Fully On-Chain Blockchain Systems: A Feasibility Analysis Based on LogLog and ZipZap

by

Mario Felipe MUÑOZ

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT FOR A MASTER'S DEGREE
WITH THESIS IN SOFTWARE ENGINEERING
M.A.Sc.

MONTREAL, JUNE 21, 2021

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Mr. Kaiwen Zhang, Thesis supervisor
Department of IT and Software Engineering, École de Technologie Supérieure

Mr. Mustapha Ouhimmou, Thesis Co-supervisor
Department of Systems Engineering, École de Technologie Supérieure

M. Abdelouahed Gherbi, President of the board of examiners
Department of IT and Software Engineering, École de Technologie Supérieure

M. Luis Antonio de Santa-Eulalia, External examiner
École de gestion SIMQG, Université de Sherbrooke

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON JUNE 16, 2021

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGEMENTS

The author would like to extend his deepest gratitude to Dr. Kaiwen Zhang and Dr. Aamir Shahzad, who kindly guided him through every step of his research. This thesis was only possible thanks to the depth and breadth of their technical knowledge. Their credentials made working with them a privilege, but their honest, amiable characters also made it a pleasure.

In the same vein, the author would like to thank Dr. Mustapha Ouhimmou for his invaluable insight into the logistic standards and business practices of the forestry industry. Without his professional perspective, any value that this thesis may have had would have been strictly theoretical.

In no particular order, the author would also like to thank Simonetta Ferrante, Mario Orlando Muñoz Cabrera, Mario Francisco Muñoz, Alexandra Trempe, Linda Stehr, Paul Trempe, Stephanie Trempe, Rose Desjardins, and all their family members for their unwavering patience and support.

In loving memory of Annunziata Trombetta.

Systèmes de chaînes de blocs entièrement sur la chaîne: une analyse de faisabilité basée sur LogLog et ZipZap

Mario Felipe MUÑOZ

RÉSUMÉ

Cette thèse compare deux systèmes de tokénisation distincts pour déterminer les contextes où une approche entièrement sur la chaîne est viable. Les deux systèmes ont été développés par l'auteur, mettant en vedette des fonctionnalités quelque peu homologues malgré de grandes différences dans le contexte industriel. Notez que bien que le travail sur LogLog a déjà été publié, le travail sur ZipZap est toujours en cours. Ainsi, des descriptions de ce dernier ne seront pas aussi rigoureuses qu'avec le premier.

LogLog

En matière de développement durable, de nombreux systèmes de certification vivent et meurent selon leur capacité à suivre un volume de la source à destination. En effet, les clients (et par conséquent, les auditeurs) sont de plus en plus préoccupés par l'origine des produits. Le Forest Stewardship Council (FSC), l'une des plus grandes autorités forestières de l'Amérique du Nord, a créé de nombreuses normes et cadres pour répondre à ces demandes de marché. Le FSC est responsable du suivi de la provenance de tous les volumes de bois que ses membres utilisent tout en étiquetant les volumes en conséquence. Bien que très respectée, leurs certifications s'appuient sur des tiers de confiance et des dossiers approfondis, deux caractéristiques qui n'addressent pas efficacement des problèmes tels que la corruption et les pratiques d'inventaire «créatives». Les chaînes de blocs sont une solution attrayante, car elles offrent de nombreuses améliorations nouvelles et précieuses pour les certifications à base de provenance, telles que la traçabilité accrue, l'auditabilité et l'accès à l'information. Toutefois, ces systèmes ne sont pas sans leurs limitations. Notamment, ils sont souvent inefficaces en leur capacité de stocker de grandes quantités de données et/ou de métadonnées, c'est pourquoi de nombreuses recherches suggèrent souvent une approche avec multiples bases de données où les fichiers importants sont tenus hors chaîne. Ce type de conception implique des compromis importants pour des frais de transaction plus faibles: traçabilité imparfaite (les modifications apportées aux fichiers hors chaîne ne sont pas nécessairement suivies), disponibilité plus basse (introduisant davantage de systèmes qui doivent tous être élaborés) et un niveau diminué d'intégrité informationnelle (les fichiers hors chaîne sont beaucoup plus facilement modifiés).

Contrairement à ces approches, nous proposons LogLog: un système entièrement sur chaîne qui suit des volumes de bois tout au long d'une chaîne d'approvisionnement et applique les normes de certification de la FSC. LogLog est conforme à ERC-1155 et répond aux exigences de base pour une chaîne d'approvisionnement forestière. Notre preuve de concept démontre la puissance et la flexibilité des contrats intelligents lorsqu'ils exprime une sémantique complexe spécifique au domaine relatif aux chaînes d'approvisionnement dans l'industrie forestière (comme classifier

un volume à l'aide de la matrice de catégories FSC ou calculer le pourcentage FSC d'un volume, par exemple).

Nous avons mis en œuvre avec succès deux implémentations de référence de notre conception avec Solidity et nous les avons déployées en Etherium. Notre évaluation utilise des paramètres réalistes et suggère que notre prototype est une alternative viable aux systèmes de base de données multiples actuels dans des contextes où l'intégrité des données est une exigence stricte.

ZipZap

Au cours des dernières années, l'industrie de l'énergie a de plus en plus investi dans les technologies de réseaux energetiques intelligents. Les réseaux électriques intelligents sont des réseaux électriques avec un degré élevé d'automatisation et des capacités améliorées de collecte de données. Une autre tendance de l'industrie est la décentralisation des réseaux électriques susmentionnées. Plus de données signifient plus de risques et bien qu'un degré de décentralisation plus élevé signifie une plus grande disponibilité, il entraîne également une augmentation massive de la surface d'attaque. Ces faits font partie des principales motivations des spécialistes de l'industrie à prendre en compte les systèmes de chaîne de blocs pour aider à automatiser les opérations (à l'aide de contrats intelligents) et à assurer l'intégrité des données (en raison de caractéristiques cryptographiques inhérentes aux chaînes de blocs).

Hydroquebec ne fait pas exception à cette tendance. Ayant un intérêt direct pour développer des moyens d'intégrer les DLT dans des réseaux energetiques intelligents, ils ont subventionné nos recherches sur le sujet. ZipZap est la première étape vers un réseaux energetique intelligent à base de chaînes de blocs. À son stade de développement actuel, ZipZap est une solution de chaîne de blocs pour la tokénisation de l'énergie locale, bien que sa gamme d'utilisations se développe pour inclure au moins une forme d'enchère energetique et inclure des cas d'utilisation réliés à l'IoT. En outre, tout comme LogLog, ZipZap est conforme à ERC-1155.

