University of Camerino

International School of Advanced Studies

Dany: scalability solutions for IoT smart contracts

Doctoral Thesis

PhD Candidate (XXXV Cycle):
Davide Sestili

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy in

Computer Science

Author:
Davide Sestili

Supervisor:
Prof. Leonardo Mostarda

April 2023

UNIVERSITY OF CAMERINO

Abstract

Computer Science

Doctor of Philosophy

Dany: scalability solutions for IoT smart contracts

by Davide Sestili

Smart contracts are self-executing programs that run when certain predetermined conditions are met. The advent of blockchain technologies provided the means to executing smart contracts in a decentralized fashion without the need of trusting a central authority: the correct execution of the smart contract is ensured by the consensus protocol of the blockchain on which it is deployed. Blockchain smart contracts property of trustless execution make it an interesting technology for the IoT. However, IoT applications often require processing large amounts of data which are often difficult to manage in a blockchain environment. In fact, traditional blockchains can process only few transactions per second, making it unsuitable for the needs of many IoT applications. Furthermore, traditional public blockchains require users to pay fees for every piece of data committed to the chain, this makes the use of blockchains too expensive in many IoT case scenarios, since they often involve IoT devices exchanging large amounts of messages. In this thesis, it is presented a solution for executing smart contracts that improves scalability on blockchains in terms of throughput and costs. The solution is particularly suited for the IoT but its generality make it possible to be used in a variety of scenarios not necessarily related to the internet of things.

Contents

Chapter 1

Introduction

Blockchain technology was introduced by Satoshi Nakamoto in [46], where he proposed a solution for allowing electronic payments between users without having to trust a centralized institution. The problem Nakamoto's work addressed was the problem of creating a distributed storage of timestamped documents that no user can tamper with without being detected [26]. These kinds of distributed storage systems are called *Distributed ledgers.* Bitcoin implements a distributed ledger in a peer-to-peer network of nodes that do not require to trust each other. Each node of the network is connected to a subset of other nodes and exchanges messages with them. When a user wants to include a transaction in the distributed ledger, he needs to send the transaction to a node; the node will gossip the transaction to its peers and eventually, the transaction will reach a *miner.* In the Bitcoin blockchain, a miner is a node that can append a transaction to the distributed ledger and, in order to do that, the miner performs some preliminary checks on the transaction received in order to guarantee that the transactions are compliant with the protocol specifications and the current state of the ledger (*e.g.*, no user can spend the same coin twice [18]). After that, the miner tries to include the transactions in the ledger by creating a *block* that contains them along with other data, including a reference to a previous block and a solution to a computationally expensive cryptographic puzzle, and then propagates it to the network: the process by which a miner creates a block is called *mining.*
Mining allows the bitcoin blockchain to guarantee that nodes will reach consensus with overwhelming probability on the state of the blockchain even if *bizantine nodes* [39] are present in the network.
Bizantine nodes, and the related concept of *Bizantine fault tolerance*, have been described for the first time by Leslie Lamport et al. in [38]. This seminal work described the problem of reaching a consensus on the state of a computer system in the presence of malicious actors that can send conflicting information to other parts of the system as a problem of Byzantine generals camped outside an enemy city having to reach an agreement on a common plan of action by communicating with one another by messenger. However, some of the generals may be traitors and send conflicting information, trying to prevent the loyal generals from reaching an agreement. The same work presented two solutions to the problem: one involving *oral messages* and the other involving signed messages.
Oral messages are messages that satisfy the following assumptions:

- Every oral message sent is delivered correctly.

- The receiver of a message knows who the sender is.
- It can be detected whether an oral message that should have reached a certain receiver hasn't arrived.

A solution to the problem of reaching an agreement among honest generals, each of which can communicate directly with one another, has been provided for $3m+1$ generals, where m is the number of Byzantine generals (malicious generals) that the system can withstand when only oral messages are exchanged. The solution involving unforgeable signed messages -differently from the oral message solution- can withstand any number of traitors.
However, these solutions, are built on a set of assumptions that do not hold in a public blockchain.
First of all, the solutions proposed require nodes (generals) to know in advance the identities of the nodes of the network, while in a public blockchain, nodes participating in the consensus may drop in or out of the network without having to be previously registered. Second, the proposed solutions work assuming nodes are in a synchronous network, which is a strong assumption for a public blockchain. In consensus protocol literature, there are three main network models that have been considered which differ from one another on the assumptions about message delay; these network models are:

1. Synchronous network: a network in which the maximum latency is bounded and known.
2. Asynchronous network: the maximum latency is unknown and messages may never be delivered.
3. Partially synchronous network: introduced by Dwork et al. [27]. In this network model, an upper bound on the latency exists but it is unknown.

The network model with the weakest assumptions is the asynchronous model, and devising a byzantine fault-tolerant consensus protocol that guarantees consensus under this network model would be ideal for a public blockchain. However, Fischer et al. [31] showed that any consensus protocol that aims to tolerate at least one fail-stop node is not assured to terminate in the asynchronous network model.
The bitcoin consensus protocol in fact, even though it is designed for the asynchronous network model, guarantees termination only probabilistically assuming that the majority of the computational power of the network is not held by a malicious user or by a set of malicious colluding users.
Bitcoin successfully implements an electronic payment system that does not rely on trust. However, there are some limitations, in fact the rate of transactions that the system can handle is low compared to centralized solutions; the fees to be paid to the miners are relatively high, making it unfit for processing micro-transactions; and the energy consumed for keeping alive the blockchain is extremely high.
After the success of Bitcoin, however, numerous solutions for implementing distributed ledgers have been explored, aiming at reducing the limitations of the Bitcoin blockchain, or aiming at providing a distributed ledger optimized for a specific domain.

1.1 UTXO model and Account model

Blockchains can be divided in two groups based on the type of transactions that they accept. Bitcoin transaction model is termed *unspent-transaction-output* model (UTXO) [58].
In a blockchain that supports a UTXO model, a single transaction is made up of *inputs* and *outputs*. The *inputs* of a transaction reference transaction outputs of transactions previously included in the blockchain that has not been spent by any other transaction input. The *outputs* of a transaction specify how the inputs are spent. UTXO transactions usually include signatures or authorization scripts that determine whether the inputs are allowed to spend the outputs they refer to. In a UTXO blockchain, a user can know what his balance is by summing the values associated to the unspent-transaction-outputs he is allowed to spend.
In an account-based blockchain, a user is identified by an address which is associated to a balance. Transactions in this kind of model specify: the sender address; the receiver address; the value to transfer, if any; and optionally additional information.
These two transaction models have their own advantages and disadvantages:
The UTXO model makes it easier to parallelize transaction processing and also makes it easier to implement scaling solutions such as sharding [37, 59].
The account model, popularized by Ethereum [29] instead, provides a more intuitive platform for smart contracts.

1.2 Smart contracts

The *smart contract* concept was introduced by Nick Szabo [54] back in 1994, way before the advent of blockchains. He described it as *"a computerized transaction protocol that executes the terms of a contract"*. The non-trusted nature of the blockchain proved to be the right environment for the implementation of smart contracts. In fact, already in bitcoin there was the possibility of including snippets of code written with a non-turing complete language in a transaction output that specifies the requirements to satisfy for spending it.
A blockchain smart contract is generally a piece of code stored in the blockchain that can produce a change of state when it is addressed by a transaction. Blockchain smart contracts have been used for implementing a variety of services such as decentralized crowdfunding [3] and peer-to-peer energy trading [4]. Blockchains and smart contracts often face significant problems in terms of scalability and costs, and their use with a straightforward approach is not always convenient for many applications.
Smart contracts become popularized with the advent of second generation blockchains: the most popular of which is *Ethereum* [29]. In fact, the differences between Ethereum and Bitcoin lied primarily in the transaction model employed and in the expressivity of their smart contract languages.
Ethereum is an account-based blockchain and also its smart contracts are identified with addresses. When a user wants to deploy a smart contract to the Ethereum blockchain, he has to specify the code for creating the smart contract in the *data*

field of a transaction that has no receiver address. If the transaction is committed successfully, then an instance of a smart contract is deployed on the blockchain and a univocal address is associated to it. The code that is contained in the transaction data field is in the form of *Ethereum Virtual Machine* (EVM) bytecode. The operations specified in the bytecode are executed by the Ethereum Virtual Machine of the miner that mines the block containing that transaction, and by the nodes receiving that block, resulting in a change of the world state of the blockchain.
Requesting the execution of bytecode, however, comes at a cost: in fact, Ethereum transactions also specify the maximum number of gas units that can be used for executing that transaction (gas limit), and the price of each unit of gas (gas price) expressed in *wei* (the smallest unit of Ethereum cryptocurrency); each EVM bytecode is associated with a specific *gas* number that is spent each time that instruction is executed. If the number of gas units used during the execution of a transaction exceeds the gas limit provided, then the execution aborts. The costs of fees associated to transactions are therefore proportional to the number and complexity of the operations they require to perform. This provides an incentive to miners to include these transactions into their blocks: in fact, if every transaction were to have similar fees, regardless of their computational demands, miners would have an incentive in including in a block only the transactions requiring the least operations to perform.
The introduction of gas limit and gas price in Ethereum also allows it to support sophisticated smart contracts written with a Turing complete language mitigating the risks of spamming attacks.

Ethereum contracts are stored in the ledger as EVM bytecode, a stack-based turing-complete low level language. However, it is possible to write smart contracts in high-level languages that compile down to EVM bytecode, the most popular of which is *Solidity* [4].

Solidity is a smart contracting language inspired in its syntax by javascript. Smart contracts written in Solidity look like classes of object-oriented languages that specify methods and state variables.
Methods are functions that can be called either by a user or by a contract and their execution may result in a change of the world state.
State variables are the equivalent of *attributes* in OOP languages and its values can be modified by running a smart contract method. Solidity has been used for a variety of projects including the token smart contracts of OpenZeppelin [3].
The code in 1.1 [1] shows a simple smart contract written in Solidity that acts like a wallet: anyone can send ether to it but only the owner of the contract can retrieve them.

In our work Bistarelli et. al [11], we performed an analysis on smart contracts deployed in Ethereum in order to retrieve information about the frequency of EVM instruction usage per contract and to retrieve information about the most

[1]Taken from: `https://solidity-by-example.org/app/ether-wallet/`

used high-level primitives occurring in solidity smart contracts. In Figure 1.1, it is shown an histogram representing the frequency of EVM instruction in contracts deployed in Ethereum. Figure 1.2 shows the number of occurrences of solidity primitives in smart contracts deployed on ethereum.

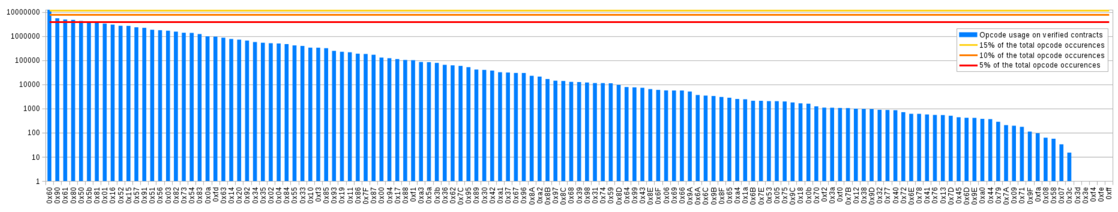


FIGURE 1.1: Histogram of opcode count on smart contracts deployed in ethereum (logarithmic scale).

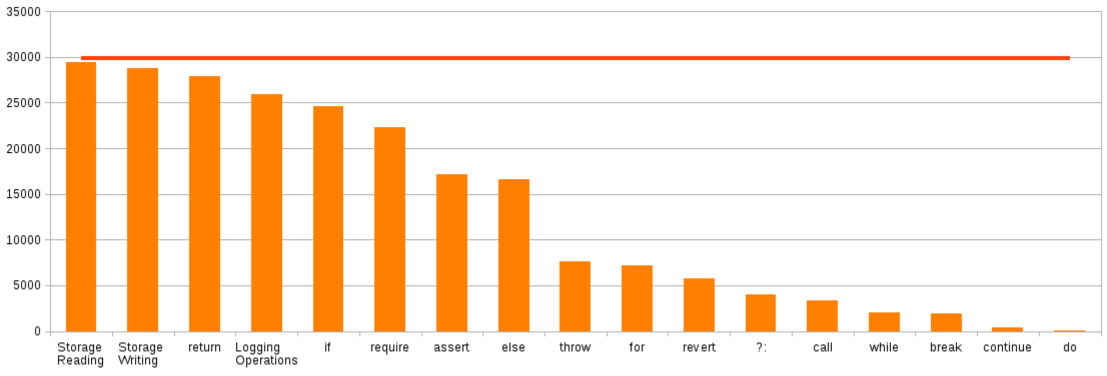


FIGURE 1.2: Histogram of solidity primitives count on smart contracts deployed on ethereum (logarithmic scale).

Algorithm 1.1 Wallet implementation in Solidity

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

contract EtherWallet {
    address payable public owner;

    constructor() {
        owner = payable(msg.sender);
    }

    receive() external payable {}

    function withdraw(uint _amount) external {
        require(msg.sender == owner, "caller is not owner");
        payable(msg.sender).transfer(_amount);
    }

