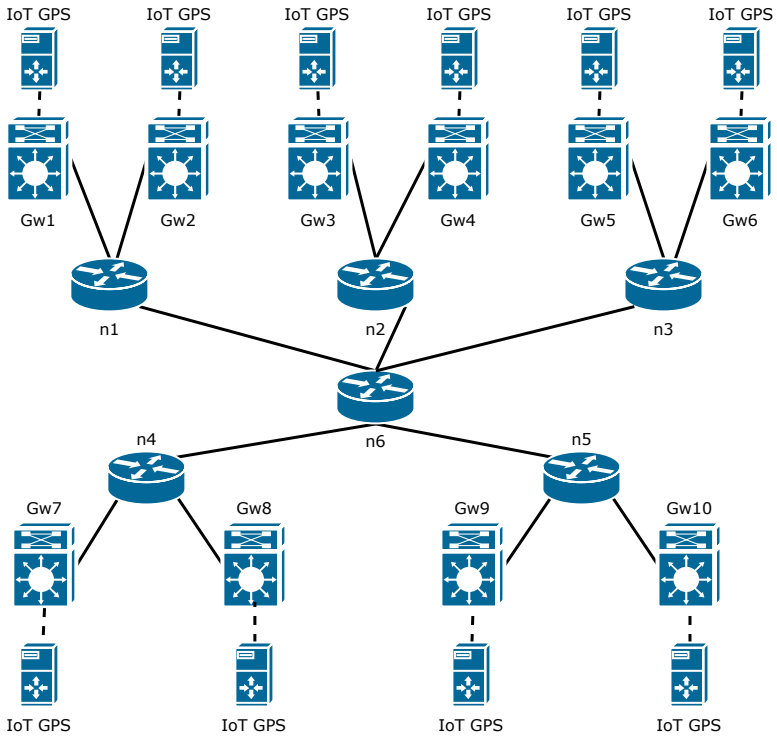

Figure 5.1 – Emulated gateways architecture

We evaluated our solution in four testing scenarios presented in Table 5.1. The used parameters were: sequential execution or parallel execution of the smart contracts operations (insertion of GPS position and distance calculation), and extra load of random data in pure-data blocks.

Table 5.1 – Evaluated scenarios

Scenario	Description
A	Sequential execution of 1,000 smart contracts without external load
B	Sequential execution of 1,000 smart contracts with an additional load of 50 devices per gateway and 100 transactions per device, *i.e.*, 5,000 transactions per gateway and 50,000 transactions in total
C	10 parallel context with 100 smart contracts transactions per context (1,000 in total) without external load
D	10 parallel context with 100 smart contracts transaction per context (1,000 in total) with an additional load of 50 devices per gateway and 100 transactions per device, *i.e.*, 5,000 transactions per gateway and 50,000 transactions in total

Scenario A was performed to establish a baseline for smart contract execution in appendable-block blockchains. Each device submitted one hundred transactions with operations. We simulated ten devices with sequential computation of smart contracts operations (total of 1,000 sequential transactions) all devices connected to the same gateway.

Scenario B performed the same number of sequential smart contracts computations as Scenario A, however, with an extra load of the normal type of transactions in every gateway. A normal transaction in this experiment is a random input of data in a pure-data block. The load was simulated through the insertion of 50 devices per gateway and 100 "normal" transactions produced by each device. In total, 50,000 additional "normal" transactions were simulated. These transactions were produced by 500 devices.

Scenario C used the proposed context-based smart contracts in appendable-block blockchains. To do that, this scenario considered ten devices connected to ten different gateways and requesting 100 smart contracts operations (1,000 operations, the same computations as in *A* and *B*) in parallel, without any extra load in the gateways.

Scenario D adopted context-based smart contracts, with the same number of transactions operations as C, but with the same extra load present in *B*.