Nous avons créé trois prototypes ZipZap: Heavyweight, Featherweight et Lightweight. Le premier est entièrement en chaîne et les deux derniers sont hybrides. Les trois ont été développés en Solidity et déployés en utilisant Ethereum. Notre évaluation utilise des paramètres réalistes et suggère que, bien qu'aucun des prototypes actuels ne soit économiquement viable, des modifications de l'échelle et/ou le choix du système de chaîne de blocs pourraient facilement entraîner une alternative viable avec des modifications minimales.

Mots-clés: chaîne de blocs, normes de tokénisation, certification forestière, tokénisation de l'énergie

Fully On-Chain Blockchain Systems: A Feasibility Analysis Based on LogLog and ZipZap

Mario Felipe MUÑOZ

ABSTRACT

This thesis compares two separate tokenization systems to ascertain the contexts where a fully on-chain approach is viable. Both systems were developed by the author, featuring somewhat homologous functionality in spite of great differences in industrial context and performance. Note that although the work on LogLog has been published already, work on ZipZap is still ongoing. Thus, descriptions of the latter will not be as rigorous as with the former.

LogLog

When it comes to sustainability, many certification systems live and die by their capacity to track a volume from source to destination. This is because customers (and consequently, auditors) are increasingly concerned with where products are sourced (*"Are these local materials?"*) and what is their environmental impact is (*"How much CO2 did this generate? Was this made from recycled goods?"*). The Forest Sustainability Council (FSC), one of the largest forestry authorities in North America, created many of its standards and frameworks to meet these market demands. The FSC is responsible for tracking the provenance of all wood volumes that its members use and labelling the volumes accordingly. Though widely respected, their certifications rely on trusted third parties and extensive record keeping, two characteristics that do not efficiently address problems like bribery and "creative" inventory practices. Blockchain technology is an attractive solution, since it offers many novel and valuable improvements to provenance-based certifications, such as increased traceability, auditability and access to information. However, distributed ledger systems are not without their limitations. Notably, they are often inefficient in their capacity to store large amounts of data and/or metadata, which is why many researches often suggest a multiple-database approach where large files are kept off-chain. That type of design entails significant trade-offs for the sake of lower transaction fees: imperfect traceability (changes to off-chain files are not necessarily tracked), lower availability (introducing more systems that all need to be up) and a diminished level of informational integrity (off-chain files are much more easily modified).

In contrast with these approaches, we propose LogLog: an entirely on-chain system for tracking wood volumes throughout a supply chain and enforcing FSC certification standards. LogLog is ERC-1155 compliant and meets the base requirements for a forestry supply chain. Our Proof-of-Concept demonstrates the power and flexibility of smart contracts when expressing complex domain-specific semantics related to supply chains in the forest industry (like determining the correct labelling for a volume using the FSC category matrix, or calculating the FSC percentage of a volume, for example).

We successfully implemented two reference implementations of our design using Solidity and deployed them over the Ethereum blockchain. Our evaluation uses realistic parameters and suggests our prototype is a viable alternative to current multiple-database systems in contexts where data integrity is a strict requirement.

ZipZap

In the last few years, the energy industry has increasingly invested into smart grid technologies. Smart grids are power grids with a high degree of automation and enhanced data collection capacities. However, another concurrent industry trend is the decentralization of the aforementioned power grids. More data means more risks and although a higher degree of decentralization means greater availability, it also results in a massive attack surface area increase. These facts have been some of the primary motivations driving industry specialists to consider blockchain systems to help automate operations (using smart contracts) and ensure data integrity (due to blockchains' inherent cryptographic characteristics).

HydroQuebec is no exception to this trend. Having a vested interest in developing possible ways to integrate DLTs into smart grids, they sponsored our research into the topic. ZipZap is the first step towards a full-fledged blockchain-based smart grid system. At its current development stage, ZipZap is a blockchain solution for local energy tokenization, although its range of use cases will expand to include at least some form of price bidding and IoT integration. Also, just like LogLog, ZipZap is ERC-1155 compliant.

We created three ZipZap prototypes: Heavyweight, Featherweight and Lightweight. The first one is fully on-chain, and the two latter are hybridized. All three were developed in Solidity and deployed using Ethereum. Our evaluation uses realistic parameters and suggests that although none of the current prototypes are economically viable, changes in scale and/or choice of blockchain system could easily result in a viable alternative with otherwise minimal data structure modifications.

Keywords: blockchain, tokenization standards, forestry certification, energy tokenization

TABLE OF CONTENTS

Page

INTRODUCTION 1
0.1 LogLog: A Blockchain Solution for Tracking and Certifying Wood Volumes 2
0.2 ZipZap: A Blockchain Solution for Local Energy Trading 5
0.3 Project Similarities 6

CHAPTER 1 BACKGROUND 9
1.1 Motivation for Selecting the FSC Certification Standard 9
1.2 FSC Classification Rules 11
1.2.1 Calculating the FSC Percentage of a Product 11
1.2.2 Calculating the FSC Credit of a Product 12
1.3 Blockchain Systems 12
1.4 Tokenization Standards 13

CHAPTER 2 RELATED WORKS 17
2.1 Blockchains for Supply Chains 17
2.2 Blockchain for the Timber Industry 18

CHAPTER 3 LOGLOG: THE DESIGN 21
3.1 Sample Use Case 21
3.2 Design Considerations of LogLog 22
3.3 WoodToken 23
3.4 Recipes 24
3.5 Transformations 24
3.6 Implementation Details 25
3.7 Data Safeguards 27

CHAPTER 4 LOGLOG: THE RESULTS 29
4.1 Gas Cost Analysis 29
4.2 Yearly Cost Comparison with Traditional Certification Process 31

CHAPTER 5 ZIPZAP: OVERVIEW 35
5.1 Background 35
5.2 Related Works: Blockchain for Smart Energy Grids 37
5.3 Zap 38
5.4 Functional Overview of Prototypes 39
5.4.1 Heavyweight 39
5.4.2 Featherweight 39
5.4.3 Lightweight 39
5.5 Implementation Details 40

CHAPTER 6 ZIPZAP: THE RESULTS 41
6.1 Heavyweight Gas Results 41
6.2 Featherweight Gas Results 42
6.3 Lightweight Gas Results 43
6.4 Time Performance 43
6.5 Cost Viability Analysis 44

