Master's Thesis Summary - Niels F. S. Vistisen

Niels F. S. Vistisen

June 2023

Decarbonising modern society has become a global priority to survive the climate crisis. There is a general agreement that part of this process is transitioning to zero-emission energy, requiring the electrical grid to incorporate such energy sources. Recent research directions inspect which opportunities distributed ledgers could bring to this area. One of the main concepts resulting from this is micro grids, and studies on their efficiency and security have shown promising results. However, one of the recent issues receiving attention from the research community is trading between multiple micro grids. Many approaches have been proposed but suffer from impaired latency and throughput or compromised security.

In this thesis, I propose a design of a relay chain-based energy market platform that utilises a distributed mediator to facilitate energy trading across heterogeneous micro grids in Denmark, called PolkEM. Furthermore, I have designed, implemented, and tested an accompanying testing environment for future micro grid implementations. Due to implementation difficulties with PolkEM, I only provide a theoretical design and discussion on its potential performance.

However, I have implemented the testing environment through a Kubernetes cluster and facilitated user configuration through Helm charts and related parameters. The testing environment can run on various hardware configurations and infrastructures because of its Kubernetes foundation.

In addition to the testing environment, I have implemented microservices for account management and logging and a trade matching algorithm for the cross-grid trading process. These implementations have been tested with unit tests and decoupled from each other and the testing environment.

I have also created numerous automation scripts to handle tedious tasks and speed up the development time. These are especially useful due to Substrate's composition, requiring several files to be repeatedly generated and modified. By automating these processes, I have accelerated them and decreased the potential for errors.

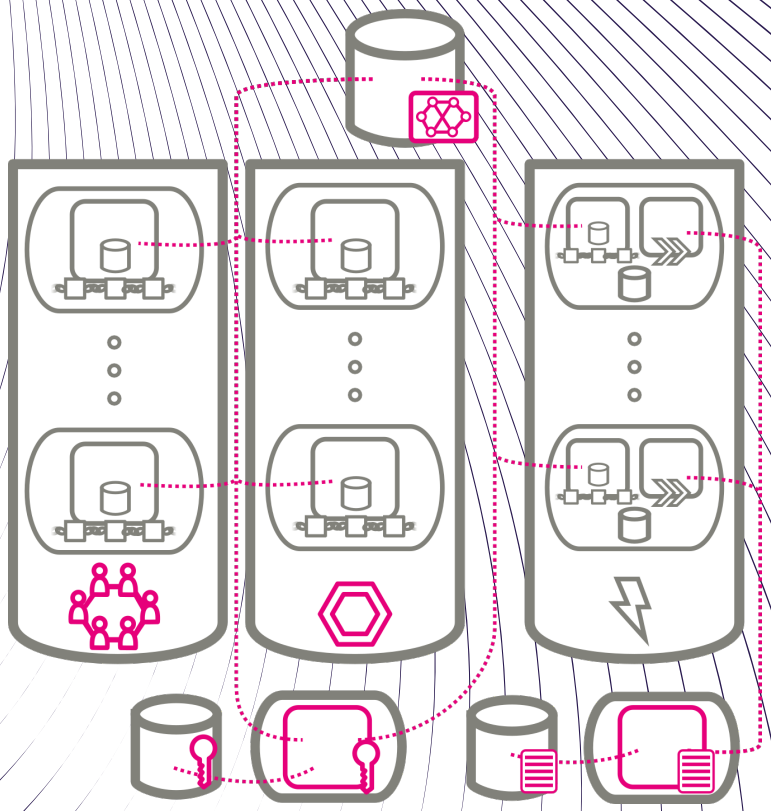
PolkEM theoretically meets its associated requirements based on a detailed discussion. Furthermore, its latency and throughput are highly competitive

with the current Danish system and decentralised alternatives proposed in related studies. I form this conclusion based on the assumption that more mature versions of Substrate and Polkadot will support the suggested micro grids and still operate efficiently. I support this assumption by comparing the practical size of the traded energy assets with the default block size used by Polkadot. I also discuss the potential computational overhead added by the needed modifications, which I deem reasonable.

I also discuss PolkEM's theoretical security properties and conclude that the platform provides high attack resilience based on its detection probability and firm punishments.

The testing environment also meets its requirements, except for malicious node injection. I did not implement the features necessary to meet this requirement due to time constraints, but I outline one approach to do so based on how I have implemented other features. I conducted an experiment illustrating how to execute tests with the testing environment and extract data from the tested micro grid's nodes.

In conclusion, PolkEM constitutes a potential design for a future distributed energy market. Whether the claims and assumptions on its performance and security properties hold in a real-world scenario is unknown, but the theoretical discussion provides promising potential for a similar system. Furthermore, the testing environment offers a way to test micro grid implementations on the platform during development and run extensive test scenarios on sophisticated infrastructures.



PolkEM

A Mediator-based Cross-Chain Energy Trading Platform with Accompanying Testing Environment

Niels F. S. Vistisen

Software, 2023-6

Master's Thesis

Department of Computer Science
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
PolkEM

Theme:
Specialisation in Distributed Systems

Project Period:
Spring Semester 2023

Project Group:
CS-23-DS-10-12

Participant(s):
Niels F. S. Vistisen

Supervisor(s):
Daniele Dell'Aglio
Michele Albano

Copies: 1

Page Numbers: 97

Date of Completion:
June 8, 2023

Abstract:

The energy sector has received increased attention recently due to the introduction of carbon-neutral sources, conflicts in Europe involving the largest fossil fuel exporters, and concerns regarding potential attacks on critical infrastructure. In this project, I propose a decentralised energy trading platform that supports distributed generators and focuses on prosumer inclusion. I use the Polkadot relay chain and a distributed mediator to facilitate trading between heterogeneous micro grids. In addition to the platform, I propose and implement a Kubernetes-based testing environment for conducting tests of future micro grid implementations. I experiment with the testing environment, proving its applicability and providing arguments for the platform's viability. I conclude that the future outlooks for the platform are promising and that the testing environment is usable for conducting complex tests of future micro grid implementations.

Contents

Preface

This report presents a platform supporting cross-grid energy trading between micro grids with an accompanying testing environment. It developed as part of my Master's Thesis in Software at AAU.

In Chapter 1, the problem is introduced. In Chapter 2, analyses of the context, preliminaries, and problem definition are presented. In Chapter 3, the proposed platform is specified with details of the used systems. In Chapter 4, the accompanying testing environment is outlined, described, and implemented. Chapter 5, present two experiments, the first covering the execution time of the exemplary trade matching algorithm and the second showing the capabilities of the testing environment. In Chapter 6, a discussion on the different aspects of the platform and testing environment are presented, concluding in whether they answer the problem definition posed in Chapter 2. In Chapter 7, the project is concluded and followed by recommendations for future development in Chapter 8. Links to the source code developed during the project are provided in Appendix B.

I want to thank my supervisors, Daniele Dell'Aglio and Michele Albano, for providing supervision and support throughout the project. I would also like to thank Brian Nielsen, Florian Lorber, Hamdi Ben Hamadou, Jiri Srba, Jonas Hansen, Torben Bach Pedersen, and Ulrik Nyman at AAU for taking the time to guide me in the pursuit of a climate-focused Master's Thesis.

Aalborg University, June 8, 2023

Niels F. S. Vistisen
<nvisti18@student.aau.dk>

Chapter 1

Introduction

Blockchains and distributed ledger technologies have received increased attention in the last decade. Their most prominent application, in terms of publicity, has been cryptocurrencies, which have created new financial markets. Recent crashes of these markets have resulted in scepticism towards the technology as a whole [1], demanding that new applications sufficiently prove that they overcome the widely acknowledged shortcomings, such as robustness, security, and efficiency [2, 3].
Alleviating these shortcomings is paramount when developing solutions for vital infrastructures, such as those in the energy sector. This sector has received additional attention from researchers and developers of distributed systems due to its transition towards more distributed and intermittent energy generation [4]. One of the main areas where this focus has offered new contributions is energy trading [5–7]. Studies on better facilitation and support for smaller producers have given rise to the concept of *micro grids* [8, 9] (MG). However, general issues persist, including scalability regarding users and security against adversaries [7, 10].
One recent development to address scalability has been the interconnectivity of multiple MGs, dividing communities into smaller groups, each with its own individual MG responsible for the limited number of users [11–13]. A promising development to support this approach is the reliance on a relay chain, facilitating parallel operation and communication between smaller chains [14]. This approach has recently been used with energy trading by Zhang [15] but achieved limited scalability, as all cross-chain transactions were facilitated and administrated directly on the relay chain.

In this study, I explore using a mediator chain to efficiently and securely facilitate energy trading between multiple MGs with heterogeneous internal logic. Supporting diverse MGs enables future developers to implement specialised features for varying communities based on the individual community's needs and capabilities, such as consumption and production volume. I provide a detailed

description of a platform called *PolkEM*, connecting this mediator to a relay chain highly inspired by Polkadot [16]. The relay chain acts as an interface for MGs to join the platform and ensure correctness. An exemplary MG is used throughout the study, as PolkEM encourages heterogeneous MG implementations and including all possible configurations would be infeasible.

Due to implementation difficulties, the project did not extend to a full implementation of PolkEM. Instead, I provide a detailed description of PolkEM and its constituents and a theoretical discussion on its applicability. Additionally, I present a testing environment capable of evaluating future MG implementations on the platform.

1.1 Motivation

The global climate crisis requires solutions for supporting the rapidly expanding global production of goods, which currently results in increased carbon emissions [17]. One aspect that could create a de-carbonising domino effect for processes is a transition to zero-emission energy. Carbon-free energy would allow other sectors to develop energy-intensive processes to replace fossil fuel-based ones, consequently allowing the zero-emission energy sector to mitigate a large part of their emissions [17]. Research enabling this transition of the energy sector is therefore highly impactful. Better facilitating energy trading for distributed, intermittent, and small-scale energy producers could increase investments in these technologies. Consequently, this would accelerate the transition, and the distribution of energy producers could enhance the infrastructure's robustness.

Blockchain technology still faces scepticism despite its public recognition regarding its application in an industrial context [2, 3, 18]. Proving that the technology can overcome the shortcomings of efficiency, scalability, and security identified in its early implementations remains a focus in the research community [10, 14]. The advancements in cryptographic systems, privacy-preserving protocols, Proof-of-Stake consensus algorithms, and multi-chain architectures show great promise, but applying these to real-world issues and proving superiority over current alternatives still constitute the frontier of the general blockchain adaptation [7, 9, 18, 19].

In this project, I aim to advance the field of decentralised energy systems. I do this by discussing the applicability of recently developed blockchain techniques and tools within the energy market and the competitiveness against the system currently used in Denmark. To back the claims made in the discussion, I have developed a testing environment for evaluating MGs on the platform. This testing environment supports various implementations, making it practical for developing and testing MG implementations for PolkEM.

The remainder of this report is separated as follows. In Chapter 2, I focus on energy trading as conducted today, the associated challenges and research directions, and will conclude with a list of requirements for PolkEM and the testing environment. I present a detailed description of PolkEM and its design in Chapter 3 and use this in Chapter 4 to describe the test environment and its implementation. I present two experiments to demonstrate the execution time of an exemplary trade matching algorithm, the test environment's capabilities, and the internals of PolkEM in Chapter 5. In Chapter 6, I discuss PolkEM's performance and the testing environment's completeness, leading to a conclusion on the project presented in Chapter 7. Chapter 8 gives an overview of future features and research directions.

In Appendix A, I present an overview of the development process, including several difficulties encountered while working with the Substrate framework [20].

Chapter 2

Problem Analysis

This project addresses challenges from energy trading in the current market, in addition to distributed ledger technologies, and I, therefore, provide an overview of the energy trading dynamics and some of the research directions within these fields.

2.1 Energy Trading

This section illustrates the responsibilities assigned to energy trading mechanisms and how the system currently operating in Denmark handles them. The section is based primarily on Christian Dahl Winther's detailed work and descriptions, presented in his book *Visual Guide to the Power Grid* [21]. Understanding these concepts is important for designing a system accommodating the growing demand for more energy and electrical grid features.

Electricity has become a vital commodity to everyday life across the planet, used, for instance, for heating, cooling, and industrial processes. [22]. Energy trading ensures availability, facilitates payments to energy producers, and balances the grid frequency to prevent white- and black-outs. These responsibilities are critical for the grid's continued operation and to meet the constantly fluctuating demand from consumers. In recent years there has also been an increasing interest in incorporating renewable energy sources into the market and improving the conditions for prosumers, who consume and produce electricity on a small scale.

2.1.1 Current Danish Model

The current system applied in Denmark utilises a highly centralised market to facilitate energy trading. The process conducts trades across two markets, the

day-ahead and the intra-day. A rough estimate is achieved through the day-ahead market and then fine-tuned based on the actual consumption and production using the intra-day market, allowing for better forecasting and proactive frequency adjustments. The day-ahead market relies on predictions from participants, while the intra-day market is mainly in place to handle the inevitable deviations of these predictions from actual consumption and production.

The frequency is balanced using information and communication technologies, but grid operators are still needed to verify results and react to unforeseen events. Any deviations from these predictions result in fines for the participants, which incentivises accuracy and precision. Parts of these fines compensate those who help stabilise the frequency. Participants can do this by increasing consumption during spikes or production during dips.

Computing the trades in either market is facilitated at a centralised point. This centralisation ensures a single source of truth, limits market participation to well-known actors, and achieves disputeless data validation. However, it also introduces a single point of failure, as attacking the trading system can bring down the entire grid. Applying redundancy to the system through backup instances only mitigate this risk partially, as the price of each instance likely limits the number of backups to a few.

Furthermore, as prosumers cannot participate directly in the market, their prices are controlled solely by large retailers operating as intermediaries. This separation results in unfavourable selling and purchasing prices for prosumers, who have no other way of market participation.

Despite energy retailers operating under laws and regulations, violations and abuse of power still occur. One example is the recent price spikes during the European energy crisis, where Danish energy companies had record high profits [23], suggesting an abuse of their position. Furthermore, multiple energy retailers are now suspected of cartel formation, collaborating to raise market prices [24].

Centralising the balancing makes its associated computations quick, as there is only one source of truth, alleviating the need to achieve consensus across entities. During the day-ahead market, the balancing algorithm is limited to 10 minutes, during which it must compute the optimal match of trades and share these with the participating actors. Each actor has 20 minutes to accept those that involve them before they become binding. During the intra-day market, trades are constructed continuously to fine-tune the balancing. The time between bid submission and trading varies across Denmark but is between five and 60 minutes. Hence, in the best-case scenario, a bid is traded within five minutes of its submission.

The points presented above illustrate prominent issues of the current market but also highlight qualities to incorporate into alternative systems. Centralisation introduces a single point of failure and trust-based trading, among other things,

to consider during the design. The main qualities inherited from centralisation are data validation, low trading delays, and the combination of rough estimates and fine-tuned balancing through the two-market model. A good balance between solving the issues and preserving the advantages should be the aim of an alternative system.

2.2 Related Work

The possibility of decarbonising large parts of our industrial processes through electricity from renewable sources has created interest from academics and practitioners. This interest has resulted in studies of both theoretical approaches and the practical applicability of decentralised market structures that have proven to improve the facilitation of distributed and intermittent production.

One study revolving around the applicability of a blockchain-based energy market conducted in Switzerland is Quartierstrom [19]. In the study, a Swiss community of 37 households participated in decentralised energy trading to determine how well local production could cover their consumption. Quartierstrom is built on the BFT engine *Tendermint* [25]. Even though the results were promising, receiving outstandingly positive feedback from the participants, the team abandoned the blockchain technology in the following study due to poor scalability [18]. The argument was that the platform did not scale sufficiently and encountered issues when reaching 500 participants.

For PolkEM, I utilise a Nominated Proof-of-Stake (NPoS) consensus approach to overcome these scalability limitations.

Scalability has been one of the main concerns regarding blockchain technologies and has resulted in numerous studies aiming to implement more efficient consensus approaches and better energy trading systems [7, 10]. One presented by Zhang [15] proposes utilising a relay chain to decrease latency and increase throughput while ensuring correctness for trades between participating MGs. The relay chain acts as a validation mechanism for the MGs and as a facilitator of cross-grid trading as part of its block authoring. Despite satisfactory latency, throughput, and security results for the tested networks, the study concludes that depending on the relay chain to facilitate cross-grid trading is not scalable in a real-world scenario.

I propose only using the relay chain for validation and incorporating a distributed mediator to facilitate cross-grid trading, alleviating the presented concerns.

Blockchain systems and microservice architectures share commonalities regard-

ing testing, as they both have autonomous processes likely distributed across multiple devices. The autonomy and distribution increase the testing difficulty due to the introduction of decentralised communication and the associated error sources. These include networking failures and non-responsive devices. Testing each node or service through unit-, component-, and integration tests still play a role for these architectures, but they cannot detect faults introduced by deploying the system on a distributing infrastructure. Therefore, explorative and scenario-based system tests are crucial for evaluating these systems, as these can identify potentially overloaded services and architectural bottlenecks. [26]

I present a testing environment for evaluating MG implementations on PolkEM, providing a general way to evaluate different designs and approaches with support for diverse and intricate test scenarios.

2.3 Preliminaries

This project builds on the research I conducted during my ninth semester. In the following, I re-introduce the main aspects to make this project self-contained. I advance the presented contribution by offering a more detailed description of PolkEM, in-depth arguments for its performance, and a testing environment.

2.3.1 Polkadot Terminology

I incorporate a relay chain, highly inspired by Polkadot, as part of PolkEM and therefore inherit some of its terminologies. I here list these with short descriptions.

- **Relay Chain**: An aggregating blockchain used to validate state transitions and, in turn, blocks of connected parachains.
- **Parachain**: A blockchain connected to a relay chain, depending on it for validation and interconnectivity with other parachains.
- **Validator**: A relay chain node participating in the validation process by joining a validator set, authoring relay chain blocks, executing approval protocols, and voting on the finality of relay chain blocks.
- **Validator Set**: A set of validators associated with a specific parachain for a limited time, responsible for initial validation of state transitions received from the associated parachain.
- **Collator**: A parachain node responsible for collating relevant block data and state transitions to the associated validator set.

- **Voting Set**: The set of validators responsible for voting on the validity of parachain blocks before they are accepted by the relay chain and on the finality of relay chain blocks.
- **Nominator**: A blockchain node that stakes tokens to help a collator or a validator reach the required stake, receiving part of the fees collected by their nominated collators and validators in return.

A validator can be part of a validator set and the voting set concurrently.

To increase readability and cohesion, I explain the remaining terminology, such as the specific approval protocols, as they become relevant throughout the report, continuously providing more details.

2.3.2 9. Semester PolkEM Architecture

The architectural result of my ninth-semester project, see Figure 2.1, consists of a relay chain that connects a set of MGs and an intermediary grid connection to a distributed mediator. The hexagon in the middle is the relay chain, and the components to the left are MGs. The two MGs depict two different implementations, MG_n containing a single type of node, the dots, and MG_1 extending this with a second type, the triangles. The dots between MG_1 and MG_n symbolise the inclusion of an arbitrary number of additional MGs. The top-right component is a grid-intermediary parachain to facilitate trading with the centralised grid. The bottom right depicts the mediator chain, which acts as the cross-chain trading facilitator for the remaining entities. The rectangles in each chain are relay chain validators connected to the specific chain for state transition and block validation. Additionally, these validators also facilitate the message passing between the parachains.

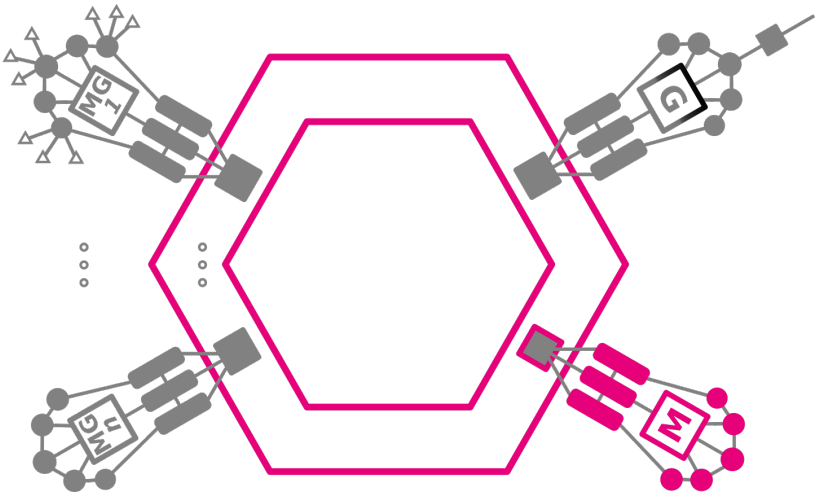


Figure 2.1: Architectural design presented in my ninth-semester project.