During the execution of the scenarios, we captured several different metrics. We called those metrics T1, T2, T3, T4, T5, and T6:

- **T1**: Time to perform consensus, insert a new block (first time that device is connected) and replicate it to all gateways;
- **T2**: Time to add and replicate a device block to all gateways (after consensus);
- **T3**: Time to perform consensus, insert a new special block for smart contracts and update EVM with the new smart contract bytecode;

- **T4**: Time to insert a transaction in the blockchain;
- **T5**: Time to run a smart contract in the EVM and update the blockchain;
- **T6**: Time to run all smart contracts evaluated (1,000 contracts operations).

All metrics are represented by an average time of ten repetitions for each scenario. A confidence level of 95% was achieved.

5.4 Data

Table 5.2 presents a summary of the obtained results, organized by scenarios in the rows and metrics in the column. Values represent the average time in milliseconds. All used data are available at Github[3].

Table 5.2 – Results summary Source: Nunes *et al.*[31]

Scenario	**T1** (in milliseconds)	**T2** (in milliseconds)	**T3** (in milliseconds)	**T4** (in milliseconds)	**T5** (in milliseconds)
A	19.54 ± 0.24	46.53 ± 0.48	61.24 ± 0.65	0.97 ± 0.003	29.17 ± 0.03
B	90.98 ± 0.77	279.64 ± 2.53	305.22 ± 33.45	2.81 ± 0.008	64.18 ± 0.40
C	32.40 ± 1.43	85.59 ± 3.55	111.03 ± 4.45	1.20 ± 0.10	34.47 ± 0.10
D	102.67 ± 0.87	320.24 ± 2.93	340.03 ± 40.04	3.02 ± 0.01	79.58 ± 0.72

5.5 Analysis

We noted that context-based smart contracts execution can impact the consensus algorithms. In column (**T1**), in Table 5.2, we can see an increase in the time spent to perform consensus in scenarios without an extra load in the gateways. The increase is of $\approx$ 65% using context-based smart contracts (*scenario C*) when compared to a sequential smart contracts execution (*scenario A*). In the case of scenarios with the extra load (“‘with normal transactions"), we can observe that consensus time is increased by less than 13% when compared to the context-based approach in *Scenario D* with the sequential one in *Scenario B*. This can be explained by the time required by the gateways to process smart contracts, affecting the time to perform consensus.

The metric **T3**, which represents the time to replicate a context-based block to all gateways, and **T2**, which represents the time to replicate a pure-data block to all gateways, gave us an insight about the block type replication time. In all scenarios, the time required

[3]https://github.com/HenryNunes/Context-based-Smart-Contracts-ForAppendable-block-Blockchains.git

was higher than the average presented in **T3** than in **T2**. That can be explained by the extra load from the communication and processing performed in the EVM. Additionally, a lower difference was observed when comparing *Scenario D* to *Scenario B* ($\approx$ 11%), than comparing *Scenario C* to *Scenario A*($\approx$ 81%).

Time to insert transactions in the blockchain (**T4**) was less affected than block insertion by the proposed context-based smart contracts solution, as presented in Table 5.2. *Scenario C* increased $\approx$ 23% over *Scenario A*, and *Scenario D* increased $\approx$ 7.5% when compared to *Scenario B*. Comparing the usage of context-based smart contracts adoption in **T3** and **T4**, we can observe that the impact in mean time to insert transactions (**T4**) was much lower than in block insertion (**T3**).

One important measure is how much the processing of smart contracts is affected by individual executions (**T5**). We can observe that *Scenario C* increased in $\approx$ 18% over *Scenario A* and *Scenario D* increased in $\approx$ 23% over *Scenario B*. This shows that the parallel approach proposed in context-based smart contracts affect individual insertions. This can be explained by the larger number of messages exchanged by nodes. Nonetheless, 79.58ms (in the worst case) to receive the result of the computation is still very good considering the application, usually GPS devices have an update rate of one update per second [14].