CHAPTER 7 LESSONS LEARNED 47
7.1 Fully On-Chain Systems 47
7.2 Computational Complexity and Spatial Complexity 48

CHAPTER 8 FUTURE WORK 49
8.1 LogLog: Multi-token Approach 49
8.2 LogLog: Yearly-Agreement Extension 49
8.3 LogLog: Data Restructuring 50

CONCLUSION 51

APPENDIX I LOGLOG 53

APPENDIX II ZIPZAP 73

BIBLIOGRAPHY 102

LIST OF FIGURES

Page

Figure 0.1 FSC Labels used in wood-derived products 2

Figure 1.1 Typical FSC Supply Chain 10

Figure 3.1 Partial Follow-through of a Volume 22

Figure 4.1 Gas Cost Results 29

Figure 6.1 Heavyweight Gas Costs 41

Figure 6.2 Featherweight Gas Costs 42

Figure 6.3 Lightweight Gas Costs 42

Figure 6.4 Gas Cost Comparison 44

Figure 6.5 Time Performance Metrics 45

LIST OF ABBREVIATIONS

FSC	Forest Stewardship Council
CoC	Chain of Custody
DLT	Distributed Ledger Technology
ERC	Ethereum Request for Comments. A term used for application-level standards in Ethereum
P2P	Peer-to-Peer. A type of distributed network
PV	Photovoltaic. Used interchangeably as PV unit(s), PV cell(s) and PV generator(s), for example

LIST OF SYMBOLS AND UNITS OF MEASUREMENTS

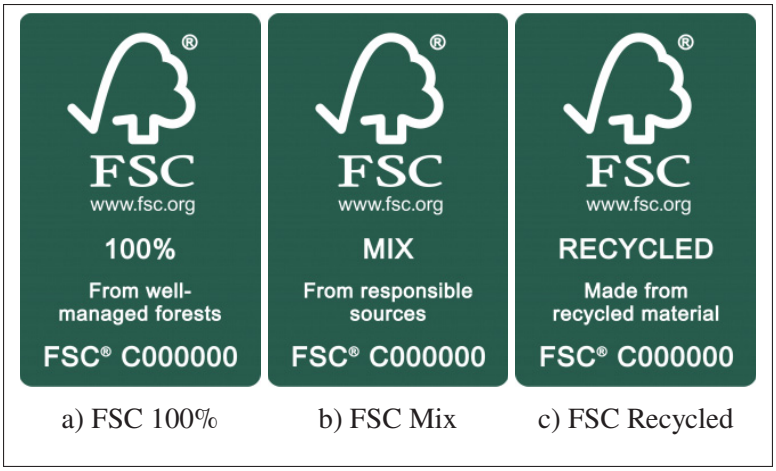
CAD	Canadian Dollars
Gas	The unit used to express the computational cost of executing operations on Ethereum, not to be confused with ether (ETH)
ETH	Also known as ether. It is the cryptocurrency used in Ethereum to pay for gas, among many other things

INTRODUCTION

Regardless of their scale and structure, supply chains often exhibit two seemingly contradictory characteristics: they are as susceptible as they are resilient. Their resilience stems from the fundamental economic law of supply and demand. Although individual members of a supply chain may be eliminated through competition, legislative changes or localized disasters, as long as there is a demand for the goods offered, the supply chain as a whole will persist. However, the same market that drives the supply chain is inherently volatile, and this volatility is compounded by each echelon of producers/suppliers in a phenomenon dubbed "The Bullwhip Effect" (Lee, Padmanabhan & Whang, 2004). These economic distortions are primary caused by heuristics-based decision making: since agents of the supply chain have incomplete and inaccurate information about past, current and future market trends, they each take imperfect measures to reap the greatest profits. Thus, an effective way to counter this phenomenon is to increase the amount and quality of information available to the supply chain as a whole.

There have been several analog means of enriching supply chains with information. Product labelling and certifications serve as excellent examples. However, throughout the current century, the overarching trend has been one of digitization, so it is only natural that we consider digital solutions to this industrial problem. Regardless of the industry, tokenizing goods is a fantastic strategy for increasing the accuracy of records and the amount of information known about any one item at any point throughout a supply chain. In other words, tokenization makes it easier to implement better forms of labelling and certification.

With this in mind, this thesis takes a look at two different industrial contexts in which tokenization generates significant value (the timber industry and the energy industry) and showcases two corresponding DLT systems: LogLog and ZipZap. Of these systems, LogLogis the only completed one, with a published paper, whereas ZipZap is still in active development. Therefore, the bulk of this thesis will be centered around LogLog, with ZipZap being considered as a



a) FSC 100% b) FSC Mix c) FSC Recycled

Figure 0.1 FSC Labels used in wood-derived products
Taken from FSC (2020c)

secondary use case. Sections 0.1 and 0.2 introduce these two systems separately and section 0.3 establishes the common elements between them.

0.1 LogLog: A Blockchain Solution for Tracking and Certifying Wood Volumes

The trade of illegal timber has tremendous negative impacts on local and international markets as well as our shared ecosystem (Gan, Cerutti, Masiero, Pettenella, Andrighetto & Dawson, 2016). This phenomenon has driven the continuous development of sustainable forestry standards and legislation. However, these standards often utilize simple auditing systems, where members are asked to accurately keep records so certified auditors can accurately inspect them (Cashore & Gale, 2006). Naturally, such standards come with a number of issues, such as the meticulous nature of the demanded documentation and the substantial trust placed on auditors, which have driven demand for improving existing certification models and exploring alternatives.

One of the most important forestry certification authorities in North America is the Forest Stewardship Council (FSC), which oversees the management of 150 million acres of forest and more than 3500 companies in both the US and Canada (FSC, 2020b). Their Chain of

Custody (CoC) standard uses Certification Bodies (i.e. trusted third parties) to assess the regulatory compliance of FSC members and of applicant companies (FSC, 2020a). Traditionally, Certification Bodies are solely responsible for identifying anomalies and ascertaining whether applicants have set effective controls in place to separate materials from different sources. Once a certificate is issued, it remains valid for five years, a period during which members are minimally expected to maintain extensive records of their processes and inventories (Council, 2016). Currently, these records are often paper-based and whenever auditors want to access them, they have to either travel to the audited premises or have the documents sent over. Also, paper-based records are susceptible to fraud: it is often difficult to establish if a document was ever generated, or which between two documents is the original, so repudiating actions is much easier for wrongdoers. Figure 0.1 shows the product labels used in products regulated via the Chain of Custody.