    function getBalance() external view returns (uint) {
        return address(this).balance;
    }
}
```

1.3 Internet of Things and Blockchains

The Internet of things (IoT) is characterized by heterogeneous small physical devices that can communicate with each other in order to reach common goals [7]. IoT applications can be found in a plethora of domains: for example, IoT solutions have been employed in the context of smart cities [6] and supply chains [9]. The distributed non-trusted nature of blockchains, along with their smart contract execution capabilities, provides a practical solution for addressing the limitations of traditional IoT applications. For instance, blockchains can provide a platform for ensuring the integrity of data produced by IoT devices without having to trust a centralized entity, and may also provide a platform for performing computations on these data, saving thus the computational resources of low-power IoT devices [40]. However, the integration of IoT with blockchains faces significant challenges: for instance, traditional blockchains can process only a small number of transactions per block, with blocks being appended on the chain at a rate of one block each several minutes, making it not particularly suited for those IoT applications that require low latency times. Furthermore, public blockchain transactions require the sender to pay some fees to the miner in order to expect it to be included in a block; since IoT applications often require devices to exchange many small data messages, the costs of interacting with the blockchain can become prohibitively high.

1.4 Motivation and research questions

Traditional blockchains and smart contracts don't scale well in terms of throughput. The number of transactions that a public blockchain can process in a given time is often low, making it unsuitable for many applications.
Blockchain smart contracts would provide a suitable environment for executing logic for the IoT, where many untrusting devices are required to perform complex interaction. However, the low throughput and high costs associated with blockchain interaction make them practically unsuitable for this task. Solutions for improving the performances of distributed ledgers and solutions for reducing the burden on the blockchain while executing smart contracts and the costs associated with it without compromising the properties guaranteed with the use of blockchains in the first place have been proposed. However, the proposed solutions do not address the IoT, where it is required to execute logic on lots of data. Therefore, this thesis' research question can be summarized as follows:

- **Research Question:** Can we have a scalable solution for IoT smart contracts while maintaining a high level of security in smart contract execution?

1.5 Thesis contribution

The main contribution of this thesis lies in the specification of *Dany*, a second layer protocol that enables blockchains to support smart contracts that require the elaboration of lots of data without incurring in massive scalability issues.
Dany provides a platform for addressing problems related to the integration of IoT and blockchains limiting the issues that can come out from applying well known blockchain solutions.

1.6 Thesis organisation

The thesis is organised as follows:

- Chapter 2 provides a summary of the state of art regarding scalability solutions for distributed ledgers.
- Chapter 3 provides the description of a use-case scenario in which solution requires the integration of IoT with the blockchain.
- Chapter 4 describes *Dany*: a novel protocol for executing scalable smart contracts.
- Chapter 5 describes the implementation of the Dany protocol.
- Chapter 6 discusses the results obtained by running an implementation of *Dany*.
- Chapter 7 provides conclusions and future directions.

Chapter 2

State of art on scalability solutions

As blockchain technologies become more and more popular, scalability issues become more evident as well.

Croman et al. [22] state that *scalability* is a term that doesn't relate to a singular property of a system, but it is a term that relates to several quantitative metrics. The metrics mentioned for the Bitcoin blockchain include among others:

- **Maximum Throughput**. The maximum rate at which the blockchain can confirm transactions.
- **Latency**. The time taken for a transaction to be confirmed.
- **Bootstrap Time**. The time taken for a new node to download and process the history necessary to validate new transactions.
- **Cost per confirmed transaction**. The cost required to the whole system to commit and validate a transaction.

When talking about transaction throughput, public blockchains are able to process a small number of transactions per second; this is especially evident when compared to centralized solutions enabling electronic payments.

Geordiadis [32] found that the exact upper bound of the transaction throughput of the bitcoin protocol was 27 tx/s in 2019. Bez et al. [10] used a synthetic test environment for ethereum to reach the conclusion that the Ethereum version active during their research could reach a maximum transaction throughput of approximately 15 tx/s. Significantly increasing the size of a block, or minimizing the time required for a block to be mined, may appear as possible improvements to the problem of low throughput in blockchains. However, these solutions cannot be adopted light-heartedly because of the *verifier dilemma* [43].
The verifier dilemma states that in blockchains that allow miners to create expensive blocks (blocks that either contain many transactions or computationally expensive transactions) honest miners are vulnerable to exhaustion resource attacks by malicious miners. In this attack, a malicious miner adds one ore more computationally expensive transactions in a block. These transactions do not cost any gas fees to the miner because all gas fees are collected by the miner of the block. Other miners, however, incur in a dilemma: *Should we verify the large block or not?*

If they do not verify it, they risk to mine a new block that will be later invalidated. If they verify it, however, they will take a longer time to do so, making them less likely to win the mining race. Minimizing the time required for a block to be mined incurs in a similar problem since now miners have less time available to validate a received block before being able to start mining the following one.
The problem of scalability is often seen as part of a trilemma (Blockchain trilemma) that states that it is hard to design a blockchain that achieves optimal levels of three desirable properties of blockchains simultaneously, namely, scalability, security and decentralization.
However, solutions to mitigate the problem of scalability while trying to maintain at acceptable levels the other two properties of the trilemma have been devised. These solutions can be divided into two categories: On-chain solutions and Off-chain solutions. The solutions that are described as On-chain solutions are those solutions that involve the redesigning of the consensus protocol used by the blockchain. On-chain solutions to scalability include consensus protocols alternative to proof-of-work protocol, such as Proof-of-stake protocols. Off-chain solutions, instead, do not require a redesigning of the underlying consensus protocol: They can be used on top of the blockchain of reference which will act as an adjudicator when unexpected behaviours occur.

2.1 On-chain approaches to scalability

On-chain approaches to scalability are those approaches that require either changes in the codebase of the distributed ledger network, or a complete redesign of it. This kind of solutions include adopting a consensus protocol alternative to PoW, partion the data stored in the blockchain into multiple *shards* stored by different groups of nodes, and adopting distributed ledger technologies alternative to the blockchain.

2.1.1 Consensus protocols

The first consensus protocol used in a public blockchain is the *Nakamoto consensus*, implemented in the Bitcoin blockchain [46]. The *Nakamoto consensus protocol* is a Proof-of-work protocol, which means that consensus is achieved on the chain of blocks that required more computational power to be generated, making it infeasible to change the history of the blockchain unless the majority of the computational power of the network is owned by a malicious user or by a set of coordinated malicious users. In particular, the Bitcoin consensus protocol requires nodes participating in the consensus protocol (miners) to create blocks by finding a nonce that, when hashed with other block information, including information about the previous block (Fig 2.1 shows how blocks are linked to each other in Bitcoin), returns a value containing a number of leading 0 bits. Finding the nonce that satisfies this requirement is computationally expensive, but, checking whether it is correct, requires very little computational power. Every node that receives blocks generated from the network will choose the longest chain as the "real" blockchain and can easily prove that the nonce of the blocks satisfies the

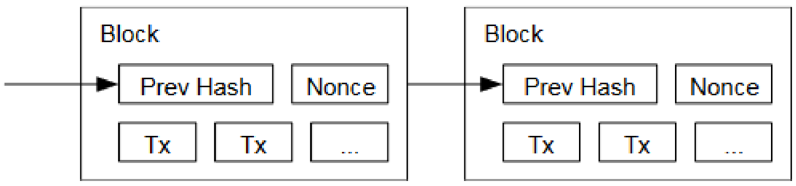


FIGURE 2.1: Picture taken from [46]. Blocks in the bitcoin network are linked to the previous one by referencing the hash of that block, and they contain a nonce proving that computation has been performed for its creation.

requirements.
Proof-of-work is considered to be a very secure class of consensus algorithms for blockchains. However, these type of blockchains require massive costs to be maintained. Other consensus protocols have been proposed and they mainly fall into two categories: *Proof-of-stake* and *proof-of-authority* consensus protocols. Less popular alternative to these classes of consensus protocol have also been proposed, such as proof of space [47] and proof-of-elapsed time [12].

Proof-of-Stake consensus protocols

In proof-of-stake consensus protocols, block proposers are not required to perform heavy computation to have the chance of proposing the new block, rather, they are chosen based on the amount of cryptocurrency they own or they have staked on the network. The costs of maintaining a proof-of-stake blockchain are thus minimal compared to proof-of-work blockchains. However, doubts about the security assumptions of some of these consensus protocols have been raised. **Ouroboros** [35] is a proof of stake consensus protocol stated to be secure under the following conditions:

- The network is synchronous.
- A number of honest stakeholders are available to participate in the consensus protocol as needed in an epoch.
- Stakeholders do not remain offline for long periods of time.

In Ouroboros, time is divided into units called *slots*. At most one *block* can be published at each slot by a leader node.
The committee members appointed to run the protocol for selecting the leader of each slot, along with their relative stake, is updated at each *epoch*, which is a time interval comprising several slots, and the probability of a stakeholder to be selected as the leader of a slot depends on its stake. During an epoch, the stakeholders run a multiparty coin flipping protocol [52] to select, for each slot, its leader. The coin-flipping protocol adopted by Ouroboros allows to tolerate a number of adversaries trying to interfere with the protocol: as long as the stake is owned by an honest majority, attackers cannot meaningfully interfere with the leader selection mechanism.

Since selected slot leaders should be online, and this may not be always possible for little stakeholders, Ouroboros implements a delegation scheme allowing stakeholders to delegate other nodes to sign messages on their behalf using a proxy signing key. The proxy signing key can be revoked at any time and they can also be issued with limitations, allowing delegators to decide in advance which is the range of slots a delegate can use the proxy keys.
Algorand [17] is a blockchain that requires nodes minimal computational power to actively participate in the generation of blocks and that can process way more transactions per seconds than Bitcoin at a lower latency. Algorand is proven to fork with extremely low probability, making it possible to practically consider all the transactions included in a valid block as final (differently from bitcoin and proof-of-work protocols in general). Even in case of network partitions, although forks may happen, they are assured not to be longer than one block in length. In Algorand, a block is assembled by a leader node l_r that is selected at each new block by means of a *verifiable random function* [44]. Blocks are then validated by a set of *verifiers* (SV_r) which members, for the r-th block, depend on a value Q_{r-1} contained in the previous block. This quantity has the property of being unpredictable and not influentiable by a very powerful adversary as the one described in [44]. A verifiable random function can be intuitively implemented as a mapping $m \to H(sig_i(m))$ where $H()$ is an ideal hashing function, and $sig()$ is an ideal signature function; The correctness of the mapping can be proved by providing $sig_i(m)$. Thanks to the verifiable random function, the quantity Q_r can be generated for each new block as the hashed signature, produced by the leader node, of Q_{r-1} ($Q_r = H(sig_{l_r}(Q_{r-1}))$). In Algorand, determining whether a user is a verifier or not depends both on the Q_r quantity and the number of coins owned in the system: the higher the amount of money owned, the higher the probability that you can satisfy the requisites for being in the SV_r set. For its dependence on the money owned in the network, Algorand's consensus protocol can be considered related to Proof-of-stake protocols, even though it doesn't require participants to stake assets in order to become a participant of the consensus protocol.

Gasper [14] is the idealized version of the Proof-of-stake consensus protocol adopted by Ethereum 2.0. Gasper is constructed by combining Casper-FFG [13] and the LMD-GHOST fork choice rule [60]
Casper-FFG is a proof of stake finality system, which allows a network to prove that certain blocks are *final* and therefore cannot be superseded by a competing fork. In Casper-FFG there are blocks that are called *checkpoints*. A block is a checkpoint if its block-height in the blockchain is a multiple of a certain predetermined number. Checkpoints are the blocks that can be finalized in Casper-FFG. To move from a finalized block to a more recent one, stakers have to commit *attestations* stating their will to move from the previous checkpoint to another one in the same chain. A checkpoints block B is called *justified* when there are attestations voting from $A \to B$, with A being another justified block, with a total weight of at least two thirds of the total validator stake. If there are sufficient attestations for $A \to B$ with B being the checkpoint immediately following A, and A is already *justified*, then A is also *finalized*. The genesis block is *justified* and *finalized* by default.

To deter stakers from creating attestations aimed at finalizing different concurrent blocks, in Gasper other participants are allowed to submit the attestation proposed by the malicious staker proving their malevolence slashing thus their deposits.
Since Casper-FFG does not deal with fork-choices, Gasper needs a fork choice rule to determine which is the canonical chain in case of forks. This is achieved with a slightly modified version of the LMD-GHOST fork-choice rule that states that the canonical chain is the chain with the largest number of attestations. Block generation is also not specified either in Casper-FFG and LMD-Ghost. In order to deal with block generation, for each epoch, the stakers are pseudorandomically assigned to a committee; each committee is then assigned to a single slot in the epoch and the first validator in the committee is required to propose a block which is then attested by the other members of the committee.

Proof-of-Authority consensus protocols

Proof-of-authority (PoA) [24] is a family of consensus algorithms particularly suited for consortium and permissioned blockchains whose relevance is due to the improved performances of its algorithms when compared to non-PoA consensus algorithms, especially PoW algorithms. In Proof-of-authority algorithms, only a set of pre-registered authorities (validators) are allowed to participate in the block proposal mechanism. This is often done by making validators take turns for creating (mint) blocks and by employing either a block finalization protocol or an appropriate fork-choice rule to reach a consensus on which is the authoritative chain. Two prominent consensus algorithms falling in this category that are implemented in the Ethereum client Besu are *Clique* [55] and *IBFT 2.0* [51].
Clique [55] is a proof-of-authority consensus algorithm available in Besu. In Clique, a validator can assemble a block (*mint* a block) by simply signing it and propagating it to the network. Clique validators are supposed to take turns in proposing blocks and the protocol deterministically determines the preferred proposer of a block. However, if the preferred proposer does not propagate the block in time, other validators may propose their blocks after waiting an amount of time depending on their position in the list of validators and on the height of the chain. Validators are however allowed to propose a block only if they have not minted any of the $floor(SIGNER\_COUNT/2) + 1$ previous blocks, where $SIGNER\_COUNT$ is the number of registered signers in the blockchain. Since multiple validators may propose conflicting blocks to be added to the head of the chain, forks may appear and should be handled. To handle forks, Clique selects the fork with the highest cumulative score as the main chain. The score of a block depends on whether it has been proposed by the preferred validator for that block or by a different validator, with blocks proposed by the preferred one having a higher score number than the other ones.
IBFT 2.0 [51] (Figure 2.2 is a Proof-of-Authority consensus protocol inspired by the PBFT consensus algorithm [16] that provides immediate finality, which means that a transaction, when included in a valid block, will not be superseded by any conflicting transaction. The protocol is proved to work correctly in a partially synchronous network when at most 1/3 of the total amount of *validators* are byzantine. In IBFT 2.0, the nodes participating in the generation of blocks are

called *validators.* A set of pre-established validators is contained in the genesis block of the blockhain, but they can be added or removed thanks to a voting system. In order to generate a new valid block, a multiple-phase protocol is run among the validators which will eventually allow validators to accept a block and to assemble a certificate certifying the validity of the block. In the first phase of the protocol, a validator p (proposer) is selected among the set of validators of the network possibly in a round-robin fashion. Validator p will then broadcast to the other validators a *propose message* signed by himself containing information about the new block and a *round change certificate.* When non-proposing validators receive the *prepare message*, they check its correctness and will later multicast to all the validators a *prepare message* signed by him, containing, along with other information, the hash of the block and the *round-change certificate* signalling other validators their willingness to accept the previously proposed block. When a validator v receives at least a quorum of agreeing *prepare messages* (2/3 of the validators), then v will assemble a *commit message* aimed at creating a finalisation seal for the block. The commit message contains in fact a *commit seal* which is the signature of validator v over the proposed block and the round number (which is 0 if it was to reach the commit phase in the first round). Once the *commit message* has been assembled, it is multicasted to the set of validators, just like the previous messages. By obtaining at least a quorum of different commit seals for the same block, a validator can assemble a block finalisation proof and, in this way, end the finalisation protocol by multicasting the block and the proof to the other validators.
Since there may be situations in which a block finalisation proof cannot be created, for instance, because the proposer is byzantine, then the protocol allows to change the proposer by creating a round change certificate. Similarly to the other parts of the protocol, when a validator has reasons to believe that the finalisation protocol will not be concluded successfully in the current round, a round-change sub-protocol run among the validators will be run. The objective of this sub-protocol is that of assembling a *round-change certificate* that will be used in a subsequent run of the block-finalization protocol to prove that the previous round did not end successfully and that a new proposer has been selected.

2.1.2 Sharding

Sharding is a technique originally adopted for scaling large databases by partitioning them into separate units called *shards*, that can be separately stored into multiple machines. In traditional blockchains, the entire sequence of blocks has to be stored by each node participating in the consensus protocol, making the storage requirements of a node that wants to participate in the blockchain always higher, no matter how many nodes participate in them. Transaction processing is also not parallelized, since each node has to process every transaction committed to the blockchain, and this means that throughput cannot be increased with new nodes joining the network. Dividing the blockchain into multiple separated chains seems, at first sight, a good idea for decreasing the storage requirements of nodes and for increasing the throughput of the network. However, in a decentralized

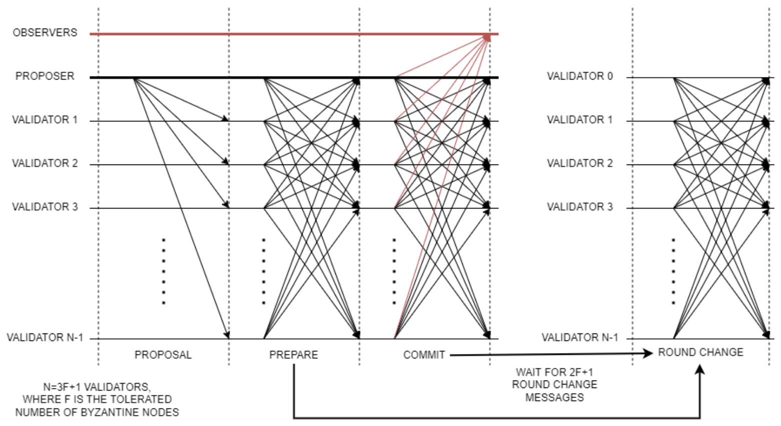


FIGURE 2.2: IBFT 2.0 block finalisation protocol: A finalized block is generated with a three phase protocol. In the proposal phase, a proposer node send a propose message to the other validator nodes; when a validator receives the propose message, it enters the prepare phase and will send prepare messages to all the other validator nodes; a validator enters the commit phase when he has received a quorum of agreeing prepare messages, in this phase he will send to the other validator nodes a commit message for the proposed block. When a validator receives a quorum of commit messages, he can assemble a block finalisation proof. Whenever a validator cannot move to the following phase within a time limit, will start a sub protocol aimed at creating a round change certificate, that will be used to designate a new validator node as the proposer for the current block.

environment with the possibility of Byzantine actors, such as the environment on which blockchains operate, sharding solutions taken from the world of traditional databases do not work, meaning ad-hoc solutions for blockchains had to be designed. A first sharding protocol for blockchains was proposed by *Luu et al.* [42] with the name of *Elastico*. Following that, more efficient sharding solutions for blockchains have been proposed, such as *Omniledger* [37] and *RapidChain* [59].

Omniledger [37]: is proposed as a distributed ledger that aims to solve scalability problems with the use of shards. Shards in Omniledgers are made up of groups of validators with the role of processing only their assigned subset of UTXO transactions. To deal with the possibility of having some shards with a majority of malicious validators, single validators are not allowed to select the shard they want to participate in, but they are assigned to a random shard after each *epoch*. In order to ensure that the assignment of validators to shards is truly unpredictable and not influenceable by malicious validators, the generation of the *randomness* is achieved by running a multi-party protocol called RandHound [53]. Since RandHound operations need to be orchestrated by a leader node, the leader node is better to be unpredictable as well, and this is achieved with a sub-protocol in which validators compute a *ticket*, which is the result of signing with their private key data pertaining to the state of the blockchain, and multi-cast them, allowing nodes to select deterministically the leader which will be the validator producing the lowest-value valid ticket. To enable cross-shard transactions, Omniledger uses a protocol (*Atomix*) that allows to atomically process transactions targeting multiple shards by following the following steps:

1. The client that is requesting a transaction gossips a *cross-shard transaction* on the network, eventually reaching every shard the transaction is targeting.

2. When the *cross-shard* transaction is received by the leader of a shard containing an output spent by the transaction, if the transaction is valid it sends back a *proof-of-acceptance* message, momentarily locking the output. If the transaction is invalid, a *proof-of-rejection* is sent instead.

3. If the client receives from every shard leader a *proof-of-acceptance*, then he will be able to assemble an *unlock-to-commit transaction* and commit it to any shard involved. Similarly, if the client receives at least a *proof-of-rejection*, he will be able to unlock the locked funds by propagating an *unlock-to-abort* transaction.

The consensus protocol used among validators in a shard is called *ByzCoinX*, an enhancement of ByzCoin [36], originally built on PBFT, allowing shards to parallelize block commitments at a fast rate.

RapidChain [59] is a sharding protocol aimed at loosening the limitations of previous sharding protocols; in particular, it is designed to limit the communication overhead inherent in previous protocols and it increases the number of byzantine validators the protocol can support up to 1/3 fraction of its participants, which is higher, for instance, than Ominledger's Byzantine fault tolerance, since the latter supports only 1/4 fraction of faulty validators out of the entire validator pool. RapidChain also addresses the problem of the generation of an *initial common randomness* that may be insecure. Generally, this common randomness is provided in the form of a genesis block, while in RapidChain it is generated by running a bootstrapping protocol that requires the exchange of $\mathcal{O}(n\sqrt{n})$ messages, where n is the number of participants of the protocol. RapidChain protocol can be divided into three main components: bootstrap, consensus and reconfiguration. The first component run is the bootstrap and then the protocol works in *epochs* where each epoch consists of multiple iterations of consensus followed by a reconfiguration phase. During the bootstrap phase, the initial set of participants run a committee selection protocol among them to select a committee of nodes called *root group*. The members of the root group are responsible of generating random bits that will be used for establishing a *reference commitee* whose role is that of dividing the participants into committees responsible for each shard. Intra-committee consensus is achieved thanks to a synchronous consensus protocol making use of a gossiping protocol designed for gossiping large messages inspired by the IDA protocol in [5]. RapidChain adopts a 4-phase synchronous consensus protocol, instead of a classical PBFT protocol that guarantees consensus even in a partially-synchronous network, because it decreases the amount of data to be exchanged to reach consensus. During the course of an epoch, a node that wants to join the consensus protocol needs to solve a PoW puzzle dependent on the randomness generated for that epoch and send a transaction containing solution of that puzzle with the public key they are going to use. If the solution is sent before the end of the epoch, then the transaction is accepted and the node is going to be part of the validator set for the next epoch. At the end of an epoch, the protocol goes into the *reconfiguration phase*. During the reconfiguration phase, nodes perform a protocol to generate a new randomness r_{i+1} which will be used as a seed for randomly assigning new nodes to different committees and to choose the nodes to evict from each committee in order to assign them to different committees.

In order to enable cross-shard transactions, RapidChain developers observed that Omniledger cross-shard protocol that requires the user proposing a transaction to run a protocol that will eventually allow him to assemble a proof proving the possibility of committing that transaction, incurs in a large communication overhead. Therefore, RapidChain adopts a different approach in which a users can simply send a transaction to any committee which will then route the transaction to the *output committee* (The committee responsible for committing that transaction outputs to their own shard). The output committee leader will then create additional transactions each of which spends an input of the original transaction and will route them to the committee holding the inputs of these transactions. If these additional transaction are added successfully to the right shard, then the output committee will commit an additional transaction spending the outputs of the transactions previously created.

2.1.3 IoT blockchain ecosystems

A number of native blockchain solutions aiming at solving specific issues related to the Internet of things world have been devised. The solutions vary not only in the implementation details, but most importantly on the problems they try to solve. Two notable IoT blockchain solutions for different issues related to the world of the IoT are *IoTeX* [56] and *Helium* [33]

IoTeX [56] is a blockchain ecosystem devised especially for the IoT. The proposers of IoTeX claim that benefits can arise from the interaction of the IoT and the blockchain. In fact the blockchain provides the property of decentralization, partially addressing the privacy concerns arising from having a centralized party monopolizing the market. Blockchains, also, by providing smart contract capabilities, allows to extend the functionalities of IoT devices that are often shipped with hard-coded logic included. However, the interaction of the IoT with the Blockchain has some challenges to be addressed, including privacy challenges and scalability challenges. IoTeX aims at providing a solution for integrating the real-world data of the IoT with the blockchain trying to address the aforementioned challenges. They do so by creating an ecosystem of blockchains, in which multiple blockchains, called *subchains*, are connected to a *root chain*. Subchains should be specialized to manage only certain kind of IoT data that is of interest in the ecosystem improving upon the scalability of the ecosystem: if all the IoT data had to be managed by a single blockchain, its size would increase at a too fast rate. The *root blockchain* that connects all of the subchains allows the blockchains of the ecosystem to communicate when needed. The root blockchain is designed as a UTXO-model blockchain because it is more suitable for applying existent privacy-preserving techniques such as ring signature, and ZK-SNARKs for hiding information about the sender of a transaction, the receiver, or other information. In order to allow resource constrained IoT nodes to participate in the consensus of the blockchain, IoTEX implements a delegated proof-of-stake consensus algorithm. In delegated proof-of-stake consensus protocols, participants are allowed to choose delegates to represent their portion of stake in the network, freeing thus low-resource IoT nodes them from executing the most resource exhausting parts of the protocol.

Helium [33] is a decentralized wireless network whose objective is that of enabling devices anywhere in the world to connect to the internet and to geolocate themselves without the need of power hungry hardware. The way Helium achieves this is by creating a blockchain whose miner nodes are devices providing wireless connectivity in a certain area. These miners run the blockchain using a consensus protocol called *Proof-of-coverage* protocol, that requires them to create proofs that they are providing network coverage in a certain area before being hopefully elected to be part of an asynchronous byzantine fault tolerant consensus group responsible of generating blocks in a given epoch. The Helium blockchain provides miners an incentive to provide network coverage, in fact, devices requesting to send or receive data through the internet need to commit transaction that will cost them fees that are collected by the miners.

2.1.4 Other Distributed Ledger technologies

Blockchains are not the only way to implement a Distributed ledger. In fact, it has been shown that distributed ledgers may benefit from being implemented without resorting to the implementation of a blockchain. *Directed acyclic graphs* are often presented as an alternative to blockchains especially in situations where many small IoT devices are required to interact with a distributed ledger.

Directed acyclic graph