Finally, as presented in Figure 5.2 from Nunes *et al.*[31], time to perform calls for every smart contract (**T6**) was reduced when using context-based smart contracts in parallel execution (*Scenarios C* and *D*) compared to "traditional" sequential execution (*Scenarios A* and *B*). Differently to the other metrics, **T6** is presented in seconds for each scenario. *Scenarios A* and B required an average of $31.22s \pm 0.43s$ and $70.40s \pm 0.97s$, respectively. Although, *Scenarios C* and *D* required only $3.46s \pm 0.12s$ and $6.96s \pm 0.48s$. While context-based approach (*Scenarios C* and *D*) presented an impact in the main operations (**T1**, **T2**, **T3**, **T4** and **T5**) of the blockchain, in **T6** required around 10% of the time to perform all smart contracts, when compared to the the sequential approach (*Scenarios A* and *B*).

Based on the presented results, we can answer our research question:

- **How appendable-block blockchain with smart contracts can impact the performance and scalability of applications?** The main advantage of scalability and performance of the context-based model is the possibility to insert blocks in parallel. In this work, we tried to measure if this benefit was real. The analysis presented of metrics **T1** through **T5** turns evident the increase in latency arising from the use of the context-based model in parallel, both in transaction insertion time and block insertion time. This can impact negatively several applications that depend on low latency for operations and turns the use of the context-based model not viable. Nonetheless, **T6** presents a significant decrease in total time to insert all transactions in the blockchain when inserting them in parallel using the context-based blockchain. This improvement has a great impact on many applications. Important to note that to have these benefits

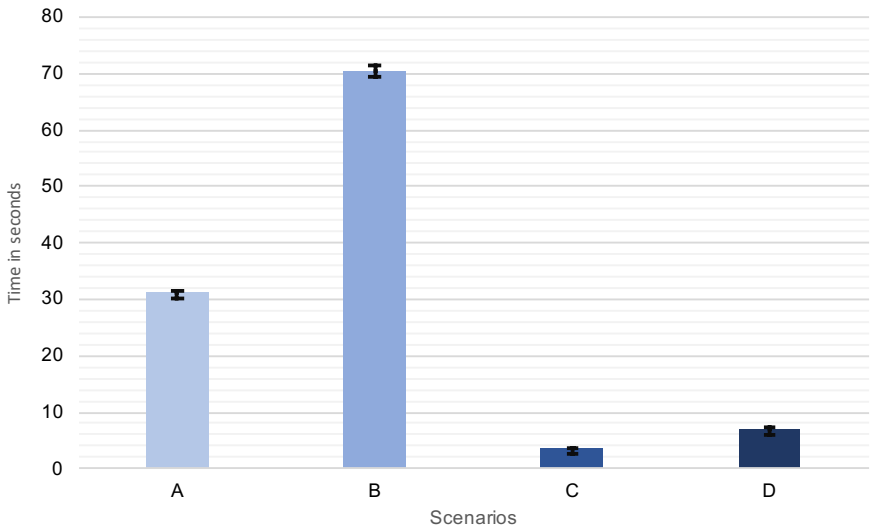

Figure 5.2 – **T6**: Time to execute 100 calls for each 10 different smart contracts

an application must be partitioned in as many blocks with context as possible. Thus the use is highly dependent on the application using the model.

6. RELATED WORK

There has been increasing interest in the area of smart contracts by the academia. To show that, Figure 6.1 presents the number of works published since 2012 using the string “smart contracts” in the IEEE, ScienceDirect and ACM databases. The result is the volume aggregate of this search in those databases. On the chart, it is evident the increasing number of works. We will present part of these work and what areas they are focusing. However, there is no work that considers a new model for smart contract execution. The reason is that the model for smart contract execution is stable. The development of the Context-based model was feasible only because of the changes that the appendable-block blockchain brought to the blockchain data structure.