In order to address the shortcomings of this type of certification system, distributed ledger technologies (DLTs) have often been proposed as a solution, generating substantial international interest (Vilkov & Tian, 2019). In trust-poor contexts such as this one, DLTs provide an attractive set of benefits (Nikolakis, John & Krishnan, 2018):

- Digitisation of records.
- Increased traceability of goods.
- Transactional non-repudiation.
- Increased transparency and access to information for consumers and authorities alike.

Despite the possible advantages, existing blockchain solutions for supply chain provenance do not adequately address all the business needs of the timber industry. Among our studied papers, those that addressed the problems related to wood product certification did not propose a working prototype, but only high-level designs at best (Düdder & Ross, 2020; Nikolakis *et al.*, 2018). On the other hand, those papers which did provide adequately performing prototypes were

crafted with generalized use-cases in mind and only suggest certification-specific features as a possible extension to their work (Gallersdörfer & Matthes, 2018; Westerkamp, Victor & Küpper, 2020). Furthermore, they rely on off-chain data storage for lower transaction fees, which has several drawbacks: imperfect traceability (changes to off-chain files are not necessarily tracked), lower availability (introducing more systems that all need to be up) and a diminished level of informational integrity (off-chain files are much more easily modified).

In this thesis, we present LogLog: an auditing system for upholding FSC certification standards and tracking wood provenance using DLTs. LogLog operates by tokenizing wood volumes, along with all FSC-required data (such as ownership and location) related to each volume. Changes to these volumes, such as merging, splitting or any other form of transformation, are entirely regulated by smart contracts. Hence, our contributions are as follows:

1. We propose LogLog, a solution that accurately implements FSC regulatory standards using smart contracts that are executed entirely on-chain, demonstrating the capacity of DLTs to express complex domain-specific validation logic related to the forest industry.
2. We provide two reference implementations of our design (one using ERC-721 and one using ERC-1155) entirely written in Solidity and deployed on an Ethereum environment. To the best of our knowledge, LogLog is the first working blockchain-based prototype specifically built to enforce FSC standards.
3. We evaluate our solution and present results demonstrating the feasibility of our approach in terms of operating gas costs compared to traditional multiple database solutions. To the best of our knowledge, this is the first thesis to evaluate the transactional performance of ERC-1155.

0.2 ZipZap: A Blockchain Solution for Local Energy Trading

ZipZap is part of a larger research effort lead by HydroQuebec. The latter's primary goals are, among others, to answer the following questions:

- What are the most valuable use cases that blockchain systems can create or enhance in a smart grid context?
- What are the characteristics of a blockchain system that is well-suited for the aforementioned use cases?
- What would the performance of such a system be in the context of a small-scale, DLT-based local energy exchange?

ZipZap is a first step towards answering these questions. In its current state, ZipZap is an energy tokenization blockchain system for local energy exchanges, although functionality extensions are being considered. ZipZap operates by tokenizing energy quantities as they are created by different types of generators (biodiesel engines, PV cells, wind turbines, etc.). The owners of the generators can then consume their own energy, and sell the excess to their neighbours. All energy-related operations are tracked by ZipZap, allowing users and auditors to verify the provenance of all energy created and spent by a household. Hence, our contributions are as follows:

1. We propose ZipZap, an energy tokenization solution for local energy exchanges.
2. We provide three reference implementations of our design: Heavyweight, Featherweight and Lightweight. The first being a fully on-chain approach, and the two latter using hybrid designs.
3. We evaluate our solution and present results demonstrating the feasibility of our approach in terms of operating gas costs compared to energy generation costs.

0.3 Project Similarities

Despite some key differences, LogLog and ZipZap share many common elements that allow a meaningful comparison to be established. For starters, both projects followed the same research methodology:

1. First, an exploratory literature review of similar systems was conducted. We used this opportunity to identify high-level similarities and shortcomings of current approaches to better grasp where our research could generate the most value.
2. Then, we would consider the different architectural choices required for the prototypes. Generally, these revolved around different tokenization standards and blockchain platforms. We made our decisions primarily with ease of implementation in mind, which consequently lead us, in both cases, to choose Ethereum as a platform due to its flexibility, vast documentation and large software library.
3. We would then develop multiple prototypes for each system. After a first prototype was created, its performance would be immediately measured (with a emphasis on gas performance and latency, whenever applicable). Subsequent prototypes would try to improve on these metrics either by using different standards or by changing the code structure, in typical iterative manner.

Similarly, both projects also had the same high-level objectives:

1. The main goal was to improve the traceability and transparency of supply chains, which would in turn increase their overall efficiency by mitigating most negative impacts of the bullwhip effect due to limited informational quality and availability (Lee *et al.*, 2004).
2. One secondary goal was to identify some ideal conditions for fully on-chain systems
3. Another secondary goal was to study the performance impact of tokenization standards
4. Our last secondary goal was to develop economically viable fully on-chain systems

It should come as no surprise then that both projects tackle the same high level problem. Namely, the issue is that most, if not all, supply-chain solutions that integrate DLTs follow a hybridized model that results in a more complex system architecture, reduced availability and reduced informational integrity. In other words, the problem was that in spite of all their advantages, current DLT systems have to make some non-trivial design compromises to stay financially viable and little to no work had been done in exploring the costs and performance of fully on-chain systems.

However, despite these similarities there are some differences to address in order to better justify the comparisons made through this thesis. First, there is the question of scale. LogLogis meant to be an end-to-end supply chain system, covering all echelons of the supply chain, whereas ZipZap is currently only being considered for local energy exchanges (single neighbourhoods). This discrepancy in scale is in turn offset by the discrepancy in the complexity and scope of their respective functionalities. LogLog is meant to be an auditing and certification system, whereas ZipZap aims to eventually become a real-time transactional energy system.

The rest of this thesis abides to a clear structure: Chapter 1 provides all the preliminary information necessary to understand CoC certification and the basic principles of operation behind blockchain systems. Chapter 2 presents some of the existing research on topics pertinent to LogLog, both practical and theoretical. Chapter 3 presents LogLog and describes its design decisions, their motivations and the implementation of its components. Chapter 4 provides an evaluation of the performance of LogLog and how it compares to a hybrid approach. Chapter 5 gives an overview of ZipZap as a system and also includes condensed versions of its respective background and related works sections. Chapter 6 shows the experimental results for all ZipZap prototypes. Chapter 7 describes the key takeaways of our research, and Chapter 8 explains possible venues for further research and development. We close this thesis with a brief conclusion.