In a *Directed acyclic graph* (DAG) distributed ledger (Figure 2.3), transactions are not bundled inside an ever growing list of blocks, instead, they are stored as vertices of a graph. In DAG distributed ledgers it is often the case that transactions do not require the intervention of external miners or validators to be added to the ledger; rather they are committed immediately by the issuer. This kind of distributed ledgers tend to be more scalable than other distributed ledger technologies mainly for the following reasons:

1. Transactions can be added to the DAG in an asynchronous way, allowing for higher throughput levels.

2. Storage requirements are lower since nodes participating in the network are generally not required to keep in storage the entirety of the DAG.

3. Committing a transaction is often less expensive, both in terms of computational power required than in terms of fees to be payed.

DAG distributed ledgers include *Nano* [41] and *Byteball* [19], but the most popular one is IOTA [50].

IOTA [50]: is a cryptocurrency especially suited for machine-to-machine payments in which transactions are stored in the *Tangle*, a directed acyclic graph for storing transactions. In IOTA, a node can issue a transaction by referencing (validating) it to two other non conflicting transactions already in the tangle, chosen with a tip selection algorithm. Transactions referenced in such a way increase their chances of being preferred in case multiple conflicting transactions appear in the tangle. In fact, the tangle does not guarantee that it does not contain conflicting transactions, however, in case conflicting transactions exist, nodes need to agree on which transaction is going to be *orphaned*. The transactions orphaned in the tangle are the transactions that will likely not be validated by any incoming transaction coming from a valid node. Deciding which of the conflicting transaction a node should make an orphan is done by running multiple times the *tip selection algorithm*: the transaction that is selected fewer times is the orphaned one. The tip selection algorithm is the algorithm used by nodes to select the transactions to validate when issuing a new transaction. The correct specification of the tip selection algorithm is crucial for guaranteeing the safety of the system and in IOTA it is implemented as a *Markov Chain Monte Carlo* (MCMC) algorithm which protects the network from attacks such as the *parasite chain attack*: an attack in which

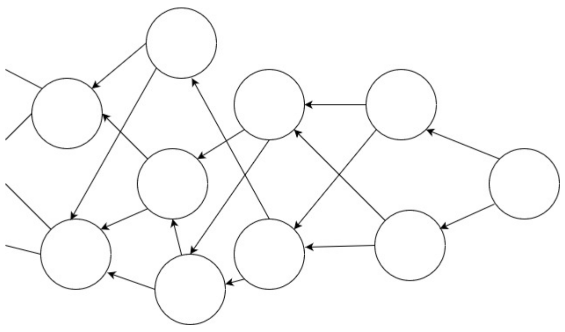


FIGURE 2.3: Example of a directed acyclic graph: In distributed ledgers based on DAGs each transaction can be represented as a vertex: the edges that connect a transaction to another represent the transaction validated by the former.

a user builds a chain with the objective of validating a transaction that conflicts with another one previously stored in the tangle.

2.2 Off-chain approaches to scalability

Off-chain approaches to scalability are those approaches that can be implemented on top of an existing blockchain without having to change any core component of the blockchain. These solutions usually allow users to perform transactions that will be elaborated and stored outside the blockchain while not having to trust any third party. Off-chain solutions to scalability include *payment channels*, *state channels*, secure delegation of the execution of smart contracts, and *sidechains*.

2.2.1 Payment channels

Payment channels offer a solution to scalability problems of the blockchain by allowing users to exchange multiple financial transactions while committing only a few pieces of information on-chain. In payment channels, parties are usually required to lock funds on the blockchain (channel setup) that can be retrieved when certain conditions are met (closing channel, Figure 2.4). These conditions usually include committing proofs that both parties agree to close the contract and redistribute funds in a certain way, or, if a party is uncooperative, to execute a dispute resolution mechanism that rewards the honest participant. After locking the funds, transactions can be exchanged by the parties without committing them to the blockchain.

Payment channels proposed for the bitcoin network include the Lightning network [49] and the protocol described by Decker et al. [25].

Bitcoin transactions are made of inputs and outputs: The inputs are references to other transactions' outputs and provide the funds to spend in the transaction; the results of a transaction contain the instructions for using the bitcoins associated with them. The combined amount of bitcoins associated with a transaction's outputs cannot exceed the number of bitcoins referenced by its inputs. Transaction outputs can be referenced only once by another transaction input, preventing

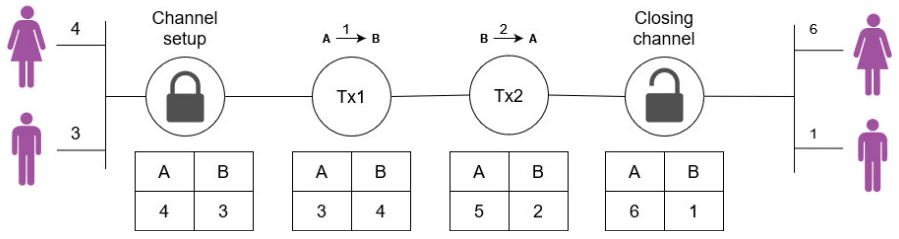


FIGURE 2.4: Example of a payment channel. Tokens are locked by both parties in the channel setup phase, they will then be able to exchange multiple transactions off-chain and eventually will be able to retrieve on-chain their balance in the closing channel phase.

thus double spending.
In Bitcoin, payment channel are often instantiated with the aid of *Timelocks* associated with transactions; a timelock is a primitive that specifies either the minimum Unix timestamp or block height after which the transaction can be included in a block. This means that a transaction with a certain Timelock in the future can be superseded by a transaction with a lower timelock spending the same outputs. In a unidirectional payment channel, time-locked transactions can be used for setting up an expiration date to the payment channel, after which, locked money will be refunded. A *refund* transaction with a specified timelock can be assembled by the parties to ensure that funds will not be locked indefinitely in the output of the transaction funding the payment channel but will eventually be refunded if the receiver end of the payment channel doesn't commit a more recent transaction. Both Lightning protocol [49] and Decker et Al. [25] improve upon payment channels by allowing the setting up of duplex micropayment channels on Bitcoin, they also allow the routing of payment through multiple hops, allowing users that did not set up an on-chain payment channel to use a route of payment channels connecting one end to the other.

2.2.2 State channels

State channels expand the capabilities of payment channels allowing the execution of smart contracts off-chain. State channels are implemented in a similar way to payment channels: opening a channel requires locking funds in a blockchain smart contract and transactions are exchanged by the participants outside the blockchain. The main difference lies in the transaction data: these transactions are not necessarily simple financial transactions, but they can carry other pieces of information, allowing the execution of custom smart contract logic outside the blockchain.

Dziembowski et al. [28] proposed a formal construction of general state channel networks. In their proposal, state channels are either ledger channels or virtual channels: Ledger channels are state channels that are created by committing a *state channel contract* on the blockchain; virtual state channels are channels built on top of two other state channels connecting the two ends of the channel to a third intermediate party, these channels do not require a smart contract to be deployed on the main chain. Following a recursive approach, virtual state channels can be created on top of other virtual state channels, enabling a party to be linked with any other party for which there exist an indirect path of ledger channels connecting them. Interaction with the smart contract is performed off-chain by the parties by exchanging signatures on new states of the smart contract without interacting with the blockchain at all. Disagreements occurring in a ledger channels are resolved by contacting the channel smart contract: the channel smart contract redistributes the funds after receiving the latest agreed state from the parties; if a party submits an old state, the other one can prove him wrong by submitting a more recent one. Disagreements occurring in a virtual channel require interacting with the intermediate party using the channels each party has previously established with it.

Tom Close proposed **Nitro** [20], a protocol for constructing arbitrary state channel networks. Nitro allows the construction of state channels that allow parties to fund them and retrieve the coins owed to them at any moment in time. Nitro also allows the construction of secure virtual channels that are funded by other state channels, allowing parties create channels among without interacting with the blockchain if at least an intermediary exists. Nitro allows the construction n-party state channels, allowing the off-chain execution of multi-party smart contracts.

Force Move [21] is a protocol designed for running a restricted set of n-party state channels. Setting up a Force Move state channel requires deploying an adjudicator contract that is responsible for holding the funds of the state channel, redistributing them when necessary, and solving disputes; and a library that specifies the rules of the smart contract. Once the state channel is opened, the parties will be able to update the state of the contract by signing a new state that is, according to the library previously mentioned, a legal transition from the previous one. The order in which parties are allowed to interact with the smart contract, however, is predetermined, limiting thus the range of applications developable with the Force Move protocol. This limitation enables the protocol to offer mechanisms to solve disputes arising in a n-party state channel. In fact, if a participant decides to not interact with the contract when it is its turn, then the other participants can commit n consecutive states, each signed by a different participant, to the adjudicator, proving that the last committed state is a valid state, and forcing the unresponsive participant to respond, either with a new valid state before a timeout expires, or with a proof that the committed state is not the most recent one. If the unresponsive participant does not respond with a valid move in time, then the contract closes and the locked funds are redistributed according to the rules specified in the blockchain library.

2.2.3 Delegated computation

Delegating the computation of smart contracts to an external party, or to a set of external parties, is a possible improvement to the problem of scalability in blockchains. Delegating the execution of smart contracts may be necessary for those smart contracts whose logic is not supported by the blockchain of reference and also proves beneficial for computationally expensive smart contracts that would require paying very high fees if they were to be executed on the first layer. Blockchains owe their importance to decentralization, and smart contracts executed on the blockchain are deemed to be secure by the participants exactly because the result of their execution is not centralized. This means that delegating smart contract execution to an external third party should be done carefully and require designing solutions that guarantee the correct execution of smart contracts. Solutions in which the execution of smart contracts is delegated to external parties include the following:

YODA [23] is a solution for enabling off-chain execution of computationally intensive smart contracts (CIC). In YODA only a randomly selected subset of nodes (Execution set) compute the smart contract and publish a digest of the result on the network, then the likelihood of each proposed solution is determined using an algorithm run by the miners which result depends on the number of digests received and the number of byzantine nodes tolerated in the system. When the likelihood computed for a specific proposal exceeds a certain threshold, then that proposal is considered the correct one, otherwise, a new execution set is chosen until the likelihood of a specific hypothesis doesn't exceed the threshold. YODA provides thus a solution to executing complex smart contracts, with minimal burden on the main blockchain, however, the likelihood of accepting a wrong solution, while minimal, still exists, if a large portion of YODA nodes behaves maliciously.

ACE [57] is a protocol that allows the execution of complex smart contracts by delegating their execution to a set of service providers (execution set). A smart contract in ACE has to specify which service providers are part of its execution set and how many agreeing service providers are needed to accept a proposed result. The salient feature of ACE is that it also allows the execution of interconnected smart contracts, which are smart contracts that call other smart contracts run by different execution sets. A state change resulting from the execution of a smart contract that does not involve other smart contracts is accepted by the blockchain if a number of agreeing state changes are committed to the blockchain by the previously appointed service providers. Users request the execution of a smart contract by committing a transaction on-chain. Service providers are required to listen to the blockchain for intercepting transactions requiring the execution of a smart contract they are responsible for. When a transaction targets the execution of a smart contract that involves other smart contracts, all the execution sets of the involved smart contracts are required to execute their code. To do so, the user that requests the execution of a smart contract can pre-execute locally its code in order to check what are the involved smart contracts, then he will commit to the blockchain transactions targeting every involved smart contract. State updates will be accepted by the miners only if the state changes committed by the execution sets are coherent and are committed by a quorum of each execution set involved.

Arbitrum [34] is a system that allows the execution of scalable smart contracts. Arbitrum smart contracts are executed by a pre-appointed set of parties called *managers*. Arbitrum requires a single honest manager to ensure that the execution of the smart contract is correct (*any-trust* guarantee). When the execution of a smart contract is requested by a party, each manager computes the resulting state-change and, if every party agrees on the result of the computation, it is communicated to the blockchain along with the signature of every manager and it is accepted by it if the preconditions stated in the message hold: these state-change messages are called *unanimous assertions*. When unanimous assertions cannot be assembled, managers are allowed to publish a *disputable assertion* signed by a single manager. Once a disputable assertion is published, it can be disputed by ohter managers. To do so, a *bisection protocol* is employed, in which the asserter and the disputer compete to prove the other wrong. At each step of the protocol, the asserter is required to divide the previous assertion into two equally complex assertions. The disputer will chose the sub-assertion he wants to dispute and the protocol continues, with the asserter dividing that sub-assertion and letting the disputer chose one division, until one of the parties doesn't respond within a certain time limit, or the assertion has been divided so many times that the disputer is required to compute a single operation whose correct result can be proved to the blockchain.

2.2.4 Sidechains

Sidechains and Childchains have been proposed as a solution to scalability issues related to blockchains and to reduce the costs of fees to be paied when moving crypto assets. The main idea of sidechains is that of *connecting* two blockchain toghether allowing them to transfer assets from one to the other and viceversa. Adam Back et al. [8] proposed the technology of pegged sidechains for transferring assets from one blockchain (parent chain) to another (sidechain). In their work they observed that the cryptocurrency is conceptually independent from the blockchain providing it:

The core observation is that "Bitcoin" the blockchain is conceptually independent from "bitcoin" the asset: if we had technology to support the movement of assets between blockchains, new systems could be developed which users could adopt by simply reusing the existing bitcoin currency [8]

They call blockchains that can receive the currency of another one as *pegged sidechains* and provide a set of properties they should satisfy which are:

- Sidechain should allow to move assets back to the parent blockchain by the owner of the assets;
- Assets should be moved without counterparty risks;
- Transfers between blockchains should be atomic;
- A bug in one sidechain should not impact the parent chain;
- Blockchain reorganisations should be handled cleanly, even during transfers; any disruption should be localised to the sidechain on which it occurs. In general, sidechains should ideally be fully independent.

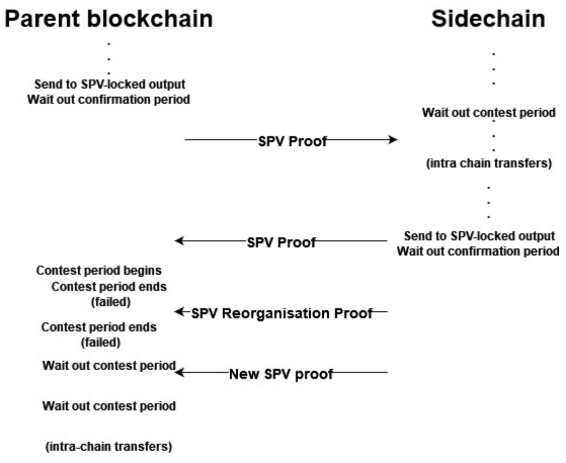


FIGURE 2.5: Example of a communication with a pegged sidechain as described in [8]

- Users of a blockchain should not be required to track pegged sidechains they don't use.

The solution proposed for enabling the instantiation of sidechains is called the *two-way peg*. A two-way peg is a mechanism that allows assets to be moved from one blockchain to another. The proposed two-way peg mechanism involves the use of *symplified payment verification proofs* (SPV-proofs). An SPV-proof is an ideally short piece of data that provides proof that a certain amount of tokens have been *SPV-locked* in a certain blockchain. To do so in a Bitcoin-like blockchain, the SPV-proof has to contain a list of blockheaders used for demostrating the proof-of-work and a proof that the transaction was commited in one of the blocks. Since forks can happen in these kind of blockchains, a sidechain receiving the tokens of a parent chain requires the SPV-proof to prove that the transaction has also been buried under a sufficient number of blocks. After that, the token is still made available in the sidechain since a *contest period* has to be waited in which any user can contest the SPV-proof provided by providing a proof that a block reorganisation has occurred, meaning that a portion of the blockchain locally stored in one node has been overtaken by one in which more proof-of-work work has been done.

Another solution that can be classified in the sidechain solutions category is **Plasma** [48]. Plasma is a framework that allows financial transactions and computation to be outsourced from a blockchain (root chain or parent chain) to another blockchain (child-chain) in order to reduce the burden on the root blockchain and possibly reducing fees. In Plasma, a child chain can be connected to a parent chain by means of a smart contract deployed on the parent chain. The smart contract has the role of enforcing the consensus rule of the child chain and allows users to move their funds from the parent chain to the child chain by and viceversa. Plasma child-chains periodically commit merkelized commitments about the blocks they are producing to the parent chain, allowing the parent chain to be freed from the burden of processing the the transactions commited in the child chain. Child chain security is ensured by allowing users to perform an interactive game in which they can prove to the parent-chain whether certain funds that are about to be withdrawn have already been spent or not. Users are in fact required to keep track of the child-chains they are interested in; as long as they can keep track of them their funds will not be stolen. While moving the computation to a faster and less-decentralized blockchain is already a good solution for scalability issues, Plasma allows the construction of child-chains on top of other child-chains, allowing the creation of trees of blockchains for improving scalability.

2.2.5 Diversity

Diversity [15] is a second layer solution we proposed in 2021 addressing the problem of scalability for the execution of smart contracts in the IoT.

Diversity assumes that a set of users are interested in running a smart contract off-chain that requires the elaboration of streams of data provided by a set of predetermined IoT devices that in Diversity are called sensors; the elaboration of such streams may result in actions to be performed on a specific blockchain, these actions can be either the logging of data on the blockchain or a redistribution of funds previously locked on a on-chain smart contract. In order to do that, Diversity requires users to commit to a blockchain of reference a smart contract specifying a set of devices responsible of performing the elaboration of such streams (intermediate nodes); a set of sensors providing the data to be elaborated; and the specifics of the elaboration to be performed, which includes the way in which intermediate nodes should assemble windows of data from the streams they are receiving, and the function to be applied to such windows (*off-chain* function), which is the function that will determine whether a window has triggered a blockchain action and what this action is (e.g., moving funds previously locked on the smart contract or logging data on the blockchain).

Once the smart contract is deployed on chain, the referenced intermediate nodes will start assembling the windows according to the specifics of the smart contract and at the end of the window period, it will execute the off-chain function. Once the off-chain function has been executed, the intermediate nodes will run a protocol among them (fig 2.6) in order to generate a proof that a unanimous agreement has been reached on the result of the computation; This protocol ensures two properties:

- A blockchain action will be performed if and only if unanimous agreement could be reached;
- Lazy intermediate nodes will not be able to generate the proof if they didn't actually perform the computation;

In order to guarantee this, intermediate nodes are required to perform three phases, namely (i) proof of computation; (ii) computation agreement checking and (iii) SC update; In the proof of computation phase, every intermediate node assembles a message (proof of computation message):

$$p_{t,i} = H(f(w_i[t])||S_{t,i}||ID_i)||t||a||H(w_i[t]) : i \tag{2.1}$$

where:

- $H()$ is an hash function;
- $f()$ is the off-chain function;
- $w_i[t]$ is the t-th window assembled by the intermediate node i;
- $S_{t,i}$ is a secret, this is used by the intermediate node to prove later on that it computed the off-chain function and thus is not lazy;
- ID_i is a unique identifier of the intermediate node;
- t specifies the index of the window on which the computation was performed;
- a is the address of the on-chain smart contract;
- :i is the signature of node i on the message.

This message will be used in the remaining phases to prove that the node really performed the computation without learning it from other intermediate nodes. At the end of the *proof of computation* phase, every intermediate node broadcasts to every other intermediate node the message $p_{t,i}$ and, when it receives the proof of computation of other nodes, it will append its own signature to it and send it back to the sender. Once an intermediate node has received the proof of computation of all the other intermediate nodes, the agreement checking phase starts. In this phase, every intermediate node broadcasts to the other nodes a message $c_{t,i}$ signed by itself:

$$c_{t,i} = [f(w_i[t])||S_{t,i}] : i \tag{2.2}$$

This message allows other intermediate nodes to know what is the result $f(w_i[t])$ computed by the intermediate node i for the t-th window. The message $c_{t,i}$ and the message $p_{t,i}$ allow intermediate nodes to prove whether node i was lazy or not. In fact, every intermediate node that receives a message $c_{t,i} = [f(w_i[t])||S_{t,i}] : i$ can hash its content $f(w_i[t])||S_{t,i}$ in order to check whether it agrees with the $p_{t,i}$ previously received or not, proving thus that the node has not been lazy.

Once a node has received every agreement-checking message and proof-of-computation message, he can begin the *smart contract update* phase. In the smart contract update phase, a node can perform a blockchain action by committing to the blockchain a proof that unanimous agreement has been reached on a given result $f(w_i[t])$. This is done by sending to the blockchain the proof-of-computation message signed by every intermediate node, and its own agreement-checking message $c_{t,i}$.

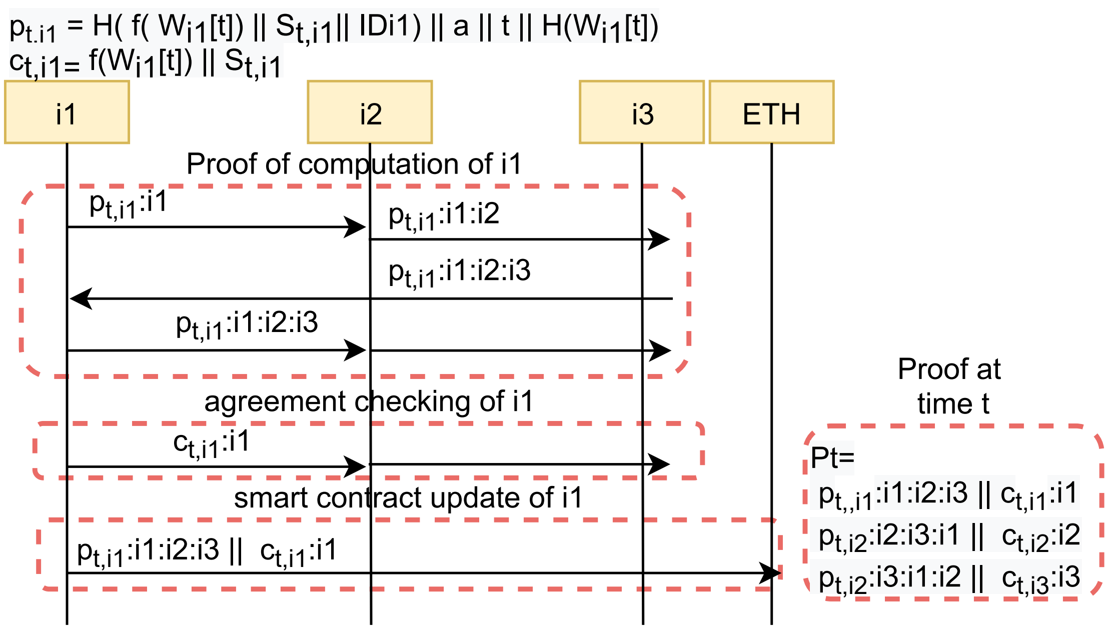

$p_{t.i1}$ = H(f(W_{i1}[t]) || $S_{t,i1}$|| IDi1) || a || t || H(W_{i1}[t])
$c_{t,i1}$= f(W_{i1}[t]) || $S_{t,i1}$
i1
i2
i3
ETH
Proof of computation of i1
$p_{t,i1}$:i1
$p_{t,i1}$:i1:i2
$p_{t,i1}$:i1:i2:i3
$p_{t,i1}$:i1:i2:i3
agreement checking of i1
$c_{t,i1}$:i1
smart contract update of i1
$p_{t,i1}$:i1:i2:i3 || $c_{t,i1}$:i1
Proof at time t
Pt=
$p_{t,,i1}$:i1:i2:i3 || $c_{t,i1}$:i1
$p_{t,i2}$:i2:i3:i1 || $c_{t,i2}$:i2
$p_{t,i2}$:i3:i1:i2 || $c_{t,i3}$:i3

FIGURE 2.6: Three phase protocol of Diversity

Chapter 3

Pollutant emission control system

This chapter presents an industrial IoT case scenario in which the use of blockchain technologies has been deemed appropriate for providing specific guarantees to the parties involved. Discussing this case study will make it clear how certain IoT case studies may benefit from the introduction of blockchains.
Furthermore, it also provides a useful starting point for understanding the kind of problems that *Dany* can address.

3.1 Pollutant emission control case study

The incineration of waste for producing energy significantly contributes to the total energy supply of the European Union. The waste-to-energy process that incinerators perform, involves the burning of waste products and this results in the generation of polluting substances.
Incinerators can have a significant impact on the dispersion of pollutants in the air, so their behaviour is regulated by law. In fact, the *Italian and european emission regulations* establishes thresholds on the concentration of *carbon monoxide* that can be emitted. The law limit values of carbon monoxide concentration are 150mg/Nm3 , 100mg/Nm3 and 50mg/Nm3 as an average value over 10 minutes, 30 minutes and 1 day.
These values are calculated using a normalization formula:

$$Es = \frac{21 - Os}{21 - Om} Em \tag{3.1}$$

in which Es is the concentration of emission calculated to the reference oxygen content; Em is the concentration of measured emission; Os is the reference oxygen concentration (11%) and Om is the measured oxygen concentration. In order to determine whether the incinerators are behaving correctly, meaning that their pollution emission rates are below the law limit values, it has been decided to measure their emissions every 5 seconds and using these measures it is possible to calculate the minute average emission (E_{min}) using the following formula:

$$E_{min} = \frac{\sum_1^n Es_i}{n} \tag{3.2}$$

in which Es_i is the instantaneous emission measured every 5 seconds, and n is the number of readings performed in a minute (12 in our case). The E_{min} value

thus calculated is used for implementing an *emission control policy* that will make sure that incinerators do not exceed the law limit values of pollution emissions. The measured values of carbon monoxide concentration is also of interest of an health authority body that will use the measured values in order to check when the law limit values are exceeded and place fines to the incinerator owners proportional to the number of times these values have been exceeded.
In this situation we have three entities that are interested in the same data, which are the *incinerator owner* the *health authority body*, and the *energy consumers* (i.e. industries) that are interested in certifying that the energy they use is green. Both the incinerator owner and the health authority have strong reasons for not trusting each other. In fact, the incinerator owner would gain an advantage by hiding or counterfeit readings showing that his incinerators have polluted more than it is permitted by law. On the other hand, the health authority may not be trusted by the incinerator owners since it would get an economic incentive from placing unjust fines. Legally, the reading data should be kept for 5 years, therefore a question arises: who should keep the reading data? In a first version of the system, a third party, which was the company that installed the sensors, had the role of managing the reading data and exposing them to the parties involved in the case scenario in order to allow them to perform their activities.
This solution was found inappropriate by the incinerator companies and the health authority that would prefer the use of a blockchain for its properties of immutability and decentralization: a blockchain solution would also allow external industries interested in the data to have the same guarantees on the data required by the health authority and the incinerator owners (Figure 3.1).

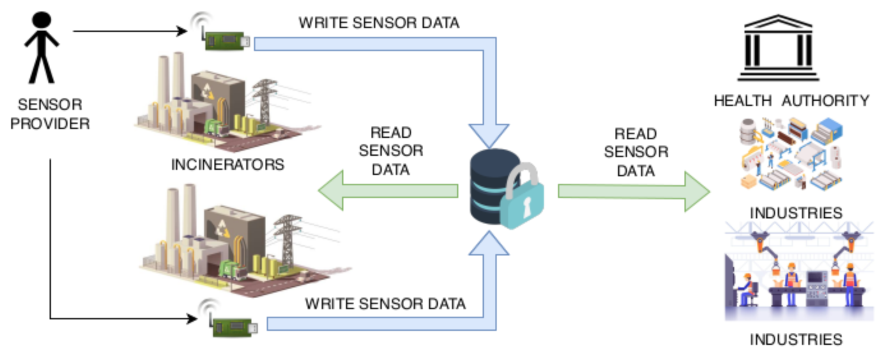


FIGURE 3.1: Graphical description of the pollution case study

3.2 Blockchain solution

Using a public blockchain, such as Ethereum, to hold the sensor reading data and to execute the smart contract responsible of fining the incinerator owners in case of misbehaviour, is unfeasible. In fact, monitoring a single incinerator would require committing to the blockchain around 500 transactions every 5 seconds. The transaction throughput of Ethereum is estimated to be around 10 tx/s, which is too low for our application. Even if we were to adopt other public distributed ledger with higher throughputs we would still incur in the problem of fees. Submitting transactions to public blockchains requires the submitter to pay a fee that is proportional to the computational efforts required to execute the code that the transaction triggers on the blockchain. We estimated that a single incinerator system would be responsible of producing 125TB of data in 5 years; committing this amount of data to a public blockchain, if possible, would cost millions of dollars. For these reasons, the solution involving the use of a public blockchain has been discarded, however, custom consortium blockchains can provide a solution to the problems inherent public blockchains; in fact, consortium blockchains can be configured to support high transaction rates without any fees.

3.3 Evaluation of a local blockchain solution with Hyperledger Besu

Hyperledger Besu [1] is an Ethereum Client that enables users to design their own blockchain solution. Hyperledger Besu supports various consensus protocols including Proof-of-Work protocols, proof-of-stake protocols and proof-of-authority protocols. In order to understand the feasibility of a permissioned blockchain solution to the problem of data management in the pollutant emission control case study, we have decided to use Hyperledger Besu for implementing multiple consortium blockchains with varying parameters in order to perform tests on them that would help us understand whether these blockchains would be able to handle the high transaction throughput that the pollutant emission control case study requires.

3.3.1 Blockchain configurations

The configurations of the blockchains vary in the following parameters:

- **Consensus protocol:** either IBFT 2.0 or Clique.
- **Validator number:** The number of nodes that are registered as validators in the blockchain (10, 15 and 20).

The *block gas limit*, which represents the total amount of gas that can be consumed in a given block, has been set to the maximum value allowed by Besu in every tested blockchain. The *block period*, which is the minimum amount of seconds that a validator has to wait in order to propose a new block, has been set to 1, which is the minimum allowed by Besu. Every node of the blockchains has been deployed on a single physical machine and cross-node communication was enabled via host networking. The host machine has the following characteristics:

- **Operating System:** Ubuntu 20.04.3 LTS,
- **CPU:** 2 CPU: Intel(R) Xeon(R) Gold 6256 CPU @ 3.60GHz,
- **RAM:** 128GB

3.3.2 Benchmark configuration

The benchmark runs were performed with Hyperledger Caliper, a blockchain performance benchmark framework. The tests performed with Caliper involved sending loads of transactions to a single validator node of the network at fixed rates. The send rates used in our tests are around 1, 5, 10, 20, 30, 40, 50, 100, 250 tx/s. Each combination of blockchain configuration and send rate was targeted once in a benchmark run, meaning that a total of 54 benchmark runs were performed.

The transactions were contract method calls calling the *open* method of the *simple.sol* smart contract (Code 3.1), provided by default by Caliper. The metrics obtained by running the tests are:

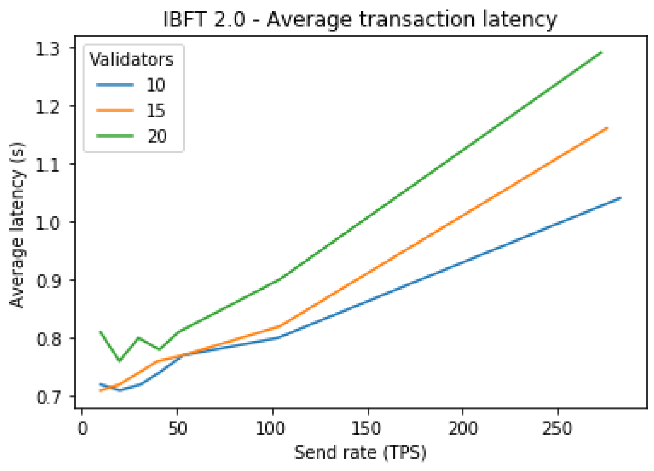
- **Transaction throughput (TPS):** the number of transactions successfully committed to the blockchain per second.
- **Transaction latency:** The amount of time a transaction takes from the moment it is sent to a node to when it is successfully committed to the blockchain.

Algorithm 3.1 Benchmark smart contract

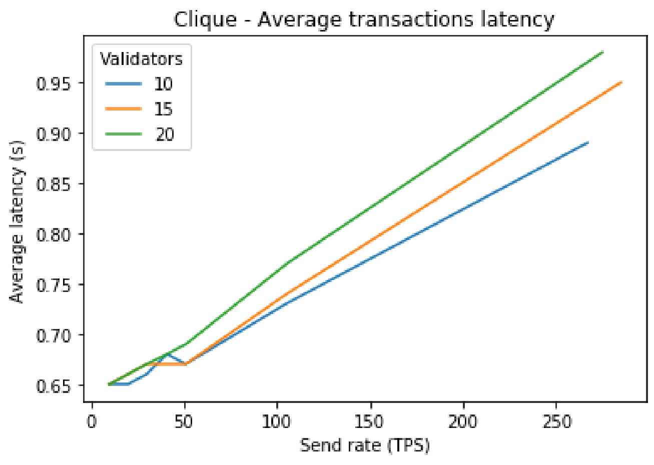
```
pragma solidity >=0.4.22 <0.6.0;