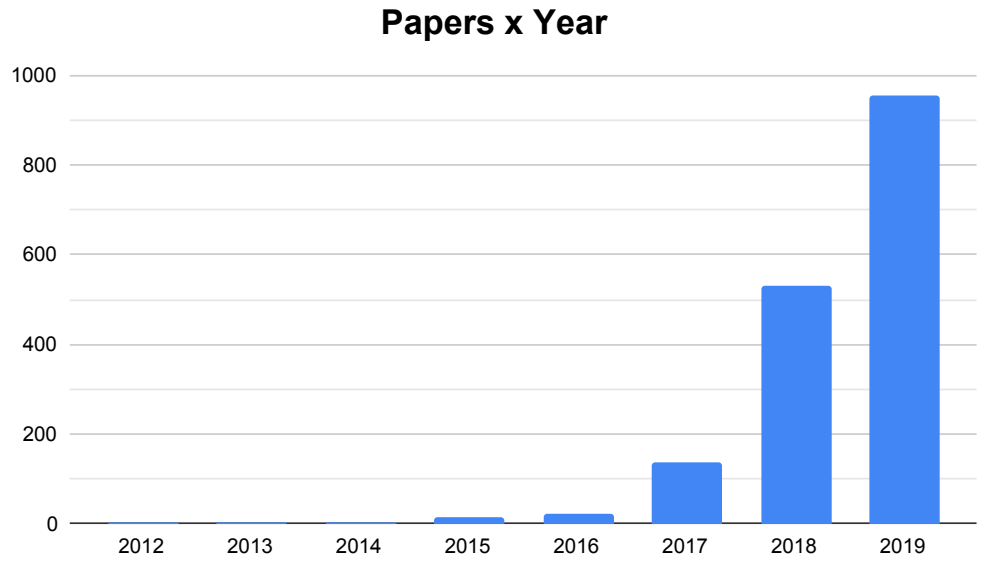

Figure 6.1 – Smart contract papers published by year

Part of those works focuses on exploring applications that can make use of smart contracts. There are multiple areas of applications, one of them is the use of a smart contract for auditorship, for example, Renner *et al.* [33] proposed a framework to help prevent and control alteration of files in cloud-based storage without the need of third parties. A user can detect any change to the file during the retrieval from a cloud-based storage service, additional changes to it are auditable.

The area of resource allocation is another application that has several works related to smart contracts. Predominantly they include negotiation Machine-to-Machine (M2M), which allows dynamic allocation of resources. Since some blockchains include a cryptocurrency, the negotiation can include automatic payments and transactions of digital currencies. One such case is the work of Scoca *et al.* [36], where a smart contract dynamically negoti-

ates contracts between cloud service provides. Those contracts change accordingly to the load needed by the customer and availability by the service provider.

IoT and its sub-domains are an important area of application with works relating to smart contracts. In the work of Wu *et al.* [38], a blockchain using smart contracts is proposed as a source of information distribution across an energy grid network. That allows smart grids to auto-adjust according to energy requirements from different zones in an automated fashion. This solution can potentially improve the efficiency of the energy network, reducing the need for investment in more extensive networks and diminishing environmental footprint.

Although, as mentioned in Section 2.3, there are known problems to smart contracts use, such as scalability (latency and throughput), privacy, and security. Mitigating those problems can expand the domains of application that can use smart contracts. For instance, the work of Mavridou [25] helps by providing a Finite State Machine (FSM) tool that can help in the design of secure smart contracts. The FSM represents a smart contract, the tool aids by providing part of the source code necessary, which avoids common known security issues, the developer can then finish the program using the generated code as a framework. However, the use of a formal method in the development, as the creation of the FSM, can slow the development process. Zhou *et al.* [39] proposes a tool that can analyze smart contracts, verify if logical risks exist, and create a topological diagram of the relationship between parts of the program. It is important to note that it detects risks, which include error and other types of risks. To assert if it is an error, a skilled developer is necessary to analyze the risk. Additionally, the topological analyzes depend too on a developer to detect possible problems with the contract. Lee and Cho [21] propose to increase the security in smart contracts by introducing news commands in Solidity, a language designed for developing smart contracts, and adding news flags to the EVM. These new commands and flags should prevent important vulnerabilities.