2.3.3 Byzantine Fault Tolerance

One theoretical problem limiting the scalability and security of distributed systems is the *Byzantine Generals Problem*, initially presented in 1982 [27]. I describe its fundamental aspects here, as it affects PolkEM's security and design. I have slightly changed the formulation to fit the context of a blockchain system, but the technical characteristics are maintained.

The problem depicts a set of nodes attempting to agree on a value, assumed to be binary in this example. One or more nodes may be adversaries aiming to alter the resulting agreement, either preventing it or deciding the agreed-upon value, by sending inequivalent messages. All proposed values are stored locally for each node as $v_i(1), ..., v_i(n)$, where n is the number of nodes, i is the local node, and $v_i(j)$ is the value received from node j.

Correctness can be ensured by meeting the conditions presented below.

1. $v_i(1), ..., v_i(n)$ must end up being equivalent across all well-behaved node
2. Given that a well-behaved node k proposes the value m, each well-behaved node, i, must end up with $v_i(k) = m$

All messages are forwarded to ensure that each node sends the same message to the other nodes. However, this introduces the main part of the Byzantine Generals Problem. Having three nodes, two well-behaved, A and B, and one adversary, C^A, A and B cannot identify C^A. For instance, assume that A receives two contradictory messages for $v_A(C^A)$, one directly from C^A and one forwarded by B. Whether C^A sent two different messages or B forwarded an altered value, $v_B(C^A)'$, is impossible for A to know.

(a) C^A sends contradicting messages.

(b) B alters the value received by C^A, misbehaving.

Figure 2.2: Depiction of the two cases where A receives contradictory values for $v_A(C^A)$. From the perspective of A, the two cases are identical.

Introducing a third well-behaved node, D, allows A to identify C^A as an adversary because A receives messages as described in one of the three cases shown in Equation 2.1, where

$$\forall_{i \in \{A,B,D\}} v_A(i) \equiv v_B(i) \equiv v_D(i)$$

$$\begin{cases} v_A(C^A) \neq v_B(C^A) \equiv v_D(C^A) & A \text{ received a different value from } C^A \\ v_A(C^A) \equiv v_D(C^A) \neq v_B(C^A) & B \text{ received a different value from } C^A \\ v_A(C^A) \equiv v_B(C^A) \neq v_D(C^A) & D \text{ received a different value from } C^A \end{cases} \tag{2.1}$$

In either of these cases, A ignores all values $v_{C^A}(1)...v_{C^A}(n)$ sent by C^A. Furthermore, a second adversary would increase the required number of well-behaved nodes by two to maintain correctness and decisively identify all adversaries. Lamport describes this problem in more detail in the original paper [27], and I will not provide further details here.

Based on the above, $\frac{2}{3}n + 1$ nodes in a network must be well-behaved to guarantee a consensus on a correct value. If a distributed algorithm can ensure the two presented conditions, given $\frac{2}{3}n + 1$ well-behaving nodes, it is called *Byzantine Fault-Tolerant* (BFT). The problem has become increasingly relevant in recent years due to the growing demand for reliant and secure distributed systems, such as those leveraging blockchains.

2.4 Problem Definition

The issues I have identified above include limited influence from prosumers, a single point of failure, and purely trust-based trading. These issues could originate from high centralisation and advocate for a more decentralised approach utilising distributed ledger technologies is creditable. The paradigm shift to a distributed market brings several concerns and technical challenges. The main problems I have identified and tackled in this project concern the design details of PolkEM and a testing environment to evaluate the MGs implemented for it.

The problem definition of this project is:

> ***How can a relay chain-based energy market platform utilise a distributed mediator to facilitate energy trading across heterogeneous micro grids in Denmark, and how can a testing environment be implemented to evaluate these heterogeneous implementations on the platform?***

2.4.1 PolkEM

The requirements for PolkEM are described in detail below and summed up in Table 2.1, using three categories, **scalability**, **security**, and **additional features**.

Latency: The time from bid submission to trade inclusion, referred to as Δ_l, must be competitive with the five minutes achieved in some parts by the current system. Whether this limit is desirable can be debated, but I use it as a target as it is achievable with the current system.

Throughput: In 2022, there where 2.788.291 households in Denmark [28]. Assuming that each of these will trade individually, PolkEM must achieve a throughput of 2.788.291 bids. Based on the latency requirement specified above, the time delay Δt_i between bid submission, i, and its inclusion in a trade must be at most $300s$. The required throughput can therefore be formalised as the set of delays $\Delta t_1, ..., \Delta t_n$, sorted in ascending order, where $n = 2.788.291$ and $\Delta t_n \leq 300s$.

Privacy: Actors in the system must be unable to obtain any participant's personal data. Such data includes address and individual long-term consumption and production data.

Transparency: Any action performed on the platform should be traceable to a specific account.

Attack Resilience: The platform must remain operational in the presence of adversaries under the assumption that at least $\frac{2}{3}n + 1$ of the authority nodes remain correct and well-behaved. One of the prominent attacks on MG-based energy markets is false data injection, as imbalances in frequency could have catastrophic consequences [29]. Therefore, resilience against this specific type of attack is crucial. I mention some additional attacks of concern and how PolkEM attempts to prevent them throughout the report.

Direct Prosumer Trading: Prosumers should be able to participate in trades without depending on a single third-party intermediary in control of pricing. Reliance on single entities to perform parts of the trading process is acceptable, given that prosumers can influence their pricing.

Heterogeneity: The platform must be able to support diverse communities only based on a few assumptions about the requirements, such as relevant market information and local trade construction.

The requirements presented here are vague and must be specified further for any meaningful implementation evaluation. However, these requirements will suffice, as I only propose the overall design.

Scalability		
R_P1	Latency	$\Delta_l \leq 300s$
R_P2	Throughput	$\forall_{i \in [1,2.800.000]} \Delta t_i \leq 300s$
Security		
R_P3	Privacy	Personal data stays hidden
R_P4	Transparency	Actions and actors are visible
R_P5	Attack Resilience	Continued operation despite adversaries
Additional Features		
R_P6	Prosumer Trading	Independent of price-controlling intermediaries
R_P7	Heterogeneity	Limited assumptions about communities

Table 2.1: Table of requirements for the PolkEM platform.

2.4.2 Testing Environment

The requirements of PolkEM outline a foundation for constructing and evaluating a competitive platform. However, they demand a sophisticated testing environment to determine whether a given MG is compatible and can achieve acceptable performance. Such an environment introduces additional technical challenges and requirements to ensure the extraction of high-quality results. I present a set of requirements for the testing environment below using three aspects, *basic functionality*, *scalability*, and *security*. Table 2.2 sums up these requirements. To limit the scope, I have focused on Substrate-based implementations [20].

The testing environment must include the basic functionalities of instantiating the MG under test, the mediator, and the relay chain that facilitates communication between them. After successfully instantiating the chains, the testing environment must also be able to connect these chains and include a mechanism for extracting data from the nodes. It must also incorporate a way to run sidecar applications[1] responsible for triggering relevant actions. Portability is another vital feature, enabling the testing environment to run on various setups. This feature allows the testing environment to function as a development and evaluation tool by supporting local development environments and more sophisticated infrastructures.

[1]Sidecar applications in this context refer to applications connected to a blockchain node, responsible for initiating its actions

After implementing these basic functionalities, the environment must also meet some requirements that allow it to test the scalability of the implementations. These requirements include instantiating an arbitrary number of chains and nodes, as well as the ability to determine the timing of events. Meeting these requirements means the testing environment can support load- and stress-testing. Extraction of event timings allows the user to reason about an implementation's scalability through the relation between latency or throughput and the number of MGs and nodes.

The security category I present here is limited, as testing MGs and the platform for security vulnerabilities using a quantitative methodology is insufficient for proving attack resilience. Instead, I have implemented the testing environment as an extensible foundation to accommodate specialised user configurations. Therefore, it must incorporate node accounts for testing against authorisation- and access-related vulnerabilities. Furthermore, it must facilitate the injection of modified nodes to represent adversaries. These features enable the environment to simulate attacks and demonstrate how the platform and the MGs under test react to these. This level of abstraction leaves the attack specification to the user, increasing the complexity of defining scenarios but expanding the range of supported configurations.

Basic Functionality	
R_T1	Support for blockchains with arbitrary logic
R_T2	Instantiation of a relay chain for inter-chain communication
R_T3	Connection between chains with arbitrary logic and the relay chain
R_T4	Extraction of data from nodes without relying on the implementation details
R_T5	Execution of sidecar applications interconnected with running nodes
R_T6	Portability across setups and infrastructures
Scalability	
R_T7	Support for an arbitrary number of chains
R_T8	Instantiation of an arbitrary number of nodes
R_T9	Incorporation of event timings as part of extracted data
Security	
R_T10	Inclusion of valid accounts for participating nodes
R_T11	Injection of adversary nodes

Table 2.2: Table of requirements for the testing environment.

In the following chapters, I present the design and implementation details of PolkEM and the testing environment and how they meet the listed requirements.

The handling of energy offers and requests rarely differs. Therefore, the term *energy asset* is introduced to refer to instances of either whenever PolkEM processes them identically.

Chapter 3

PolkEM

In this chapter, I introduce the PolkEM energy trading platform that leverages a relay chain and a distributed mediator. The relay chain and mediator enable PolkEM to conduct trading across multiple MGs with heterogeneous internal logic. The relay chain is based on Polkadot [30] and inherits its features. Furthermore, I use the Substrate [20] framework for the mediator to leverage the framework's components and protocols.

In Section 3.1, I give an overview of the energy asset submission and trade inclusion process from the prosumer perspective. Substrate and Polkadot's block creation and validation details are provided in Section 3.2 to form a foundation for discussing PolkEM's scalability and security. I finish the chapter by presenting and describing the main novelty of PolkEM, the mediator facilitating cross-grid trading, in Section 3.3. I discuss the presented concepts and details in Chapter 6.

PolkEM aims to support interconnectivity with the current grid operators through an intermediary parachain. This chain conducts block authoring and achieves finality identically to the MGs. Therefore, I only mention the MGs in the following sections, but the approach applies to the grid-intermediary chain as well.

A blockchain-based system's performance and security are rooted in the block creation and validation approach. PolkEM leverages the Substrate framework, which separates block authoring and finalisation, and inherits this property. The separation allows for increased chain growth and for multiple blocks to reach finality at a time.

3.1 Energy Asset Process Flow

Ensuring low latency, high throughput, and sound security in a blockchain-based system requires a deep understanding of multiple computer science areas, including networking and consensus. Before explaining the protocols PolkEM use to

achieve these properties, I list some requirements for the MG implementations that ensure their compatibility with the platform. After this, I provide an overview of the process flow for submitting an energy asset from the perspective of a prosumer.

3.1.1 Micro Grid Trading

Most energy trading is assumed to conclude locally, as it will involve fewer participants and less computational power, hence be cheaper. The internal logic for each MG is undefined due to the goal of supporting heterogeneity. However, each MG must communicate with the relay chain to participate in cross-grid trading, which is possible in two ways. The MG can either meet a set of requirements and connect directly to the relay chain or ignore these and connect to an intermediary chain. Using an intermediary chain includes additional steps that could decrease the performance and potentially introduce security risks. I provide an exemplary MG that connects directly to the relay chain for illustrative purposes.

The requirements for MG implementations that wish to communicate directly with the relay chain are listed below.

- The runtime must be representable as a Web Assembly (WASM) binary.
- The initial state of the MG blockchain, also known as the genesis state, must be exportable to a format compatible with the relay chain.
- The chain's internal logic must consist of a state machine using state-transition functions.
- The chain must be able to collate its state to relay chain validators through a node known and trusted by these validators.

Participants in the exemplary MG can submit energy assets through the following steps, illustrated in Figure 3.1.

1. An MG node receives a call to submit an energy asset, e.g. from a smart meter.
2. The node submits an energy asset storage transaction to the transaction queue and propagates it through the network.
3. During the authoring of the next block, queued transactions are materialised on the chain.
4. The collator forms trades, potentially accounting for surplus consumption or production through the mediator.
5. Resulting trades are stored on the chain, transferring tokens from buyers to sellers.

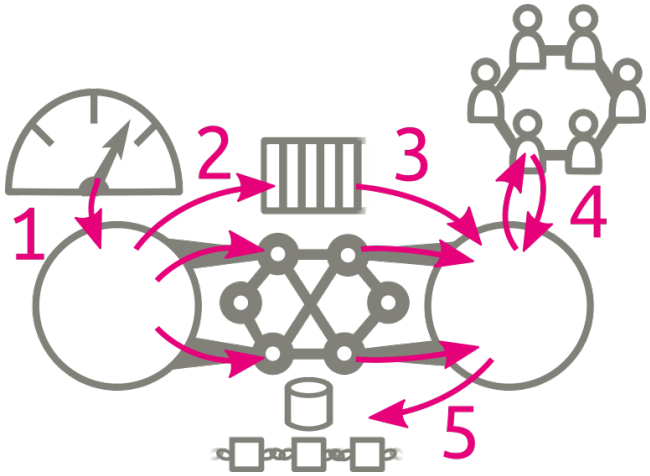


Figure 3.1: Process flow of an energy asset from the perspective of an MG node.

During step four, all surplus energy assets are aggregated into one and included in a message for the mediator.

The exemplary MG records all local energy assets included in the cross-grid trading process. After a trade is constructed with the aggregated energy asset, the following MG block author constructs transfers to all sellers based on the kept records. The aggregated price is calculated based on the constituent assets' amounts and prices. This approach allows each local seller and buyer to trade at the price they set but could result in undesirably high or low aggregated pricing, which would decrease the chances of being included in MG-to-MG trades. One solution is to restrict local prices to a range.

Future MG implementations might leverage various trading mechanisms, such as double auctions, to resolve most of their trades locally. More advanced trading mechanisms could require more blocks between each mediator message, potentially increasing the latency between bid submission and trade inclusion. However, parts of the existing system work on a 60-minute delay, allowing for $\sim$ 600 six-second blocks. The involved stakeholders should decide the approach most suitable for a given MG.

3.1.2 Energy Assets

Implementation-specific transaction data influences the size of each transaction but will include some fundamental elements that take up at most 200 bytes [31]. If each energy asset consists of the *amount* of energy, encoded as a u16 integer, and the *price* per kilowatt-hour, encoded as an f32 floating point, the total size will be at most 206 bytes. Assuming the buyer or seller is included separately from the origin account submitting the asset as a Substrate `MultiAddressAccount` [32], another 16 bytes are

added. For simplicity and to increase the space available to MG implementations' additional data, I will use a transaction size of 512 bytes. This assumption enables MGs to include miscellaneous data relevant to their needs and ensures that the presented discussion account for this to some extent. As discussed in Chapter 6, this overestimation does not considerably hurt the performance of PolkEM.

The default Substrate block length of 5.242.880 bytes [33] allows each block to hold 10.240 of the energy assets defined above. The documented size of empty and nearly empty blocks, i.e. blocks without transactions, is between 25.128 and 196.000 bytes. I will therefore use 9,850 transactions per block as an estimate, allowing for 199.680 bytes of miscellaneous data [34].

The figure in Figure 3.2 illustrates how long it would take to include the 2.788.291 million energy assets locally, not considering cross-grid trades and how additional MGs affect this delay.

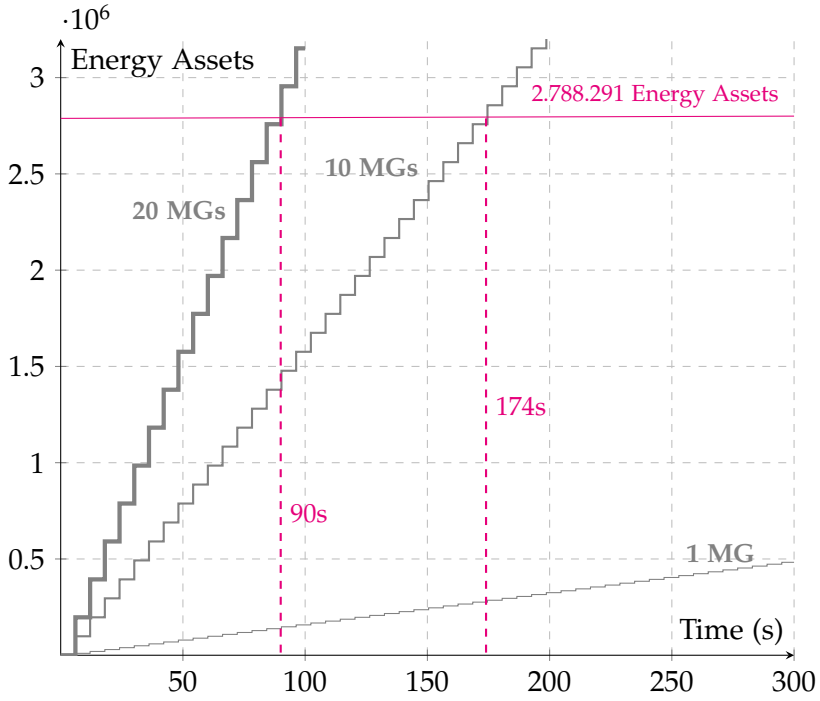


Figure 3.2: Graph showing energy assets included in blocks. The two vertical lines show how long it would take to include the target number of energy assets in local trades using 10 and 20 MGs.

3.1.3 Trade Finalisation

The local trades included in a parachain block must be materialised on a chain through block finalisation on the relay chain, following the steps listed below.

1. A parachain collator collates the relevant parachain block data to a member

of the associated validator set, V_i.

2. The receiving validator propagates the collated data through the set.

3. V_i decides the validity of the parachain block. If invalid, V_i ignores the block and skips the remaining steps. If valid, V_i submits the collated data to the relay chain queue for inclusion.

4. The submitted data is approved or denied through the process described later in Section 3.2.

5. A validator authors a relay chain block referencing the collated data and awaits finalisation.

6. The voting set casts votes on potential chains extending the relay chain to finalise its associated blocks.

7. Assuming that the block reaches finality, all referenced parachain blocks do too. Hence, materialising all trades in the associated MGs.

The performance and security of these steps are impossible to reason about at this level of abstraction. Therefore, I present detailed descriptions of block authoring, validation, and finalisation below.

3.2 Block Process Flow

As mentioned in the introduction of this chapter, the authoring and finalisation of blocks are separate parts of the process. Furthermore, additional protocols ensure the validity of collated parachain blocks and further improve performance and security. The seven steps listed below describe the process of conducting cross-grid trading and are depicted in Figure 3.3.

1. An MG collator collates a block, A, containing a message for the mediator to V_i.

2. V_i decides whether or not to back the received block. If they do, the data is forwarded to a queue on the relay chain, making the members of V_i liable for its validity.

3. Several protocols, described in Section 3.2.3, are executed to validate the block. If it is deemed valid, it is finalised on the relay chain and the contained message is forwarded to the mediator.

4. A collator on the mediator chain processes the received MG messages and forms cross-grid trades during block authoring. The trades are then validated through a voting process involving all mediator nodes. If approved, a new block, B, is authored with signatures from all supporting mediator nodes and messages for each MG involved in a trade.

5. A collator collates the relevant data from both blocks to the associated validator set, V_j.

6. Both blocks go through the procedure described in steps two and three and is finalised.

7. The messages for the MGs are forwarded to the recipients and their associated cross-grid trades are materialised on the MG chain.

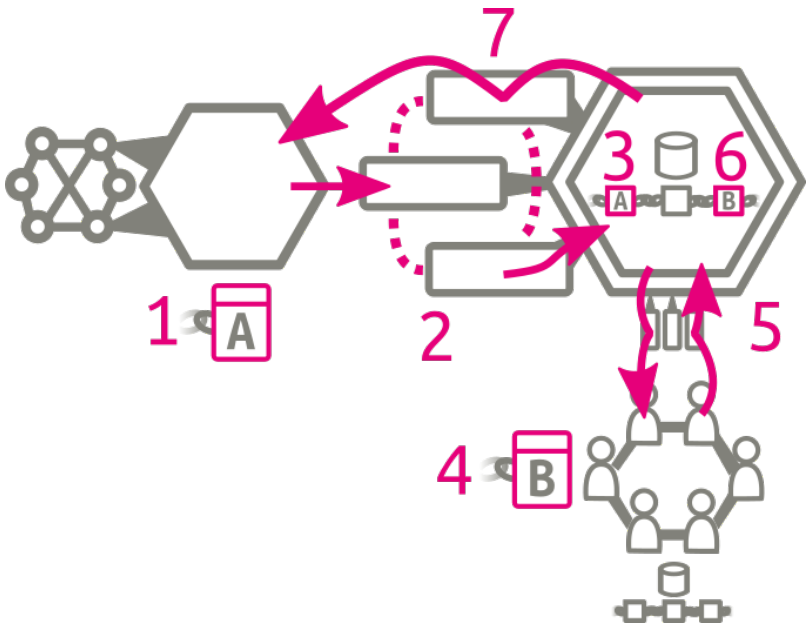


Figure 3.3: Process flow of blocks containing aggregated energy assets across the PolkEM system.