contract simple {
    mapping(string => int) private accounts;

    function open(string memory acc_id, int amount) public
        payable {
        require(int(msg.value) == amount, 'No commitment made
            by the caller.');
        accounts[acc_id] = amount;
    }

    function query(string memory acc_id) public view returns (
        int amount) {
        amount = accounts[acc_id];
    }

    function transfer(string memory acc_from, string memory
        acc_to, int amount) public {
        accounts[acc_from] -= amount;
        accounts[acc_to] += amount;
    }
}
```

3.3.3 Benchmark results

The results obtained from the benchmark runs are shown in Figures 3.2a 3.2b 3.3a 3.3b. It is possible to observe that both the average transaction throughput and transaction latency are affected by the number of validators in the network, with performance decreasing as the number of validators of the networks increase. It



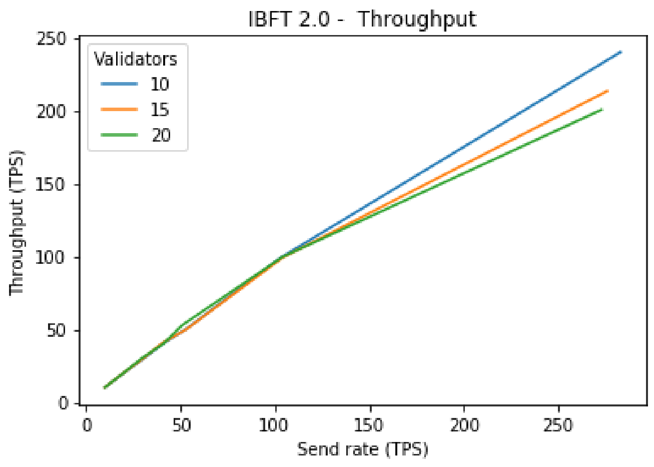
(A) IBFT 2.0.



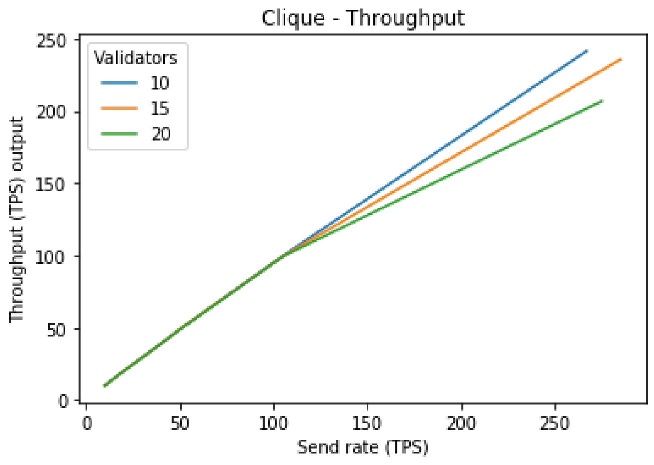
(B) Clique.

FIGURE 3.2: Average transaction latency measured in IBFT 2.0 (A) and Clique (B) blockchains with varying send rates and validators number.

is also possible to observe that the average transaction latency increases as the send rates increases, even though it is always very low (lower than 1 second). The transaction throughput of the blockchains tends to approximate the benchmark run transaction send rate, with a slight decrease in performances as the send rate increases up to around 250 tx/s. Unfortunately, it was not possible with our setting to use higher send rates in order to reach an upper limit on transaction throughput.



(A) IBFT 2.0.



(B) Clique.

FIGURE 3.3: Average throughput of IBFT 2.0 (A) and Clique (B) blockchains with varying send rates and number of validators.

3.4 Evaluation of a distributed blockchain solution using Hyperledger Besu

In the work we published *Performance analysis of a BESU permissioned blockchain* [45], we gained additional information about the performance of real-world consortium blockchains. In particular, we tested a real permissioned blockchain having four validator nodes dislocated in different regions of Italy using Hyperledger Caliper and a Custom benchmark tool developed by us. The custom benchmark tool we designed allowed us to gain insight into the abnormal behaviours detected that we couldn't inspect with Hyperledger Caliper.

3.4.1 Blockchain configuration

The blockchain network tested is built up as a private permissioned network. Permissioning is implemented by the Consensys Dapp mechanism where new nodes can be onboarded if added into the permissioning smart contracts by administrators. Administrator can be registered as well and can manage the inclusion of new nodes or new administrators with their cryptographic credentials. The consensus protocol used is IBFT 2.0 with four validator nodes. Nodes are run by different and independent organizations which may adopt different access policies and firewalls and different machines and software. They are geographically distributed in different sites so that internet interaction can play a true role in the exchange of information among them. Every organization implements a node by means of the same docker-container technology but BESU versions run by the nodes can be different.

3.4.2 Hyperledger Caliper

Hyperledger Caliper [2] is a blockchain performance benchmarking tool for Hyperledger Besu and other Ethereum-like blockchain technologies. Caliper allows to set up a workload specifying the nodes to which send requests (via web socket) and information about the transaction load. Transaction loads are completely customizable, allowing the user to define the fields of the transactions that will be sent, as well as the number of transaction in the load and the send rate: the number of transaction to submit per second. The metrics measurable with Caliper are:

- Read latency: the time elapsed between a request is submitted and a reply is received;
- Read througuput: the total amount of read operations successfully submitted for which a reply has been received per second;
- Transaction latency: the amount of time a transaction requires to be part of the blockchain;
- Transaction througuput: the rate at which valid transactions are committed to the blockchain.

3.4.3 Custom benchmark tool

The custom java benchmark tool we developed allows us to send custom ethereum transactions to a node at a steady configurable rate and retrieve various metrics. The application allows us to configure various parameters to be used in a benchmark run, including:

- *Transaction send delay*: Number of milliseconds the application waits in order to send a new transaction after sending the previous one;
- *Transaction number*: number of transactions to be sent in a given run;
- *Timeout*: the number of milliseconds after which a transaction is considered to be not included in the blockchain if its transaction receipt hasn't been received;

- *Communication protocol*: The protocol employed for communicating with the node (either HTTP or Web socket);
- *Node address*: The node IP address and port number used to communicate with the blockchain node;
- *Sender key*: Ethereum account private key used for signing transactions.

The transactions that can be sent by the application are either simple value transfer transactions or smart contract method calls, more specifically, the smart contract method call that the application natively supports is the transfer method of the ERC20 smart contract. In case the application is run in the smart contract method call mode, the application will first commit a new ERC20 smart contract, wait for its inclusion in a block, and then will start sending the method call transactions. After a benchmark run, the application outputs the average transaction delay – which is the average number of milliseconds elapsed from when a transaction has been sent to when it has been included in a valid block and the number of blocks that have been generated during the run. Furthermore, for every transaction committed successfully, the application outputs information regarding the transaction and the block that contains the transaction. This information contains: block number; the timestamp when the transaction was sent to the node; the timestamp of the block; the ethereum address of the validator that proposed the block; the transaction delay.

3.4.4 Benchmark tests performed

The tests performed involved the use of Hyperledger Caliper and of the Custom benchmark tool. Both tools were used to send loads of transactions targeting a particular smart contract deployed on the network at fixed rates. The smart contract targeted with Caliper is the *simple.sol* (code 3.1 smart contract that comes by default with the tool, while the smart contract called with the custom benchmark tool is the IERC20 smart contract, and the method called was the *transfer* method.

Caliper results

Hyperledger Caliper shows measured metric values for three different blockchain interactions: *open*, *query*, and *transfer*. The *open* and *transfer* tests are performed with gas-consuming transactions to the smart contract. The *query* test is a call to the view function of the smart contract. Transactions were sent for a programmed time of 100 seconds at the different send rates of 0.2, 1, 2, and 10 transactions per second (TPS). An additional test at the rate of 100 transactions per second was set up but was not carried out by the tool. Caliper metric results are summarized in Tab. 3.1 where we report the results of the *transfer* test.

The results show how the average latency time of the transactions tends to increase with the increase of the send rate. This value should not depend on the number of transactions and should be close to half the block time (which in our case is 6 seconds). However, it seems that the throughput is close to optimal.

TABLE 3.1: Results of the execution of Hyperledger Caliper tests at four different transaction send rates. Only the results relating to the *transfer* tests are reported.

Tx Send Rate (TPS)	0.2	1	2	10
Maximum latency (s)	5.20	6.66s	12.39	13.42
Minimum latency (s)	0.23	0.64	0.69	0.97
Average latency (s)	2.70	3.58	6.41	7.23
Throughput (TPS)	0.2	0.9	1.9	9.5

TABLE 3.2: The blocks metrics of the test 1 are reported in each column of the table for a different value of the transaction sending rate.

Tx Send Rate (TPS)	0.2	1	2	10	100
Number of Tx	20	100	200	1000	10000
N. of Blocks	16	17	19	19	24
Total Block Time (s)	90	96	108	108	139
Average block time (s)	6.0	6.0	6.0	6.0	6.043

Caliper therefore allows us to detect an anomaly in latency metrics but does not provide further information to understand finer details.

Custom tool results

We configured the Custom tool to send transactions using WebSocket as a protocol, to use the closest node (in our case, the local host) as destination, and to use a single sender key. We set proper values for the parameters in order to have

$$\textit{transaction number}/\textit{transaction send delay} = 100$$

in other words, transactions were sent for a programmed time of 100 seconds. Different values for *transaction send delay* were set in order to send transactions at the different send rates of 0.2, 1, 2, 10, and 100 transactions per second (TPS). We performed the test twice (i.e., test 1 and test 2 in the following) to verify the repeatability of the observations. After running the tests, our custom tool produces the results in the form of serialized data on a file that can be easily processed. All of the data shown below are the results of processing the custom tool's data without the need for any additional queries to the blockchain.

Tab. 3.2 and Tab. 3.3 report the block metrics of the two tests performed with the same configuration of the tool. In these tables we can read for each send rate: the number of blocks created during the test; the total time elapsed between the first and last block created; and the average block time.

Thanks to the reports generated by our Custom Java Benchmark tool, we can understand how the blockchain network acquires transactions and how and when these are inserted into new blocks. For both tests, the average block time is stable for any TPS (6 seconds).

Table 3.3: The blocks metrics of the test 2 are reported in each column of the table for a different value of the transaction sending rate.

Tx Send Rate (TPS)	0.2	1	2	10	100
Number of Tx	20	100	200	1000	10000
N. of Blocks	17	18	18	19	25
Total Block Time (s)	96	102	102	108	144
Average block time (s)	6.0	6.0	6.0	6.0	6.0

Table 3.4: Test 1. The values of the transaction confirmation delays are reported in each column of the table for a different value of the sending transaction frequency parameter.

Tx Send Rate (TPS)	0.2	1	2	10	100
Maximum latency (s)	5.99	6.289	12.235	13.001	22.112
Minimum latency (s)	0.897	0.190	0.402	0.337	4.720
Average latency (s)	3.405	3.206	5.246	6.759	12.638
Throughput (TPS)	0.208	0.997	1.779	8.927	69.136

The tool allows the evaluation of latency metrics and to compare the results with the measurements made by Caliper. The transaction latency (i.e., the time interval between the instant of sending the transaction and the instant of confirmation) allows us to verify when the transactions are correctly inserted in the blocks. The latency is expected to be a local maximum if the transaction is requested immediately after creating a block. Conversely, minimal latency is expected if the transaction is sent close to the creation of a new block. Furthermore, an average latency of close to 3 seconds is expected for each test (or half of the block time established during the blockchain's genesis, which in our case is 6 seconds). As for the number of transactions per block, each block is expected to contain the same number of transactions, except for the first block and the last block. Any noticeable difference between expected and measured data is worth further investigation and can be traced back to network node operational anomalies.

The Tab. 3.4 and Tab. 3.5 show the latency measures of test 1 and 2 respectively. The results confirm the values of the Hyperledger Caliper metrics obtained for the transaction send rates up to 10 TPS.

In both tables, for rates up to 10 TPS, the minimum delay remains below one second. At a rate of 100 TPS the minimum delay increases considerably up to 4.7 and 2.8 seconds for test 1 and test 2 respectively. For the rates of 2, 10, and 100 TPS, a significantly higher than expected average latency is observed, and, for both test 1 and test 2, the average latency increases with higher send rates.

Therefore, an anomaly is found for the transaction send rates of 2 transaction per second and above. The data produced by our Custom benchmark tool allows the examination, transaction by transaction, of the latency, the block number, and the block's miner. This allows for delving into the reasons for the increasing in the latency. To investigate the causes of this anomaly, we consider the smallest sending rate that causes anomalies and the highest send rate, i.e. 2 TPS and 100

TABLE 3.5: Test 2. The values of the transaction confirmation delays are reported in each column of the table for a different value of the sending transaction frequency parameter.

Tx Send Rate (TPS)	0.2	1	2	10	100
Maximum latency (s)	6.917	6.664	12.196	12.774	18.528
Minimum latency (s)	0.833	0.567	0.289	0.377	2.835
Average latency (s)	3.781	3.596	5.092	6.580	10.712
Throughput (TPS)	0.199	0.960	1.855	8.879	66.454

TPS. By looking at the data provided by our tool, we were able to understand what caused these anomalies. In fact we could observe that with the send rate of 2 TPS what happens is that two validators seem to propose blocks with more transactions than the other two validator (Figure 3.4), the situation became even worse in the 100 TPS scenario, since the validators that were producing smaller blocks in the previous test were now producing empty blocks (Figure 3.5).

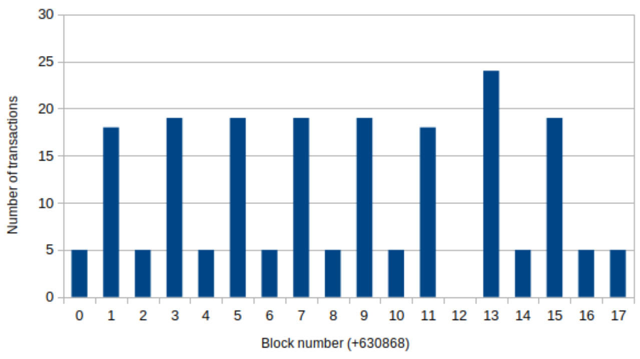


FIGURE 3.4: Number of transactions per block in the test 1 at a transaction send rate of 2 TPS.

3.5 Discussion

The results observed from our benchmark runs performed in the locally deployed blockchain suggests that a permissioned blockchain can be used for managing the data generated by a single incinerator. However, these results were obtained in a synthetic environment in which network delays are not taken into consideration; performances are likely to degrade in a real-world scenario where nodes of the network are geographically distant and communication between them is not instantaneous. It is also not clear what would be the maximum number of incinerators our blockchain solution can handle, since in our setting we were limited to sending transaction at a maximum rate of around 250 tx/s, which is not enough for reaching saturation in any of the explored blockchain configurations.
A more extensive performance analyis on Hyperledger Besu used for private blockchains has been conducted by Caixan Fan et al. [30]. Their testes consisted, simarly to ours, in sending loads of transactions with Caliper to Besu blockchain nodes that

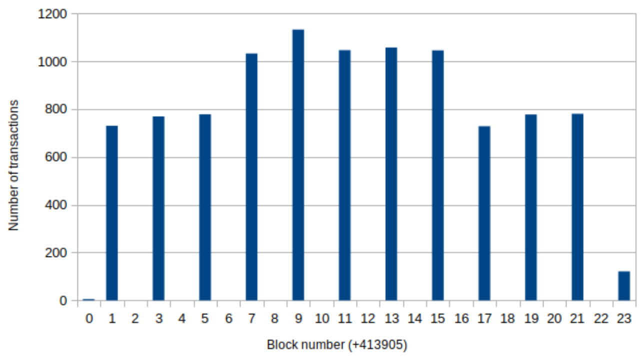


FIGURE 3.5: Number of transactions per block when sending transactions at 100 TPS.

were deployed on the same machine. Their setting enabled them to send up to 1,500 transaction per second addressing the same *simple.sol* smart contract used in our tests. The maximum throughput reached by the most performant blockchain configuration was less than 500 tps, with degrading performances as the complexity of the computation triggered by the transactions increased. These results suggest that a straightforward consortium blockchain solution to the pollution management case scenario is not ideal if the number of incinerators to be monitored is large. The tests we performed on the real-world permissioned blockchain suggests that in real world scenarios the high throughput of permissioned blockchains detected in a synthetic environment may not be achievable. These results suggest us that devising a scalable solution for executing smart contracts in case scenarios where the number of transactions to be processed is high, would provide a suitable solution for these case scenarios.

Chapter 4

Dany

This chapter describes *Dany* (**D**ecentralized st**A**te cha**N**nel s**Y**stem), a solution for executing secure and scalable smart contracts particularly suited for the IoT. Smart contracts that require the elaboration of continuous streams of data cannot be implemented in a straightforward approach in many blockchains, in fact, such smart contracts would put an unnecessary burden on the blockchain. The data processed by a blockchain smart contract is embedded in transactions and the throughput of blockchains is often quite low, making it impossible to process all the incoming data. This is especially true in public blockchains since they tend to have lower throughputs than private ones, making them unsuitable for executing IoT smart contract, especially if multiple similar smart contracts are deployed in the same chain. Another problem faced by IoT smart contract is related to transaction fees: sending transactions to a blockchain smart contract, such as an Ethereum smart contract, costs fees that are generally proportional to the amount of data sent and the computational power required to execute the triggered smart contract procedure. In order to solve these problems it is possible to off-load the execution of the smart contract from the *blockchain layer* to another layer that will interact with the blockchain in few occasions. The solutions that implement this approach are called either *off-chain solutions* or *second-layer solutions*. Second-layer solutions, however, are not to be implemented naively since they should provide security guarantees similar to those of traditional smart contracts while providing at the same time scalability improvements. With *Dany* we provide a second-layer solution in which the computation of smart contracts is off-loaded to a second-layer that we call the *intermediate node layer*. The intermediate node layer is composed of devices (intermediate nodes) that are required to execute a smart contract that processes data coming from a third layer called *sensor layer*. Smart contracts state updates will be performed off-chain only if a unanimous agreement can be reached by the intermediate nodes. If a unanimous agreement cannot be reached, then a dispute resolution protocol allows them to prove to the blockchain what is the correct state. This means that Dany provides an *any-trust* guarantee, that is a single honest intermediate node is enough for enforcing the correct execution of the smart contract.

The solution described in this chapter is a further development of Diversity, that was first presented in *Cacciagrano et al* [15].

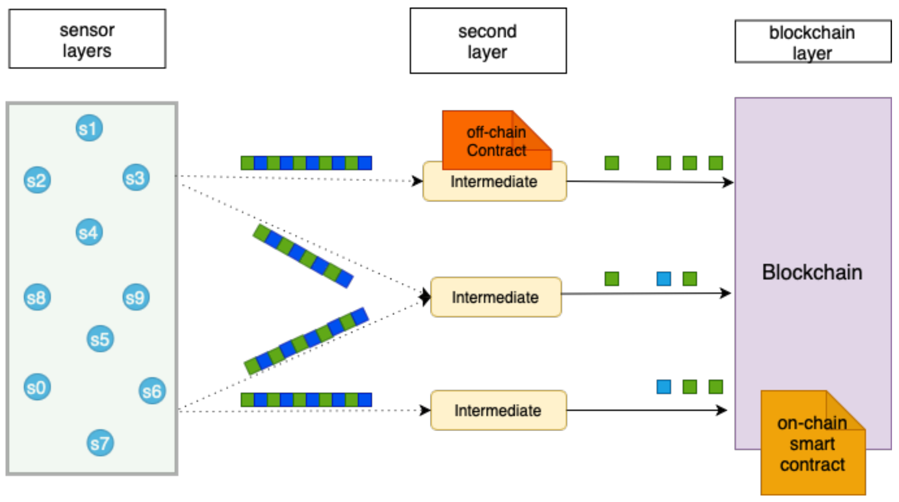


FIGURE 4.1: Dany architecture

4.1 Dany at glance

Figure 4.1 shows the architecture of the Dany system. This has a three layered architecture that is composed of the following layers: (i) a main blockchain; (ii) a set of distributed intermediate nodes; (iii) a set of IoT sensor devices.

The blockchain layer can contain one or more on-chain smart contracts. The use of Dany requires that a set of users specifies a smart contract. This specifies a logically centralised state machine that takes as an input a stream of sensor data and performs a transition from its current state to the next one. This transition may cause an action to be performed by using the on-chain smart contract. This can move funds on the reference blockchain.

Each user that has been involved in the smart contract definition can propose an intermediate node. Intermediate nodes constitute a distributed layer of devices that read the on-chain smart contract and execute it in a decentralised fashion. In the rest of the thesis, the local copy of the on-chain smart contract is referred to as off-chain smart contract. The basic idea is that the intermediate nodes should simulate the state machine that is defined in the on-chain smart contract in a decentralised fashion (this decentralised execution defines the state channel). More precisely, a Dany secure protocol ensures that the local state of each honest intermediate node will be the same as the logically centralised one.

The sensor layer is composed by a set of sensors that forward data to the intermediate nodes. Sensors are assumed to be trusted. The origin of sensor messages can be always authenticated and their integrity proved.

In the next sections we explore all the layers in detail.

4.2 Sensor layer

The *sensor layer* is composed of devices that we call *sensors*. A sensor in Dany is assumed to be a device with a small amount of memory and limited CPU computation capabilities. A sensor has connectivity and cryptographic capabilities since is able to send signed messages to the *Intermediate nodes.* Sensors in Dany are assumed to be *trusted*, which implies the following:

- Sensors do not have a byzantine behaviour, meaning that a sensor does not send different readings to different *intermediate nodes.*
- Sensors cannot be tampered with (e.g., they are tamper-proof devices).