In the area of privacy, Kosba *et al.* [18] presented a framework that keeps privacy in transitions using the blockchain. In the framework, two parties can keep private communication using a third party. Although this third party is not a trusted party, if it acts maliciously during the communication, the two original parties will detect. Furthermore, the third party cannot compromise privacy. For the development, the framework provides a compiler that simplifies the utilization of the protocol. Benhamouda *et al.* [3] explores the utilization of multi-party computation (MPC) using Hyperledger Fabric smart contracts and zero-knowledge proofs. It is a generic solution that can be used in multiple applications. For example, to keep some information confidential in an auction. Performance is acceptable, despite that the authors consider that performance improvements are feasible. In the work of Cheng *et al.* [7], the Ekiden platform is developed, which uses Trusted Execution Environments (TEEs) to keep confidentiality in the processed Smart Contracts and increase performance when compared to the Ethereum blockchain.

Finally, regarding scalability, the work of Gao *et al.* [13] proposed a divide-and-conquer strategy for partitioning smart contracts into smaller fair parts that can be processed by multiple nodes at the same time. Additionally, they proposed that not all nodes need to process all the smart contracts. If just part of the nodes processes a divided partition of the smart contract distributed randomly, the security of the system would be very close to the same as if all nodes processed all smart contracts. The authors argue that the use of integer linear programming solver for partitioning the smart contracts can impact performance, all the tests were made with samples with less than 1,000 smart contracts.

In summary, in this chapter, we presented basics concepts to understand how a traditional blockchain, smart contracts, and appendable-block blockchain work. Moreover, we had a briefing about related works in the area of smart contracts. With this knowledge, we proceed to the next chapter, where we will discuss our research plan.

7. CONCLUSION

In this work, we present a new model to execute smart contracts in blockchains. We also verified the impact that the adoption of the context-based model for smart contracts can have on applications. The main focus was to apply the context-based model in the appendable-block blockchain, where it can explore the synergy of block parallelism native of the appendable-block blockchain. Although it can be used on other immutable-block blockchains that support smart contracts.

As a result of the work, the context-based model proved that can have a positive impact on certain applications by significantly decreasing the time to insert multiple transactions in the blockchain. This result also answer our research question, this model can impact applications by allowing higher performance output. However, this model should not be seen as a silver bullet solution. The performance depends on applications that can be split in multiple contexts. Additionally, there is an increase in latency for individual blocks and transactions in the blockchain that can be have a great impact for some applications.

We list the following as the most important contributions of this work: formalization of the appendable-block blockchain using mathematical functions and set theory, previous works included formalization but not using the same mathematical tools; conception and formalization of the context-based model for smart contracts, the model is new and is the most important contribution of this work; a proof of concept implementation of the model using a real blockchain, SpeedyChain, which uses the appendable-block blockchain model; finally, an experiment to measure the performance of the model comparing sequential insertion versus parallel insertion and analysis of the performance. For society, the use of smart contracts can have a positive impact on applications and is the only technology that allows the existence of completely decentralized applications. The model presented can help in a number of these applications by improving performance, throughput, and making the use in scenarios not previously possible to smart contracts because of performance issues applicable.

There are several directions that can be explored as further work: i) the performance analysis using a real application should generate better insights of the performance for a production environment, and a more realistic result; ii) a study of the consequences of unbalanced insertions in the available contexts, such as, when one application in one specific context inserts more transactions than other application in the same blockchain; iii) a comparison of performance with a blockchain that used the immutable-block blockchain model; and, iv) an examination of the impact of different consensus algorithms on the context-based model.

REFERENCES

[1] Ahrenholz, J.; Danilov, C.; Henderson, T. R.; Kim, J. H. “Core: A real-time network emulator”. In: 27th IEEE Military Communications Conference (MILCOM), 2008, pp. 1–7.

[2] Atzei, N.; Bartoletti, M.; Cimoli, T. “A survey of attacks on ethereum smart contracts (sok)”. In: 6th International Conference on Principles of Security and Trust (POST), 2017, pp. 164–186.