Before describing the authoring, validation, and finalisation processes, I present how Substrate and Polkadot reason about time.

3.2.1 Time

Executing logic across distributed entities, such as nodes in a blockchain, introduces challenges due to their separation. One such challenge is consistently keeping track of time. Substrate aims to overcome this by using delays from block observations. Time zero differs between nodes but is based on the time each node

observed its first block. Any future measure of time originates from this and uses fixed delays to reason about future points in time. The delays are set in the genesis state and are therefore agreed upon by all participating nodes. Substrate achieves loosely kept time through this 'synchronised' time zero and locally applied delays. Polkadot defines the units listed below. I use the presented values, as these are the current configuration of Polkadot.

- **Slot** [6s]: Only one block can be produced in each slot[1]. Due to the link between blocks and slots, the name *block time* is also used.
- **Tick** [0.5s]: Ticks further divide slots. This granularity allows the validators to reason about the state of a block throughout a slot.
- **Epoch** [4h or 24.000 *slots*]: Epochs are used to split up the assignment of authors. At the start of each epoch, participating nodes reach a consensus on a new *epoch random seed* used for the pseudo-random assignments described in Section 3.2.4.
- **Era** [24h or 6 *epochs*]: A given set of validators is active for an era at a time. The system selects this set for a given era during the last epoch of the previous one.

I use these delays in the following sections to describe the timings of different protocols and events.

3.2.2 Micro Grid Block Authoring

The exemplary MG uses the *Authority Round* (Aura) [35] consensus mechanism for block authoring and relies on the relay chain for finalisation. A single node authors the block for a given slot and is selected through a simple round-robin-based author-assignment approach. The nodes propagate the submitted transactions through the network as described in Section 3.1. Each node keeps track of the received transactions using a local transaction queue. The designated author constructs a new block with as many queued transactions as possible. The author then adds this block to the chain and waits on a collator to collate it for finalisation. The relay chain validators finalise the parachain block by referencing it in a finalised relay chain block.

Before referencing any parachain block on the relay chain, the validators complete an approval process of the parachain block.

[1] Due to the randomness in the assignment of authors, it is possible that no author assignment exists for a given slot. Furthermore, the system will discard additional blocks when encountering multiple author assignments in one slot.

3.2.3 Approval of Parachain Blocks

Multiple validators must approve a parachain block before referencing it on the relay chain. This section presents several protocols, described in a white paper from the Substrate team [36], that ensure the availability and validity of each referenced parachain block.

The most impactful efficiency and security improvements offered by PolkEM, compared to other blockchain-based cross-chain trading architectures [6, 19], comes from its Polkadot-inspired relay chain. Conducting block validation with a smaller set of trusted nodes instead of the given MG is more efficient due to fewer nodes required to reach consensus. All validators must put out a substantial stake, incentivising well-behaviour and improving security, further discussed in Section 6.3. However, an incentive is not equivalent to assurance and trusting these nodes without additional measures would not be advisable. Instead, the validation process includes five protocols to ensure correctness and well-behaviour. Validators execute these in sequence for each collated parachain block but can participate in their execution for multiple blocks concurrently. The protocols are Collation, Backing, Availability, Approval Checking, and Disputes.

Collation is the process of a collator submitting a block to be finalised on the relay chain, as described in Section 3.2.2. The collated data consists of the parachain block header and a *Proof-of-Validity* (PoV), used to validate the proposed state transition. The PoV ensures that the state transition is valid, and the header ensures that the exact transition results in the proposed state.

Backing is performed by the validator set assigned to the given parachain. These validators check the collated data to determine its validity and, if it is valid, register it on the relay chain for further approval. By backing a parachain block, the validators risk punishment if the block is proven invalid during one of the later protocols.

Availability ensures that the data needed to perform the validation check for a parachain block remains available. Relay chain blocks reference the collated headers but not the PoVs. Instead, the PoV is split into chunks and distributed between the backing validators. Other validators can then retrieve these at a later time. Any subset of these with a given size can reconstruct the PoV. By only requiring a subset and distributing them to multiple nodes, reconstruction of the PoV becomes resilient to node failures and denial of possession from malicious or faulty nodes. The protocol ensures that validators can perform checks later, decoupling the different processes.

Approval Checking requires non-backing validators to decide the validity of a parachain block through the execution of the state-transition function. The check is started by fetching a large enough subset of the PoV chunks and reconstructing the PoV. Then the checker retrieves the block header and the previous parachain state from the relay chain. After gathering all this data, each checker verifies the PoV, executes the state transition, and verifies that the output matches the fetched header. The checkers either approve the block and place it in a queue to be referenced in a future relay chain block or denies it, initiating a dispute. If the checkers disagree on the validity, even with a single opposing checker, they initiate a dispute as well.

This overview of the process is further detailed and discussed in Section 6.2.2 and subsubsection 6.3.2.

Disputes are used to handle conflicting opinions. They function as a fail-safe in the rare cases where validators disagree on the validity of a parachain block. Due to the rarity of these cases, disputes are very infrequent. When one is triggered, the entire set of validators must vote either *for* or *against* the subjected parachain block. All backers and approving checkers vote *for* automatically. When $\frac{2}{3} + 1$ of the validators have reached an agreement, all opposing validators, and their nominators, are fined a considerable amount of their staked tokens. The process of fining a participant is also called *slashing* and can result in a deduction equivalent to the stake of the fined participants.

3.2.4 Relay Chain Authoring

In this section, I describe the authoring process on Polkadot, which use the *Blind Assignment for Blockchain Extension* (BABE) protocol [37]. BABE builds on Ouroboros Praos [38], differentiating itself by not depending on a network time protocol for tracking time. The details of Ouroboros Praos [38] are superfluous in the context of this project, as the explanation of BABE will cover the essential aspects.

The author assignment of BABE uses a Verifiable Random Function (VRF), for which each participant, i, holds a keypair (sk^i, pk^i). All public keys are stored on the chain to facilitate checks of VRF results. At the start of each epoch, e_m, each participant executes the VRF locally for each slot, sl_k, in e_m to determine when to author blocks.

$$VRF_{sk^i}(r_m||sl_k) \rightarrow (d, \pi) \tag{3.1}$$

Equation 3.1 shows the VRF's signature, which takes a concatenation of an *epoch random seed* (ERS), referenced as r_m, and sl_k as input. The private VRF key, sk^i, is used locally to generate values specific to the given validator. The output

consists of a numeric value, d, and a proof of correctness, π. The numeric value determines whether the validator should author a block for slot sl_k by comparing it to a threshold, τ. τ is deterministically recalculated at each epoch and stored on the relay chain to be consistent for all participants. Other participants can verify the result using pk^i and π.

$$\tau = 2^{\ell_{VRF}}(1 - (1 - c)^{\frac{1}{n}}) \tag{3.2}$$

In Equation 3.2, ℓ_{VRF} is the bit size of d, c is a constant, and n is the number of participants. τ ensures that each validator has the same probability of being assigned for each slot. When authoring a block, the author includes the associated VRF output to allow future checks.

The ERS for epoch e_m is generated from the ERS of e_{m-2}, namely r_{m-2}, as well as the epoch number, m, and a concatenation, ρ, of all authoring VRF outputs from that epoch. e_m uses e_{m-2} and not e_{m-1} to ensure that the verification keys of validators wanting to participate as authors have been included on the chain before including them in an epoch. Without this shift, the keys of a validator that joined at the end of e_{m-1} might not be available. Hence, the threshold would be incorrect and affect the set size n, resulting in skewed VRF results.

The genesis state includes the ERS values for the first two epochs.

The author of a given slot constructs a block by retrieving queued parachain block headers. The associated blocks have already been approved at this phase, making the headers safe to reference in the relay chain block. The authored block is then added to the chain and awaits finalisation.

3.2.5 Relay Chain Finalisation

I here describe how Polkadot achieves absolute finality through GRANDPA [39].

The process of approval checking and authoring described above ensures the validity of every parachain block referenced on the relay chain and the steady growth of the chain. However, it only achieves *probabilistic finality*. The main drawbacks are the risk of invalidating forks and added latency from requiring each block to reach sufficient depth in the chain. To alleviate these issues, Polkadot use the *GHOST-based Recursive Ancestor Deriving Prefix Agreement* (GRANDPA) finality gadget [39], which allows chains of blocks to reach *absolute finality*.

GRANDPA uses rounds with an associated primary and a voting set, G. At the start of round r_i, the primary broadcasts the block, B_{i-1}, determined as finalisable during round r_{i-1}.

After a predefined networking delay, each voter broadcasts a pre-vote on a block B_i. All honest voters should vote on a block that extends B_{i-1}. However, B_i

can be multiple blocks ahead of B_{i-1}. If B_i gets finalised, all blocks in the chain between B_{i-1} and B_i are too.

Each voter determines the highest block that can be finalised based on the pre-votes and broadcasts a pre-commit to this block. This block has either received the most pre-votes or part of chains connecting B_{i-1} to proposed blocks that collectively have the most pre-votes. After $\frac{2}{3}|G| + 1$ pre-commits for B_i has been received, the voters broadcast a commit to that block.

The first step of round r_{i+1} requires the new primary to broadcast B_i as finalised to G. The members can identify a broadcast of any other block B' as invalid due to the previously received commits to B_i. The broadcast ensures that all voters agree on the latest finalised block, B_i, from which to extend the chain during r_{i+1}.

By essentially voting on chains rather than blocks, GRANDPA achieves higher throughput compared to other Byzantine fault-tolerant consensus algorithms.

As soon as a block reaches absolute finality, the weight of the relay chain is behind it, making the process of invalidating it infeasible.

3.3 Cross-Grid Trading

The sections above describe the process flow of PolkEM, which builds on a configuration of Substrate and Polkadot. In this section, I discuss the mediator chain, which is the primary novelty of the PolkEM system and is responsible for orchestrating cross-grid trading. The process includes energy assets forwarded from MGs through the *Cross-Chain Messaging Protocol* (XCMP) [40] offered by Polkadot. The cross-grid trading has considerable implications for the performance and security of PolkEM, making its design paramount to the platform's value. I use XCMP, a consortium chain approach, considerate selection of participants, and consensus on matched trades to achieve both qualities.

When an MG is unable to reach a local equilibrium between consumption and production, it can forward the surplus to the mediator using XCMP through the construction of a *cross-chain message* (XCM) [41]. XCMP requires parachain block authors to construct outgoing messages as part of block authoring, after which they are placed in an egress queue by the associated validator set. After finalising the parachain block, the validator set processes the egress queue and moves the messages to an ingress queue associated with the receiving parachain. Block authoring at the recipient chain includes processing any ingress queue messages. Hence, the authoring of parachain blocks involves both constructions of outgoing messages and processes of incoming ones. Message processing is part of the chains' state-transition functions and therefore inherits its verifiability.

The mediator encompasses multiple participants, each staking a considerable amount of tokens. The large stake requirement increases security on the mediator in the same way as on the relay chain. However, in contrast to the relay chain, the mediator is introduced as a consortium chain. Such a chain contains a smaller set of meticulously selected nodes, inhibiting adversaries from joining the network and members from adding additional nodes to achieve a majority. Hence, this drastically decreases the risk of Sybil attacks where adversaries join the network with numerous proximity accounts to gain the majority [42]. Furthermore, selecting participants with conflicting goals, such as energy retailers that wish to trade their assets, introduces natural market dynamics and further incentivises inspection of the resulting trades. This assumption is backed by research on Nash Equilibria within non-cooperative energy markets [43], assuming one exists for the proposed model.

Figure 3.4 depicts the mediator's process flow, which consists of the steps listed below.

1. Messages sent by other parachains are forwarded from the relay chain to the mediator through its validator set and placed in an ingress queue.

2. A collator authors a new block, A, by processing the messages in the ingress queue and conducting trade matching of the received energy assets.

3. The resulting block is propagated through the chain, ensuring backing from at least $\frac{2}{3} + 1$ of the nodes. A node backs the trades by submitting a signature on the set.

4. Assuming that $\frac{2}{3}n + 1$ signatures on the trades, a collator authors a new block, B, with outgoing messages for the trade constituents, meaning MGs and the grid-intermediary parachain. Each message contains any potential payment. The relevant data is collated to the associated validator set, forwarding the outgoing messages on finalisation.

The block authoring is based on Aura and incorporates the trade matching process, which follows an algorithm defined within the state-transition function. Because of this, the execution of this algorithm becomes part of the verification process, ensuring any deviations are usable as proof of misbehaviour. The algorithm is deterministic and verifiable through the blocks PoV, block header, and the previous state.

The mediator forwards any energy assets not matched between MGs to the grid-intermediary parachain by constructing trades with the grid's current selling and purchasing prices. These trades are similar to the ones between MGs, except the grid-intermediary chain controls the price. The pricing between assets in a single round of trading is assumed to fluctuate inconsiderably, as matching other

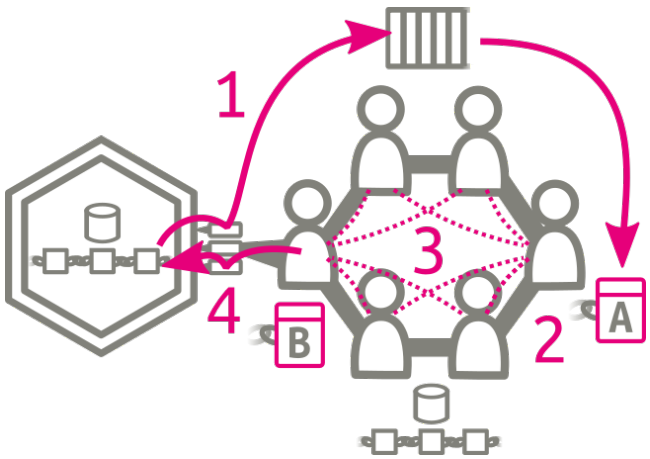


Figure 3.4: Process flow of the cross-grid trade matching on the mediator.

participants' pricing increases each MG's chances of being included in an MG-to-MG trade. This claim is further supported by the Nash Equilibrium theory [44], assuming that one exists, as sellers can only change their strategy.

I propose a simple algorithm, but a more sophisticated one might be required, incorporating other market dynamics. I present a simplified algorithm outline in Algorithm 3.1. The simplification is specifically the deviation between requested and offered amounts. For the actual implementation, any surplus amount is stored in a new energy asset and matched during the following iteration of the while loop on lines 3-11. I have developed the algorithm to maximise the trades between MGs, neglecting price optimisation.

Algorithm 3.1 Match received energy assets to create a set of trades T.

Input: Requests, ER, Offers, EO, and Grid Equivalents, (GR, GO)
Output: Set of trades, T

1: $sort(ER)$ and $sort(EO)$; ▷ Ascending order on price
2: **let** $ri \leftarrow 0$, $oi \leftarrow 0$, and $T \leftarrow \emptyset$;
3: **while** $ri < |ER| \wedge oi < |EO|$ **do**
4: **if** ER[ri].price >= EO[oi].price **then**
5: **let** $t \leftarrow (ER[ri], EO[oi])$;
6: $ri \leftarrow ri + 1$;
7: $oi \leftarrow oi + 1$;
8: **else**
9: **let** $t \leftarrow (ER[ri], GO)$;
10: $ri \leftarrow ri + 1$;
11: $push(T, t)$; ▷ Append t to T
12: **while** $ri < |ER|$ **do** ▷ Handle remaining requests
13: **let** $t \leftarrow (ER[ri], GO)$;
14: $push(T, t)$;
15: $ri \leftarrow ri + 1$;
16: **while** $oi < |EO|$ **do** ▷ Handle remaining offers
17: **let** $t \leftarrow (GR, EO[oi])$;
18: $push(T, t)$;
19: $oi \leftarrow oi + 1$;
20: **return** T;

The algorithm can sort the two energy asset sets, ER and EO, in $O(n \times log(n))$ by using Quick Sort. The three while loops, line 3-19, adds $O(n)$ as each iteration processes at least one element. I have implemented and benchmarked the algorithm in Chapter 5. This implementation includes the logic for processing matches between assets with deviating amounts. Any trade involving the grid is priced purely by the grid, as done in the current Danish system.

The mediator achieves the consensus on the resulting set of trades through a voting process that requires backing from $\frac{2}{3}n + 1$ of the nodes. Hence, the process is tolerant to Byzantine faults. Furthermore, any node attempting to construct a set of trades in a way that diverges from the defined algorithm will face slashing, disincentivising misbehaviour. The consensus process initiates when the block containing the trade set is proposed and concludes before the authoring of the next block. Doing this gives the nodes a few seconds to reach a consensus on the trades' validity. During this time, each node executes the algorithm, compares the result with the set in the proposed block, and submits a signed approval, assuming the

two results are equivalent. This design requires every mediator node to perform the trade matching while the block authoring and associated computations are conducted once at the assigned author. Furthermore, the results shown in Chapter 5 indicate that the computational load of running the trade matching algorithm is negligible.

I further discuss the latency and throughput aspects of the mediator in Section 6.2 and its security properties in Section 6.3.

Chapter 4

Testing Environment

In this chapter, I describe the design and implementation of the testing environment, intended as a development and verification tool for MG implementations operating on PolkEM. I base it on a Kubernetes cluster, which I outline with its constituents in Section 4.1. I then present automation scripts and configuration options available to the user in Section 4.2.

Basing the testing environment on a Kubernetes cluster allows it to run on various infrastructures and hardware configurations. I use Helm charts [45] to enable user configuration of various parameters, such as the number of nodes in each chain and their specifications. I expect users to download the testing environment repository before using it. In the repository's root directory, I have defined two Helm charts, a *parachain* chart configurable to support the given MG implementation and a *PolkEM* chart responsible for launching the environment itself. The PolkEM chart consists of configuration files for the microservices and two subcharts, one for the relay chain and one for the mediator. As the mediator connects to the relay chain like any other parachain, the mediator sub-chart is a symbolic link to the parachain chart, ensuring the two remain interchangeable. I define any mediator-specific configurations within the parent chart. The charts' content is depicted partially in Figure 4.1.

I have included separate deployments for the boot nodes to instantiate specific services for these. These services allow the remaining nodes to communicate with their associated boot node and are critical for instantiating the chain networks. The `relay-chain-chart` directory has an identical structure. However, the content of each chart file differs between them.

The user can provide parameters to each Helm chart to configure the testing environment during instantiation. Hence, the user can supply their chain specifications, Docker images, and similar.

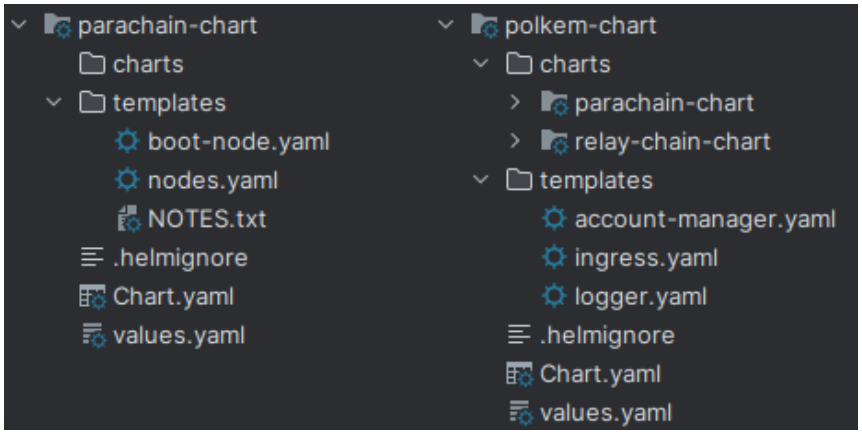


Figure 4.1: Screenshot of the PolkEM repository's Helm charts with the parachain Helm chart expanded.

I have structured the code for each microservice and node in separate repositories to decrease coupling and facilitate the interchangeability of the components. The testing environment uses Docker images constructed within the repositories and supplied to the Helm charts to instantiate each component. This way, they can be replaced or extended without extensive modifications to the testing environment. Each repository includes automated scripts to perform repetitive tasks, such as building and uploading Docker images and generating chain specifications. The primary repository is called *PolkEM* and contains the Helm charts, the testing environment code and configurations, scripts to increase ease of use, and the chain specifications. I expect users to modify the chain specifications to test their MG chains.

I provide a list of the repositories below. They are all publicly available and open source.

- **PolkEM**: The main repository containing the testing environment.
- **PolkEM Relay**: The relay chain, based on Polkadot.
- **PolkEM Mediator**: The mediator, based on the Substrate parachain template.
- **PolkEM AM**: The account manager.
- **PolkEM Logger**: The logging agent.
- **PolkEM MG**: The exemplary MG, based on the Substrate parachain template.
- **PolkEM Runner**: The sidecar application developed to communicate with the exemplary MG nodes.