CHAPTER 1

BACKGROUND

In order to understand the scope of our solution, it is imperative to understand the role that FSC standards play in forestry supply chains. Figure 1.1 is an example of an FSC supply chain, where certified members exchange goods between each other, but also with non-certified entities. Within the CoC framework, all certified members must provide process and inventory records while also accurately labelling and tracking all wood volumes they handle. It also introduces a categorization system based on the percentage of FSC-certified materials present in a volume, as well as a credit-based alternative system used in contexts where FSC-certified materials are scarcer. With this rigorous set of controls, the CoC system is a great candidate to test the flexibility and expressiveness of smart contract languages. It is important to have a basic understanding of these controls because LogLog automatically applies them to tracked volumes as part of its main smart contract. Therefore, subsections 1.2 to 1.2.2 cover some of the most important sets of operations and regulatory checks enforced by the FSC.

We briefly justify our interest in blockchain systems by explaining how they work and how they can generate value when it comes to various forms of certification in subsection 1.3. In the case of LogLog, our choice of implementing CoC certification instead of some other forestry certification has an obvious impact on our feature set and performance metrics. Similarly, tokenization standards influence many of the aspects of *WoodTokens*. Following a specific standard (or not) has a number of repercussions on the performance and compatibility of our solution. Subsection 1.4 serves as both an introduction to some of the current standards used for Ethereum tokens, as well as a justification for the standards used by *WoodTokens*.

1.1 Motivation for Selecting the FSC Certification Standard

There are several reasons why we chose to implement the FSC CoC framework as opposed to any of its numerous competitors. FSC being the most widespread framework in North America (FSC, 2020b) makes our research more attractive to our local market, but the CoC standard also

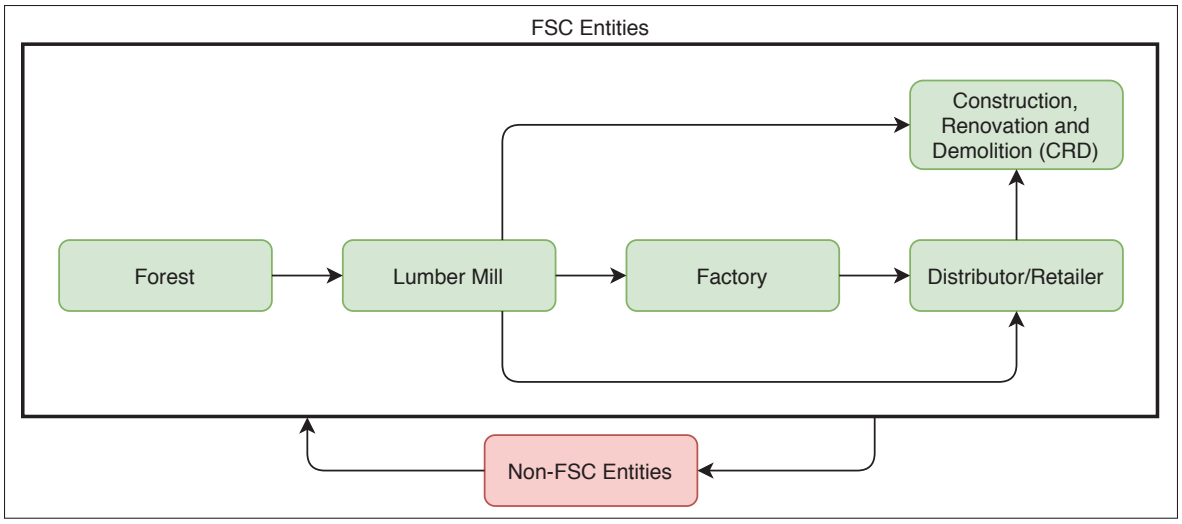


Figure 1.1 Typical FSC Supply Chain

places a greater emphasis on international trade than others (Sugiura & Oki, 2018), making it just as interesting to a global user-base. However, CoC accreditation has at least two key shortfalls (Sugiura & Oki, 2018):

- It is relatively expensive: in some cases certification can cost between USD 9K and USD 28K in addition to the mandatory yearly audit fees (Sugiura & Oki, 2018). Consequently, we assume that companies interested in FSC certification have the capital required to invest in audit automation software and that both members and certification bodies are incentivized to invest in such software to reduce certification costs and increase membership numbers, respectively.
- Obtaining and maintaining the certification is quite effort-intensive due at least partly to its rigorous bookkeeping and auditing requirements. In fact, abiding to the CoC protocol (which falls under the Monitoring and Assessment FSC principle) is one of the top three causes of non-conformity for FSC members (Rafael, Fonseca & Jacovine, 2018). This makes it clear that facilitating in either of those processes and increasing the integrity, transparency and correctness of records generates additional value on top of any cost-saving attributes our solution may have.

Table 1.1 FSC Category Matrix
Adapted from Council (2016)

Inputs	FSC 100%	FSC Mix Credit	FSC Mix %	FSC Recycled Credit	FSC Recycled %	Pre-consumer reclaimed paper	Post-cons. Recl. wood & paper	FSC Controlled Wood
FSC 100%	FSC 100%	FSC Mix Credit	FSC Mix %	FSC Mix Credit	FSC Mix %	FSC Mix 100%	FSC Mix 100%	FSC Controlled Wood
FSC Mix Credit	FSC Mix Credit	FSC Mix Credit	FSC Mix %	FSC Mix Credit	FSC Mix %	FSC Mix Credit	FSC Mix Credit	FSC Controlled Wood
FSC Mix %	FSC Mix %	FSC Mix %	FSC Mix %	FSC Mix %	FSC Mix %	FSC Mix %	FSC Mix %	FSC Controlled Wood
FSC Recycled Credit	FSC Mix Credit	FSC Mix Credit	FSC Mix %	FSC Recycled Credit	FSC Recycled %	FSC Recycled Credit	FSC Recycled Credit	No FSC claims allowed
FSC Recycled %	FSC Mix %	FSC Mix %	FSC Mix %	FSC Recycled %	FSC Recycled %	FSC Recycled %	FSC Recycled %	No FSC claims allowed
Pre-consumer reclaimed paper	FSC Mix 100%	FSC Mix Credit	FSC Mix %	FSC Recycled Credit	FSC Recycled %	FSC Recycled 100%	FSC Recycled 100%	No FSC claims allowed
Post-cons. Recl. wood & paper	FSC Mix 100%	FSC Mix Credit	FSC Mix %	FSC Recycled Credit	FSC Recycled %	FSC Recycled 100%	FSC Recycled 100%	No FSC claims allowed
FSC Controlled Wood	FSC Controlled Wood	FSC Controlled Wood	FSC Controlled Wood	No FSC claims allowed	No FSC claims allowed	No FSC claims allowed	No FSC claims allowed	FSC Controlled Wood

Thus, we will compare the gas costs of our solution to these estimates to determine the economic feasibility of LogLog.