[3] Benhamouda, F.; Halevi, S.; Halevi, T. “Supporting private data on hyperledger fabric with secure multiparty computation”. In: IEEE International Conference on Cloud Engineering (IC2E), 2018, pp. 357–363.

[4] Bocek, T.; Rodrigues, B. B.; Strasser, T.; Stiller, B. “Blockchains everywhere - a use-case of blockchains in the pharma supply-chain”. In: IFIP/IEEE Symposium on Integrated Network and Service Management (IM), 2017, pp. 772–777.

[5] Branco, V.; Lippert, B.; Nunes, H.; Lunardi, R.; Zorzo, A. “Avaliação do uso de smart contracts para sistema de saúde colaborativa”. In: 17a Escola Regional de Redes de Computadores (ERRC), 2019, pp. 9–16.

[6] Branco, V. S.; Lippert, B. H.; Lunardi, R. C.; Nunes, H. C.; Neu, C. V.; Zorzo, A. F.; Pirolla, D.; Bernucio, R.; Spacov, S. “Modelo de negócio para saúde colaborativa usando smart contracts: caso tokenhealth”, *Revista Brasileira de Computação Aplicada*, vol. 12, Apr 2020, pp. 134–144.

[7] Cheng, R.; Zhang, F.; Kos, J.; He, W.; Hynes, N.; Johnson, N.; Juels, A.; Miller, A.; Song, D. “Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts”. In: IEEE European Symposium on Security and Privacy (EuroS P), 2019, pp. 185–200.

[8] CoinMarketCap. “Top 100 cryptocurrencies by market capitalization”. Source: https://coinmarketcap.com/, Apr 2019.

[9] Ethereum Foundation. “Ethereum documentation”. Source: http://ethdocs.org/en/latest/index.html, Feb 2019.

[10] Ethereum Foundation. “Solidity documentation”. Source: https://solidity.readthedocs.io/en/v0.6.3/, Mar 2020.

[11] Foundation, E. “Ethereum white paper”. Source: https://github.com/ethereum/wiki/wiki/White-Paper, Mar 2020.

[12] Gao, W.; Hatcher, W. G.; Yu, W. “A survey of blockchain: Techniques, applications, and challenges”. In: 27th International Conference on Computer Communication and Networks (ICCCN), 2018, pp. 1–11.

[13] Gao, Z.; Xu, L.; Chen, L.; Shah, N.; Lu, Y.; Shi, W. “Scalable blockchain based smart contract execution”. In: IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS), 2017, pp. 352–359.

[14] Huang, J. Y.; Tsai, C. H.; Huang, S. T. “The next generation of gps navigation systems”, *Communications of the ACM*, vol. 55–3, Mar 2012, pp. 84–93.

[15] Johnson, D.; Menezes, A.; Vanstone, S. “The elliptic curve digital signature algorithm (ecdsa)”, *International Journal of Information Security*, vol. 1–1, Aug 2001, pp. 36–63.

[16] Johnson, G. “The babylonian method and its properties”. In: Quod Erat Demonstrandum Chicago’s Youth Math Symposium (QED), 2013, pp. 1–11.

[17] K. Christidis, M. D. “Blockchains and smart contracts for the internet of things”, *IEEE Access*, vol. 4, May 2016, pp. 2292–2303.

[18] Kosba, A.; Miller, A.; Shi, E.; Wen, Z.; Papamanthou, C. “Hawk: The blockchain model of cryptography and privacy-preserving smart contracts”. In: IEEE Symposium on Security and Privacy (SP), 2016, pp. 839–858.

[19] Kurka, D. B.; Pitt, J. “Smart-cpr: Self-organisation and self-governance in the sharing economy”. In: 2nd IEEE International Workshops on Foundations and Applications of Self* Systems (FAS*W), 2017, pp. 85–90.

[20] Lee, D. “Hash function vulnerability index and hash chain attacks”. In: 3rd IEEE Workshop on Secure Network Protocols (NPSec), 2007, pp. 1–6.