Neither the mediator nor the MG is functioning, and their produced blocks contain no energy assets. Hence, they do not conduct energy trading. Furthermore, cross-chain communication is not set up. However, the environment does show how the chains produce blocks and receive updates on their finalisation from the relay chain. Furthermore, the microservices and sidecar applications are all functioning as intended.

4.1 Kubernetes Cluster

As mentioned, I build the environment on a Kubernetes cluster, allowing tests to be executed on various infrastructures and making it highly portable. Figure 4.2 depicts the environment's architecture. The dotted lines illustrate communication between different services in the cluster.

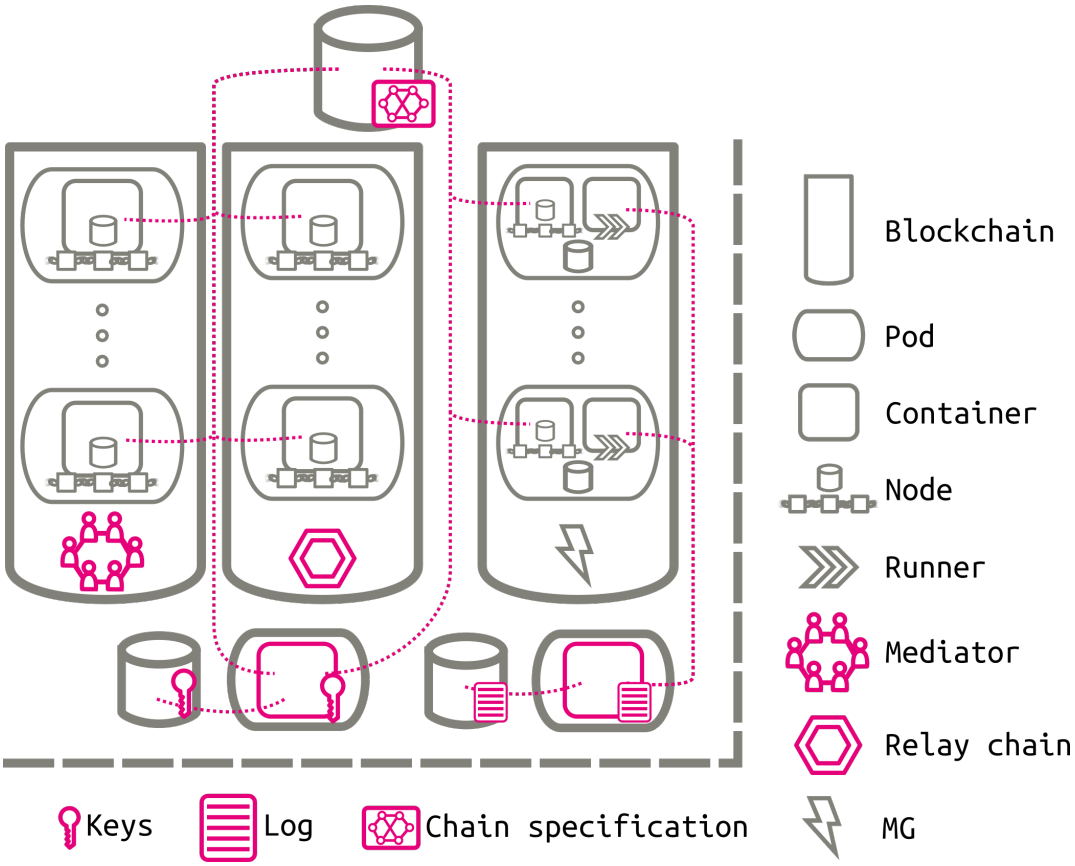


Figure 4.2: Architectural overview of the testing environment.

The cluster contains three blockchain networks, the relay chain, the mediator, and the MG under test. However, it does support multiple MGs running concur-

rently on PolkEM. The pods within a given network are all interconnected and have read-only access to the chain specification for the associated chain. A pod is the smallest deployable unit in Kubernetes and can contain multiple containers. Each pod in the testing environment also connects to an account manager service to retrieve its private key, used for connecting to the blockchain network with the correct account.

The two pods running outside the chains are the account manager, marked with a key, and the logging agent, marked with a file. They are used to provide the nodes with private keys and for logging any relevant data, respectively.

The testing environment stores the chain specifications, generated keys, and logs in volumes outside the pods. It requires the user to define persistent volumes before launching it. For local development, these volumes can be linked to directories on the local machine through the `create-kind-cluster.sh` script and references to these, also called claims, can be created by executing the `polkem-launch.sh` script. The chain specifications are loaded within each node from the persistent volume, while generated keys and logs can be extracted during tests or afterwards. Kubernetes offers *ConfigMaps*, which allows the user to inject data into each pod. However, this was not a viable option for the chain specifications because of their size.

Each pod within the MG blockchain contains two containers, a node and a runner. The node container is the blockchain node participating on the chain, while the runner is a separate sidecar application that triggers actions in the node container. These two containers run in the same pod to prevent runners from interfering with one another and connecting to the wrong nodes. Each pod also contains a volume to facilitate shared data between the two containers, such as the account information.

In addition to the elements in Figure 4.2, I have included an ingress control, which opens an external connection to the account manager and MG boot node. I used this for debugging purposes but it could also be useful during test execution.

4.1.1 Microservices

One of the main goals of creating the testing environment is to facilitate the testing and validation of MG implementations. Therefore, I introduce the *account manager* and *logging* microservices. Both services are written in Rust to avoid using a new language.

The account manager serves a single endpoint providing a GET method with a URL parameter, `ID`. The account manager uses the ID to generate a 64-character hex-decimal string. The string is returned to the client and saved in its local `keys`-directory. Any generated string is compatible with Substrate-based chains and used as the private key for the requesting node. The account manager also writes

all generated strings to a file, `keys/{ID}`, for future inspection and reference. Access to these is critical, as they must be known when constructing and debugging the chain specifications.

The testing environment uses the logging service to output data from the cluster nodes. Its endpoint provides a POST method with two URL parameters, `log` and `ID`. It uses these to specify which log to extend and the originating node, respectively. It expects the body of each POST request to be a string and inserts it into a log entry. Each entry is of the form "`{ID} {BODY} {DATE}@{TIME}`" and the entries are separated by the line-breaks. This way, the logger becomes a single source of truth regarding the time for the logged events.

4.1.2 Pod Initialisation

In addition to the components presented above, each blockchain pod includes a set of initialisation containers. These containers are responsible for configuring the given pod, ensuring it is ready for the node before initiating its container.

I use three initialisation containers for each pod containing a node. The first one waits for services that the node depends on, such as the account manager or associated boot node. The second requests the node's private key from the account manager. The third enables the node to take part in protocols on the chain by creating a keystore with seven public keys generated from the private key. Without these initialisation containers, the nodes would be unable to communicate or construct blocks.

The third initialisation container is likely to be different for some MG implementations and is therefore specified by the user through a Helm chart parameter, specifying the Docker image to use.

I present a snippet of the blockchain pod definition in Listing 4.1.

```
...

initContainers:
- name: {{ .Values.name }}-wait-for-boot-node-init
  image: busybox:1.28
  command: [ "sh", "-c", "until (nc -nvw1
    {{ .Values.name }}-boot-node-service 9933 2>&1 |
    grep -q open); do echo Waiting for boot node...;
    sleep 5; done"
  ]

- name: {{ .Values.name }}-node-key-init
  image: busybox:1.28
  ...
```

```
- name: {{ .Values.name }}-keystore-init
  image: {{ .Values.parachainInitImage }}
  ...

containers:
- name: {{ .Values.name }}-node
  image: {{ .Values.parachainImage }}
  ...
  args: [ "--collator", "--force-authoring",
    "--name=$(NAME)", ...
  ]
  ...

{{ if ne .Values.runnerImage "" }}
- name: runner
  image: {{ .Values.runnerImage }}
  ...
{{ end }}
...
```

Listing 4.1: Snippet from the stateful set defined in the `nodes.yaml` file included in the parachain chart. "..." is inserted where I have excluded lines from the snippet.

After all three initialisation containers have exited successfully, the pod initiates the node and runner containers, beginning the block authoring process.

4.2 Automation & Configuration

I have developed numerous automation scripts and configuration options to increase the usability and applicability of the testing environment. The user can launch the testing environment through a Bash file named `polkem-launch.sh`, which starts the microservices, relay chain, and mediator. The script requires that a Kubernetes cluster is running, is addressable through the Command Line Interface (CLI) tool *kubectl*, and contains an ingress configuration.

The PolkEM repository also contains a configuration file for *Kind* [46], used for running local Kubernetes clusters. Furthermore, it includes a pre-configured ingress controller and a script for setting up the Kind cluster. The Kind configuration file, the ingress controller, and the script can instantiate a local version of the testing environment with a single command, only requiring the installation of *kubectl*, *Kind*, and *Helm*.

I have also created scripts for handling repetitive tasks related to modifying the

microservices and nodes. A `build.sh` script is included in all application repositories and is used to generate the associated Docker image with all the required artefacts. I have also included a `build_init.sh` script in the chain-related repositories for generating the configurable initialisation container. Furthermore, these repositories also included a `generate_chain-spec_and_genesis.sh` script that generates the chain specification and extracts the chain's genesis state and WASM. The user must update the chain specification and genesis files every time the code is modified and included in the relay chain's specification. Hence, keeping them up to date can become repetitive.

In the next chapter, I present two experiments I conducted to test the trade matching algorithm and the testing environment.

Chapter 5

Experiments

I present two experiments. Firstly, I evaluate the implemented mediator trade matching algorithm's performance by executing a set of benchmarks. Secondly, I use the testing environment to extract proposed and finalised blocks from a running MG implementation. The MG implementation does not include any energy asset logic.

I conducted both experiments using a *Lenovo YOGA 730-15IWL* laptop with the hardware specification shown in Table 5.1.

Hardware Specification	
CPU	Intel i7-8565U - 4 cores (8 threads) at 4.6 GHz
GPU	NVIDIA GeForce GTX 1050 Mobile
Memory	16 GB - Dual-channel at 2400 MHz
Storage	512 GB NVMe SSD
OS	Ubuntu 22.04.2 - x86_64

Table 5.1: Table with the hardware specification of the laptop used for the experiments.

5.1 Mediator Trading Benchmarks

I have designed the mediator with a trade matching algorithm intended to be executed at every mediator node for each trading round to ensure the correctness of and consensus on the resulting trades. Because of the multiple executions in each round, the algorithm's performance could impact the time between each set of trades.

The presented implementation is simple and focuses on balancing requests and offers with as little dependence on the grid as possible. I have implemented it as a Rust library, allowing reusability across projects through Rust's crate system. I present the implementation in Appendix C.2, which slightly deviates from the outlined algorithm to be more idiomatic. I conducted a series of benchmarks based on different energy asset sets to provide an idea of the algorithm's execution time.

I based the benchmarks on the values proposed in Section 6.2, listed below.

1. One asset for each of the 1,888 proposed MGs
2. Ten assets for each of the 1,888 proposed MGs
3. Block transaction limit (9,850 energy assets)
4. Large over-approximation (4,000,00 energy assets)

Benchmarks one, two, and three are reasonable cases under the assumptions presented in Section 6.2, while the fourth benchmark pushes the algorithm performance.

5.1.1 Setup

The benchmarks were conducted using the `criterion.rs` Rust crate [47] and with randomly generated energy assets. I provide the code executed during the first benchmark in Listing 5.1. I used similar methods for the remaining benchmarks.

```
fn one_asset_per_mg(c: &mut Criterion) {
    let (mut requests, mut offers) =
        generate_requests_and_offers(944, 944);

    let (grid_request, grid_offer) =
        generate_grid_request_and_offer();

    c.bench_function("One asset for each of the 1,888 proposed MG",
    |b| {
        b.iter(|| {
            generate_trades(
                black_box(&mut requests),
                black_box(&mut offers),
                black_box(&grid_request),
                black_box(&grid_offer)
            )
        })
```

```
    }
    );

    let reqs_json = serde_json::to_string(&requests).unwrap();
    let mut file = std::fs::File::create(format!(
        "requests-{}.json",
        Local::now().format("%Y-%m-%d@%H:%M:%S").to_string()
    )).unwrap();

    file.write_all(reqs_json.as_bytes()).unwrap();

    let offs_json = serde_json::to_string(&offers).unwrap();
    let mut file = std::fs::File::create(format!(
        "offers-{}.json",
        Local::now().format("%Y-%m-%d@%H:%M:%S").to_string()
    )).unwrap();

    file.write_all(offs_json.as_bytes()).unwrap();
}
```

Listing 5.1: The code executed for the first benchmark.

The requests and offers are serialised and exported after the benchmark to evaluate the properties of the input. I only did this for the first benchmark to avoid generating the large files needed for the succeeding benchmarks' inputs.

5.1.2 Execution

I executed the benchmarks through a terminal using the `cargo bench` command, with no other active processes running on the machine.

As seen in Listing 5.1, the energy assets are randomly generated. I used ranges to generate the values for both the amount and price. I used the values 200 and 1,000 as the range for the requests' and offers' amounts, while the price ranges were 1.8 - 3.5 and 2.0 - 4.0 for the requests and offers, respectively. The values are superfluous in the results, as the benchmarks focus on the execution time. Furthermore, the algorithm runs with the aggregated assets from each MG, making the possible amount range immense. I illustrate the distribution of the generated assets in Figure 5.1.

The `criterion.rs` crate provides configuration options to tailor the benchmarks. I used the configuration seen below, where *measurement time* is the least number of seconds spent on each benchmark, *sample size* is the least number of executions for each benchmark, and the *confidence level* is the target for how con-

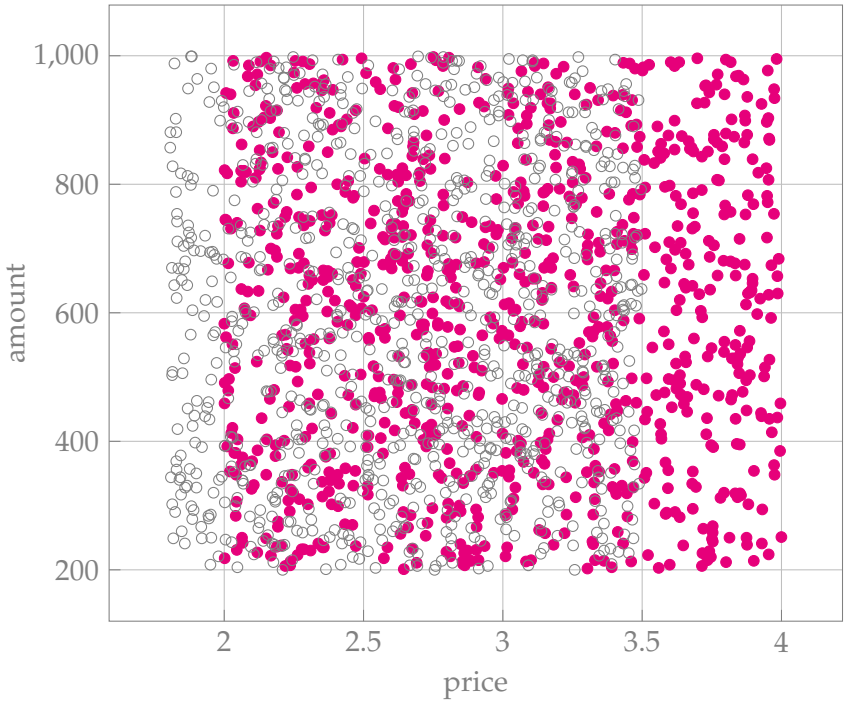


Figure 5.1: Scatter plot depicting the distribution of the randomly generated requests, grey, and offers, pink, used for the first benchmark. The units for amount and price are superfluous for the benchmarks, as the focus is on execution time.

clusive each benchmark result should be.

```
Criterion::default()
    .measurement_time(Duration::new(60,0))
    .sample_size(30)
    .confidence_level(0.98);
```

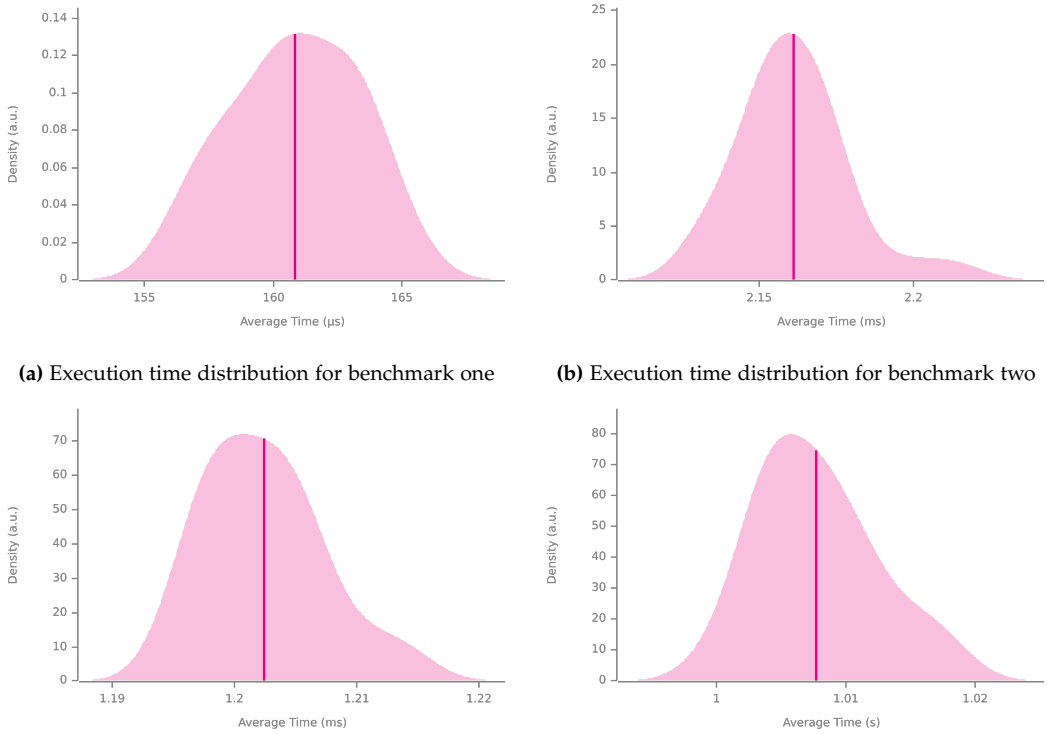
Criterion runs additional iterations automatically if the specified time limit allows it. A target confidence level of 0.98 means that the benchmark aims to have at most a 2% probability of not covering edge cases. It is important to point out that the confidence level relates to the execution environment and is unrelated to the inputs and execution paths of the function itself.

5.1.3 Results

I present each benchmark's result with additional execution details in Listing 5.2. Furthermore, I show their associated distribution graphs in Figure 5.2.

```
Benchmarking: One asset for each of the 1,888 proposed MG
    Warming up for 3.0000 s
    Collecting 30 samples in estimated 60.070 s (371k
       iterations)

    time:   [158.40 µs 159.51 µs 161.01 µs]

Benchmarking: Ten assets for each of the 1,888 proposed MG
    Warming up for 3.0000 s
    Collecting 30 samples in estimated 60.940 s (28k
       iterations)

    time:   [2.1602 ms 2.1672 ms 2.1759 ms]

    Found 1 outliers among 30 measurements (3.33%)
    1 (3.33\%) high mild

Benchmarking: Block transaction limit (9,850 energy assets)
    Warming up for 3.0000 s
    Collecting 30 samples in estimated 60.225 s (50k
       iterations)

    time:   [1.1994 ms 1.2018 ms 1.2046 ms]

Benchmarking: Large over-approximation (4,000,00 energy assets)
    Warming up for 3.0000 s
    Collecting 30 samples in estimated 72.168 s (60
       iterations)

    time:   [1.0059 s 1.0077 s 1.0097 s]
```

Listing 5.2: Benchmarks results for the trade matching algorithm. I have modified the output to increase readability.



(a) Execution time distribution for benchmark one

(b) Execution time distribution for benchmark two

(c) Execution time distribution for benchmark three

(d) Execution time distribution for benchmark four

Figure 5.2: Distribution of execution times for the four trade matching algorithm benchmarks.

As seen in the results above, running 30 samples of the trade matching algorithm using 1,888 energy assets give an upper bound execution time of 161.01 microseconds. For trading with 18,880 and 9,850 assets, this increases to 2.1602 and 1.1994 milliseconds, respectively. I also ran a benchmark using a large over-approximation of 4,000,000 energy assets, equivalent to more than one energy asset from each of the 2,788,291 individual households in Denmark. This benchmark resulted in an upper bound execution time of 1.0097 seconds.