1.2 FSC Classification Rules

Whenever two or more volumes are transformed together as part of a single industrial process (for example, using different kinds of planks to assemble a piece of furniture), the *FSC category* of the output must be calculated according to a comprehensive matrix. Table 1.1 shows the matrix that dictates how any mix between two volumes, each belonging to an *FSC category*, must be classified. In cases where there are more than two volumes being mixed, the matrix is applied recursively on the output of each binary transformation until all input types are accounted for. Should any set of transformations result in a volume being labelled as either FSC Percent or FSC Credit, then the following calculations must take place, respectively.

1.2.1 Calculating the FSC Percentage of a Product

$$\frac{\sum FSC\_volumes}{\sum total\_volumes} * 100 \; = FSC\% \tag{1.1}$$

It is also worth nothing that only volumes with at least 70% FSC content are eligible for FSC labelling.

1.2.2 Calculating the FSC Credit of a Product

$$\sum credited\_input\_volumes = credited\_output\_volume \tag{1.2}$$

Just as above, only volumes with at least 70% FSC content are eligible for FSC labelling. All volumes leftover from such an operation are immediately classified as FSC Controlled Wood. In contrast with most other categories, transformations resulting in this category label create two new volumes instead of a single new volume.

1.3 Blockchain Systems

Throughout this thesis, the terms *distributed ledger* and *blockchain* are used interchangeably. Since these terms are etymologically derived, respectively, from the *function* and the *form* of blockchain systems, we offer the following informal, simplified definitions:

- A distributed ledger is a set of records collectively managed and shared by its users.
- A blockchain system/network is an ordered collection of data blocks that are connected to one another via some cryptographic tool. Most commonly, each block contains the hash of all the data stored in the previous block.

Although the terms blockchain and cryptocurrencies are often conflated, blockchain systems have countless uses beyond creating alternative currencies, as one can infer from our brief definitions. For the sake of brevity and conciseness, we will limit our explanation of DLTs solely to the one used by LogLog: the Ethereum network.

Ethereum is a public, general-purpose blockchain (Wood, 2020). In this case, *public* means that the system is entirely ran and maintained by its users as opposed to a single entity or some cadre of companies. Ethereum is also described as *general-purpose* because it uses a Turing-complete language (Solidity) to write data onto its blocks. This characteristic allows users to develop

complex software that runs on the Ethereum network, software that is otherwise known as *smart contracts*. Because they are transparent and self-executing, smart contracts allow multiple trust-poor parties to reliably exchange, store and transform funds and information.

There is, however, one major limitation to this subtype of DLT: gas costs. Gas is the unit of measurement used to express the computational cost of executing a set of operations on Ethereum (Wood, 2020). Unsurprisingly, gas costs scale directly with computational complexity, but, perhaps less intuitively, they also scale with spatial complexity. Deploying a contract that contains a single fixed-size array, for example, costs less than deploying another with multiple dynamic arrays. This is because data blocks have a predetermined, limited size, and the larger the data structures used by the smart contract, the more block space is taken up by it, thus increasing the cost of *deploying* the contract. Gas costs are also monetary costs, so they can be compared to other forms of maintenance and operational costs by converting them to CAD.

It is worth noting that smart contracts by themselves are not enough to safely exchange large wood volumes: a means to digitize the aforementioned volumes and append relevant metadata to each of them is also required. Thus, we consider the tokenization of wood volumes.

1.4 Tokenization Standards

Before explaining tokenization standards we must first cover crypto-assets. The Financial Conduct Authority defines crypto-assets as "cryptographically secured digital representations of value or contractual rights that use some type of distributed ledger technology (DLT) and can be transferred, stored or traded electronically" (Authority, 2019). Hence, by their very definition, the value of a crypto-asset can be backed by goods or services, unlike fiat currency. We can consider a *WoodToken* to be a crypto-asset whose value is backed by a *specific* volume of wood. We stress *specific* because, in a CoC context, no two wood volumes are the same, seeing as they all have different provenance and ownership records. If we were to think of them as a currency, it would not be a fungible one since wood volumes are not interchangeable. Naturally, *WoodTokens* are also not fungible.

We define tokenization as the process used to map assets to digital tokens. In the context of crypto-assets and cryptocurrencies, there are many existing tokenization standards with different use cases in mind. The ERC-20 standard, for example, is one of the most widespread ones on Ethereum, counting over 270,000 compatible tokens (Etherscan, 2020b). Although it facilitates token transactions within and between different smart contracts, ERC-20 tokens are fungible, which makes them inadequate for LogLog. Instead, we decided to comply to the ERC-721 standard.

Within the Ethereum network, the ERC-721 standard is not as widespread as ERC-20, but it is nonetheless the de-facto choice for non-fungible tokens, as shown by the fact that many of the most popular DApps on the network use it (including CryptoKitties and Sorare, for example) (Etherscan, 2020a). The ERC-721 standard was specifically designed to create tokens that represent ownership over an asset, and to allow wallets and other token-management software to seamlessly transact the aforementioned tokens (Entriken, Shirley, Evans & Sachs, 2018). Because LogLog fits entirely within the proposed use cases of ERC-721, it would be out of the question to use an entirely home-brewed tokenization system in an industrial setting when such a thoroughly tested and widely compatible standard already exist. Abiding to it, however, does entail at least one significant problem: previous research has shown that fully implementing the ERC-721 standard incurs greater gas costs than simply using home-brewed tokens (Westerkamp *et al.*, 2020). This is most likely because full ERC-721 compliance involves several groups of functions, not all of which are always used. Some of the most important function groups considered are:

- Metadata functions (*name*, *symbol*, *tokenURI*, etc.): they are not strictly necessary in our case given that the metadata is inseparable from the token and therefore does not need additional referencing.
- Enumeration functions (*tokenOfOwnerByIndex*, *totalSupply*, etc.): these are all integral to LogLog since being able to search tokens by their unique index is a necessity.