[21] Lee, S.; Cho, E. “A modified smart contract execution enviroment for safe function calls”. In: IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), 2019, pp. 904–907.

[22] Lunardi, R. C.; Michelin, R. A.; Neu, C. V.; Nunes, H. C.; Zorzo, A. F.; Kanhere, S. S. “Impact of consensus on appendable-block blockchain for IoT”. In: 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous), 2019, pp. 1–10.

[23] Lunardi, R. C.; Michelin, R. A.; Neu, C. V.; Zorzo, A. F. “Distributed access control on iot ledger-based architecture”. In: IEEE/IFIP Network Operations and Management Symposium (NOMS), 2018, pp. 1–7.

[24] Malatras, A. “State-of-the-art survey on p2p overlay networks in pervasive computing environments”, *Journal of Network and Computer Applications*, vol. 55, Apr 2015, pp. 1–23.

[25] Mavridou, A.; Laszka, A. “Tool demonstration: Fsolidm for designing secure ethereum smart contracts”. In: International Conference on Principles of Security and Trust (POST), 2018, pp. 270–277.

[26] Michelin, R. A.; Dorri, A.; Steger, M.; Lunardi, R. C.; Kanhere, S. S.; Jurdak, R.; Zorzo, A. F. “Speedychain: A framework for decoupling data from blockchain for smart cities”. In: 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous), 2018, pp. 145–154.

[27] Milutinovic, M.; He, W.; Wu, H.; Kanwal, M. “Proof of luck: an efficient blockchain consensus protocol”. In: 1st Workshop on System Software for Trusted Execution (SysTEX), 2016, pp. 1–6.

[28] Mirobi, G. J.; Arockiam, L. “Service level agreement in cloud computing: An overview”. In: International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT), 2015, pp. 753–758.

[29] N. Fotiou, G. C. P. “Smart contracts for the internet of things: Opportunities and challenges”. In: European Conference on Networks and Communications (EuCNC), 2018, pp. 256–260.

[30] Nakamoto, S. “Bitcoin: A Peer-to-Peer Electronic Cash System”. Source: www.Bitcoin.Org, Jan 2020.

[31] Nunes, H.; Lunardi, R.; Zorzo, A.; Michelin, R.; Kanhere, S. “Context-based smart contracts for appendable-block blockchains”. In: IEEE International Conference on Blockchain and Cryptocurrency (ICBC), 2020, pp. 9.

[32] Peercoin Foundation. “Peercoin documentation”. Source: https://docs.peercoin.net/#, Feb 2019.

[33] Renner, T.; Müller, J.; Kao, O. “Endolith: A blockchain-based framework to enhance data retention in cloud storages”. In: 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), 2018, pp. 627–634.

[34] Robusto, C. C. “The cosine-haversine formula”, *The American Mathematical Monthly*, vol. 64–1, Jan 1957, pp. 38–40.

[35] Sabounchi, M.; Wei, J. “Towards resilient networked microgrids: Blockchain-enabled peer-to-peer electricity trading mechanism”. In: IEEE Conference on Energy Internet and Energy System Integration (EI2), 2017, pp. 1–5.

[36] Scoca, V.; Uriarte, R. B.; Nicola, R. D. “Smart contract negotiation in cloud computing”. In: 10th IEEE International Conference on Cloud Computing (CLOUD), 2017, pp. 592–599.

[37] Tschorsch, F.; Scheuermann, B. “Bitcoin and beyond: A technical survey on decentralized digital currencies”, *IEEE Communications Surveys Tutorials*, vol. 18–3, Jun 2016, pp. 2084–2123.

[38] Wu, X.; Duan, B.; Yan, Y.; Zhong, Y. “M2m blockchain: The case of demand side management of smart grid”. In: IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS), 2017, pp. 810–813.

[39] Zhou, E.; Hua, S.; Pi, B.; Sun, J.; Nomura, Y.; Yamashita, K.; Kurihara, H. “Security assurance for smart contract”. In: 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), 2018, pp. 1–5.