Sensors are referenced by Dany smart contracts stored in the *blockchain layer* and send messages to every *intermediate node* referenced in those same contracts. The *sensor layer* can be considered a trusted source of information. Dany assumes that the connection between the sensors and the intermediate nodes is *unreliable*, meaning that sensor messages can be lost during transmission. In fact, we designed Dany in such a way that it should be able to provide *liveness* and *safety* even in the case of unreliable sensor message communication.

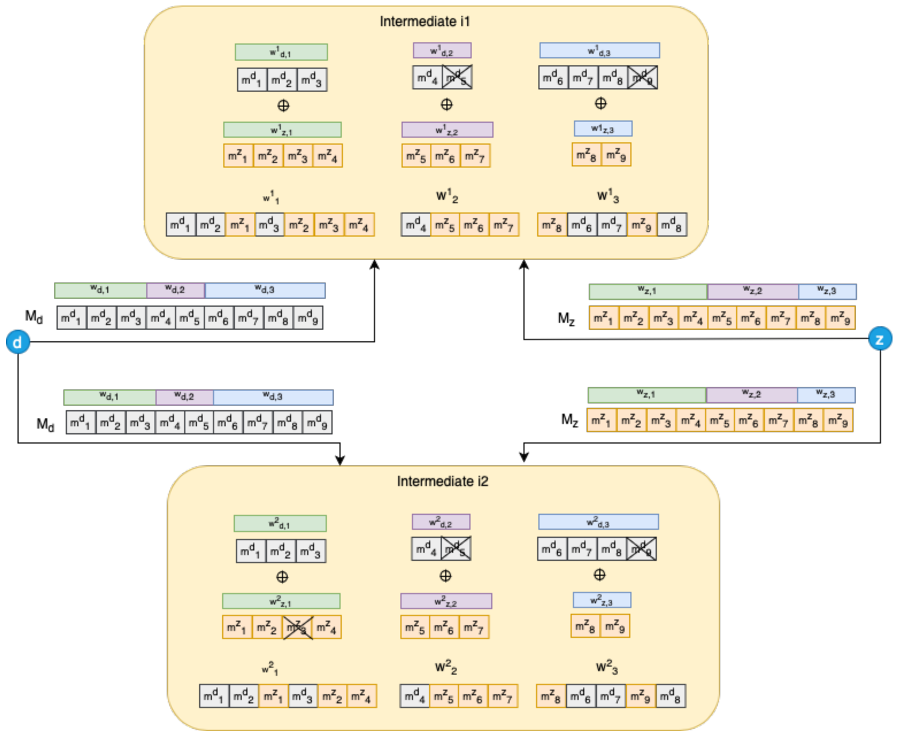


Figure 4.2: Windows and traces generated by sensors and received by intermediate nodes: Sensors d and z generate the traces M_d and M_z respectively. These are received by intermediate nodes $i1$ and $i2$ who divide them into windows and merge them. Intermediate nodes may lose some sensor messages, possibly making their merged local windows different, e.g., w_1^1 is different from w_1^2

.

4.2.1 Sensor messages and windows

A message sent by a sensor d to an intermediate node can be described as a tuple:

$$m_{d,q} = \langle id_d, q, data_{d,q}, t_q, type_d \rangle \sigma_d \tag{4.1}$$

where:

- q is a sequence number;
- id_d: is the public key of the sensor, or any other information that can identify univocally the sensor;
- $data_{d,q}$: is the data payload sent by the sensor;
- t_q: is the timestamp when that sensor message was sent;
- $type_d$: represents the type of data transmitted by the sensor (e.g., humidity, temperature, etc..);
- σ_d: is the signature of sensor d.

A sensor generates a **sensor trace** that is a stream of sensor messages. We use r_d to denote the sensor trace generated by d and $m_{d,0} \cdot \ldots \cdot m_{d,q} \cdot m_{d,q+1}, \ldots$ denotes the sensor messages in r_d. We specify here that each message contained in a stream of sensor messages r_d is an element of M_d, with M_d being the set of all possible messages coming from the sensor d. The sensor messages are broadcast to a set of intermediate nodes $\{i_1, \ldots, i_m\}$. These use the trace r_d to run all smart contracts that include the sensor d. It is worth mentioning that the timestamp t_q of a message $m_{d,q}$ can also be used for ordering the sensor messages obtaining a trace that is r_d. For the sake of presentation we often use $m_{d,t}$ to denote the message $m_{d,q}$ that is we replace the sequence number q with its timestamp t. Figure 4.2 shows the sensor traces r_d and r_z that are related to the sensors d and z. These are sent to the intermediate nodes i_1 and i_2.

Traces coming from different sensors can be merged into a single one. Merged traces are used by the intermediate nodes for executing Dany smart contracts.

Definition 1 *Let r_d and r_z be two sensor traces that are related to the sensors d and z. A merged trace that is denoted with $r_d \oplus r_z$ is a sequence of sensor messages such that:*

1. *each message in the traces r_d and r_z appears exactly once in the merge $r_d \oplus r_z$;*
2. *a message m_{d_k,t_k} is in the merge $r_d \oplus r_z$ iff m_{d_k,t_k} is either in r_d or r_z;*
3. *if m_{d_j,t_j} and m_{d_k,t_k} are two messages in $r_d \oplus r_z$ and $t_j < t_k$ then m_{d_j,t_j} appears before m_{d_k,t_k} in the merge;*
4. *if m_{d_j,t_j} and m_{d_k,t_k} are two messages $r_d \oplus r_z$ with $t_j = t_k$ and $id_{d_j} < id_{d_k}$ then m_{d_j,t_j} appears before m_{d_k,t_k} in the merge $r_d \oplus r_z$;*

We can generalise the merge operation to a set of sensor traces. Suppose that we have a set of sensors D and $d_1, d_2, ...d_h$ are element in D. Then we denote with $r_D = \bigoplus_{k=1}^{h} r_{d_k}$ $(d_k \in D)$ the merge of all sensor traces. We use r instead of r_D when the set of sensors D is clear from the context.

We also define R_d as the set of all possible traces that can be generated by a sensor d. Let M_d be the set of all sensor messages that can be generated by sensor d, R_d can be defined as follows:

$$R_d = M_d^* \tag{4.2}$$

Where $*$ is the kleene star operator. From this definition, we can define R_D, which is the set of all possible merged sensor traces that can be generated by the sensors in a set $D = \{d_1, d_2...d_h\}$:

$$R_D = \{\bigoplus_{i=1}^{|D|} tr_i : tr_i \in R_{d_i}\} \tag{4.3}$$

As we are going to see in section 4.3, Dany smart contracts are defined over finite sub-strings of traces. We denote with $R_{D,n}$ the set of all possible substrings of the traces that can be generated by the nodes in D with a maximum lenght n.

4.3 Blockchain layer

The Dany system assumes that a set U of users $\{u_1, u_2, ...u_m\}$, are interested in running a state channel whose state transitions depend on the messages on the traces produced by a set of sensors $D = \{d_1, d_2, ...d_h\}$. The behaviour of the state channel can be described by a state machine that given a certain input returns a new state, which in our case will represent both the amount of money owned by every user in U, and a numeric value that we call *state variable*. The amount of cryptocurrency of the channel owned by every user in U is called *redistribution*. A redistribution e_U to a set of users $U = \{u_1, u_2, ...u_m\}$ can be described as an m-tuple:

$$e_U = \langle (u_1, a_1), (u_2, a_2), ...(u_m, a_m) \rangle \tag{4.4}$$

where each element a_i is a positive integer representing the amount of cryptocurrency owned by user u_i.

In order to instantiate a state channel, it is necessary to deploy first a *Dany* smart contract SC on a blockchain that specifies both the state machine, as well as the information about the the windows to be used as input to the transition function of the state machine, and the devices responsible of physically running the state channel.

A Dany smart contract SC can be described as a tuple:

$$SC = \langle I, D, K, T_0, T_w, N_w, SM \rangle \tag{4.5}$$

where:

- I: is a set of intermediate nodes $I = \{i_1, i_2, ...i_n\}$, responsible of running the state channel;
- D: is a set of sensors;
- K: is a positive integer representing the stake, which is the amount of cryptocurrency locked used to give an incentive to intermediate nodes to behave honestly;
- T_0: is the timestamp at which the state channel execution starts;
- T_w: is the window duration: windows of sensor messages are the input of the transition function;
- N_w: is the number of windows the state channel processes;
- SM: is the state machine specifying the behaviour of the state channel.

The state machine SM can also be described as a tuple $SM = \langle (S{\times}E), (s_0, e_0), R_{D,n}, F\rangle$ where:

- $S \times E$ is a set of states, where S is a set of values and E is a set of redistributions;
- (s_0, e_0) is the *initial state* of the state channel, with $s_0 \in S$ and $e_0 \in E$;
- $R_{D,n}$ is the *input alphabet*, which is the set of all possible merged sensor traces of sensors in D with maximum length n;
- F: it is the *transition function*, mapping elements in $S{\times}E{\times}R_{D,n}$ to elements in $S \times E$.

The state channel execution specified by the Dany smart contract is performed by the intermediate nodes in I: these devices should be receiving messages from the sensors in D that they will use to assemble windows of sensor messages to be used as the input of the state machine transition function in order to make the state channel transition from one state to another. A window $w(r_d, T_0, T_w, n)$ is the n-th substring of a sensor trace r_d where T_0 and T_w are the times included in the state machine definition. $w(r_d, T_0, T_w, n)$ contains all messages $m_{d,t}$ such that $T_0 + n \times T_w \leq t < T_0 + (n+1) \times T_w$. For the sake of presentation we denote a window of sensor messages with w_n when r_d, T_0 and T_w are clear from the context. Ideally, the state channel execution specified by a Dany smart contract starts at time T_0, waiting until time $T_0 + T_w$ is reached in order to allow intermediate nodes to be able to assemble the first window $w(r_d, T_0, T_w, 0)$ which will be used by them to execute a protocol that will result in the state channel transitioning from a state (s_0, e_0) to (s_1, e_1) with $(s_1, e_1) = F((s_0, e_0), w(r_d, T_0, T_w, 0))$. As we are going to see in the next section the intermediate nodes simulate a state change (s_1, e_1) when a proof of unanimous agreement of intermediate nodes could be generated for it or when the on-chain Dany smart contract accepts the state (s_1, e_1) after an on-chain dispute has been settled. Once the state channel has transitioned to a state (s_1, e_1), then another T_w period is waited in order to transition to the following state. This is done a number of times equal to N_w.

4.4 Intermediate node layer

The intermediate node layer is composed by devices called *intermediate nodes*. These devices run the state channel specified by a Dany smart contract SC.

Since state channels in Dany update their states according to the messages produced by a set of referenced sensors, the intermediate nodes running a Dany state channel are required to be able to receive messages from the referenced sensors.

It is worth recalling that the connection between the sensor layer and the intermediate node layer is not assumed to be reliable, meaning that messages may be lost during transmission from a sensor to an intermediate node. Because of this, an intermediate node i_k may have received only a portion of the messages in the sensor trace r_d.

We denote with r_{d,i_k} the trace of sensor messages coming from sensor d that have been received by node i_k, and with r_{D,i_k} the merged trace of sensor traces produced by sensors in D that has been observed by node i_k; we call this the *intermediate node trace* of i_k.

However, even though any intermediate node may only have partial knowledge of the trace generated by the sensors, the Dany system will ensure that the transition from a state (s_n, e_n) to (s_{n+1}, e_{n+1}) will be the result of the application of the transition function to the window of data of the trace r_D, that is ($(s_{n+1}, e_{n+1}) = F((s_n, e_n), w_n)$) assuming that every message in w_n has been received by at least one honest intermediate node. However, even this assumption is quite strong and may not hold in many situations. It may happen, for instance, that some sensor messages are never received by any intermediate node, making it impossible for them to perform a state transition that depends on r_D.

In these situations we believe that the best solution is to guarantee that the state channel transitions to a new state by applying the transition function to the merge of the local windows observed by the intermediate nodes. This means that if a message in r_D that should be in $w(r_D, T_0, T_w, n)$ is not received by any intermediate node, then the next state of the state channel (s_{n+1}, e_{n+1}) will be guaranteed to be equal to $F((s_n, e_n), \omega_n)$, with ω_n being the merge of all the local windows that the intermediate nodes used to transition to a new state:

$$\omega_n = \{\bigoplus_{k=1}^{|I|} w_{D,i_k} : i_k \in I\} \tag{4.6}$$

We can also write ω_n as $w(r_{D,I}, T_0, T_w, n)$ by defining $r_{D,I}$ as the merge of the local traces of the intermediate nodes in I used to transition to a new state.

$$r_{D,I} = \{\bigoplus_{k=1}^{|I|} r_{D,i_k} : i_k \in I\} \tag{4.7}$$

Intermediate nodes will ensure that the state channel transition to a reasonable state by running a protocol among them aimed at proving a certain state has been agreeed by every honest itermediate node: we call this protocol the *Dany protocol*.

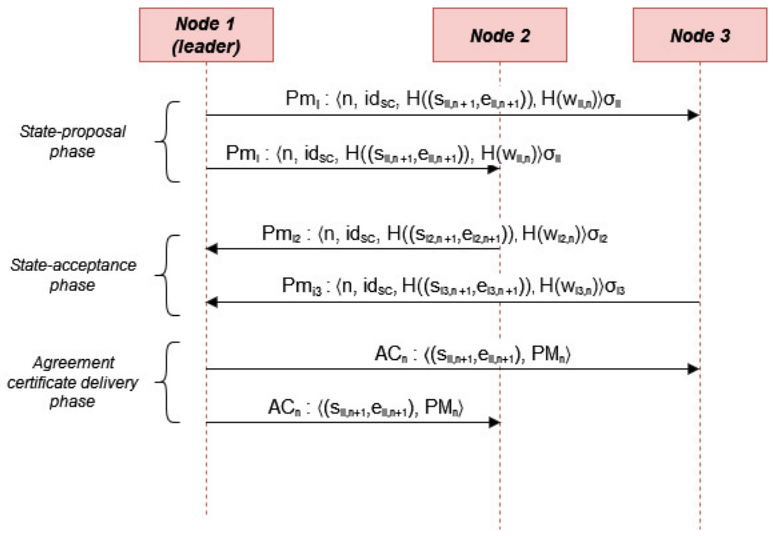

Node 1 (leader)
Node 2
Node 3
State-proposal phase
$Pm_l : \langle n, id_{SC}, H((s_{il,n+1}, e_{il,n+1})), H(w_{il,n}) \rangle \sigma_{il}$
$Pm_l : \langle n, id_{SC}, H((s_{il,n+1}, e_{il,n+1})), H(w_{il,n}) \rangle \sigma_{il}$
State-acceptance phase
$Pm_{i2} : \langle n, id_{SC}, H((s_{i2,n+1}, e_{i2,n+1})), H(w_{i2,n}) \rangle \sigma_{i2}$
$Pm_{i3} : \langle n, id_{SC}, H((s_{i3,n+1}, e_{i3,n+1})), H(w_{i3,n}) \rangle \sigma_{i3}$
Agreement certificate delivery phase
$AC_n : \langle (s_{il,n+1}, e_{il,n+1}), PM_n \rangle$
$AC_n : \langle (s_{il,n+1}, e_{il,n+1}), PM_n \rangle$

FIGURE 4.3: Message sequence showing the process of creating and distributing an Agreement certificate with Node 1 acting as the leader node for the given window

We refer to this protocol as the *Dany protocol*, which is divided into two sub-protocols, nameley the *Agreement certificate protocol* and the *Dispute resolution protocol*.

4.5 Agreement certificate protocol

The agreement certificate protocol 4.3 is the protocol run by the intermediate nodes aimed at generating a certificate proving that every intermediate node agrees on a given state (s_n, e_n): we call this proof an *agreement certificate*. Agreement certificate allow state updates to be performed off-chain without interacting with the blockchain: as long as an agreement certificate exists for a given window, intermediate nodes can be sure that the off-chain state they reached an agreement on can be enforced on-chain and that the related redistribution of funds can be performed. In order to generate such a certificate, a leader is selected each time the local time reaches $T_0+(T_w \cdot n)+\delta_w$, with δ_w being the maximum tolerable delay. Leader selection can be done in a round robin fashion without any exchange of messages. The leader i_l thus selected will then propose a new state channel state by applying the state machine transition function to the previous state previously agreed and to its local window w_{n,i_l} obtaining thus:

$$(s_{n+1,i_l} e_{n+1,i_l}) = F((s_{n,i_l} e_{n,i_l}), w_{n,i_l}) \tag{4.8}$$

We refer to this state as the leader node *proposed state* at time n.

After the intermediate node has completed this stage, he can propose its locally computed state to the other intermediate nodes by assembling a message with the following format:

$$pm_l = \langle n, id_{SC}, H(F((s_{n,i_l} e_{n,i_l}), w_{n,i_l})), H(w_{n,i_l}) \rangle_{\sigma_{i_l}} \tag{4.9}$$

where:

- $H(F((s_{n,i_l}e_{n,i_l}), w_{n,i_l}))$ is the hash of the state computed by node i_l on its n-th local window;
- $H(w_{n,i_l})$ is the hash of the n-th local window of i_l;
- n is the number of the window;
- id_{SC} is a reference to the Dany smart contract on-chain;
- σ_{i_l} is the signature applied by the leader node.

We refer to this kind of message as a *protocol message.*

The leader protocol message is broadcasted to every other intermediate node referenced by the Dany smart contract, initiating thus the *state proposal phase* of the protocol. Every intermediate node i_k will perform the following checks upon receiving such a message:

- Check whether the hash of the local state matches its own computed state for smart contract SC at window number n:
$H(F((s_{n,i_l}e_{n,i_l}), w_{n,i_l})) = H(F((s_{n,i_k}e_{n,i_l}), w_{n,i_k}))$.
- Check whether the hash of the window used to compute the new state is equal to the hash of its local window: $H(w_{n,i_l}) = H(w_{n,i_k})$.

If the checks pass, then each intermediate nodes i_k will send back its own protocol message to i_l containing the same values sent by i_l but with the signature of i_k: we refer to this protocol phase as the *state acceptance phase*

Once the leader node has received the protocol messages of all the other intermediate nodes he can create the agreement certificate.

An agreement certificate AC_n, proving a unanimous agreement has been reached by every intermediate node, can be described as follows:

$$AC_n = \langle (s_{n+1,i_l}, e_{n+1,i_l}), PM_n \rangle \tag{4.10}$$

where PM_n is a set of protocol messages such that:

- Every protocol message in PM_n have the same values for every field, except for the signature;
- Each protocol message in PM_n is signed by a different intermediate node;
- Every intermediate node in I has signed exactly one protocol message in PM_n
- for every protocol message $pm_{i_k} = \langle H(F((s_{n,i_k}e_{n,i_k}), w_{n,i_k})), H(w_{n,i_k}), n, a \rangle_{\sigma_{i_k}}$ in PM_n it is true that: $H(F((s_{n,i_k}e_{n,i_k}), w_{n,i_k})) = H(s_{n+1,i_l}, e_{n+1,i_l})$

Once the agreement certificate has been assembled by the leader node, he will broadcast it to all the other intermediate nodes: this is the last phase of the agreement certificate protocol and we refer to it as the *agreement certificate delivery phase.*

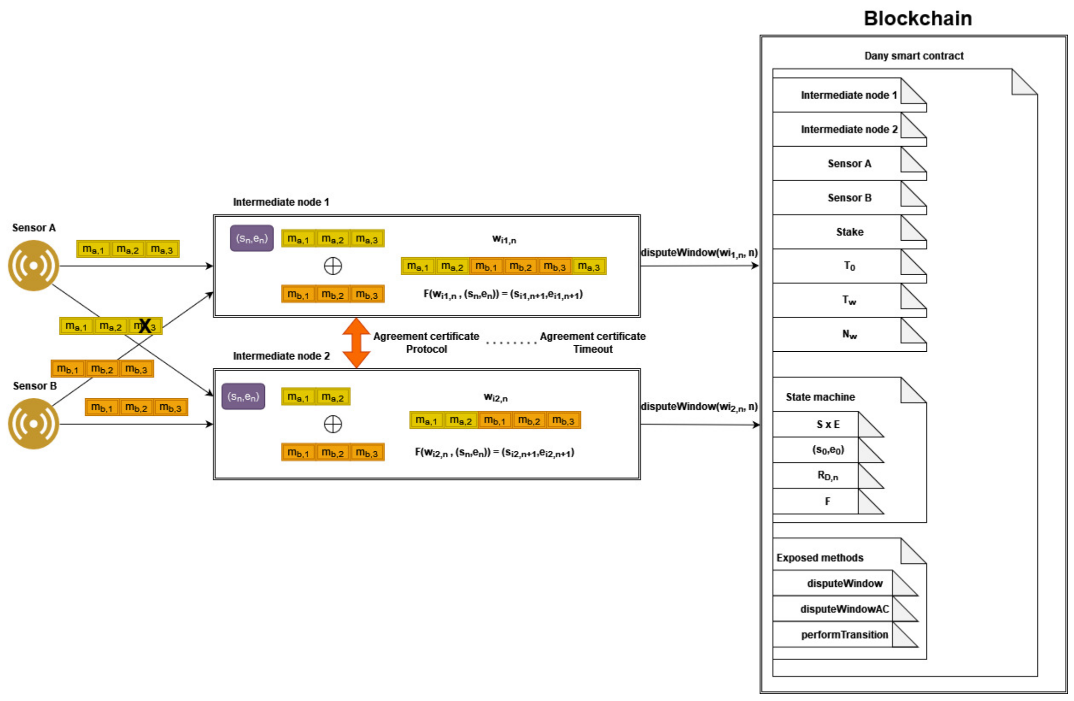


FIGURE 4.4: Starting a dispute in Dany: two intermediate nodes starting from the same state (s_n, e_n) try perform the agreement certificate protocol in order to generate a valid agreement certificate. However, they have received different windows meaning that they cannot create a valid agreement certificate in time, forcing them to provide their local window to the on-chain smart contract.

4.6 Dispute resolution protocol

A dispute resolution mechanism had to be devised for Dany in order to ensure the *liveness* and *safety* of the state channel even when intermediate nodes do not cooperate. *Liveness* informally states that ”good” things will eventually happen; while *safety* means that bad things do not happen. In our context, liveness states that the state channel is guaranteed to progress as long as there is at least one honest and working intermediate node in the system; while with *safety* we mean that any state reached by the state channel is either unanimously agreed by intermediate nodes, or is the product of the application of the transition function to the previous state and the merge of the local windows that the intermediate nodes use in order to move to the next state. In Dany, when intermediate nodes cannot reach an agreement on the state of the state channel at a certain step n, a dispute starts (Figure 4.4). An intermediate node starts a dispute either when he could not receive a valid agreement certificate for a given window before a timeout expires, or when he detects that another intermediate node has started a dispute on-chain.

When one of these situations occur, the intermediate node cannot prove that the state transition he performed is the same agreed by other intermediate nodes, nor that it has been performed with a window that contains the sensor messages received by other intermediate nodes. A solution for that is to perform the state transition on-chain and make the Dany smart contract transition to a new state. Intermediate nodes can in fact send to the smart contract their local windows and the smart contract would be able to merge these windows in order to apply the transition function to it. However, the *transition function* also requires to know what is the previously agreed state. The agreement certificate is the piece of data that allows any intermediate node to prove what is the state from which we are performing the transition. This is why it is necessary to generate an agreement certificate for every reached state (s_n, e_n), since it enables nodes to solve possible disputes occurring in the future. When an agreement certificate for a state cannot be generated, it is thus necessary to solve the dispute on-chain.

To solve the dispute, any intermediate node i_k waits an amount of time before communicating their local window, and possibly the agreement certificate for the previous state, to the Dany smart contract on-chain.

This kind of communication is performed by calling the methods exposed by the Dany smart contract. These methods are the *disputeWindow* 4.1 and *disputeWindowAC* 4.3.

Dispute resolution by providing the window of data

The *disputeWindow* method, is the method exposed by the Dany smart contract to allow any intermediate node i_k to solve disputes by providing its own local window w_{n,i_k}, allowing the Dany smart contract to transition to a state (s_{n+1}, e_{n+1}).

An intermediate node can call the *disputeWindow* method (algorithm 4.1) only if the following conditions apply:

- The blockchain time reached $T_0 + (T_W \cdot n)$ and a dispute period has not ended.
- The Dany smart contract knows the starting state (s_n, e_n) that can be used as an argument of the *state transition function.* This is the case if the method is called for the first window of the smart contract, in this situation the *state* to be passed to the *state transition function* is (s_0, e_0), which is the initial state. It is worth mentioning that the Dany smart contract at time $n + 1$ knows the state (s_n, e_n) from which he should apply the transition function if it previously performed on-chain the state transition from (s_{n-1}, e_{n-1}) to (s_n, e_n).

If these conditions are met, a node i_k calls this method providing its local window of data w_{n,i_k} and the index n. The local window thus provided will be used by the Dany smart contract to compute the merge of the local windows of the intermediate nodes (ω_n): the smart contract will in fact wait a dispute period that is increased each time a window containing more messages than the previous ones is proposed on-chain. This is done in order to allow other intermediate nodes to provide their local window, and then, after the dispute period has ended, it is ready to perform the state transition on-chain. Once the dispute period has ended, any intermediate node can make the smart contract perform the state transition by calling the exposed method *performTransition.* This perform the state transition on chain, and in the last state ($n = N_w - 1$), it will unlock the redistribution, giving the locked funds to the intermediate nodes according to e_{N_w}.

Algorithm 4.1 Dispute with window method