- Core ERC-721 functions (*balanceOf*, *ownerOf*, *approve*, etc.): although extremely useful, some functions are redundant. For example, we exclusively use *safeTransferFrom*, so *transferFrom* is unnecessary.

We built an additional prototype using the ERC-1155 standard that further increases the gas efficiency of LogLog without compromising its correctness or compatibility. ERC-1155 is entirely backwards-compatible with both ERC-20 and ERC-721 (Radomski, Cooke, Castonguay, Therien, Binet & Sandford, 2018). In exchange for some slightly higher deployment costs, ERC-1155 introduces *batch transfers*, which allow for multiple tokens to be transferred in a single transaction. In a forestry supply chain, entities are likely to purchase different kinds of wood volumes in large bulks, which results in a significant performance improvement for the system as a whole. Although not implemented in our second prototype to keep it as homologous to the first as possible, ERC-1155 also increases the expansion potential of our application, since it allows LogLog to use multiple types of *WoodTokens* or new, supporting tokens. For all intents and purposes, ERC-1155 can be seen as a net improvement over ERC-721.

Comparing the ERC-721 OpenZeppelin library (OpenZeppelin, 2020b) to their ERC-1155 version (OpenZeppelin, 2020a) makes many standard-specific improvements become immediately evident. Overall, ERC-1155 has a much smaller footprint than ERC-721 in terms of functions. Whereas ERC-721 has four core packages (*ERC721*, *ERC721Enumerable*, *ERC721Metadata* and *ERC721Full*) and four extensions (*ERC721Mintable*, *ERC721MetadataMintable*, *ERC721Burnable* and *ERC721Pausable*), ERC-1155 has a single core package and two extensions (*ERC1155Pausable* and *ERC1155Burnable*). In addition to streamlining package structure, redundant functions were also removed. For example, instead of having two kinds of transfers (a safe one and a regular one), ERC-1155 only allows safe transfers (both individual and batch).

One final advantage that ERC-1155 has over ERC-721 is its *beforeTokenTransfer* function, which automates certain checks and ensures correct behavior for some edge transfer use cases, which makes contract behaviour more transparent and consistent, but can also be overridden to follow a custom set of rules.

CHAPTER 2

RELATED WORKS

In this section, we review DLT systems, forestry supply chains, general supply chain DLT systems and timber certification DLT systems.

2.1 Blockchains for Supply Chains

The simplest form of the tackled problem is tracking a single, untampered item from source to consumer. As the item trades hands, a smart contract keeps track of its location and physical characteristics, ensuring its integrity. This is a well known example that already has at least one industrial-grade solution in IBM Food Trust, based on the DLT Hyperledger Fabric (IBM, 2020).

However, scaling the problem to track massive volumes of items poses a unique challenge in itself. The main problem to consider is that when the size of the tracked volumes is too large to realistically label each of their components, they have to be labelled as volumetric sets. However, because these sets are often transformed in many different ways, merely tokenizing them is not enough. Mechanisms must be developed to accurately represent all operations performed on these sets and any data associated with them. Due to the limitations of blockchain systems when it comes to large datasets, a great deal of effort must be spent to reach satisfactory performance metrics without compromising the integrity of the data manipulated.

Existing works have produced at least one generalized blockchain-based solution to the aforementioned issue (Westerkamp *et al.*, 2020). In said instance, researchers tracked individual volume IDs using an ERC-721-compliant smart contract to mint and manage non-fungible tokens that represent the tracked volumes. For the sake of maintaining reasonable transactional performance, they integrated off-chain data storage into their design. They also introduced the concept of *Recipes* as stand-alone contracts that capture and regulate the transformation of volumes. These *Recipes* are defined in terms of inputs and outputs, dictating how tokens

of a certain type must be produced, altered and/or consumed in any given real-world process. Unsurprisingly, *Recipes* are also an important aspect of our proposed solution.

Contrary to previous work, LogLog is tailor-made for a specific industry rather than a generalized system. LogLog emphasizes data integrity and traceability over gas performance due to its fully-on-chain nature, which is also unique.

2.2 Blockchain for the Timber Industry

Researchers have already articulated the theoretical benefits and characteristics of including DLTs as part of forestry certification programs from various legal, environmental and economic points of view. One of the most interesting contributions to this initiative has been the Evidence, Verifiability and Enforceability (EVE) framework (Nikolakis *et al.*, 2018). The EVE framework highlights many of the shortcomings of current certification methods:

- Limited ability to monitor transaction data and metadata within the supply chain.
- Limited ability to safely share information among multiple trust-poor parties (members-auditors, members-members and members-customers).
- Participating members have often unclear roles and responsibilities.
- Irregular compliance: not all members follow norms to the same extent.

The same paper also proposes a blockchain-based forest value chain framework that shows how the intrinsic properties of DLTs can help overcome these issues:

- All transactions generate an audit trail (increased auditability).
- All on-chain records are tamper-proof (increased data integrity).
- All interactions depend on successful authentication and authorization, forcing companies to establish clear, pre-determined worker profiles (increased accountability).
- Smart contracts automatically enforce compliance to norms (homogenous enforcement).

Unlike EVE, our work is not concerned with producing a novel certification system. Instead, we propose improving an existing one using DLTs: the rules and standards of the FSC were not

changed in any way and LogLog only makes them easier to follow. Also, unlike EVE, this thesis does provide reference implementations for one such system.

On a different note, there is at least one academic example of reticence towards DLTs for the timber industry (Howson, Oakes, Baynham-Herd & Swords, 2019), but even then, the authors do recognize many of the aforementioned advantages. With that in mind, forestry professionals have urged researchers to develop blockchain software tailored to the needs of their industry, outlining their motivations and requirements (Düdder & Ross, 2020).

One notable answer to their call was published in 2018 (Figorilli, Antonucci, Costa, Pallottino, Raso, Castiglione & Pinci, 2018). However, the blockchain system in question uses a particularly complex hybridized approach (boasting 15 different architectural components, all incurring additional costs and limiting availability), does not, to the best of our knowledge, implement or use any industry-specific standards (FSC, SFI, SGEC, etc.) or any tokenization standards, and does not provide any gas measurements despite running on Ethereum.

Luckily, several such systems have been developed throughout the world recently that more specifically address the needs of their respective local timber industries (Cueva-Sánchez, Coyco-Ordemar & Ugarte, 2020; Sheng & Wicha, 2021). The key difference between those systems and LogLog, are their focus on impeding illegal wood trafficking, as opposed to implementing a full-fledged certification standard, and once again, their hybridized database approach.