5.2 Micro Grid Data Extraction

For this experiment, I ran the cluster depicted in Figure 5.3, containing an MG that constructs empty blocks and perform no energy trading. Due to the lack of energy assets, the experiment shows how the testing environment can extract data from MG implementations but not PolkEM's correctness and performance.

I used the logging agent to extract proposed and finalised blocks with timestamps. I then calculated the delay between a block's proposal and finalisation and the throughput within a fixed time window with these.

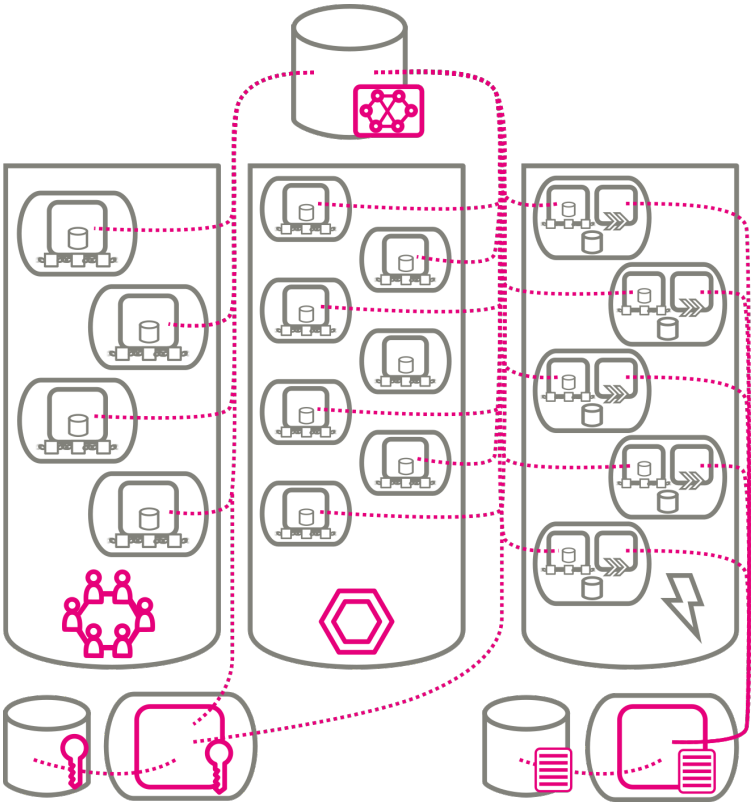


Figure 5.3: Overview of the Kubernetes cluster during the experiment's execution.

5.2.1 Setup

I started the experiment by executing the `create_kind_cluster.sh` script mentioned in Section 4.2. The script instantiated a local Kind cluster with the required persistent volume references and ingress controller objects. The volume references pointed to directories within the `/tmp/` directory on the physical machine, automatically removing the data when no longer needed.

The user must follow the five steps below to test an MG implementation in the environment.

1. Generate the MG chain specification and reference it in PolkEM.
2. Generate the MG genesis state and WASM.
3. Inject the content of the genesis files into the specification of the relay chain as a parachain.
4. Recompile the specification for the relay chain.

5. Create a Docker image of the MG node and install a release of the `parachain` Helm chart with the desired configuration.

After following these steps, I launched the blockchain nodes by executing the `polkem-launch.sh`-script. The script launched the relay chain with seven nodes, the mediator with four nodes, and instances of the account manager, logger agent, and ingress. Additionally, I included instantiation of the exemplary MG with five nodes, one of them being the boot node.

After several minutes, the entire cluster was up and running, authoring and finalising blocks in the chains.

5.2.2 Execution

I calculated the desired metrics by inspecting the log file generated by the logging agent after the cluster had been running for roughly 45 minutes. The log contained entries for each MG node's observation of proposed and finalised blocks. To obtain latency and throughput data, I removed entries logged before all five MG nodes were connected and after the 30-minute mark.

The execution itself was automated and required no manual interference. However, I connected K9s, a CLI tool for observing Kubernetes clusters [48], to verify that all nodes were running correctly and included in their associated chains.

5.2.3 Results

The extracted data included 45 initial lines preceding the fully launched MG network. These lines are present because Kubernetes sequentially starts each pod in the stateful set, waiting for pod i to be ready before initiating pod $i+1$. The results presented here will exclude these 45 lines, where only a subset of the desired nodes were connected. Additionally, I have removed 305 trailing entries to limit the experiment to 30 minutes. Finally, I excluded two blocks only logged as proposed or finalised. With the excluded lines, the result contained 1,440 log entries. I show a snippet of the log file in Appendix C.1.1. All processing and analysis of the logs were conducted using a set of CLI commands and scripts, shown in Appendix C.1.2.

All five MG nodes logged their observations, meaning each proposal and finalisation appeared five times. Hence, 288 entries are unique block proposals or finalisations, resulting in 144 blocks.

Latency

Comparing the delay between the first proposal and finalisation for each observed block gives an average delay of $\sim$15.85 seconds, with a lower and upper bound of

14 and 20 seconds, respectively. I depict the delays' distribution in Figure 5.4.

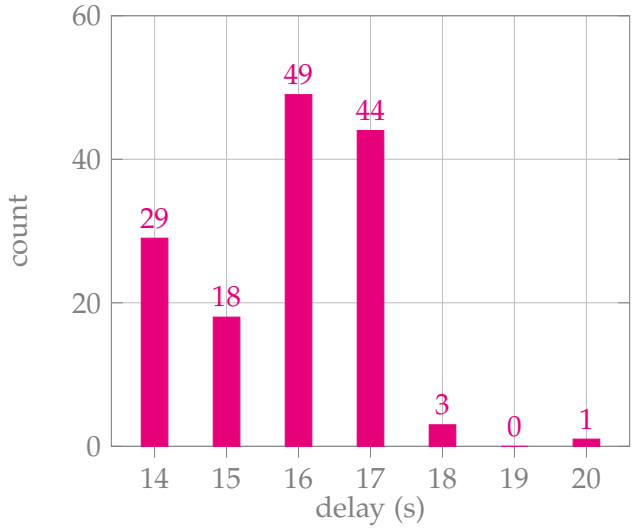


Figure 5.4: Plot of the delays recorded during the experiment and the occurrences of each delay.

Throughput

Based on the experiment, the MG produced 144 blocks during the 30-minute window. However, I excluded one block proposal and one block finalisation due to their associated finalisation and proposal not being included in the cleaned log. Therefore, the MG effectively produced 145 blocks during the experiment. I present the deduced throughput in Equation 5.1.

$$\frac{145\ blocks}{30\ minutes} \approx 4.833\ blocks/minute \tag{5.1}$$

In the next chapter, I discuss these results and the project as a whole.

Chapter 6

Discussion

In this chapter, I discuss whether PolkEM and the testing environment answer the problem definition posed in Section 2.4. The first section focuses on the overall aspects and builds on the succeeding sections that cover each aspect in more detail. I first discuss PolkEM's theoretical latency, throughput, and security based on the presented architecture and proposed implementation details. After this, I discuss the testing environment's features and performance. I conclude the chapter with a discussion on using Substrate and Polkadot, followed by a discussion of the project's scope.

6.1 Overall Viability

In this section, I discuss whether PolkEM and the testing environment answer the problem definition presented in Section 2.4. The claims and arguments made in this section are further detailed and discussed in Section 6.2, Section 6.3, and Section 6.4.

I have designed PolkEM to support direct prosumer trading and heterogeneity of MGs. The platform can support any MG built as, or connected to, a Substrate-based parachain with a viable state-transition function. Under the assumption that the MGs support prosumers, PolkEM alleviates the need to depend on a single centralised entity for buying and selling prosumer energy. I, therefore, conclude that the presented architecture ensures that PolkEM meets the two associated requirements, R_P6 and R_P7, presented in Section 2.4.1.

PolkEM allows every household in Denmark to participate in trades within a few minutes. However, most prosumers in the current system have their consumption and production settled on an hourly basis [21]. Based on this, PolkEM is capable of supporting the Danish energy market. If, for instance, cross-grid trading was

conducted every 15 minutes, the first ten could be used for local trades within each MG, allowing for more advanced trading mechanisms that, for instance, incorporate privacy and real-time traceability, such as the one presented by Hossain [49]. The ideal interval for cross-chain trading should be concluded based on real-world tests and could change as new technologies are developed and adopted.

To prevent the injection of false data, smart storage systems able to provide cryptographic signatures with proof of supply could be used. These could ensure the existence of any local supply, before submitting it. Such details must be specified within each MG, but studies on privacy-preserving and real-time traceable algorithms for MG trading have shown promising results [49, 50]. One platform-level measure useful in the prevention of false data is forecasting. By including relevant metrics, such as weather conditions, and previous consumption and production from each MG, the validators on the relay chain could check whether deviations in the submitted surplus from a given MG are within an acceptable range. Forecasts would require extensive experimentation and possibly depend on artificial intelligence to identify these ranges and deviations. However, it could be one way of protecting the platform against false data.

Comparing PolkEM to the Quarterstorm pilot project [18, 19], scalability is improved. Even if using the current Polkadot version for the relay chain, the platform would still be able to support well over 900,000 users, based on using 100 parachains and accounting for the mediator and grid-intermediary using a slot each. Hence, vastly more than the 500 users of the pilot project. To be competitive, PolkEM would need a user interface and to be evaluated in a similar real-world study. These aspects are outside this project's scope, but Substrate does provide templates for setting up web interfaces compatible with the nodes and accounts running in each chain [51].

The system presented by Wang [12], which also utilised a relay chain similar to Polkadot but without a mediator, faced scalability issues too. PolkEM resolves these by not depending on the relay chain for conducting the cross-grid trades. The authors of the referenced study presented this as their main congestion point. Based on the aspects discussed here, the mediator is assumed to alleviate the issue and improve the platform's scalability.

Establishing PolkEM in Denmark as an alternative to the current centralised system in its current state would not be feasible. The limitations come from using Polkadot as the relay chain, limiting the number of MGs to 98. One way to bypass this limitation is to introduce a multi-tier architecture. For this, I propose an additional layer of relay chains, each aggregating up to 100 MGs. These relay chains could then connect to an overarching relay chain, identical to the one presented in this project. The intermediary layer of relay chains adds another step in cross-grid trading, equivalent to a round of finalisation and authoring, increasing

the latency by 15-20 seconds based on the conducted experiments. This increase in latency would still allow cross-grid trading every 15 minutes but could also increase the computational load for tracing any misbehaviour in the system for future iterations.

PolkEM also improves the conditions for prosumers, as prices are assumed to be higher than those offered by the current retailers. The price of one kilo-watt-hour produced by a prosumer and sold to a retailer is roughly 0.25 DKK [52]. The retailer sells the same kilo-watt-hour priced at 2-3 DKK [53]. This large gap between buying and selling prices cover the retailer's operational cost, but the gap could be assumed to decrease in a more open market. Due to this, prosumers have more incentive to invest in energy generation equipment, as they can faster recoup initial costs and expect high future earnings. Increased private investment would speed up the transition to full reliance on zero-carbon electricity. Furthermore, it could also make the grid more resilient to failures, as communities become more self-sufficient at a finely-granulated level. For instance, a failure that divides a community's energy network in two is less catastrophic if both halves are partially self-sufficient. The impact is not necessarily negligible but smaller than if one of the halves was left without electricity.

I have shown that the testing environment can run a network with the required chains and provides the user with configuration options. Due to its Kubernetes foundation, it is highly portable and usable for conducting large-scale tests, given a sufficiently configured infrastructure.

Furthermore, it meets the requirements presented in Section 2.4.2, except for R_T11. How to extend it to meet this requirement is discussed in Section 6.4. The proposed approach builds on the presented implementation of the well-behaved nodes, making the necessary features' implementation straightforward.

I deem the testing environment sufficient for conducting tests of PolkEM-compatible MG implementations, based on the discussion in Section 6.4.

Based on the presented arguments, I deem PolkEM applicable to the Danish energy market under the assumption that it can be sufficiently implemented and tested. Furthermore, the testing environment is deemed sufficient as a tool for evaluating MG implementations for the platform, despite not meeting requirement R_T11. Hence, PolkEM and the testing environment answer the problem definition presented in Section 2.4.

In the following sections, I discuss the project's different aspects in further detail to support the claims made in this section.

6.2 Latency and Throughput

The performance evaluation of PolkEM is highly affected by the number of constituent MGs. Therefore, I estimate the number of MGs suitable to instantiate across Denmark. I disregard the division of the physical grid between Funen and Zealand, as well as connections to neighbouring countries in this estimate. However, running two instances of PolkEM can resolve the division, while several intermediary parachains could facilitate trading with each neighbouring country. These parachains would submit energy assets representing import and export. Based on this and the heterogeneity of the MGs, I present several assumptions and estimations that allow for a theoretical discussion of the overall performance. At the end of this section, I discuss how loosening these assumptions affects the presented arguments.

6.2.1 Assumptions and Estimations

The discussion of PolkEM's latency and throughput builds on the assumption that all MGs, the mediator, and the relay chain are implemented using the Substrate framework and configured with the default values for block times, block lengths, and similar. The team behind Polkadot has expressed on multiple occasions that these could be modified to better suit different use cases. For instance, they have mentioned that the six-second block time might be lowered to two or three seconds, affecting the latency and throughput discussed in this section [54]. However, determining the optimal configuration for each MG is outside this project's scope, and I use the default values as a balance between efficiency and security.

Furthermore, I use the scenario with the highest mediator load, where all MGs forward an aggregated energy asset to the mediator for every block they produce. In the real world, it is likely that each MG only gets involved with cross-grid trading at fixed intervals, as this would increase the chances of resolving consumption and production locally. However, the MGs are heterogeneous, placing their internal logic out of PolkEM's control.

I present the assumptions and estimations under the assumption that the relay chain uses a more mature version of Polkadot. Furthermore, I use concrete data sizes of the included elements and public statements from the Substrate and Polkadot teams.

Number of Micro Grids

To better reason about the scalability of PolkEM, I here provide an estimate on the number of MGs needed. I base this on the mentioned 2,788,291 Danish households and the division of Aalborg municipality, which includes 69 districts [55].

Some of these districts are small suburban towns, while others are parts of Aalborg city, meaning their energy consumption and production profiles vary. Due to this variety, I propose a separate MG for each district.

There are 101,888 households in Aalborg municipality [56], resulting in an average of ∼1,477 across the 69 districts. Applying a similar division across Denmark would result in 1,888 MGs, as shown in Equation 6.1.

$$\left\lceil \frac{2,788,291\ households}{1,477\ households/MG} \right\rceil = 1,888\ MGs \tag{6.1}$$

Limitation on Parachains

The mediator can support the 1,888 MGs proposed above, assuming that each MG forwards at most five aggregated energy assets in each cross-grid trading round. I derive this from the 9,850energy assets per block presented in Section 3.1.2. However, the relay chain is based on Polkadot, which currently limits the number of parachains to 100[57, 58]. This is a temporary limit used while Polkadot reaches a more mature state. No specific details on the future number of supported parachains have been published, but the Polkadot team is confident that it can be pushed far beyond 100, based on recent advancements[57, 58].

One of the limiting factors is the data stored on the relay chain for each referenced parachain block. The data is listed below with their associated bit sizes [59] and takes up 1664 bits or 208 bytes.

- The parachain ID (32-bit).
- The collator's ID and signature (32-bit and 64-bit).
- A hash of the parent block's candidate receipt (256-bit).
- A Merkle root of the block's erasure-coded pieces (256-bit).
- A Merkle root of any outgoing messages (256-bit).
- A hash of the block (256-bit).
- The state root of the parachain before block execution (256-bit).
- The state root of the parachain after block execution (256-bit).

The size of each relay chain block is 5 MB [33], allowing for a little over 25,200 referenced parachain blocks in each relay chain block. However, each relay chain block also contains data on recent disputes, active validators, members of the voting, and other state data. With 1,888 MGs, the mediator, and the grid-intermediary parachain, 7.5% of the block size would be parachain block references, assuming one reference to each parachain in every relay chain block.

Limitation on Validators

Another limitation is the number of validators, as each parachain requires its own validator set. The suggested ratio is five validators for every parachain, requiring 9,450 validators to support 1,888 MGs, the mediator, and the grid-intermediary. The conservative estimate for the current Polkadot implementation is 1,000 [57]. Hence, the proposed platform would require a 10-fold increase, likely impairing the performance. However, most of the protocols executed by the validators only utilise a subset to decrease latency.

The current voting set used for running the approval checking protocol contains 30 validators. Hence, PolkEM would use 284 if scaled proportionally to the number of validators. Message passing in a peer-to-peer network using robust multicast requires $O(n^2)$ messages, as all n nodes need to send a message to the $n-1$ other nodes [60]. With this protocol, the required number of messages increases from 900 to 80,656, increasing the block time considerably. Based on this, the ratio between the active validators and the size of the voting set would have to be modified to achieve sufficient latency and throughput. A decrease in the voting set size hurts PolkEM's attack resilience, as its size directly affects the number of malicious or faulty nodes PolkEM can tolerate. Alternatively, a more efficient messaging protocol could be used, such as Pastry, using $O(log_B(n))$ messages [61]. Pastry uses prefix routing and a typical B value of 16, resulting in a negligible increase of messages when using 284 validators instead of 30.

The relay chain could be implemented as a consortium chain to mitigate the risk of faulty or malicious nodes, but this would require an intricate screening and validation process for validator candidates. However, the process could reintroduce the issue regarding a small set of entities controlling the system.

The number of parachains and validators supported by Polkadot in the future is unpredictable. However, for the remainder of this chapter, I assume that the latency and throughput from the experiment conducted in Section 5.2 can be achieved with a more mature and potentially further modified Polkadot-based relay chain.

Most Impactful Protocols

Some of the mentioned processes and protocols have a higher impact on latency and throughput than others. For instance, block authoring is negligible compared to approval checking. Furthermore, the experiment conducted in Section 5.2 provides estimates for the authoring and finalisation. Additionally, the protocols for collation, backing, and availability, presented in Section 3.2.3, are included in these estimates. They have a minor impact on performance due to only relying on a small subset of nodes. I detach the approval checking and dispute protocols from the estimates because they were derived from a controlled environment and are not representative of the execution time of the approval checking in an uncertain

environment. Furthermore, they bypass the dispute protocol completely. These protocols are also affected by the number of validators and voters, making them a central point in the discussion of PolkEM scalability.

The protocol for cross-chain messaging was also excluded from the conducted experiment, as the experiment did not establish the required connections. Therefore, I also discuss this protocol in more detail.

6.2.2 Approval Checking

During approval checking, each validator in the voting set keeps a local representation of the given parachain block's state. This state has an overview of when each validator in the voting set is assigned to cast their vote, the tick at which the block was first observed, and the current number of approvals.

The assignments are calculated at each validator using the VRF, which takes the parachain ID and the relay chain's BABE credentials as input, see Section 3.2.4. Each assignment is associated with a delay tranche, each with multiple validators associated, used to determine when a given validator should perform approval checking. Spreading out these checks ensures that only the needed number of delay tranches and, in turn, validators are involved. After a check has passed, a signed approval message is propagated through the network, updating the parachain block's local state at each validator. In the case of a failed check, meaning that the validator determines that the block is invalid, a dispute is initiated.

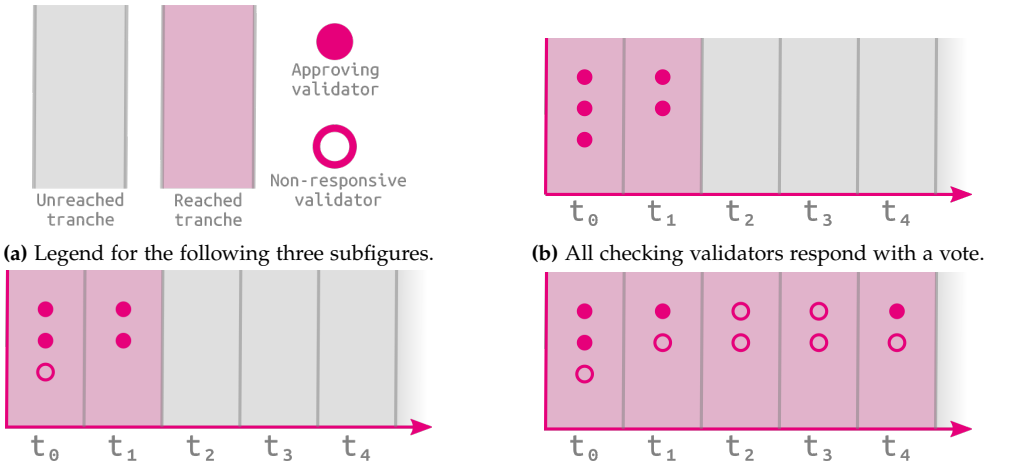
The configuration of the delay tranches influences the process's efficiency based on the parameters listed below. The Polkadot community is still discussing the best values for these parameters, as seen on GitHub [62]. I present the values used in the `v0.9.37` release.