```
\\ $\omega_n$ is a local variable stored by the Dany smart contract
    that
\\stores the merge of all the windows proposed at time n
\\$disputeBegin_n$ is a flag that specifies whether a dispute has
    been raised for the n−th window

function disputeWindow($w_n$, $n$){
     require(isDisputeTime(tx.timestamp,$n$))
     require( n $\geq$ 0)
     require( n $<$ $N_w$)
     require(windowContainsNewValidMessages($w_n$,$n$))
     $\omega_n \leftarrow \omega_n \oplus w_n$
     $disputeBegin_n \leftarrow TRUE$
     increaseDisputeTime(n)
}
```

Algorithm 4.2 pseudocode of the perform transition method exposed by the Dany smart contract

```
 function performTransition(n){
     require(disputeTimeIsOver(n))
     require($disputeBegin_n = TRUE$)

     $(s_{n+1}, e_{n+1}) \leftarrow F((s_n, e_n), \omega_n)$

     if( $n = (N_w - 1)$ ){
          unlockRedistribution($e_n$)
     }
 }
```

4.6.1 Dispute with window and agreement certificate

The scalability improvements of Dany lie in the fact that state transitions can be performed mostly off-chain, without the need to tell the blockchain anything. For this reason, it is not always the case that the smart contract on-chain knows what is the state unanimously agreed off-chain. Whenever a dispute among intermediate nodes occurs for a state (s_n, e_n), and the Dany smart contract does not know the previous state (s_{n-1}, e_{n-1}), every disputing node should call the *disputeWindowAC* method. The *disputeWindowAC* method requires three arguments: the local window of the intermediate node w_{n,i_k}; the index n; and the agreement certificate AC_{n-1}, that allows the smart contract to know what is the previously unanimous agreed state reached off-chain. The *disputeWindowAC* performs the following checks upon being called:

- check whether AC_{n-1} is a valid agreement certificate for the state (s_{n-1}, e_{n-1});

- check whether the window provided contains valid sensor messages that were not provided to the blockchain before;
- check that the dispute period for the window has not ended

If the checks pass, the smart contract will store the agreement certificate and will wait a dispute period to allow other intermediate nodes to provide their own local window. Once the dispute period has ended, similarly to what happens when the intermediate nodes call the *disputeWindow*, the *performTransition* 4.2 method has to be called, allowing the Dany smart contract to perform the state transition on-chain.

Algorithm 4.3 Dispute with window and previous agreement certificate method

```
function disputeWindowAC(
        $AC_{n-1}$ , $n$)
        {
        require( n $\geq$ 0)
        require( n < $N_w$)
        require(windowContainsNewValidMessages($w_n$ ,$n$))
        require(isDisputeTime(tx.timestamp ,$n$))
        require(AgreementCertificateIsCorrect($AC_{n-1}$ , $n-1$))
        $disputeBegin_n$ $\leftarrow TRUE$
        $this.AC_{n-1}$$\leftarrow$ $AC_{n-1}$ //the stored agreement certificate at index
             n-1 is set to be equal to the one provided as argument
        $\omega_n \leftarrow \omega_n \oplus w_n$
}
```

4.7 Proofs

This section sketches the proof of a theorem about the Dany protocol that states that the honest intermediate nodes will eventually always agree on the same n-th state of the state channel and that this state is the result of the application of the transition function to the previous state and to the merge of the windows processed by the intermediate nodes, meaning that $(s_n, e_n) = F((s_{n-1}, e_{n-1}), \omega_{n-1})$. The theorem will be proven to hold true both when disputes occur and when they do not occur.

Theorem 1 *Let $\mathcal{A}$ be a state channel defined by a Dany smart contract $SC = \langle I, D, K, T_0, T_w, N_w, SM\rangle$, with I being a non empty set of intermediate nodes where $I = \{i_1, \cdots, i_{|I|}\}$. Let W_n be the set of windows that are used by the intermediates nodes to run the n-th run of the Dany protocol. That is $W_n = \{w_{i_1,n}, w_{i_2,n}, \cdots, w_{i_{|I|},n}\}$. Suppose that $\{\omega_0, \cdots, \omega_h, \cdots, \omega_n\}$ are the sequence of merged windows that are used by Dany at each run of the protocol where each $\omega_h = \bigoplus_{j=1}^{|I|} w_{i_j,h} \in W_h$. We can prove that at step n ($0 < n < N_w$) of the Dany protocol execution all honest intermediate nodes will reach the same state (s_n, e_n) that is the state reached by the state machine SM after parsing the merged windows*

$\omega_0, \omega_1, \cdots, \omega_n$. *This parsing can be described in the following way:*

$$(s_n, e_n) = \begin{cases} (s_0, e_0) & : n = 0 \\ F((s_{n-1}, e_{n-1}), \omega_{n-1}) & : 0 < n < N_w \end{cases}$$

where F is the transition function of the state machine SM.

Sketch proof.

The theorem can be proved by induction on the number n of the protocol runs. At run $n = 0$ all honest nodes will be in the same state (s_0, e_0) that is the initial state of SM when no window has been parsed.

Suppose that at the n-th protocol run (with $0 < n < N_w - 1$) all honest intermediate nodes will have the same state (s_n, e_n) and

$$(s_n, e_n) = \begin{cases} (s_0, e_0) & : \mathrm{n} = 0 \\ F((s_{n-1}, e_{n-1}), \omega_{n-1}) & : 0 < n < N_w \end{cases}$$

We need to prove that after one run of the protocol (i.e., $n + 1$) all the honest nodes will reach the same state (s_{n+1}, e_{n+1}) that is equal to $F((s_n, e_n), \omega_n)$. This proof can be done by considering the following two complementary set of assumptions:

1. **Case A.** (i) all intermediate nodes are running (i.e., there are no node failures and the network connection is always working), and (ii) all nodes are honest (i.e., they are correctly executing the protocol), and (iii) all intermediate nodes use the same window for the computation.

2. **Case B.** (i) at least an intermediate node failed (a fail-stop model is assumed) or a node connection link is not working, or (ii) at least an intermediate node is not sending valid protocol messages. A message is invalid in the following cases: (1) the message is wrongly formatted or unexpected (see protocol messages of Figure 4.3); (2) two different nodes propose different SM states; (3) two different nodes propose two different message windows;

Case (A). Without loss of generality we assume the intermediate node k is playing the leader role at run n of the protocol. The leader node will use its n-th window $w_{i_k,n}$ in order to calculate the new SM state (s_{n+1}, e_{n+1}) that is $(s_{n+1}, e_{n+1}) = F((s_n, e_n), w_{i_k,n})$. This will allow the leader k to assemble the protocol message of Eq. 4.9 that is:

$$pm_k = \langle n, id_{SC}, H(F((s_n, e_n), w_{i_k,n})), H(w_{i_k,n}) \rangle_{\sigma_{i_l}} \tag{4.11}$$

The leader will send pm_k to all other intermediate nodes. Since each receiving node i_j is honest (i.e., assumption **Case (A).(ii)**) and performs the same computation (i.e., assumption **Case (A).(iii)**), it will send back the same computation $H(F((s_n, e_n), w_{i_j,n}))$ and the same window hash $H(w_{i_j,n})$ (state-acceptance phase). This means that all nodes use the same window that is $w_{i_j,n} = w_{i_h,n}$ for each $i_j, i_h \in I$. It is worth mentioning that having the same window $w_{i_j,n}$ for all the nodes does not necessary mean that all sensor messages are correctly received. In fact, all intermediate nodes may lose the same sensor message but still having the same window. The leader will correctly receive all messages, will change its state to (s_{n+1}, e_{n+1}) and will be able to send back the Agreement Certificate. All nodes will correctly receive the agreement certificate and will move their state to state (s_{n+1}, e_{n+1}) (since they are all honest). It is worth noticing that in this case the blockchain is not involved. Effectively, there are no disputes. We now need to prove that the new state $(s_{n+1}, e_{n+1}) = F((s_n, e_n), w_{i_k,n})$ agreed by the intermediate nodes is equal to

$$F((s_n, e_n), \omega_n) \tag{4.12}$$

This can be easily proved with the use of assumption (A iii) stating that all intermediate nodes use the same window for the computation, meaning that the merge of all intermediate node windows ω_n is the same as every intermediate node window

$$w_{i_j,n} = \bigoplus_{i=1}^{|I|} w_{i,n} = \omega_n \tag{4.13}$$

since all sensor windows are the same that is $w_{i_j,n} = w_{i_h,n}$ for each $i_j, i_h \in I$. This means that all intermediate nodes will move to the state (s_{n+1}, e_{n+1}) such that:

$$(s_{n+1}, e_{n+1}) = \begin{cases} (s_0, e_0) & : \text{n} = 0 \\ F((s_n, e_n), \omega_n) & : 0 < n < N_w \end{cases}$$

Case (B). First of all it is worth mentioning that the honest nodes will drop all invalid messages. For instance, an intermediate node that sends all invalid messages is treated as a failing node that sends no messages. For case (B) we prove that any dishonest behaviour will always lead to a dispute. Then we prove that a dispute always synchronises honest nodes to the same state (s_{n+1}, e_{n+1}). This state is equal to $F((s_n, e_n), \omega_n)$.

Without loss of generality we first assume that the leader node is honest. The leader node will use its n-th window $w_{i_k,n}$ in order to calculate the new SM state (s_{n+1}, e_{n+1}) that is $(s_{n+1}, e_{n+1}) = F((s_n, e_n), w_{i_k,n})$. This will allow the leader k to assemble the protocol message of Eq. 4.9 that is:

$$pm_k = \langle n, id_{SC}, H(F((s_{n+1}, e_{n+1}), w_{i_k,n})), H(w_{i_k,n}) \rangle_{\sigma_{i_l}} \tag{4.14}$$

The leader will send pm_k to all other intermediate nodes and wait for their reply. By assumption, one of the nodes is not running a valid protocol. This means that the leader will not receive a valid reply from all the intermediate nodes and the agreement certificate will not be generated within the allowed time, or a dispute will be issued by an intermediate node (although this is not necessary). In case the timeout expires with the leader node not being able to generate a valid Agreement certificate, then the leader node starts a dispute on-chain. If the honest node was able to generate and send a valid AC, the dishonest node can still attack the protocol by issuing a dispute message. More in general, a dispute message can be issued at any step of the protocol. On-chain dispute messages will be always monitored by an honest node that will correctly detect it and will take part to the dispute protocol resolution.

When a non-leader node j is honest and running, it will wait for the pm_k message from the leader. When no valid message is received, the node will start a dispute. When the node receives a valid message that is $H(w_{i_j,n}) = H(w_{i_k,n})$ and $H(F((s_{n+1}, e_{n+1}), w_{i_j,n}))$ is equal to $H(F((s_{n+1}, e_{n+1}), w_{i_k,n}))$ then the non-leader node will reply by signing the message to its leader. In this case the node will wait for a valid agreement certificate. When this message is not received a dispute will start. After the honest non-leader nodes receive a valid AC, the dishonest node can still attack the protocol by issuing a dispute message.

We can conclude that a dishonest node will always trigger a dispute. At this point we prove that a dispute always synchronises honest nodes in a finite time to the same state (s_{n+1}, e_{n+1}) and that this state is equal to $F((s_n, e_n), \omega_n))$. In order to prove this we need to look at the dispute resolution protocol. First we can prove that disputes are always settled in a finite time as long as at least one intermediate node is honest. This is the case because each time an intermediate node calls one of the dispute methods exposed by the Dany smart contract on-chain, namely the *disputeWindow* and *disputeWindowAC*, while providing new valid messages as arguments, then the dispute period is increased. However, since the number of sensor messages is finite, it means that the dispute methods cannot be called indefinitely meaning that the dispute period is also finite, and, as long as there is an honest intermediate node, the *performTransition* method will be called, allowing any honest intermediate node to synchronize on the same state.

We now need to prove that the state (s_{n+1}, e_{n+1}) intermediate nodes synchronized to is equal to $F((s_n, e_n), \omega_n)$. This is evident from the dispute resolution methods pseudocode. In fact, the exposed methods will collect all valid sensor messages provided by the intermediate nodes and will perform a state transition towards a new state by applying the state transition function to the previous state and to the merge of all the windows that have been provided, but this merge is actually equal to ω_n since all intermediate nodes are required to provide their local windows: if any intermediate node does not provide their window we consider their local window to be empty, even if they physically received some sensor messages.

Corollary 1 *Let $\mathcal{A}$ be a state channel defined by a Dany smart contract $SC = \langle I, D, K, T_0, T_w, N_w, SM\rangle$. Let BC be the set of states of $\mathcal{A}$ that are stored by the blockchain on which an agreeement has been reached through the on-chain dispute resolution protocol*

$$BC = \{(s_k, e_k), \cdots, (s_j, e_j)\} \tag{4.15}$$

Let $ACSET_{i_k} = \{AC_x, \cdots, AC_y\}$ be the set of agreeement certificates either received, or generated, by an honest node $i_k \in I$ for the state channel a.

$$ACSET_{i_k} = \{AC_x, \cdots, AC_y\} \tag{4.16}$$

Let S_{i_k} be the set of state channel states reached by an honest node i_k either by agreement or by dispute resolution

$$S_{i_k} = \{(s_0, e_0), \cdots, (s_n, e_n)\} \tag{4.17}$$

Then it is possible for i_k to prove that any state in S_{i_k} is a state reached by the state channel, meaning that for every (s_i, e_i) in S_{i_k} either there exists an agreement certificate AC_i in $ACSET_{i_k}$ proving it, or (s_i, e_i) has been adjudicated in the blockchain, meaning that $(s_i, e_i) \in BC$. At the end of the execution of the state channel, any honest intermediate node is able to prove what was any n-th state of the state channel (s_n, e_n).

Sketch proof. This is a corollary of theorem 1. In theorem 1 it is proved that an agreement among honest intermediate nodes for any state (s_n, e_n) is always reached in a finite time, meaning that at the end of the execution of the state channel, an honest intermediate node i_k will have reached the states $S_{i_k} = \{(s_0, e_0), \cdots, (s_n, e_n)\}$.

Agreement can be reached in two different ways, either by agreement certificate propagation, or by disputing on-chain. This means that for every (s_n, e_n) there exists at least one agreement certificate proving it, or the state has been adjudicated on-chain. In case the agreement has been reached on-chain, then $(s_n, e_n) \in BC$, In case agreement has not been reached on-chain and i_k is honest, then it means that there exist an agreement certificate and that i_k has either received it, or assembled it. This is because the Dany protocol specifies that each time an agreement certificate is not received within a period of time, then any honest intermediate node should start a dispute on-chain. In this case, i_k can prove what is the n-th state of the state channel by providing the agreement certificate AC_n in $ACSET_{i_k}$, since it contains a proof that unanimous agreement has been reached among intermediate nodes.

4.8 Monetary incentive

The security of Dany comes from the fact that any party involved in the execution of the Dany smart contract can enforce the system to behave correctly. The scalability enhancements, instead, come from requiring parties to interact with the blockchain only on a few occasions, such as when money should be transferred from the Dany smart contract to a certain recipient or when there is a dispute. Solving disputes on the blockchain, however, is computationally expensive, and is also monetary expensive if the blockchain on which the *Dany smart contract* is deployed makes users pay for fees whose cost is proportional to the amount of computation requested. Ideally, to prevent nodes from purposefully communicating wrong information to cheat the system, a penalty should be given to the wrongdoers when they are proven malicious. However, in Dany it is not possible to know whether a node is communicating wrong information because he is malicious or because he hasn't received certain messages. For this reason, a different incentive system has been devised. The incentive system aims at providing the following properties:

- Intermediate nodes do not have an incentive in proposing wrong data, i.e incomplete windows, to the blockchain.
- Intermediate nodes do not have an incentive in waiting for other parties to solve the dispute for them (lazy nodes are not incentivized).

The Dany incentive system consists in requiring intermediate nodes to lock a certain amount of money in the Dany smart contract at the moment of deployment (stake) that will be used to incentivize their correct behaviour. The staked money is then used to repay the transaction costs borne by the disputers when they call the smart contract methods for solving disputes. Each time a smart contract method is called successfully by an intermediate node, the smart contract equally detracts the staked money from every registered intermediate node on the contract. Suppose intermediate nodes A, B and C have staked 5 tokens each in a Dany smart contract and suppose also that a transaction costs 3 tokens of fees, if node A calls a dispute method, then he is refunded of the three tokens required to pay for transaction fees, but the money is equally detracted from everyone's stake, meaning that after executing the method every party has now 4 tokens each. For this reason, a party is not encouraged to wait other parties to start a dispute since money would be detracted from the stake nonetheless if a dispute occurs. It may be argued that intermediate nodes do not have an incentive in disputing a wrong result on the blockchain at all since it would detract partially their money, however, intermediate nodes should be run only by the parties interested in the correct execution of the smart contract, if no party has an incentive in enforcing the correct execution of the smart contract, then no one can complain about its incorrect execution. With this solution, parties do not have an incentive in proposing wrong windows to the blockchain, as long as every party believes that at least one of them is honest, then every wrong window proposed would be challenged on-chain, detracting staked money from every intermediate node, even from the dishonest node. This incentive system, however, makes delegating the execution of a smart contract to a set of external intermediate nodes insecure. In fact, if no intermediate node has an incentive in making the smart contract execute correctly, then they may collude in order to gain the maximum rewards, or the least expenditures, from the smart contract execution, since there is no penalty for the intermediate nodes that behave incorrectly and no incentive to the intermediate nodes that detect the incorrect behaviour of these nodes. For these reasons, it is better that intermediate nodes be managed by the parties of the contract.

4.9 Risks

This section discusses the security risks related to the use of Dany related either to inappropriate configurations of the Dany smart contracts or to possible attacks.

4.9.1 Inappropriate window size

It is crucial to choose an appropriate *window size* T_w when deploying a Dany smart contract in order to ensure the correct behaviour of the system. Choosing an appropriate window size depends on the send-rate of sensor messages; on the characteristics of the blockchain used as *blockchain layer*, the most important characteristics of which being the *block size* and *block period* in Ethereum-like blockchains; and on the complexity of the *transition function* F. Setting a window size so big that it is likely to make intermediate nodes assemble windows containing more sensor messages than the ones processable by the on-chain Dany smart contract may make it impossible to solve potential disputes correctly.

4.9.2 Blackhole attack

A *packet drop attack* or *Blackhole* attack occurs when the attacker reprograms a node on the network to make it drop certain messages. In Dany a blackhole attack can be performed in various scenarios. if the blackhole attack is performed by blocking the messages sent by some sensor to a specific intermediate node, then the intermediate node that is the victim of the attack may not be able to receive the sensor messages required to assemble a window, making it disagree with the state proposed by other intermediate nodes of the network. This generates a dispute for every window for which a message has been dropped, meaning that the improvement in scalability coming from using Dany is nullified. If the attacker manages to perform a blackhole attack by making every intermediate node not receive certain sensor messages, then disputes are not started, but the agreed state transitions would not be dependent on real-world data anymore. Even though an attacker cannot forge false sensor messages without having a sensor secret key, he could none the less influence the outcome of the smart contract execution by dropping certain messages. If the attacker manages to block an intermediate node from receiving messages from the other intermediate nodes, then it may be possible that the attacked node is never able to receive the agreement certificate of any window, having thus to start a dispute for every window, similarly to what happens when he does not receive the sensor messages. Blackhole attacks may also be performed between an intermediate node and a blockchain node, in this case, the *any-trust* property of the system cannot any more be guaranteed since disputes cannot be resolved on-chain.

4.9.3 Off-line intermediate node

If an intermediate node goes offline for a prolonged period of time, the *any-trust* property of the system cannot be any more guaranteed. The scalability enhancements provided by the solution will also be hampered since the other intermediate nodes executing the Dany smart contract will not be able to generate valid agreement certificates for the processed windows, and when this happens they should communicate the local window to the blockchain.

4.9.4 Inappropriate dispute period

Dispute periods should be fine-tuned taking into consideration the specific blockchain used for deploying the *Dany* smart contract. Different blockchains may differ in terms of block period: the period of time after which a block is usually appended to the chain. Any Dany smart contract should not have a dispute period that is smaller than the block period of the blockchain on which it is deployed, or that approximates it. A dispute period that is too small may make intermediate nodes unable to dispute in time a result proposed to the blockchain, either because they couldn't assemble a transaction in time, or because they have received information about a block containing a dispute transaction for the Dany smart contract too late.

4.9.5 Inappropriate transition function

The *transition function* is applied both by the intermediate nodes when computing a state transition and by the blockchain when a dispute occurs. The *transition function* therefore has to be computable by the underlying blockchain. In fact, it is not true that every imaginable function can be executed by any blockchain. Even the blockchains that can be programmed using a Turing-complete language may not be able to compute computationally expensive functions. If the transition function requires too much computation to be processable by the blockchain, then a wrong state could be proposed to the Dany smart contract and other intermediate nodes would not be able to prove it wrong.

4.9.6 Stake is too low

If the money staked by intermediate nodes is low, then it may become impossible to use them in order to solve disputes. If the money staked gets used in its totality, then intermediate nodes may incur in a situation in which it would be convenient for them not to dispute certain wrong proposed results. let's assume a smart contract without staked currency is run by three intermediate nodes: *Alice*, *Bob* and *Charles*. Suppose that Alice is the leader node for a given window and communicates to the blockchain an incomplete window that penalizes both Charles and Bob. In this situation, either Charles or Bob should dispute the window by providing its window of data to the blockchain, however, providing a window of data to the Dany smart contract on-chain may cost a significant amount of fees that would not be refunded since it does not contain any staked money. If Bob assumes that Charles will eventually solve the dispute, then Bob has an incentive not to communicate anything to the blockchain since even though solving the dispute is a positive outcome for Bob, solving the dispute without contacting the blockchain is even more convenient. The same is also true for Charles, making it thus possible to make the Dany smart contract transition to an incorrect state. It may be argued that since both Charles and Bob are not following the protocol and Alice is malicious, then it is not necessary to provide any guarantee in this situation because every party is behaving incorrectly, however, let's suppose that Bob is an honest non-lazy intermediate node while Charles is lazy and never disputes wrong state transitions that could be disputed by Bob. In a situation like this, Bob will be unjustly penalized since the costs of its dispute resolution method calls will be borne only by him.

4.9.7 Blockchain reorganizations

Reorganizations, or *reorgs*, occur locally in a blockchain client when an accepted sequence of blocks is overtaken by a different sequence. Reorganizations can occur most notably in those blockchains that do not implement a consensus protocol that guarantees block finality, such as proof-of-work consensus protocols. Blockchain reorganizations may hinder the security of the Dany protocol since it may happen that the block containing the transaction that solved a certain dispute is overridden by another block. In these situations, it may not be possible to solve certain disputes since the blockchain may be reorganized after the dispute period has ended. To mitigate the possibility of incurring in this problem, it is suggested to either set a dispute period high enough to make the probability of out-of-dispute-period reorganizations extremely low or to deploy the *Dany smart contract* on a blockchain that guarantees transaction finality.

Chapter 5

Intermediate nodes implementation

This chapter describes the procedures followed by the intermediate nodes in order to elaborate sensor messages, create agreement certificates and interact with the blockchain. Intermediate nodes' behaviour can be described by the internal data structures they hold and the procedures they execute. Section 5.1 describes the data structures intermediate nodes hold to represent Dany smart contract states. Section 5.2 describes the procedures executed by the intermediate nodes.

5.1 Data structures

We assume that intermediate nodes start with knowledge about the Dany smart contract to operate, these information can be thought to be kept in a list of objects of type *Contract* as shown in Figure 5.1. The *Contract* class attributes reflect the information of the smart contract stored on-chain and provide a reference to the *Dany* smart contract on-chain. Any *Contract* object has the following attributes:

- **currentState**: the current state of the state channel;
- **address**: is a reference to the Dany smart contract deployed on-chain, such as an Ethereum address. We are assuming that the blockchain layer is instantiated by an account-based blockchain;
- **intermediateNodes**: references a sequence of IntermediateNode objects containing information about the intermediate nodes, they are used for signature verification purposes;
- **sensors**: references a sequence of sensor objects representing the sensors referenced by the *Dany* smart contract, they are used for signature verification purposes;
- **windows**: reference to a sequence of *Window* objects;
- **contractState**: can assume the values: *READY*,*EXECUTING*, *COMPLETED*. It represents whether contract execution has yet to start (READY), has started and is executing (EXECUTING) or it's execution is completed, meaning that the time period of the last window of the contract has ended;

- **numberOfWindows**: the number of windows processable by the referenced Dany smart contract: this is equivalent to N_w in the Dany smart contract formalization;
- **disputeTime**: the dispute period of the Dany smart contract.

Every Contract object also implements a method called *transitionFunction*, this is equivalent to the transitionFunction F defined in the Dany smart contract.

Intermediate nodes can keep track of the messages pertaining to a certain window of data with *Window* objects. *Window* objects are always referenced to by a *Contract* object and they allow the intermediate node to store the data required to perform the transition function, updating thus the state of the state channel run by them, and to keep track of the messages exchanged by intermediate nodes used for creating the *AgreementCertificate*. The *Window* class has thus the following attributes:

- **sensorMessages**: the sensor messages that comprise this window;
- **startTimestamp**: the timestamp when window execution start;
- **windowSize**: the temporal size of the window;
- **windowIndex**: the index of the window (whether it is the first one, the second one and so on);
- **protocolMessages**: the protocol messages received from the intermediate nodes referencing the window;
- **state**: the state computed by the intermediate node on this window: equivalent to (s_n, e_n);
- **agreementCertificate**: the agreement certficate either generated or received by other intermediate nodes;
- **windowState**: a value that provides information about the operations to perform on the window.

The attribute *windowState*, in particular, offers information to the intermediate node on how to treat the window. This attribute can take the following values:

- **NOTREADY**: means that the window period hasn't started, therefore sensor messages cannot be added to the window;
- **READY**: the window period has started and is not finished yet, therefore sensor messages can be added to the window;
- **FINISHED**: the window period has ended and the intermediate node can use it for executing the *off-chain function*;
- **ACWAITING**: the intermediate node is waiting to receive an agreement certificate;

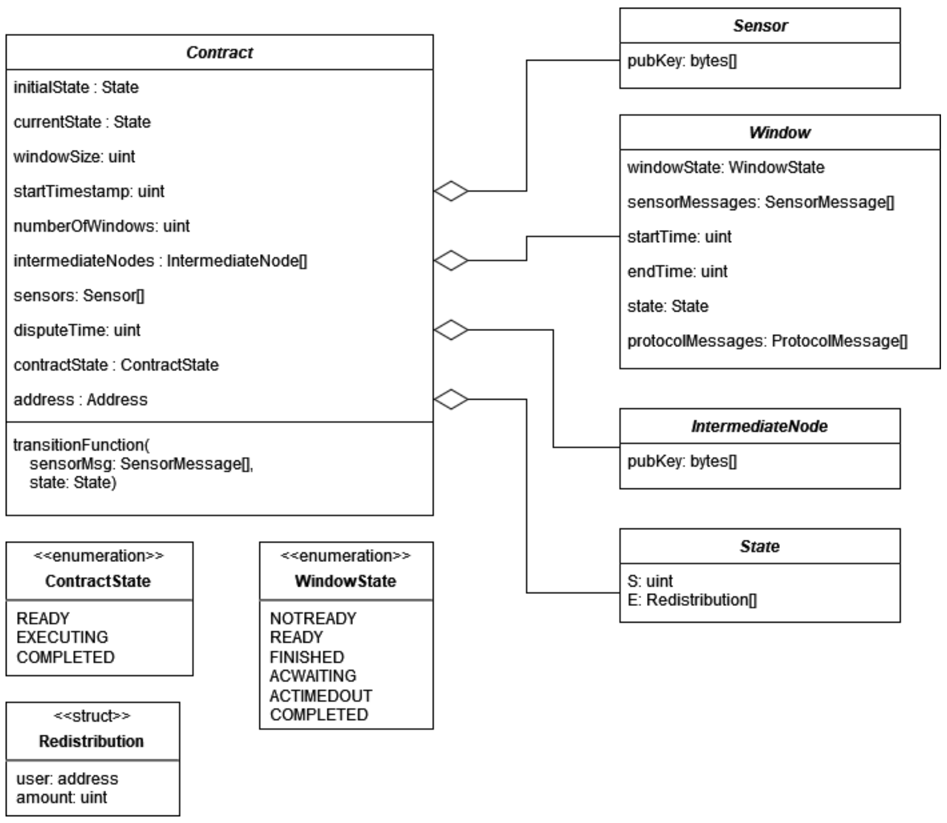


FIGURE 5.1: Class diagram representing a contract in intermediate nodes

- **ACTIMEDOUT**: the agreement certificate has not been received in time, therefore a dispute should be started on chain;
- **ACCOMPLETED**: the agreement certificate for the window has been either received or generated successfully.

5.1.1 Message structures

Messages received and sent by the intermediate nodes can be modeled as classes specializing a generic *Message* class 5.2. The types of messages an intermediate node can handle are *protocol message*, *leader protocol message*, *sensor message*, *Agreement certificate* and *Tick*. In particular, the leader protocol message class specializes the protocol message class because it uses the same fields any protocol message has. The *tick* message is never mentioned in the previous section because it is an internal message generated locally by the intermediate node at short regular intervals: it is an artifact used for timing purposes.

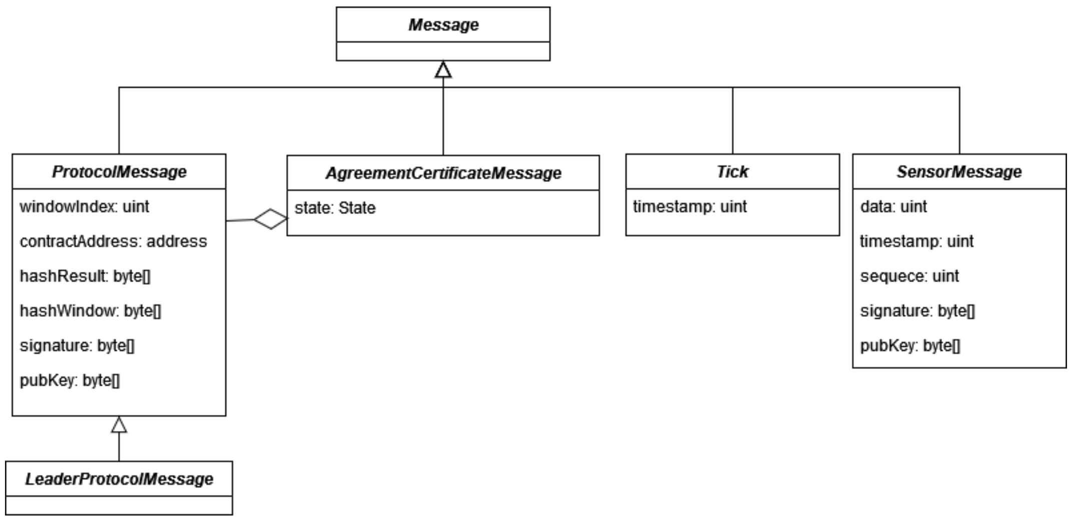


FIGURE 5.2: Class diagram representing the type of messages that are processed by intermediate nodes

5.2 Procedures

This section describes the procedures performed by intermediate nodes when receiving messages. Intermediate node behaviour depends on the messages it receives, as it is shown in 5.1 When a *sensor message* is received, the intermediate node should put it in the appropriate window. To do so, the procedure *parseSensorMessage* 5.2 is executed.

Algorithm 5.1 Receive message procedure