[40] Zorzo, A. F.; Nunes, H. C.; Lunardi, R. C.; Michelin, R. A.; Kanhere, S. S. “Dependable iot using blockchain-based technology”. In: Latin-American Symposium on Dependable Computing (LADC), 2018, pp. 1–6.

ATTACHMENT A – Publications

Here we list the production associated with this master's dissertation.

- **Context-based Smart Contracts For Appendable-block Blockchains:** This publication is directly related to this dissertation. The concepts, models, development, and experiment are presented in this dissertation are presented in this paper in a more summarized way. **Nunes, H.; Lunardi, R.; Zorzo, A. F.; Michelin, R.; Kanhere, S. "Context-based smart contracts for appendable-block blockchains". In: IEEE International Conference on Blockchain and Cryptocurrency, 2020, accepted.**

- **Impact of Consensus on Appendable-Block Blockchain for IoT:** This contribution is an analysis of the performance yield using different consensus algorithms in an Appendable-block blockchain. The analysis includes scenarios with different number of devices and gateways present in the network. **Lunardi, R. C.; Michelin, R. A.; Neu, C. V.; Nunes, H. C.; Zorzo, A. F.; Kanhere,S. S. "Impact of consensus on appendable-block blockchain for IoT". In: 16th EAIInternational Conference on Mobile and Ubiquitous Systems: Computing, Networkingand Services, 2019, pp. 1–10.**

- **Performance and Cost Evaluation of Smart Contracts in Collaborative Health Care Environments:** This work is an expansion of the work bellow. The expansion includes more details about the architecture and more complex performance analysis. **Lunardi, R. C.; Nunes, H. C.; Branco, V.; Lippert, B.; Neu, C. V.; Zorzo, A. F."Performance and cost evaluation of smart contracts in collaborative health careenvironments". In: 14th International Conference for Internet Technology and SecuredTransactions (ICITST-2019), 2019, pp. 1–6.**

- **Avaliação do uso de Smart Contracts para Sistema de Saúde Colaborativa:** This work is the result of a project that aims to introduce gamification in a healthcare solution. The solution relays heavily on blockchain use, all the business logic exists as smart contracts in an Ethereum blockchain. The paper presents an overview of the project, the solution's architecture, a financial analysis, and a performance analysis comparing the use of a public test network or a private instance of the blockchain. Publish on: **Branco, V.; Lippert, B.; Nunes, H.; Lunardi, R.; Zorzo, A. F. "Avaliação do uso de SmartContracts para Sistema de Saúde Colaborativa". In: 17a Escola Regional de Redes deComputadores, 2019.**

- **Dependable IoT Using Blockchain-Based Technology:** This paper proposes a novel architecture for blockchains. The proposed architecture is generic, and any blockchain can fit in its model. It is composed of four layers: The Network layer, the Data layer,

the Consensus layer, and Application layer. **Zorzo, A. F.; Nunes, H. C.; Lunardi, R. C.; Michelin, R. A.; Kanhere, S. S. “Dependableiot using blockchain-based technology”. In: 2018 Eighth Latin-American Symposium onDependable Computing (LADC), 2018, pp. 1–9.**

Another publication related to other topics is listed next:

- **Model-Based Testing in Agile Projects: An Approach Based on Domain-Specific Languages:** This paper consists of the development of a Domain-Specific Language, the Aquila language. This language is based on Behavior-driven development, it generates automatic scripts for testing in web applications. The novelty is the use of Model-based testing with Agile teams. A focus group study is developed to assert language utility.

  Zanin, A.; Zorzo, A. F.; Nunes, H. "Model-Based Testing in Agile Projects: An Approach Based on Domain-Specific Languages". In: 2020 Iberoamerican Conference on Software Engineering (CIbSE), 2020, accepted.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Graduação
Av. Ipiranga, 6681 - Prédio 1 - 3º. andar
Porto Alegre - RS - Brasil
Fone: (51) 3320-3500 - Fax: (51) 3339-1564
E-mail: prograd@pucrs.br
Site: www.pucrs.br