- **MAX_TRANCHES**: The maximum number of delay tranches that indirectly dictates the time it takes to check each parachain block when the entire voting set is involved. The default value is 89, meaning the last validator would check the block 44.5 seconds after observing its availability. During normal execution, this tranche will not become relevant.
- **MIN_CHECKERS**: The minimum number of passed checks required for a parachain block to be approved. The default value is 30. Delay tranche zero, starting at tick zero, is designed to have this number of validators, ensuring that, in the best case, only very few delay tranches are reached.
- **NO_SHOW_SLOTS**: The number of slots available for each validator to conduct the approval check before casting their vote. The default value is two,

meaning each validator has 12 seconds to perform the check. This is equivalent to an entire relay chain block and is unlikely to become necessary. It is important to note that the approval checking and the block time are not connected. Successful execution of the approval checking protocol places the checked parachain block in the inclusion queue, waiting for authoring of the next relay chain block. The two timings are unrelated, but both increase the finalisation time for parachain blocks.

The delay tranches $(t_1, ..., t_{MAX\_TRANCHES})$ are only used if needed, for instance, if a checker disappears or does not submit an approval in time. The approval checks in each delay tranche are only triggered if the minimum number approvals are still missing at the start of that delay tranche.

Using delay tranches results in a best-case approval time close to the checking time and propagating the result. Under poor network conditions, such as a large set of malicious nodes or high networking delays, the protocol will compromise on latency to ensure correctness by utilising more delay tranches. Figure 6.1 depicts three scenarios, each reaching a different number of delay tranches.



(a) Legend for the following three subfigures.

(b) All checking validators respond with a vote.

(c) One checking validator becomes non-responsive.

(d) Multiple checking validators become non-responsive, requiring five delay tranches.

Figure 6.1: Depiction of reached delay tranches in three scenarios, requiring a minimum of four checkers. [63]

The three presented scenarios illustrate how few delay tranches are used during approval checking, only requiring additional tranches in rare cases where multiple checkers become non-responsive.

As mentioned, the approval checking protocol can take up to 44.5 seconds if, for instance, a large subset of the voting set disappears. However, this is not a regular event because tranche zero already assigns the minimum number of checkers, and using each delay tranche is only necessary if enough validators disappeared

during the previous one. Due to this, the latency of the approval checking is not very high. I include an additional ten delay tranches to the experiment, accounting for a less controlled environment. This results in an additional latency of five seconds. I retrieved the results from a controlled environment, and the five-second extension is assumed to account for minor deviations expected from a real-world execution.

Needing to rely on the last delay tranche, triggered at tick 89, is very improbable, and I spend no further time discussing its effect on latency and throughput. Suffice it to say that it would increase the time to conduct approval checking to 44.5 seconds, still keeping the trading time presented in Section 6.2.5 within the target of five minutes.

6.2.3 Disputes

The frequency of disputes is expectedly low, and their impact on latency and throughput is not critical to the general latency and throughput of PolkEM. However, they are not infeasible. They compromise latency and throughput, primarily through delayed finality for parachain blocks. Disputes stall finality because they mean that validators disagree on the parachain states, making it impossible to determine which state to extend the chain from.

Triggering a dispute means that at least one validator, either backing the parachain block or participating in the voting set, disagrees with another validator regarding the block's validity. A dispute requires at least $\frac{2}{3}n + 1$ validators to agree on the block's validity by casting votes. The process consists of the following steps.

1. Collect all statements about the associated parachain block, including those for backing, approval checking, and disputes.
    - If a statement from validator v_i is collected, they do not participate in the remaining steps.
2. Fetch data using the availability protocol.
3. Extract relevant information from any recent relay chain blocks.
4. Perform the validation check using all fetched data.
5. Issue a statement of dispute participation, containing the result of this check.

Step four is expected to have no considerable impact on the delay, as it can be performed at each validator in parallel, starting when each validator has collected all necessary data. Therefore, the execution time primarily depends on time

needed to fetch all previous statements on the block's validity, step one, and required data for the check, steps two and three. I cannot provide a concrete execution time for these steps, but I can deduce an estimate from a recent event on the Polkadot main chain [64].

The event occurred when a validator started disputing valid parachain blocks, initiating 60,614 disputes in 20 hours. The event increased the block time on the relay chain to 18 seconds, 50%, and stalled the finality of all parachain blocks. The increase in the block time came from the need to include all disputes in every block. Despite the high number of disputes and increased block sizes, the validators were able to resolve each dispute quickly. The disputes' distribution was not uniform across the 20 hours, but assuming this, ∼50 disputes were resolved every minute, see Equation 6.2. [64]

$$\left\lfloor \frac{60,614\ disputes}{20\ hours \times 60\ minutes/hour} \right\rfloor = 50\ disputes/minute \tag{6.2}$$

These number give an average resolution time of ∼1.19 seconds. The 20 hours included a window during which no disputes were initiated, further decreasing this delay.

Based on the presented arguments and real-world event, PolkEM's theoretical overall latency and throughput estimates do not include disputes. However, it could be interesting to conduct experiments on their frequency and impact on PolkEM at a large scale.

6.2.4 Cross-Chain Message Passing

As I introduce a mediator parachain to facilitate cross-grid trading, the latency and throughput of message passing between connected MGs and this mediator becomes one of the main latency and throughput concerns of PolkEM.

As I mentioned in Section 3.3, the message parsing process involves the construction of a message during block authoring on the sending chain, collating the block, having it finalised, and then processing the messages during block authoring on the receiving chain. I use the upperbound block delay encountered in the conducted experiment, presented in Section 5.2.3, and assume that the approval process will require an additional ten delay tranches, as mentioned in Section 6.2.2. Hence, the messaging delay, Δ_{XCMP}, is 50 seconds, as shown in Equation 6.3. The 50 seconds cover passing a single message from an MG to the mediator.

$$\Delta_{XCMP} = 20\text{s} + 5\text{s} + 20\text{s} + 5\text{s} = 50\text{s} \tag{6.3}$$

The equation includes the two timings twice because both an MG block and

a mediator block must reach finality to ensure correctness of the process. Based on this, each round of cross-grid trading would take 100 seconds, as the process includes one message from each MG to the mediator and a response containing the cross-grid trades from the mediator to each MG.

It is important to note that I assume the mediator block containing the calculated trades gets finalised before authoring of the block with mediator node signatures starts. The authoring of the second block could begin while the first block is going through finalisation. The experiment in Section C.1 supports this assertion, containing logging proposed blocks before the previous one reaches finality. If authoring of the second block occurs before the first block reaches finality, the presented delay is decreased.

Basing this value on the results of an experiment conducted in a local Kubernetes cluster without any transactions makes it unlikely to match the real-world performance. However, the live Polkadot chain consistently achieves a 12-second block time, except during extraordinary events or attacks, while aggregating 100 parachains [65] and using 297 validators [66]. A 12-second block time would result in Δ_{XCMP} decreasing to 36 seconds, as the authoring would take six seconds for each of the two blocks, while finalisation would require an additional 12 seconds. The overall latency of cross-grid trading would then be 76 seconds. I use the calculated 100 for the remainder of this section.

6.2.5 Summary

Based on the latencies of approval checking, disputes, and message passing, I present an overview of PolkEM's overall latency and throughput.

The latency of trading an energy asset on PolkEM is highest when depending on cross-chain trades from the mediator, giving a latency equivalent to 100 seconds, as shown in Equation 6.4. This calculation is based on the highest delay encountered during the test environment experiment, accounting for an additional ten delay tranches during approval checking.

$$\Delta_l = 2\Delta_{XCMP} = 100\text{s} \tag{6.4}$$

The average finalisation delay of the experiment was $\sim$15.85 seconds, which would result in a latency of 83.4 seconds, still accounting for ten additional delay tranches.

Based on the above calculation, PolkEM theoretically meets the latency requirement, R_P1, presented in Section 2.4.1.

The throughput of PolkEM is lowest in the case where all MGs submit an energy asset for each block, as it increases the number of assets passed to and processed by the mediator. This case forwards 1,889 messages to the mediator, including the grid-intermediary parachain. Each passed message aggregates the surplus energy assets of the sending MG, processed on the mediator during the authoring of the following mediator block. Using the energy asset and block sizes presented in Section 3.1.2, the mediator can process 9,850 energy assets within a single block. Matching these assets and constructing trades can be performed during block authoring without significant overhead, as shown in Section 5.1.

Based on these arguments, the theoretical throughput of PolkEM is 18,596,800 transactions per 100 seconds, as seen in Equation 6.5.

$$9,850\text{t} \times 1,888 = 18,596,800\text{t} \tag{6.5}$$

Based on this calculation, PolkEM theoretically meets the throughput requirement, R_P2, presented in Section 2.4.1. Furthermore, a configuration using half as many MGs would still meet this requirement, and so would one with twice as many MGs. I have depicted these hypothetical situations in Figure 6.2, together with the proposed 1,888 MGs.

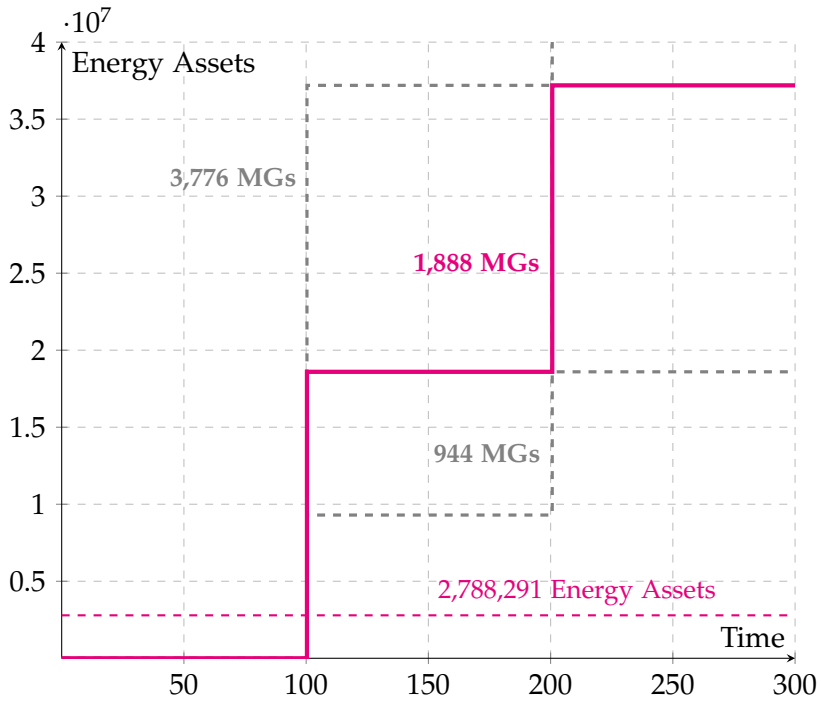


Figure 6.2: Graph showing the number of energy assets included in trades per second for PolkEM, utilising different numbers of MGs. The dotted pink line represents the target number.

The numbers presented here assume that MGs cannot forward energy assets to the mediator during the consensus process. However, this could be possible, as the processed message size is far below the maximum size of the block. Hence, constructing trades and agreeing on them could be done during a single block.

6.3 Security

The relay chain and the Substrate framework highly influence the security of PolkEM because of its dependence on these for most of the functionality. Additionally, PolkEM cannot assume the secureness of each MG, and the relay chain will therefore be the platform's only protection from malicious MG nodes. In the following sections, I present discussions on PolkEM's privacy and traceability, followed by one on its attack resilience.

6.3.1 Privacy and Traceability

PolkEM is only responsible for validating the state transitions of MGs and cross-grid trades. Therefore the privacy of each prosumer is ensured, from the relay chain's perspective, by aggregating energy assets into a single MG level one. Traceability at this level only includes validation checks for each parachain block and can only trace the actions of the MGs. Hence, each MG must incorporate sufficient mechanisms to ensure these properties at the prosumer level. Based on this, PolkEM meets requirements R_P3 and R_P4 but not at the prosumer level.

In my ninth-semester project, I argued that any misbehaviour detected during the validation of its blocks could result in a fine of the MG, which would incentivise improved mechanisms to avoid collating invalid blocks. Studies on different privacy-preserving trading mechanisms that ensure traceability in the case of misbehaviour have been conducted [49, 67]. One approach is to use reputation scores, which affect how trades include each participant. The relay chain could implement similar scores. These would be modified based on the validity checks conducted on each MG's collated blocks. Each MG would then be liable for all its participants and have to incorporate traceability mechanisms to trace the actions of each node.

It could also be desirable to conduct screenings of MGs wanting to participate on the platform. During this screening, validators could inspect each MG's source code, checking for the implementation of traceability, attack detection, and attack mitigation features. However, this is outside the scope of this project.

One way for MGs to incorporate traceability in the current system would be to store all energy assets aggregated in each collated block while it awaits finality, as mentioned in Section 3.3. If the validity check fails, the MG would have a record of all involved accounts and could start running checks on their recorded actions.

Assuming that PolkEM should enforce traceability from the relay chain down to the individual MG nodes, a more sophisticated communication protocol between the relay chain and MGs is needed. One example would be the cross-grid trading mechanism presented by Wang [12], which utilises smart contracts and a verification subgroup in each MG. Implementing a slightly modified version that communicates with the relay chain instead of other MGs would force each MG to incorporate the mechanism into its internal logic. Each collated energy asset, and its constituents, would be traceable due to the inclusion of signatures from the originating account. However, it would compromise PolkEM's support for heterogeneity, increase the size of collated energy assets, and extend the collation process with several steps.

The increase in energy asset size could be acceptable, as a minor decrease in the theoretical throughput of PolkEM would not harm its viability. Furthermore, if each MG does not participate in cross-grid trades for every block they produce, a slightly higher latency would not be an issue. This approach could therefore be further investigated and compared to other alternatives.

Based on the presented options, I suggest implementing different approaches and conducting a comparative analysis based on their latency and throughput. Furthermore, I propose evaluating these results in the context of each approach's effect on heterogeneity.

6.3.2 Attack Resilience

The relay chain uses NPoS, demanding that each validator has placed a stake and is backed by several nominators, each with an additional stake. On Polkadot, each validator has a combined stake of 2 billion DOT behind them, ∼12 million US$ at the time of writing [68][1]. Each detected attempt to attack the platform would result in the slashing of 10 - 20 validators[2]. The percentage deducted from each validator's account dependens on the severity of their misconduct, but approving a block later determined invalid results in 100%, equivalent to a total of 120 - 240 million US$ across the slashed validators.

The main security argument of NPoS consensus algorithms, including Polkadot, is the Gambler's Ruin [69]. The idea is that by requiring each participant's stake to exceed the potential profit of an attack and ensuring a high probability

[1] Fluctuations in the actual value of a DOT affects this value and a potential crash would dissipate any security implications of the stake. However, based on the all time low, ∼ 2.69 US$ [68], and all time high, ∼ 55.00 US$ [68], evaluations, the staking behind each validator has stayed between ∼5.38 million and ∼110 million US$. A required stake equivalent to any value within this bound would make the feasibility of an attack low.

[2] This is based on Polkadot with 900 validators. The number of slashed validators is affected by multiple factors, including the size of the active validator set and the number of checkers

of being detected, attackers will never be able to make a profit, regardless of their strategy.

Requiring high stakes behind each validator disincentivises profit-based attacks. However, this is only true if the probability of detection is high. Furthermore, it will not necessarily prevent attacks that aim to bring down the grid. Such profitless attacks conducted by foreign states are a considerable concern for vital infrastructure, such as the energy sector, as documented in the Danish National Risk Profile of 2022 [70].

Based on the importance of detection, I focus on the mechanisms and mitigation techniques used by PolkEM to increase the detection probability. The main protocols for this are approval checking and disputes.

Approval Checking

In addition to what I describe in Section 3.2.5, the approval checking increases the security of block finalisation and detection of malicious behaviour through the properties listed below.

1. Assignments are kept secret until each checker reveals itself, which makes it impossible to initiate targeted Denial-of-Service attacks faster than these reveals are received. Alternatively, the attack must target all validators in the set, increasing its cost and overhead.

2. Assignments are deterministically generated to increase the chances of including an honest nodes as checkers in the set and make assignments verifiable.

3. A checker broadcasts their intention to check a block before recovering the necessary data. Other nodes are alarmed if the checker disappears after this broadcast, which triggers additional checks.

4. Disappearance of a checker triggers an assignment of multiple new checkers for the block. Resulting in an increased number of checkers for each disappeared node.

These properties ensure that a potential attacker needs to control all selected checkers or silence every other validator for an attack to succeed. If any well-behaved node performs a check, it will conclude that the block is invalid, initiating a dispute. Hence, the approval checking process is highly secure, assuming that at least $\frac{2}{3}n + 1$ checkers are correct and well-behaved. Due to this, the configuration for the voting set and the minimum number of checkers hugely impacts PolkEM's security.

Disputes

Initiation of disputes trigger a vote to determine the validity of a parachain block. This vote requires at least $\frac{2}{3}n + 1$ validators to agree on its validity, based on BFT [27]. When a subset of this size is well-behaved and correct, all opposing nodes are slashed. These nodes have either voted an invalid block valid or a valid block invalid. These two cases constitute the majority of attacks that alter the chain's progress and state. Under the presented assumption, disputes are able to ensure the validity of all referenced parachain blocks and correctness of the relay chain.

Based on the presented security properties, PolkEM meets the attack resilience requirement R_P5, assuming that the number of faulty or malicious nodes is below the number of checkers required for the approval checking process.

6.4 Testing Environment

The intentional use of the testing environment extends beyond the experiment conducted in Section 5.2. Therefore, I evaluate how well it supports the intended use cases. I discuss its features, design, performance, and whether it meets the requirements presented in Section 2.4.2.

6.4.1 Features and Requirements

I have implemented the testing environment to form a solid foundation for conducting tests of PolkEM-compatible MG implementations. As shown in Section 5.2, I can launch the testing environment with an instance of the relay chain and mediator. Furthermore, these chains are connected, collating and finalising blocks. I also instantiated and connected an exemplary MG with a different internal logic. Based on this, the testing environment meets the requirements R_T1, R_T2, and R_T3.

The experiment also represents how the testing environment facilitates data extraction through the logging agent and provides a general way to instantiate sidecar applications that trigger node actions. Even though the presented application only subscribed to blocks, the user can similarly trigger actions on the node. Furthermore, the logging agent incorporates timestamps for logged events to enable the user to reason about the timing of these, as done in Section 5.2.3. The testing environment, therefore, meets requirements R_T4, R_T5, and R_T9.

I built the testing environment as a Kubernetes cluster, allowing it to run on any infrastructure and hardware configuration supporting Kubernetes. Furthermore, I introduce configurability through Helm charts to fit these infrastructures. Utilising Helm charts also enables the user to configure the number of nodes and instantiate

multiple MGs based on a generic chart specification. These design choices ensure that the testing environment meets requirements R_T6, R_T7, and R_T8.

I introduced an account manager that deterministically provides each node with a private key, allowing each node to have a specific account to use for connecting to it. Based on this, the testing environment meets requirement R_T10.

This leaves requirement R_T11, "*Inject nodes with different definitions into each chain*". I presented this requirement to support malicious node injection to test the security aspects of MG implementations. Unfortunately, I was unable to implement this feature during the course of this project. However, it could be a separate stateful set and use the chain specification of the MG. Given the time, I would extend this stateful set with additional configuration parameters in the parachain Helm chart. These would be `malicious_node_image`, for specifying the Docker image of the malicious node, `malicious_node_count`, to configure the number of malicious nodes to instantiate, and `malicious_node_runner_image`, allowing the user to specify the sidecar application to run alongside malicious nodes. How to construct the Docker image to join the chain and also misbehave is too intricate for this discussion and could be an interesting study in its own right.

6.4.2 Usability

One of the crucial aspects of a testing environment is its usability. I did not conduct any usability tests during the project. Therefore, I discuss this aspect based on the number of steps and configurable parameters instead. In addition to this, I also consider the complexity of understanding these steps.

To set up a working instance of the testing environment, I followed the steps outlined in Section 5.2.1. I consider the generation of the chain specification and referencing it as a parameter for the parachain Helm chart an easy step. The same goes for the genesis state and WASM strings' generation, as the exemplary MG repository contains a script for performing this. Building a Docker image of the MG node and referencing it when using the parachain Helm chart is considered a reasonably straightforward step, as a script for this is also available. Hence, the step of injecting the genesis strings into the relay chain specification and recompiling it remains.

This injection and compilation step is tedious, as the WASM string is roughly 1.2 megabytes and must be copied into the right line in the relay chain specification. During the project, I had to modify the allowed memory usage of the integrated development environment before the file could be analysed, making the step highly cumbersome.