```
procedure receive(Message msg){

    if(msg.type is SensorDataMessage){
        parseSensorMessage(msg)
    }

    if(msg.type is Tick){
        parseTick(msg)
    }

    if(msg.type is ProtocolMessage){
        parseProtocolMessage(msg)
    }

    if(msg.type is AgreementCertificateMessage){
        parseAgreementCertificateMessage(msg)
    }
}
```

5.2.1 parseSensorMessage procedure

parseSensorMessage takes a sensor message in input and performs some preliminary checks to ensure the sensor message was sent by a registered sensor of the contract. The checks performed are to ensure that the sensor message is correctly signed
($require(isCorrectlySigned(msg)$ in the pseudocode) and that the timestamp of the sensor message falls between the start time of the smart contract and its end time ($startTimestamp + (numberOfWindows \cdot windowSize)$).

If the checks pass, then the window that should contain it is selected and the sensor message is added to its *sensorMessages* list.

Algorithm 5.2 Parse sensor message procedure

```
parseSensorMessage(SensorMessage msg){
    require(isCorrectlySigned(msg))
    require( Contract.startTimestamp $\leq$ msg.timestamp)
    require(Contract.startTimestamp + (Contract.windowSize $\cdot$
        Contract.windowNumber) $\geq$ msg.timestamp)

    window $\leftarrow$ getWindow(msg.timestamp)
    addSensorMessageWindow(msg,window)
}
```

5.2.2 parseProtocolMessage procedure

ProtocolMessages are the building block of *Agreement certificates.* When an intermediate node receives a protocol message it should do the following:

- Assemble the agreement certificate, in case the intermediate node is the *leader* of the window;
- Send back its own protocol message to the intermediate node if the state proposed in the received message is the same as the one computed locally;
- Store it in the appropriate *Window* object in order to assemble an agreement certificate later on when the node has received every intermediate node protocol message.

In order to do so, the protocol message is passed as an argument to the *parseProtocolMessage* procedure 5.3. The *parseProtocolMessage* performs the following checks:

- checks whether the protocol message is signed correctly by a registered intermediate node;
- checks whether it refers to a valid window for the Dany smart contract (the window index n is between 0 and *window_number*);

If the checks pass, then the procedure will retrieve the window object that the protocol message refers to ($getWindowProtocolMessaage(pm)$ in the pseudocode) and will check whether the received protocol message is a *leader protocol message* or a *standard protocol message*. A leader protocol message is a protocol message referencing window w_n sent by a node i for which $leader(w_n) = i$, with $leader : W \rightarrow I$ being a function that maps windows to nodes. Leader protocol messages should be received only by non-leader nodes and these receiving nodes should send back a response to the sender in the form of a standard protocol message. On the other hand, non-leader protocol messages should be received only by leader nodes which will use them to generate an agreement certificate. For this reason, leader protocol messages and standard protocol messages are handled by two different procedures, namely *parseLeaderProtocolMessage* and *parseStandardProtocolMessage*.

Algorithm 5.3 Parse protocol message procedure

```
parseProtocolMessage(ProtocolMessage pm){
    require(pm.window_index >= 0
    require(pm.window_index < Contract.numberOfWindows)
    require(isCorrectlySigned(msg))

    window ← getWindowProtocolMessage(pm)
    if(pm is LeaderProtocolMessage){
        parseLeaderProtocolMessage(pm, window)
    }else{
        parseStandardProtocolMessage(pm,window)
    }
}
```

5.2.3 parseLeaderProtocolMessage procedure

The *parseLeaderProtocolMessage* 5.4 makes these preliminary checks before continuing:

- checks that the receiving intermediate node is not the leader for that window. If the receiving node is the leader then it should not be possible for him to receive a leader protocol message and therefore should ignore them if they arrive;
- checks whether the state of the window is *FINISHED*.

Leader protocol messages received before a window has reached the *FINISHED* state should be ignored since the intermediate nodes may not have received the required sensor messages to complete the window. Similarly, the other window states (ACWAITING, ACTIMEDOUT, COMPLETED) imply that a valid leader protocol message for that window has been received in the past, this is why a new leader protocol message is ignored if the window is in one of these states.
If the checks pass, the intermediate node assembles a *protocol message* that is sent back to the leader node ($send(protocolMessageAnswer, sender(pm))$ in the

pseudocode). The state of the window object is also changed to *ACWAITING*, meaning that the intermediate node is now waiting to receive a valid agreement certificate.

Algorithm 5.4 parse leader protocol message procedure

```

parseLeaderProtocolMessage(LeaderProtocolMessage pm, window){
    require(! thisNodeIsLeader(window))
    require(window.state = FINISHED)

    protocolMessageAnswer ← generateProtocolMessage(window)
    send(protocolMessageAnswer, sender(pm))
    window.windowState ← ACWAITING;
}
```

5.2.4 parseStandardProtocolMessage procedure

The *ParseStandardProtocolMessage* 5.5, requires that the receiving node is the leader of the window and that the window state referenced by the protocol message is in the *FINISHED* state. This is because regular protocol messages interest only the leader node that has the role of using them in order to generate and propagate an agreement certificate and they would not be able to create such an agreement certificate if the window state is not *FINISHED*. With this procedure, the leader stores the *valid* protocol message received by the intermediate nodes. The leader node considers a protocol message to be *valid* if the message *hashState* and *hashWindow* are the same as the hash of the state and the hash of the window that the leader node stores locally.

Algorithm 5.5 Parse standard protocol message procedure

```
parseStandardProtocolMessage(ProtocolMessage pm, window){
    require(thisNodeIsLeader(window))
    require(window.windowState ← FINISHED)

    addProtocolMessageToWindow(pm, window)

    if(requiredProtocolMessagesForAgreementCertificate(window))
        {
        AC ← generateAgreementCertificate(window)
        window.AgreementCertificate ← AC
        broadcast(AC)
        window.windowState ← COMPLETED
    }
}
```

5.2.5 parseAgreementCertificate procedure

The *parseAgreementCertificate* 5.6 procedure is executed by intermediate nodes when an agreement certificate message is received. Upon receiving an agreement certificate, the intermediate node checks that the agreement certificate is valid. If that is the case, then the node will retrieve the window referenced by the agreement certificate ($window \leftarrow getWindowACMessage(msg)$ in the pseudocode) and will further check that its state is *ACWAITING*, meaning that the node is waiting to receive an agreement certificate for the window but hasn't received one yet. The agreement certificate is stored in the appropriate window object and the window is set to the *COMPLETE* state, meaning that no further actions are required to be performed.

Algorithm 5.6 Parse agreement certificate procedure

```
parseAgreementCertificate(AgreementCertificateMessage msg){
     require(isACValid(msg))

     window ← getWindowACMessage(msg)
     parseACWindow(msg,window)
}

parseACWindow(AgreementCertificateMessage msg, WindowInfo
    window){
     require(window.state = ACWAITING)
     require(msg.state = window.state)

     window.agreementCertificate ← toAgreementCertificate(msg)
     window.state ← COMPLETED
}
```

5.2.6 parseTick procedure

Ticks are used to change the window state when certain time limits are exceeded. In fact, the *parseTick* procedure 5.7, when called, retrieves every *window* object of every contract and depending on the state of the window performs some checks before changing their state if needed. The window state transitions performed are the following:

- from **NOTREADY** to **READY** state if the tick timestamp is larger than the window *startTimestamp* attribute;
- from **READY** to **FINISHED** state, if the tick timestamp is larger than the window startTimestamp + windowSize;
- from **FINISHED** to **COMPLETED** if the intermediate node is the leader node. before transitioning the intermediate node assembles a *leader protocol message* and sends it to the other intermediate nodes;
- from **FINISHED** to **ACTIMEDOUT** if the tick timestamp is larger than a timeout period waited by the node for receiving a valid agreement certificate;

- from **ACWAITING** to **ACTIMEDOUT** if the tick timestamp is larger than a timeout period waited by the node for receiving a valid agreement certificate;
- from **COMPLETED** to **ACTIMEDOUT** if if the tick timestamp is larger than a timeout period waited by the node for assembling a valid agreement certificate.

Algorithm 5.7 Parse tick procedure

```

parseTick(Tick tick){
    foreach(window in windows){
        parseTickWindow(tick,window)
    }
}

parseTickWindow(Tick tick,Window window){
    switch(window.windowState)

        case NOT_READY:
        if(tick.timestamp >= window.startTimestamp)
        {
        window.windowState ← READY
        }

        case READY:
        if(tick.timestamp >
        (window.startTimestamp + window.windowSize))
        {
        window.windowState ← FINISHED
        }

        case FINISHED:
        window.state ← applyTransitionFunction(window)
        async start disputeWindowProcess(window)

        if(isThisNodeLeader(window))
        {

        leaderProtocolMsg ← generateLeaderProtocolMessage(
            window)
        broadcast(leaderProtocolMsg)
        window.windowState ← COMPLETED