Given the time, I would propose automating all five setup steps by requesting the MG node binary as input for the testing environment and programmatically generating the chain specification, genesis state, and WASM. As I already developed most of the necessary scripts, this could be achieved by triggering these from a combined entry point. The only step not already automated is injecting the content of these files into the relay chain specification. However, the specification is a JSON file, easily deserialisable into an object. The object could then be modified through its fields and re-serialised. This way, the user would not have to understand the underlying coupling between the chains' specifications and genesis states.

Automating these steps would increase usability by only requiring the user to provide the MG binary and set a few parameters when triggering the parachain Helm chart.

6.5 Depending on Substrate and Polkadot

I have based all the blockchain implementations on the Substrate framework and the relay chain on Polkadot. This dependence has resulted in several issues, as both these systems are actively under development. I discuss the development process and how working with these technologies has impacted the project in more detail in Appendix A. However, I will discuss whether a solution based on these could form a viable platform for the energy market.

One of the main advantages of Substrate is its modularity, as it enables developers to base their implementations on pre-built components and swap them out as needed, making the framework highly extensible and flexible. However, due to its immaturity, there is limited documentation on configuring complex systems. Furthermore, the number of modules and components can seem overwhelming due to the philosophy of complete decoupling. Decoupling is generally seen as a quality characteristic but can cause a lack of cohesion and decreased usability, as it increases complexity and steepens the learning curve.

Substrate achieves modularity and low coupling through several libraries using generic parameters. Each library contains a set of primitives, instantiated based on the provided generic type. However, these primitives are introduced in multiple libraries to avoid coupling between them, such as the `AccountId`. The definitions are incompatible by default, despite often being identical. Hence, the user must manually specify conversion logic when multiple libraries need to process the same primitive. Keeping track of used definitions and defining conversion logic are tedious, worsened by the following factors.

First, the framework's size results in very slow indexing for development environments, which prevents jumping between references and their definitions, mak-

ing it very difficult to locate a given reference's definition.

Secondly, many of these libraries utilise re-exportation of the primitives, which means that the underlying implementations are identical, but because each library defines them, they are incompatible.

Thirdly, the lack of documentation makes it difficult to understand the expected use of each module. The use of more complex structures composed of multiple primitives increases the difficulty further.

One of Substrate's founders, Gavin Wood, has mentioned that the lack of documentation is by design. The intention is to foster more innovation and spark conversations in the community [71]. He made the following statements during a talk at the Polkadot Decoded 2022 conference [71]. I have removed repeated words and half-finished sentences to increase the readability.

> "*[...] all opinions, ultimately, are going to be design errors. We do it when it's necessary.*"
>
> - Gavin Wood, Polkadot Decoded 2022 [71]

> "*We build reference implementations but stay away from official implementations. As soon as you label something official, you're putting a limitation on where your ecosystem can go.*"
>
> - Gavin Wood, Polkadot Decoded 2022 [71]

> "*We are not going to give everybody a manual in the ecosystem, 'this is how you do x, this is how you do y, this is what happens if you need to do z', no we're not. [...] Therefore, we have to have a much more open approach.*"
>
> - Gavin Wood, Polkadot Decoded 2022 [71]

He follows up the last quote with a statement on how anyone that provides a direct description of how to do something is "*probably giving you a flawed design*" [71]. I will not discuss whether the assumption that fixed opinions inevitably result in errors here, but the decision to avoid official implementations and actively avoid the creation of manuals does increase the learning curve drastically.

A more open approach to developing solutions necessitates a well-structured and responsive community. Substrate aims to achieve this through its stack exchange forum, but as no official documentation is available, most answers are people's own approaches or source code references. How much this helps solve the posed issues is questionable.

Substrate and Polkadot are both currently under active development, which means that changes are continuously released to improve the systems and fix errors. The releases can contain breaking changes, resulting in tools developed by

the community breaking too and becoming obsolete. For instance, a new Substrate release might break one or more tools, requiring the user to deduce which version is compatible across all the tools they intend to use during development. The current state also means that critical bugs and vulnerabilities are still present.

Based on these points, one could argue that Substrate is unsuitable for the development of PolkEM. However, one of the primary selling points of using the framework for this project is its fundamental integration with Polkadot. The integration allows the mediator and future MG implementations to connect to the relay chain by basing them on the parachain template provided by the framework.

If I were to redo this project with the gained knowledge, I would reconsider relying on Substrate. I could have implemented the essential features of PolkEM using general Rust libraries and research articles on the Polkadot protocols. I would have set up a simple peer-to-peer network using the `libp2p` library, developed by Parity Technologies [72], and then extended it with the relevant protocols. This approach could have resulted in a bare-bones prototype but would not have included all the features available in Substrate and Polkadot, such as Aura and BABE. It would also have taken time away from the testing environment implementation.

Whether Substrate and Polkadot will mature and foster a community able to support new developers, as Gavin Wood envisioned, remain to be seen. These aspects are critical for the viability of this more open development approach, as it necessitates a mature community and ecosystem. If enough competent developers are engaged, the number of good answers to general questions on the forum will increase and could eventually become unofficial documentation.

6.6 Scope of The Project

My initial goal for the project was to develop a primitive prototype of PolkEM. I envisioned a relay chain, the mediator, a grid-intermediary parachain, and numerous exemplary MGs. Such a system is an ambitious goal for a Master's thesis project, especially considering I would implement it alone. I assumed that Substrate and Polkadot would facilitate quick implementations of each chain with some simplifications, allowing the project to focus on analysing different configurations and comparing the system's performance with and without the mediator.

Unfortunately, the state of an ecosystem around Substrate were not as expected, and I had to pivot multiple times during the process. The project presented in this report is still quite ambitious, but I hope the results spark new research ideas for decentralised energy markets.

In terms of the project's scope, I am satisfied with the parts presented here,

given the circumstances. I argue that the original prototype would have been achievable with more mature versions of Substrate and Polkadot. The project might have benefited from a narrower scope, but I argue that a combined platform, leveraging recent technological advancements, is the next step for distributed energy market research.

Chapter 7

Conclusion

In this project, I aimed to propose an energy trading platform that leverages a distributed mediator to facilitate cross-grid trading and an accompanying testing environment. I have presented both, demonstrated some of their capabilities, and discussed whether they answer the problem definition. Based on the listed requirements and discussion, both contributions are, in general, satisfactory.

The platform requires Substrate and Polkadot to reach a more mature state before being an appropriate replacement for the centralised system currently used in Denmark. However, the points in the discussion provide potential modifications and improvements that would make PolkEM a competitive system.

The testing environment supports the intended scenarios with the ability to configure numerous parameters to fit each one. I cannot determine the overhead of running extensive simulations on sophisticated infrastructures through the experiments presented in this project. However, many commercial systems build and run on Kubernetes, and I expect the testing environment to scale sufficiently. Furthermore, I did not implement malicious node injection during the project but described a potential approach to do so.

Based on the above, I deem PolkEM constitutes a solution to the problem of distributed energy trading, assuming that the underlying systems reach the required maturity. Additionally, I conclude that the testing environment sufficiently provides a way to evaluate MG implementations intended to run on the platform.

Chapter 8

Future Work

I present some selected features and considerations for extending the contributions in future iterations. I start by briefly discussing general research directions, followed by feature suggestions to help evaluate PolkEM as a platform and improve the testing environment. I have excluded the features necessary to meet requirement R_T11, as I included these in Section 6.4.

8.1 Future Research Directions

The main focus of future iterations should be the implementation of PolkEM. A prototype is necessary to conclusively answer the questions I have only answered under limiting assumptions and theoretical estimates in this project. The prototype should include a modified version of Polkadot as the relay chain, the mediator, and the grid-intermediary parachain. These can be connected using the testing environment and evaluated by instantiating and connecting various MG implementations.

Following this, I would get input from researchers and practitioners in game theory and commodity trading markets to improve PolkEM's market dynamics. The platform might benefit from using a more elaborate algorithm on the mediator or restructuring the way MGs forward their assets to optimise different metrics.

Finally, I would vouch for conducting a real-world pilot project similar to the one in Quarterstorm [19] to collect real-world data on PolkEM's performance and security. Such a project would require a user interface to be available for the participants, achievable with the Substrate frontend template [51].

In the following sections, I will propose implementation details of concrete features for both PolkEM and the testing environment.

8.2 Modified Relay Chain

The relay chain presented in this project uses the default configuration of Polkadot. However, changing the various parameters and underlying mechanisms would be highly advisable to tailor them to PolkEM. I suggest conducting several experiments using a prototype of PolkEM and the developed testing environment with various configurations to determine which parameters could improve the performance and security of the platform.

More specifically, I suggest comparing different values for the voting set size, the total number of validators, the parachain validator set size, and the block times. These four metrics constitute the main factors potentially inhibiting the scalability of PolkEM based on the current Polkadot implementation. Therefore, it is critical to determine the optimal scalability configuration that maintains a high attack detection rate and punishment of detected malicious nodes.

In addition to these four parameters, a large set of MGs should be instantiated on the platform to conduct load- and stress tests. Due to Polkadot limiting the number of connected parachains to 100, the relay chain will likely need further modifications to allow such tests.

8.3 Optimised Mediator

PolkEM's main novelty is the mediator, and how it could be optimised is therefore of interest. I propose merging the two blocks authored for each round of cross-grid trading. Authoring of the first block includes processes of incoming MG messages and execution of the trade matching algorithm. Authoring of the second block mainly consists of collecting the signatures from all mediator nodes, either backing or disputing the set of trades.

Merging these two blocks would mean that only one round of authoring and finalisation would be necessary for one cross-grid trading round. Furthermore, the relay chain would not have to finalise blocks that might include erroneous trades. The finalisation of such blocks could be a potential risk with the current setup, as the first block does not require any backing from the remaining mediator nodes. The first block does not share the unsupported trades with the MGs but does add unnecessary computations to the process.

One drawback of merging the two blocks is achieving consensus on the set of trades during the authoring process. However, because the mediator uses a single author for each block, a leader-based consensus algorithm can be used without the additional overhead of selecting a leader separately. One such algorithm is Raft, a simplified and modern version of Paxos [73], which could achieve this consensus. As shown in Section 5.1.3, the trade matching can be conducted within one second, leaving several seconds for the consensus.

Based on the above, I highly suggest experimenting with Raft to achieve single-block cross-grid trade matching and backing. If consensus is unachievable using six-second slots, it might be valuable to extend these, considering merging the two blocks might still decrease the overall delay.

8.4 Automated Chain Specification Creation

The final feature I suggest is the automation of all chain specification-related tasks. I have already automated the generation of the chain specifications and the genesis states, but injecting these into the relay chain's specification remains tedious.

By automating these tasks, I alleviated the need for the user to construct and inject chain specifications for all chains after each modification. Additionally, the user will not have to run multiple automation scripts in a specific sequence to ensure up-to-date versions.

I propose to develop an application to handle this. As Substrate uses JSON files for the chain specifications, most programming languages could easily deserialise them and allow rigid modification of the resulting objects through named fields. Wanting to represent the specifications as objects makes simple scripts less enticing than a programming language built to handle such abstractions. Furthermore, using Rust, the struct defined by Substrate to represent these specifications can be directly used, decreasing the associated workload.

I suggest that the application takes an MG node binary as input, from which it can then generate the MG chain specification and genesis state files. After this, it can deserialise the relay chain's specification and modify its fields to include the MG chain's genesis state data. After this, the relay chain's specification can be re-serialised and used in the testing environment.

A further addition to this application could be to generate the Docker image for the MG node binary.

Bibliography

[1] Hannah Smith. *Bitcoin crash: what's behind crypto collapse?* URL: https://www.thetimes.co.uk/money-mentor/article/is-bitcoin-crash-coming/. 11-03-2023.

[2] Johannes Sedlmeir et al. "Recent Developments in Blockchain Technology and their Impact on Energy Consumption". In: *CoRR* abs/2102.07886 (2021). arXiv: 2102.07886. URL: https://arxiv.org/abs/2102.07886.

[3] Shahid Tufail et al. "A survey on cybersecurity challenges, detection, and mitigation techniques for the smart grid". In: *Energies* 14.18 (2021), p. 5894. DOI: 10.3390/en14185894.

[4] Jim McNiel. *For a clean energy future, our relationship to the grid must change. Here's how*. 2022. URL: https://www.weforum.org/agenda/2022/07/clean-energy-future-relationship-to-grid-must-change. 19-10-2022.

[5] U.S. Department of Energy. *QER Chapter II Resilience April 2015*. Tech. rep. U.S. Department of Energy, 2015. URL: https://www.energy.gov/sites/prod/files/2015/08/f25/QER%20Chapter%20II%20Resilience%20April%202015.pdf.

[6] Andrija Goranovic et al. "Blockchain applications in microgrids an overview of current projects and concepts". In: *IECON 2017 - 43rd Annual Conference of the IEEE Industrial Electronics Society, Beijing, China, October 29 - November 1, 2017*. IEEE, 2017, pp. 6153–6158. DOI: 10.1109/IECON.2017.8217069.

[7] Esteban A Soto et al. "Peer-to-peer energy trading: A review of the literature". In: *Applied Energy* 283 (2021), p. 116268. DOI: 10.1016/j.apenergy.2020.116268.

[8] Chenghua Zhang et al. "Peer-to-Peer energy trading in a Microgrid". In: *Applied Energy* 220 (2018), pp. 1–12. ISSN: 0306-2619. DOI: https://doi.org/10.1016/j.apenergy.2018.03.010. URL: https://www.sciencedirect.com/science/article/pii/S0306261918303398.

[9] Qiang Wang, Rongrong Li, and Lina Zhan. "Blockchain technology in the energy sector: From basic research to real world applications". In: *Computer Science Review* 39 (2021), p. 100362.

[10] Tonghe Wang et al. "Challenges of blockchain in new generation energy systems and future outlooks". In: *International Journal of Electrical Power & Energy Systems* 135 (2022), p. 107499.

[11] Apoorvaa Deshpande and Maurice Herlihy. "Privacy-Preserving Cross-Chain Atomic Swaps". In: *Financial Cryptography and Data*. Ed. by Matthew Bernhard et al. Cham: Springer International Publishing, 2020, pp. 540–549. ISBN: 978-3-030-54455-3.

[12] Longze Wang et al. "Dynamic adaptive cross-chain trading mode for multi-microgrid joint operation". In: *Sensors* 20.21 (2020), p. 6096.

[13] Disha L. Dinesha and P. Balachandra. "Conceptualization of blockchain enabled interconnected smart microgrids". In: *Renewable and Sustainable Energy Reviews* 168 (2022), p. 112848. ISSN: 1364-0321. DOI: https://doi.org/10.1016/j.rser.2022.112848. URL: https://www.sciencedirect.com/science/article/pii/S1364032122007304.

[14] Qiheng Zhou et al. "Solutions to Scalability of Blockchain: A Survey". In: *IEEE Access* 8.1 (2020), pp. 16440–16455. DOI: 10.1109/ACCESS.2020.2967218.

[15] Shaomin Zhang and Cong Hou. "Model of decentralized cross-chain energy trading for power systems". In: *Global Energy Interconnection* 4.3 (2021), pp. 324–334. ISSN: 2096-5117. DOI: https://doi.org/10.1016/j.gloei.2021.07.006. URL: https://www.sciencedirect.com/science/article/pii/S2096511721000670.

[16] Keith Alfaro. *Architecture*. URL: https://wiki.polkadot.network/docs/learn-architecture. 24-12-2022.

[17] Bill Gates. *How to avoid a climate disaster*. Allen Lane, 2021. ISBN: 9780241448304.

[18] Liliane Ableitner. *District electricity 2.0*. URL: https://quartier-strom.ch/index.php/homepages/quartierstrom2-0/. 22-11-2022.

[19] Swiss Federal Office of Energy SFOE. *Community energy network with prosumer focus*. Tech. rep. Swiss Federal Office of Energy SFOE, 2020. URL: https://www.aramis.admin.ch/Default.aspx?DocumentID=66041&Load=true.

[20] Parity Technologies. *The Blockchain Framework for a Multichain Future*. URL: https://substrate.io/. 18-11-2022.

[21] Christian Dahl Winther. *Visual Guide to the Power Grid*. 1st ed. The Visual Power Grid Company, 2020. ISBN: ISBN 978-87-971959-0-1.

[22] U. S. Energy Information Administration. *Electricity explained*. URL: https://www.eia.gov/energyexplained/electricity/use-of-electricity.php#:~:text=Electricity%20is%20an%20essential%20part,machinery%2C%20and%20public%20transportation%20systems. 05.05.2023.

[23] Emil Søndergård Ingvorsen. *Regeringen vil kræve ekstraskat af energiselskaber for milliardbeløb – men stenrige elhandlere går fri*. URL: https://www.dr.dk/nyheder/penge/regeringen-vil-kraeve-ekstraskat-af-energiselskaber-milliardbeloeb-men-stenrige. 05.05.2023.

[24] Peter S. Mygind. *Konkurrencestyrelsen undersøger 49 elselskaber for at drive kartel*. URL: https://energiwatch.dk/Energinyt/Energiselskaber/article15616526.ece. 05.05.2023.

[25] Interchain GmbH. *What is Tendermint*. URL: https://docs.tendermint.com/v0.34/introduction/what-is-tendermint.html. 25.05.2023.

[26] Werner Schütz. "Fundamental issues in testing distributed real-time systems". In: *Real-Time Systems* 7.2 (1994), pp. 129–157.

[27] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* (1982), 382–401. ISSN: 0164-0925.

[28] Danmarks Statistik. *Husstande, familier og børn*. URL: https://www.dst.dk/da/Statistik/emner/borgere/husstande-familier-og-boern#:~:text=februar%202022-,1.,der%20var%202.397.075%20husstande. 03.05.2023.

[29] Ahmed S. Musleh, Guo Chen, and Zhao Yang Dong. "A Survey on the Detection Algorithms for False Data Injection Attacks in Smart Grids". In: *IEEE Transactions on Smart Grid* 11.3 (2020), pp. 2218–2234. DOI: 10.1109/TSG.2019.2949998.

[30] Web3 Foundation. *A scalable, interoperable & secure network protocol for the next web*. URL: https://polkadot.network/technology/. 03-11-2022.

[31] ParityTech. *Transaction Construction and Signing*. URL: https://wiki.polkadot.network/docs/build-transaction-construction. 10.05.2023.

[32] Parity Technologies. *Address Formats*. URL: https://docs.substrate.io/reference/address-formats/. 02-01-2023.

[33] ParityTech. *Weights*. URL: https://spec.polkadot.network/id-weights#defn-polkadot-block-limits. 10.05.2023.

[34] *What is an empty block's size?* URL: https://substrate.stackexchange.com/questions/2390/what-is-an-empty-blocks-size. 24.05.2023.

[35] OpenEthereum. *Aura - Authority Round - Wiki*. URL: https://openethereum.github.io/Aura. 20.04.2023.

[36] Jeff Burdges. *Availability and Validity*. URL: https://research.web3.foundation/en/latest/polkadot/Availability_and_Validity.html. 19.04.2023.

[37] Joe Petrowski. *Polkadot Consensus Part 3: BABE*. URL: https://research.web3.foundation/en/latest/polkadot/block-production/Babe.html. 19-04-2023.

[38] Bernardo "David et al. "Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain". In: *Advances in Cryptology – EUROCRYPT 2018*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Springer International Publishing, 2018, pp. 66–98. ISBN: 978-3-319-78375-8.

[39] Alistair Stewart and Eleftherios Kokoris-Kogia. "GRANDPA: a Byzantine Finality Gadget". In: *CoRR* abs/2007.01560 (2020). URL: https://arxiv.org/abs/2007.01560.

[40] Web3 Foundation. *XCMP Overview*. URL: https://research.web3.foundation/en/latest/polkadot/XCMP/. 02-01-2023.

[41] Filippo. *Cross-Consensus Message Format (XCM)*. URL: https://wiki.polkadot.network/docs/learn-xcm. 29-12-2022.

[42] John R Douceur. "The sybil attack". In: *Peer-to-Peer Systems: First InternationalWorkshop, IPTPS 2002 Cambridge, MA, USA, March 7–8, 2002 Revised Papers 1*. Springer. 2002, pp. 251–260.

[43] Mousa Marzband et al. "Non-cooperative game theory based energy management systems for energy district in the retail market considering DER uncertainties". In: *IET Generation, Transmission & Distribution* 10.12 (2016), pp. 2999–3009. DOI: https://doi.org/10.1049/iet-gtd.2016.0024.

[44] Elodie Adida and Georgia Perakis. "Dynamic Pricing and Inventory Control: Uncertainty and Competition". In: *Operations Research* 58.2 (2010), pp. 289–302. DOI: 10.1287/opre.1090.0718.

[45] Helm Authors. *Charts*. URL: https://helm.sh/docs/topics/charts/. 24.05.2023.

[46] The Kubernetes Authors. *kind*.

[47] Jorge Aparicio and Brook Heisler. *criterion.rs*. URL: https://github.com/bheisler/criterion.rs. 30.05.2023.

[48] Imhotep Software LLC. *k9s - Kubernetes CLI To Manage Your Clusters In Style!* URL: https://k9scli.io/. 04.06.2023.

[49] Md Tamjid Hossain, Shahriar Badsha, and Haoting Shen. "Privacy, Security, and Utility Analysis of Differentially Private CPES Data". In: *IEEE Conference on Communications and Network Security, CNS 2021, Tempe, AZ, USA, October 4-6, 2021*. IEEE, 2021, pp. 65–73. DOI: 10.1109/CNS53000.2021.9705022.

[50] Qingyou Yang et al. "A privacy-preserving and real-time traceable power request scheme for smart grid". In: *2017 IEEE International Conference on Communications (ICC)*. IEEE. 2017, pp. 1–6.

[51] Parity Technologies. *Substrate Front End Template*. URL: https://github.com/substrate-developer-hub/substrate-front-end-template. 06.06.2023.

[52] Solcelle Konsulenten. *Salg af solcellestrøm*. URL: https://solcellekonsulenten.dk/salg-af-solcellestroem/. 03.06.2023.

[53] Energytilsynet. *ANALYSE AF KONKURRENCEN PÅ DETAILMARKEDET FOR EL*. URL: https://forsyningstilsynet.dk/media/5011/analyse_af_detailmarkedet_for_el.pdf. 08.05.2023.

[54] ParityTech. *Frequently Asked Questions (FAQs)*. URL: https://wiki.polkadot.network/docs/faq#what-is-the-block-time-of-the-relay-chain. 10.05.2023.

[55] COWI and Økonomikontoret. *Befolkningsprognose 2016 - 2028 for Aalborg Kommune fordelt på Bydel*. URL: http://apps.aalborgkommune.dk/statistik/webaarbog/aarbog16/struktur/Befolkning/Prognose/bydele.html. 01.06.2023.

[56] Aalborg Kommune. *Antal Husstande fordelt efter beboede (personer) pr. 28. marts 2014*. URL: http://apps.aalborgkommune.dk/statistik/webaarbog/Boliger/Struktur/Postnumre/Husstande/husstande.html. 01.06.2023.

[57] Gavin Wood. *Avalanche and Polkadot*. URL: https://polkadot.network/blog/polkadot-roadmap-roundup. 02.06.2023.

[58] Polkadot. *Polkadot Roadmap Roundup*. URL: https://polkadot.network/blog/polkadot-roadmap-roundup. 02.06.2023.

[59] Parity Technologies. *Candidate Types*. URL: https://github.com/paritytech/polkadot/blob/master/roadmap/implementers-guide/src/types/candidate.md#candidate-descriptor. 02.06.2023.

[60] Emanuele Fusco and Andrzej Pelc. "Communication Complexity of Consensus in Anonymous Message Passing Systems". In: vol. 137. Dec. 2011, pp. 191–206. ISBN: 978-3-642-25872-5.

[61] Eng Keong Lua et al. "A survey and comparison of peer-to-peer overlay network schemes". In: *IEEE Communications Surveys & Tutorials* 7.2 (2005), pp. 72–93. DOI: 10.1109/COMST.2005.1610546.

[62] eskimor. *Better approval voting paramters*. URL: https://github.com/paritytech/polkadot/issues/7208. 01.06.2023.

[63] Rob Habermeier. *Polkadot v1.0: Sharding and Economic Security*. URL: https://polkadot.network/blog/polkadot-v1-0-sharding-and-economic-security. 01.06.2023.

[64] Andrei Sandu. *Polkadot Dispute Storm - The Postmortem*. URL: https://forum.polkadot.network/t/polkadot-dispute-storm-the-postmortem/2550. 01.06.2023.

[65] Filippo. *Parachains*. URL: https://wiki.polkadot.network/docs/learn-parachains. 01.06.2023.

[66] Filippo. *Frequently Asked Questions (FAQs)*. URL: https://wiki.polkadot.network/docs/faq#why-will-polkadot-have-only-1000-validators-while-other-projects-have-hundreds-of-thousands. 01.06.2023.

[67] Ahmed Samy et al. "SPETS: Secure and Privacy-Preserving Energy Trading System in Microgrid". In: *Sensors* 21.23 (2021), p. 8121. DOI: 10.3390/s21238121.

[68] Coin Market Cap. *Polkadot DOT*. URL: https://coinmarketcap.com/currencies/polkadot-new/. 19.04.2023.

[69] Columbia University. *Gambler's Ruin*. URL: http://www.columbia.edu/~ks20/FE-Notes/4700-07-Notes-GR.pdf. 19.04.2023.

[70] Danish Emergency Management Agency. *National Risk Profile 2022*. Tech. rep. URL: https://www.brs.dk/globalassets/brs---beredskabsstyrelsen/dokumenter/krisestyring-og-beredskabsplanlagning/2022/-national-risk-profile-2022-.pdf. 02.06.2023.

[71] Gavin Wood. *Gavin Wood, Polkadot founder: XCM v3 | Polkadot Decoded 2022*. URL: https://www.youtube.com/watch?v=K2c6xrCoQOU&list=WL&index=6&ab_channel=Polkadot. 04.06.2023.

[72] Parity Technologies. *Crate libp2p*. URL: https://docs.rs/libp2p/latest/libp2p/. 04.06.2023.

[73] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm". In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 305–319.

[74] Grammarly, Inc. *Grammarly: Free Writing AI Assistance*. URL: https://grammarly.com. 07.06.2023.

Appendix A

Development Process

In this appendix, I describe the project's development process. I present different techniques and tools I have used to ensure quality and stay on schedule.

A.1 Planning and Task Management

Working on the project alone has meant that most organisational techniques and tools have been superfluous. Due to this, I have not used Kanban boards or Gantt charts but instead kept task lists of varying abstraction levels. Section A.4 present an example of a high abstraction level list, while Figure A.1 consists of two more concrete lists. I also used sticky notes to provide a visual overview and allow me to re-prioritise tasks.

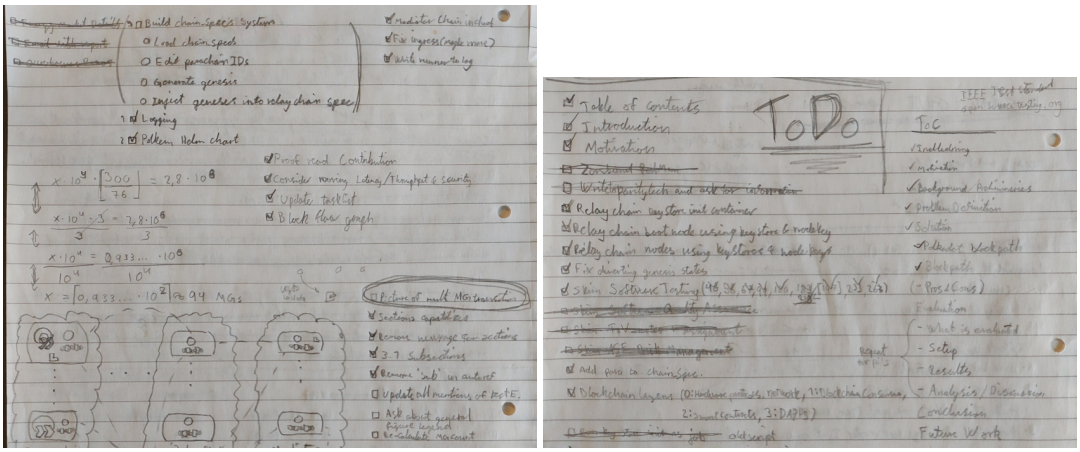


Figure A.1: Pictures of handwritten lists used during the project.

I decided to use handwritten lists and sticky notes as they give a more tactile feel when created and finshed, either crossing them out or taking down a sticky note. As I was the only one who needed to be aware of the current tasks and their progress, I decided that more elaborate, software-based tools would provide negligible advantages and additional overhead.

To ensure I was on track, I revised my current task list every morning and ended every week with a look at the overall plan. These revisions helped me carry out relevant tasks first and allowed me to adapt the overarching schedule if one week's items exceeded their estimated duration or if I completed them quicker than planned.

A.2 Substrate Issues and Pivoting

Another advantage of having disposable lists is easy pivoting. This attribute was useful due to the several hurdles encountered while using Substrate to develop PolkEM's chains.

As no official documentation is available for Substrate, I encountered numerous issues that impacted the overall schedule. I initially had an overview similar to the one in Section A.4 with the features I wanted to implement and the project-related tasks. I assigned each to one of the project's weeks to ensure I could reach the set goals. I did not expect to follow the original plan perfectly, but I intended to use it as an overview of milestones and then continuously evaluate whether I was keeping up or had to adapt my plan.

Table A.1 lists several of these issues with their solutions. Each one increased the implementation time of the chains, as I derived their solutions from reading source code, reverse-engineering Substrate, or waiting for answers from forum users.

Issue	Solution
Deprecated guides	Read through source code
Missing `setjmp.h` header file	Remove `jsonrpsee` library from the runtime
Double implementation of `panic!`	Remove `sp_io` library from the node
Keys generate different addresses	Modify the chain specification code in the node
No authoring	Manually add the seven public keys to the keystore
Deviating genesis states	Forces each node to wait for their associated boot node
Code error in `validate_transaction`	-

Table A.1: Table showing several issues encountered while connecting a parachain to the relay chain. I did not resolve the last issue, and is the main reason for the lack of traded energy assets during the experiments.

Issues such as the ones listed above meant that I had to revise the overall plan multiple times during the project. My original problem definition only mentioned PolkEM but was rewritten to include the testing environment. I spent two months mainly working on solutions to Substrate-related issues, while waiting for answers to questions I had posted on the Substrate forum. These attempts highly impaired the extent of the presented implementations.

A.3 Quality Assurance

Because I was responsible for planning, producing, and reviewing every aspect of the project, I applied processes to enforce multiple iterations for each item. I separated these iterations using other tasks to divide the given item's production and review processes.

I used a trunk-based branching approach and pull-requests for the code to ensure I reviewed changes before merging them into the main branch. Pull-requests generally enforce reviews by multiple team members before changes reach production. However, by preserving this process while working alone, I considered changes from both the perspective of a developer and a reviewer. For this to work sufficiently, the development and review processes should be conducted at different times, allowing for a sufficient context switch and fresh perspective.

I used a similar approach for this report, writing one day and reviewing another. I also used Grammarly [74] to get additional feedback on my writing, manually rewriting sections highlighted as unclear or too passive.

I further increased the code's quality by conducting experimental and unit tests. The prior consisted of executing the different components with various inputs and verifying the results manually, while I used the unit tests to validate the components' units and their behaviour. I have included one of the unit tests used for the trade matching algorithm in Listing A.1. All unit tests follow the Assign, Act, Assert pattern.

```
#[test]
fn cheap_request_is_sold_to_the_grid_and_
everything_else_is_peer_to_peer() {
    let mut requests = Vec::<EnergyRequest>::from([
    EnergyRequest { amount: 10, price: 1.9, buyer: "buyer_1".into
() },
    EnergyRequest { amount: 10, price: 2.0, buyer: "buyer_2".into
() },
    ...,

    EnergyRequest { amount: 10, price: 2.8, buyer: "buyer_5".into
() }
    ]);
    let mut offers = Vec::<EnergyOffer>::from([
    EnergyOffer { amount: 10, price: 2.0, seller: "seller_1".into
() },
    ...,

    EnergyOffer { amount: 5, price: 2.7, seller: "seller_5".into()
 }
    ]);

    let grid_request = EnergyRequest { amount: 10, price: 2.0,
buyer: "grid".into() };
    let grid_offer = EnergyOffer { amount: 10, price: 2.0, seller:
 "grid".into() };

    let trades = generate_trades(&mut requests, &mut offers, &
grid_request, &grid_offer);

    // Assert that six trades are formed, only one involving the
grid, and the grid is the seller
    assert_eq!(trades.len(), 6);
    assert_eq!(trades.iter().filter(|&t| t.buyer.eq("grid")).count
(), 0);
    assert_eq!(trades.iter().filter(|&t| t.seller.eq("grid")).
count(), 1);
}
```

Listing A.1: One of the unit tests used to test the trade matching algorithm. This test verifies that the algorithm matches a request with a price lower than the cheapest offer with the grid's offer.

I have not conducted automated module or system tests, as the size of each tested application is limited. Furthermore, I performed tests covering the blockchain nodes using the testing environment, which one could classify as automated system tests without an assertion phase. Substrate offers ways to unit test parts of the nodes' logic. However, I did use this feature due to my limited node implementations.

A.4 Abstract Task List

A task list showing my intentions for PolkEM and its evaluation at one point during the project.

1. Submit offers and requests
2. Retrieve finalised block containing offers and requests
3. Implement mediator
   (a) Accept incoming XCMP
   (b) Implement democratic voting on blocks, possibly by rewriting the authoring
   (c) Forward results using XCMP
4. Connect Mediator to relay chain
   (a) Add para id, genesis wasm, and genesis state
   (b) Setup XCMP (or HRMP) channel between MG and Mediator, both ways
5. Update MG
   (a) Add XCMP forwarding (at specific intervals or simply for all offers and requests)
   (b) Accept incoming XCMP messages and include them on the chain
6. Connect a second MG
   (a) Generate second chain specification (possibly within `chain_spec.rs`) with para ID and accounts
7. Connect the second MG to the relay chain
   (a) Add para id, genesis wasm, and genesis state
   (b) Setup XCMP (or HRMP) channel between second MG and Mediator, both ways

8. Perform evaluation tests
    (a) Run `polkem-runner` in all MG pod
    (b) Use a JSON file to illustrate consumption/production data
    (c) Load JSON file and use entries as offer and request submissions
    (d) Verify finalisation in the transaction for each submitted offer/request

Appendix B

Source Code Links

The following list contains links to each repository developed during this project.

- The main repository containing the testing environment:
  https://github.com/Nielswps/PolkEM.
- The relay chain, based on Polkadot:
  https://github.com/Nielswps/polkem-relay.
- The mediator, based on the Substrate parachain template:
  https://github.com/Nielswps/polkem-mediator.
- The account manager:
  https://github.com/Nielswps/polkem-am.
- The logging agent:
  https://github.com/Nielswps/polkem-logger.
- The exemplary MG, based on the Substrate parachain template:
  https://github.com/Nielswps/polkem-mg.
- The sidecar application for the exemplary MG nodes:
  https://github.com/Nielswps/polkem-runner.

Appendix C

Experiments

This appendix contains code snippets and commands I have used to conduct the experiments in Chapter 5.

C.1 Log Processing

This section contains snippets of the log and the commands and scripts used to process the entries.

C.1.1 Log Snippet

```
energy-node-2 "Block 0x2b66...28bd was observed as proposed" 2023-05-25@09:57:48
energy-node-1 "Block 0x2b66...28bd was observed as proposed" 2023-05-25@09:57:48
energy-boot-node "Block 0x2b66...28bd was observed as proposed" 2023-05-25@09:57:48
energy-node-0 "Block 0x2b66...28bd was observed as proposed" 2023-05-25@09:57:48
energy-node-3 "Block 0x2b66...28bd was observed as proposed" 2023-05-25@09:57:48
energy-boot-node "Block 0xf3a2...ba8f was observed as finalized" 2023-05-25@09:57:50
energy-node-2 "Block 0xf3a2...ba8f was observed as finalized" 2023-05-25@09:57:50
energy-node-0 "Block 0xf3a2...ba8f was observed as finalized" 2023-05-25@09:57:50
energy-node-3 "Block 0xf3a2...ba8f was observed as finalized" 2023-05-25@09:57:50
energy-node-1 "Block 0xf3a2...ba8f was observed as finalized" 2023-05-25@09:57:50
energy-node-1 "Block 0x2b66...28bd was observed as finalized" 2023-05-25@09:58:02
energy-node-3 "Block 0x2b66...28bd was observed as finalized" 2023-05-25@09:58:02
energy-boot-node "Block 0x2b66...28bd was observed as finalized" 2023-05-25@09:58:02
energy-node-0 "Block 0x2b66...28bd was observed as finalized" 2023-05-25@09:58:02
energy-node-2 "Block 0x2b66...28bd was observed as finalized" 2023-05-25@09:58:02
```

C.1.2 Log Processing Commands and Scripts

```
# Exclude all nodes but the boot node
sed -n '/energy-boot/p' > boot_node_only

# Filter entries to only include block hash and
   timestamp
grep -Po '"Block \K[^ ].*' boot_node_only |
sed 's/ .*"//' |
sort > sorted_hashes_with_timestamp

# Exclude block hashes and get timestamps in UNIX time
grep -Po '.* \K.*' sorted_hashes_with_timestamp |
grep -Po '.*' |
xargs -I {} date -d '{}' +%s > timestamps

# Calculate delays
#!/bin/bash
while read -r p; do
read -r f;
echo "$((f-p))"
done

# Calculate average
#!/bin/bash
let a=0;
let len=0;
while read -r i; do
a=\$((a+i));
len=\$((len+1))
done
echo "\$((a/len))"

# Get delay occurrences
sort averages | uniq -c | sort -rn
```

C.2 Mediator Trade Matching

This section contains a snippet depicting the Rust implementation of the trade matching algorithm. It differs from the outline in Algorithm 3.1 by handling the

potential surplus amount remaining after matching a request and an offer. Furthermore, it utilises iterators instead of looping over array indices, as this is more idiomatic to Rust.

```
pub fn generate_trades(energy_requests: &mut Vec<EnergyRequest>,
    energy_offers: &mut Vec<EnergyOffer>,
    grid_request: &EnergyRequest,
    grid_offer: &EnergyOffer) -> Vec<Trade> {

    let mut trades = Vec::<Trade>::new();

    // Sort requests and offers and turn two iters
    energy_requests.sort_by(|e1, e2| e1.price.partial_cmp(&e2.price)
    .unwrap());

    energy_offers.sort_by(|e1, e2| e1.price.partial_cmp(&e2.price)
    .unwrap());

    let mut request_iter = energy_requests.iter();
    let mut offer_iter = energy_offers.iter();

    // Get first request and offer
    let mut req = request_iter.next();
    let mut off = offer_iter.next();

    // Create variables to store temporary surplus requests and offers
    let mut surplus_req: EnergyRequest;
    let mut surplus_off: EnergyOffer;

    // Match trades
    while req.is_some() && off.is_some() {
        match (req, off) {
            (Some(r), Some(o)) => {
                let t;

                if r.price >= o.price {
                    let average_price = r.price.add(o.price).div(2.0);

                    // Create a match for the request and offer,
                    // and handle potential surplus
                    match r.amount.partial_cmp(&o.amount)
                    .expect("Both values are numbers") {
```

```
                    Ordering::Less => {
                    // The offered amount exceeds whats requested
                        t = Trade {
                            amount: r.amount,
                            price: average_price,
                            buyer: r.buyer.clone(),
                            seller: o.seller.clone()
                        };
                        req = request_iter.next();

                        surplus_off = EnergyOffer {
                            amount: o.amount.clone() -
                                r.amount.clone(),
                            price: o.price.clone(),
                            seller: o.seller.clone()
                        };
                        off = Some(&surplus_off);
                    }
                    Ordering::Equal => {
                    // The same amount is requested and offered
                        t = Trade {
                            amount: o.amount,
                            price: average_price,
                            buyer: r.buyer.clone(),
                            seller: o.seller.clone()
                        };
                        req = request_iter.next();
                        off = offer_iter.next();
                    }
                    Ordering::Greater => {
                    // The requested amount exceeds the offered
                        t = Trade {
                            amount: o.amount,
                            price: average_price,
                            buyer: r.buyer.clone(),
                            seller: o.seller.clone()
                        };
                        off = offer_iter.next();

                        surplus_req = EnergyRequest {
```

```
                                amount: r.amount.clone()
                                - o.amount.clone(),
                                price: r.price.clone(),
                                buyer: r.buyer.clone()
                            };
                            req = Some(&surplus_req);
                        }
                    }
                } else {
                    t = Trade {
                        amount: r.amount,
                        price: grid_offer.price,
                        buyer: r.buyer.clone(),
                        seller: grid_offer.seller.clone()
                    };
                    req = request_iter.next();
                }
                trades.push(t);
            }
            _ => break
        }
    }

    // Match remaining requests with the grid
    while req.is_some() {
        let t = Trade {
            amount: req.unwrap().amount,
            price: grid_offer.price,
            buyer: req.unwrap().buyer.clone(),
            seller: grid_offer.seller.clone()
        };
        req = request_iter.next();
        trades.push(t);
    }

    // Match remaining offers with the grid
    while off.is_some() {
        let t = Trade {
            amount: off.unwrap().amount,
            price: grid_request.price,
            buyer: grid_request.buyer.clone(),
```

```
            seller: off.unwrap().seller.clone()
        };
        off = offer_iter.next();
        trades.push(t);
    }

    trades
}
```