        }else if(tick.timestamp >
        windowACTimeOutTimestamp(window))
        {
        window.windowState ← ACTIMEDOUT
        }

        case ACWAITING:
        if(tick.timestamp > windowACTimeOutTimestamp(window))
        {
        window.windowState ← ACTIMEDOUT
        }

        case COMPLETED:
        if(tick.timestamp > windowACTimeOutTimestamp(window))
        {
        window.windowState ← ACTIMEDOUT
        }
}
```

5.2.7 Dispute process

When a window turns into the *finished* state, it becomes possible to communicate to the Dany smart contract on-chain the state obtained from applying the *off-chain function* on it. For this reason, it is necessary that every intermediate node checks that invalid results are not proposed on-chain by any possible dishonest intermediate node. The process that intermediate nodes run in order to solve possible disputes arising for a certain window is the *dispute window* process algorithm (code 5.8). The *dispute process* is started asynchronously as a window transitions to the *FINISHED* state, taking as argument the Window object representing the window on which disputes may occur. The process will then wait a *preDisputePeriod*, in order to mitigate the risks of multiple nodes providing the same window to the blockchain, and then will communicate with the on-chain Dany smart contract in order to retrieve data about its state. The data retrieved are data about the window previously proposed. Intermediate nodes, in fact, need to know whether a window has been proposed on-chain because an intermediate node proposing an invalid window may make the DaNy smart contract accept it if the dispute period ends and other intermediate nodes haven't disputed it; moreover, they need to know what is the window that has been used to propose a result in order to ensure that a state transition is not performed with an incomplete window.
The dispute process performs polling on the Dany smart contract in order to check whether a window has been proposed for the given window. If a window has been proposed on-chain, then the intermediate node will dispute it if it does not contain the messages in its local window. The result is disputed by calling the appropriate exposed Dany smart contract method, the method to be called is determined by whether the blockchain knows the previous state or whether it doesn't.

Interaction with the blockchain layer doesn't end with this. In fact, a state resulting from the application of the transition function to a window can be considered *accepted* by the DaNy smart contract only when the *performTransition* method is called. For this reason, the process will wait until the dispute period for the given window has ended and then will wait for an additional *postDisputePeriod* that is different for every intermediate node referenced in the Dany smart contract in order to avoid multiple nodes calling the same Dany contract when not necessary. After that, the node checks whether the *performTransition* method has already been called by any other intermediate node. If that is not the case, then he will call it, making thus the Dany smart contract transition to a new state. The *blockchain window process* ensures thus that as long as at least one intermediate node is honest and active, the Dany smart contract on-chain accepts only correct windows.

Algorithm 5.8 dispute process pseudocode run by intermediate nodes for a given window

```
process disputeWindowProcess(Window window) {
    watingPeriodPreDispute ← getWatingPeriodPreDispute(window)
    waitingPeriodPostDispute ← getWaitingPeriodPostDispute(
        window)
    wait(watingPeriodPreDispute)
    do until(endDisputePeriod(window) < currentTime )
    {
        if(isWindowProposed(window))
        {
            if(window.state = COMPLETED)
            {
                if(H(proposedWindow(window)) ≠ H(getAC(window).
                    state))
                {
                //calls the disputeWindow method
                //of the Dany smart contract
                disputeWindow(window)
                }
            }

            if(window.state = ACTIMEDOUT)
            {
                if(windowContainsUnproposedMessages(window))
                {
                disputeWindow(window)
                }
            }
        }
        else
        {
            if(satisfiedDisputeWindowRequireStatement(window))
            {
            disputeWindow(window)
            }
            else
            {
            disputeWindowAC(window)
            }
        }
    }
    wait(waitingPeriodPostDispute)
    if( ! isResultAcceptedOnChain())
    {
    //call the perform transition method of the
    //Dany smart contract
    performTransition(window)
    }
}
```

Chapter 6

Experimental results

In this chapter we discuss the experimental results obtained from running an implementation of Dany. The results obtained allows us to better understand which are the settings in which Dany proves to be a valid solution for scalability issues inherent smart contract execution.

6.1 Scenario

The scenario taken into consideration was a pollutant emission scenario in which a sensor produces at a constant rate data measuring the levels of pollution in the air as an integer number. When the average pollution in a given window of time exceeds a predetermined threshold, a fine should be applied to a user.

6.2 Implementation

The pollutant emission scenario was simulated by implementing a *Dany smart contract* written in Solidity in a local EVM compatible blockchain. The implemented Dany smart contract is initialized with the following arguments:

- *Intermediate node addresses*: the ethereum addresses of the intermediate nodes required to run the state channel: in our configuration we initialize it with two different ethereum addresses;
- *Sensor addresses*: The ethereum addresses of the sensors required to produce the data processed by the state channel: in our case we initialize the smart contract with one sensor address;
- *Dispute time*: the dispute period of the smart contract set to 10 seconds;
- *Stake*: An amount of cryptocurrency K staked by the intermediate nodes in the contract;
- *Number of windows*: (N_w) the number of windows processable by the smart contract;
- *Start timestamp*: The timestamp when the contract execution should start;
- *Initial state*: the initial state (s_0, e_0) of the state channel: represented as a struct in solidity (code: 6.1);

- A *transition function* (code 6.2) that returns a value that depends on whether the average calculated on a window of data exceeded a given threshold or not.

The smart contract exposes the three dispute resolution methods *disputeWindow* (code: 6.3) , *disputeWindowAC* (code: 6.5) and *performTransition* (code: 6.4) allowing nodes to dispute on chain.

Algorithm 6.1 Structs used to represent the state

```

    struct State{
        uint s;
        SingleRedistribution[] e;
    }

    struct SingleRedistribution{
        address receiver;
        uint amount;
    }
```

Algorithm 6.2 Transition function written in solidity used in the Dany smart contract

```
    function transitionFunction(StateLibrary.State memory
        _previous_state, Structs.Window memory _window) public
        pure returns (StateLibrary.State memory) {


        StateLibrary.SingleRedistribution[_previous_state.e.
            length] memory redistribution;
        StateLibrary.SingleRedistribution[] memory
            redistribution = new StateLibrary.
            SingleRedistribution[](_previous_state.e.length);

        int threshold = 30;
        int sum = 0;
        for(uint i = 0; i<_window.sensor_messages.length; i++){
            sum = _window.sensor_messages[i].data.data;
        }

        int avg = sum / int(_window.sensor_messages.length);

    if(avg > threshold){

        if(_previous_state.e[0].amount >= 100){
            redistribution[0].amount = _previous_state.e[0].
                amount - 100;
            redistribution[1].amount = _previous_state.e[1].
                amount + 100;
        }
    }
```

Algorithm 6.3 DisputeWindow in Solidity

```

function disputeWithWindow(Structs.Window calldata _window,
    uint256 _window_index)
        external
    {
        require(isDisputeTime(_window_index));
        require(windowIndexIsValid(_window_index));
        require(!timestampedDisputedStates[_window_index].
            accepted);
        require(
            _window_index == 0 ||
                timestampedDisputedStates[_window_index - 1].
                    accepted
        );

        disputeWithWindowPrivate(_window, _window_index,
            current_state); //should check that current state is
            always updated

    }

    function disputeWithWindowPrivate(Structs.Window calldata
        _window, uint256 _window_index, StateLibrary.State
        memory _previousState) public {
        require(windowIsOrdered(_window));
        require(addNewValidMessagesToStoredWindow(_window,
            _window_index));

        StateLibrary.State memory computedState =
            transitionFunction(_previousState,
            timestampedDisputedStates[_window_index].window);
        addStateToStorage(computedState,
            timestampedDisputedStates[_window_index].state);

        timestampedDisputedStates[_window_index].state_proposed
             = true;
        timestampedDisputedStates[_window_index].state_proposed
             = true;
        timestampedDisputedStates[_window_index].
            lastupdated_timestamp = block.timestamp;

    }
```

Algorithm 6.4 performTransition function in solidity

```

function performTransition(uint256 _window_index) external {
        require((timestampedDisputedStates[_window_index].lastupdated_timestamp + dispute_time) < block.timestamp);
        require(timestampedDisputedStates[_window_index].state_proposed == true);
        require(timestampedDisputedStates[_window_index].accepted == false);
        current_state = timestampedDisputedStates[_window_index].state;
        timestampedDisputedStates[_window_index].accepted = true;
        if(_window_index == (number_of_windows - 1)){
            performRedistribution(current_state.e);
        }
    }
```

Algorithm 6.5 disputeWindowAC in solidity

```

function disputeWithWindowAndPreviousAgreementCertificate(
          Structs.Window calldata _window,
          Structs.AgreementCertificate calldata
              _agreement_certificate,
          uint256 _window_index
     ) external {

          require(!timestampedDisputedStates[_window_index].
              accepted);
          require(!timestampedDisputedStates[_window_index -1].
              accepted);
          require(_window_index > 0 && _window_index <
              window_size);
          require(isDisputeTime(_window_index));
          require(isAgreementCertificateValid(
              _agreement_certificate)); require(
              _agreement_certificate.signed_protocol_messages[0].
              protocol_message.window_identifier == _window_index
              -1);

          timestampedDisputedStates[_window_index -1].
              agreement_certificate = _agreement_certificate;
          timestampedDisputedStates[_window_index -1].
              agreement_certificate_proposed = true;
          addStateToStorage(_agreement_certificate.state,
              timestampedDisputedStates[_window_index -1].state);
          timestampedDisputedStates[_window_index].state_proposed
               = true;
          timestampedDisputedStates[_window_index -1].accepted =
              true;
          addStateToStorage(timestampedDisputedStates[
              _window_index -1].state, current_state);
          disputeWithWindowPrivate(_window, _window_index,
              current_state);
     }
```

6.3 Results and discussion

The results presented in Table 6.1 show the amount of gas used when calling the methods exposed by the Dany smart contract with varying number of sensor messages in a window. It is possible to observe that the number of sensor messages in a window does not affect the cost of the *performTransition* method while the costs of *disputeWindow* and *disputeWindowAC* increase as the number of messages in the window increases. This is due to the fact that *disputeWindow* and *disputeWindowAC* take the window as input and have to perform computation on it (applying the *transition function* on it) and have to perform other checks on the window. However, the *performTransition* gas cost remains constant since its behaviour is that of accepting a *state* that was previously computed by either

calling the *disputeWindow* or *disputeWindowAC* methods. The gas requirements of such methods are high; this becomes more evident if we compare these costs with the cost of logging one sensor message in a simple logging smart contract (Algorithm 6.6). Logging one sensor message in such a contract costs 23424 gas units, a significantly lower cost than the Dany smart contract method calls. Using *Dany* becomes therefore convenient only if disputes are unlikely to be raised.
In Figure 6.1 it is shown the amount of gas consumed by an intermediate node for the Dany smart contract assuming that every window contains 5 sensor messages with varying *dispute rates*. With *dispute rate* we mean the number of disputes raised when a certain number of windows has been computed. These values are also compared with the amount of gas needed for logging sensor messages using the logging smart contract. It is possible to observe that the Dany smart contract makes an intermediate node spend less gas if the number of disputes raised during the execution is inferior to one dispute every 50 processed windows, while, if disputes where to arise every 10 windows, the logging smart contract is still preferable. Scenarios in which the rate at which sensor messages are produced is fixed, or close to fixed, and the number of disputes to be raised in a given period is expected to be within certain extremes, may allow users to fine tune the Dany smart contract, requiring intermediate node owners to stake only a slightly higher amount of money they expect to spend during the execution. In a scenario in which disputes do not occur, Dany could be able to process thousands of sensor messages per second, while public blockchains can only process a very small number of transactions per seconds. However, it is not always the case that disputes do not occur. Unfortunately, if the rate of sensor messages produced is extremely high and disputes are expected to occur, it may not even be possible to implement Dany on top of a public blockchain such as ethereum. This is because blockchains such as Ethereum produce blocks with a limited block size, meaning that it is not possible to create blocks containing transactions spending more gas than the maximum allowed. A Dany smart contract that manipulates windows containing an excessive number of sensor messages may not be implemented in such blockchains since it would make it impossible to solve arising disputes. Consortium blockchains, however, may provide a suitable blockchain layer for demanding Dany smart contracts since they may be set with high block size and block creation rate. The results obtained show that it is possible to improve the scalability of a blockchain by running Dany as a second layer on top of it.

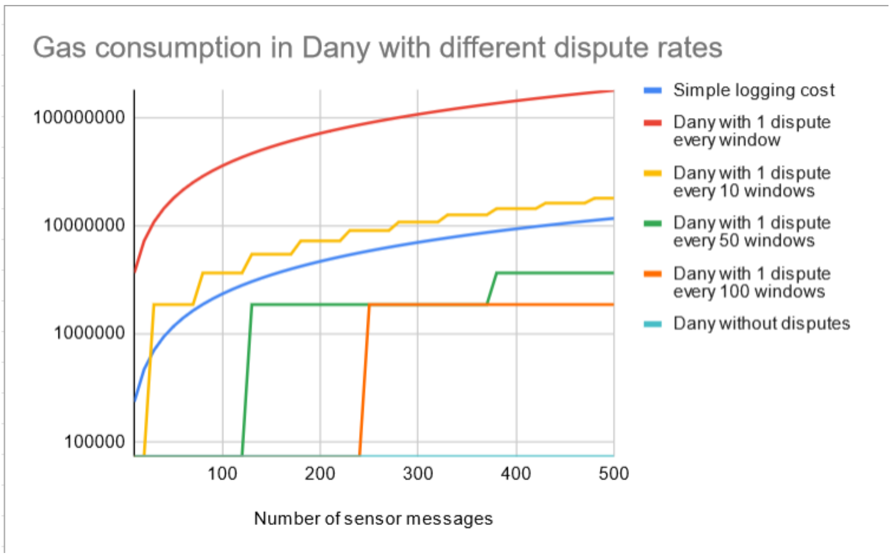


FIGURE 6.1: Graph showing the amount of gas used by an intermediate node for executing a Dany smart contract with different number of disputes. The Y-axis shows the amount of gas spent while the X-axis is the number of sensor message processed. It is assumed that sensor messages are grouped 5 by 5 in the intermediate node windows.

Algorithm 6.6 Logging smart contract written in Solidity

```


contract StorageContract{

    address sensor_address;
    address operator;
    Structs.SensorMessage[] sensor_messages_array;

    constructor(address _sensor_address, address _operator){
        sensor_address = _sensor_address;
        operator = _operator;
    }

    function addSensorMessage(Structs.SensorMessage memory
       _sensor_message) public {
        require(msg.sender == operator);
        require(isValidSensorMessage(_sensor_message));
        sensor_messages_array.push(_sensor_message);
    }

    function isValidSensorMessage(Structs.SensorMessage memory
       _sensor_message)
        public
        view
        returns (bool)
    {
        address signer = StructUtils.
           getSignerAddressOfSensorMessage(_sensor_message);
        return sensor_address == signer;
    }

}
```

size	performTransition	disputeWindow	disputeWindowAC
1	72938	371458	1018325
2	72938	565003	1211884
3	72938	758575	1405469
4	72938	952144	1599051
5	72938	1145715	1792647

TABLE 6.1: Table showing the amount of gas used per method call with varying window sizes. The column *size* specifies the number of messages in the window

Chapter 7

Conclusions and future directions

In this thesis, a novel protocol for the execution of smart contracts addressing the limits of traditional smart contract execution methods has been proposed.

The research questions addressed was the following:

- **Research Question:** Can we have a scalable solution for IoT smart contracts while maintaining a high level of security in smart contract execution?

We observed a lack of solutions for the execution of scalable IoT smart contracts in the literature. With *Dany*, we devised a solution for enabling the execution of smart contracts that are more scalable. Some of the smart contracts that would benefit from being run on Dany are in the field of the IoT. We provided a case study where the use of Dany would prove convenient such as the one described in chapter 3 and then we provided experimental results that show when the use of Dany would be convenient.

As a future direction, we think it would be useful to explore different kind of security assumptions. The *any-trust* security assumption used to build Dany may indeed be weakened if by doing so a more efficient protocol can be devised. We also think it can be useful to explore new dispute resolution mechanisms.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Hyperledger besu. `https://www.hyperledger.org/use/besu`. Accessed: 2023-02-28.

[2] Hyperledger caliper. `https://hyperledger.github.io/caliper/`. Accessed: 2023-05-2.

[3] Openzeppelin. `https://docs.openzeppelin.com/contracts/4.x/tokens`. Accessed: 2023-05-2.

[4] Solidity 0.8.17 documentation [online]. available: https://docs.soliditylang.org/en/v0.8.17/ [accessed 2023-01-25].

[5] Noga Alon, Haim Kaplan, Michael Krivelevich, Dahlia Malkhi, and Julien Stern. Addendum to "scalable secure storage when half the system is faulty. 01 2004.

[6] Hamidreza Arasteh, Vahid Hosseinnezhad, Vincenzo Loia, Aurelio Tommasetti, Orlando Troisi, Miadreza Shafie-khah, and Pierluigi Siano. Iot-based smart cities: A survey. In *2016 IEEE 16th international conference on environment and electrical engineering (EEEIC)*, pages 1–6. IEEE, 2016.

[7] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[8] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling blockchain innovations with pegged sidechains. *URL: http://www. opensciencereview. com/papers/123/enablingblockchain-innovations-with-pegged-sidechains*, 72:201–224, 2014.

[9] Mohamed Ben-Daya, Elkafi Hassini, and Zied Bahroun. Internet of things and supply chain management: a literature review. *International Journal of Production Research*, 57(15-16):4719–4742, 2019.

[10] Mirko Bez, Giacomo Fornari, and Tullio Vardanega. The scalability challenge of ethereum: An initial quantitative analysis. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 167–176, 2019.

[11] Stefano Bistarelli, Gianmarco Mazzante, Matteo Micheletti, Leonardo Mostarda, Davide Sestili, and Francesco Tiezzi. Ethereum smart contracts: Analysis and statistics of their source code and opcodes. *Internet of Things*, 11:100198, 2020.

[12] Mic Bowman, Debajyoti Das, Avradip Mandal, and Hart Montgomery. On elapsed time consensus protocols. Cryptology ePrint Archive, Paper 2021/086, 2021. `https://eprint.iacr.org/2021/086`.

[13] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[14] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. Combining ghost and casper. *arXiv preprint arXiv:2003.03052*, 2020.

[15] Diletta Cacciagrano, Flavio Corradini, Gianmarco Mazzante, Leonardo Mostarda, and Davide Sestili. Off-chain execution of iot smart contracts. In *Advanced Information Networking and Applications: Proceedings of the 35th International Conference on Advanced Information Networking and Applications (AINA-2021), Volume 2*, pages 608–619. Springer, 2021.

[16] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[17] Jing Chen and Silvio Micali. Algorand. *arXiv preprint arXiv:1607.01341*, 2016.

[18] Usman W Chohan. The double spending problem and cryptocurrencies. *Available at SSRN 3090174*, 2021.

[19] Anton Churyumov. Byteball: A decentralized system for storage and transfer of value. *URL https://byteball. org/Byteball. pdf*, 2016.

[20] Tom Close. Nitro protocol. *Cryptology ePrint Archive*, 2019.

[21] Tom Close and Andrew Stewart. Forcemove: an n-party state channel protocol. *Magmo, White Paper*, 2018.

[22] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, et al. On scaling decentralized blockchains. In *International conference on financial cryptography and data security*, pages 106–125. Springer, 2016.

[23] Sourav Das, Vinay Joseph Ribeiro, and Abhijeet Anand. Yoda: Enabling computationally intensive contracts on blockchains with byzantine and selfish nodes. *arXiv preprint arXiv:1811.03265*, 2018.

[24] Stefano De Angelis, Leonardo Aniello, Roberto Baldoni, Federico Lombardi, Andrea Margheri, and Vladimiro Sassone. Pbft vs proof-of-authority: Applying the cap theorem to permissioned blockchain. 2018.

[25] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*, pages 3–18. Springer, 2015.

[26] Massimo Di Pierro. What is the blockchain? *Computing in Science & Engineering*, 19(5):92–95, 2017.

[27] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[28] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 949–966, 2018.

[29] W Ethereum. Ethereum whitepaper. *Ethereum. URL: https://ethereum.org/en/whitepaper [accessed 2023-01-24]*, 2014.

[30] Caixiang Fan, Changyuan Lin, Hamzeh Khazaei, and Petr Musilek. Performance analysis of hyperledger besu in private blockchain. In *2022 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, pages 64–73. IEEE, 2022.

[31] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[32] Evangelos Georgiadis. How many transactions per second can bitcoin really handle ? theoretically. *IACR Cryptol. ePrint Arch.*, 2019:416, 2019.

[33] Amir Haleem, Andrew Allen, Andrew Thompson, Marc Nijdam, and Rahul Garg. Helium: A decentralized wireless network. *Helium Systems Inc., Tech. Rep.[Online]. Available: http://whitepaper. helium. com*, 2018.

[34] Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, S Matthew Weinberg, and Edward W Felten. Arbitrum: Scalable, private smart contracts. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1353–1370, 2018.

[35] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual international cryptology conference*, pages 357–388. Springer, 2017.

[36] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. *arXiv preprint arXiv:1602.06997*, 2016.

[37] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[38] LESLIE LAMPORT, ROBERT SHOSTAK, and MARSHALL PEASE. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[39] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the works of leslie lamport*, pages 203–226. 2019.

[40] Laphou Lao, Zecheng Li, Songlin Hou, Bin Xiao, Songtao Guo, and Yuanyuan Yang. A survey of iot applications in blockchain systems: Architecture, consensus, and traffic modeling. *ACM Computing Surveys (CSUR)*, 53(1):1–32, 2020.

[41] Colin LeMahieu. Nano: A feeless distributed cryptocurrency network. *Nano [Online resource]. URL: https://nano. org/en/whitepaper (date of access: 24.03. 2018)*, 4, 2018.

[42] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 17–30, 2016.

[43] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 706–719, 2015.

[44] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*, pages 120–130. IEEE, 1999.

[45] Leonardo Mostarda, Andrea Pinna, Davide Sestili, and Roberto Tonelli. Performance analysis of a besu permissioned blockchain. In *Advanced Information Networking and Applications: Proceedings of the 37th International Conference on Advanced Information Networking and Applications (AINA-2023), Volume 3*, pages 279–291. Springer, 2023.

[46] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[47] Sunoo Park, Albert Kwon, Georg Fuchsbauer, Peter Gaži, Joël Alwen, and Krzysztof Pietrzak. Spacemint: A cryptocurrency based on proofs of space. In *Financial Cryptography and Data Security: 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26–March 2, 2018, Revised Selected Papers 22*, pages 480–499. Springer, 2018.

[48] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. *White paper*, pages 1–47, 2017.

[49] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[50] Serguei Popov. The tangle. *White paper*, 1(3):30, 2018.

[51] Roberto Saltini and David Hyland-Wood. Ibft 2.0: A safe and live variation of the ibft blockchain consensus protocol for eventually synchronous networks. *arXiv preprint arXiv:1909.10194*, 2019.

[52] Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Annual International Cryptology Conference*, pages 148–164. Springer, 1999.

[53] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. Scalable bias-resistant distributed randomness. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 444–460. Ieee, 2017.

[54] Nick Szabo et al. Smart contracts, 1994.

[55] Péter Szilágyi. Eip-225: Clique proof-of-authority consensus protocol," ethereum improvement proposals, no. 225. [online serial]. available: https://eips.ethereum.org/eips/eip-225., 2017.

[56] IoTeX Team. Iotex: a decentralized network for internet of things powered by a privacy-centric blockchain. *IoTeX Team*, 2018.

[57] Karl Wüst, Sinisa Matetic, Silvan Egli, Kari Kostiainen, and Srdjan Capkun. Ace: Asynchronous and concurrent execution of complex smart contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 587–600, 2020.

[58] Joachim Zahnentferner. Chimeric ledgers: translating and unifying utxo-based and account-based cryptocurrencies. *Cryptology ePrint Archive*, 2018.

[59] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 931–948, 2018.

[60] Vlad Zamfir. Casper the friendly ghost. 2017.