CHAPTER 3

LOGLOG: THE DESIGN

The CoC standards state that each certified entity must keep an account for each category of wood volume they have in stock (Council, 2016). We propose in this thesis a form of tokenization to keep an accurate record of these volumes. The tokens, being associated to an *FSC category* and *product type*, track individual volume units. As a result, the tokens must be non-fungible. We seized this opportunity to verify the adequacy of the ERC-721 and ERC-1155 standards in this specific context. Partly due to the availability of libraries like OpenZeppelin, which we used to provide out-of-the-box ERC-721 and ERC-1155 compliance, we chose Ethereum as a testing platform. This choice of public infrastructure may save costs as long as gas performance remains reasonable, since FSC members and auditors would not be solely responsible for maintaining the required infrastructure. Furthermore, since most, if not all, information shared on the system is meant to be public, there is little incentive to consider private blockchain solutions.

One of the main differences between our current prototype and previous research examples is that its *Recipes* are part of a single contract that also encapsulates the token functionality (instead of being stand-alone contracts). Furthermore, all data is stored on-chain. This decision results in higher deployment and transactional costs, but also simplifies the architecture, provides greater data integrity and increases system availability.

3.1 Sample Use Case

Following a single wood volume along a supply chain is a good way to understand how the system functions. Figure 3.1 provides such an example, where physical processes are mapped to the processes of LogLog (in bold). In that instance, a single volume of maple trees is followed through the supply chain as it is split and transformed into different products: logs become planks and beams, which then become tables. Each of these operations triggers a homologous function in LogLog that applies the corresponding changes to the tokens used. Not shown in this illustration is the fact that when the volume reaches the end consumer (or any other non-FSC

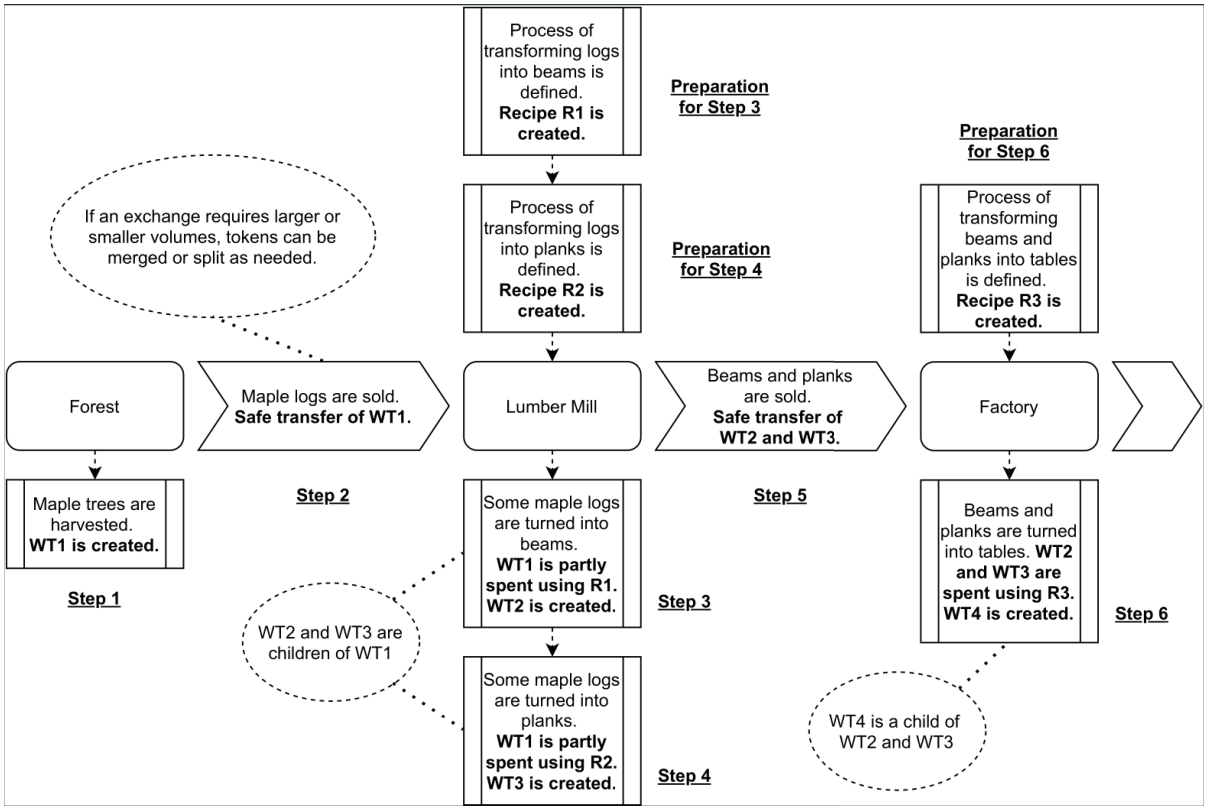


Figure 3.1 Partial Follow-through of a Volume

entity), the token associated with that volume is burnt, since it is no-longer FSC certified. Indeed, at any point in the supply chain, *FSC-certified Entities* can interact with *Non-FSC Entities* via volume exchanges. Generally speaking, all volumes sourced by the former from the latter are labelled and tracked as Controlled Wood. The only major exception to this rule is post-consumer wood and paper products, which are reintroduced into the supply chain as FSC Post Consumer Reclaimed Wood and Paper.

3.2 Design Considerations of LogLog

Among the reference papers that we consulted during research, the overall design of the proposed solutions only changed as it became more specialized. Our high-level understanding of the problem and the requirements of a viable solution were loosely based on papers that did not provide any concrete implementation examples (Düdder & Ross, 2020; Nikolakis *et al.*, 2018).

We then considered practical instances of blockchain and supply-chain integration (Helo & Hao, 2019). Comparing our requirements with the aforementioned implementation, it became evident that forestry supply chains would demand an extensive adaptation of the generalized model due to the constant transformation of goods in this context. Early in our development phase, we found more specialized models that addressed the issues associated with constant transformations (Gallersdörfer & Matthes, 2018; Westerkamp *et al.*, 2020). Still, even though they used a forestry supply chain as an example, they were not meant to be exclusively used in that context, and thus had to be even further adapted.

Drawing from these examples, we arrived at our current implementation, which is based on two key structures: *WoodTokens* and *Recipes*. In the following subsections we explain these structures and their interactions at length.

3.3 WoodToken

A *WoodToken* is the digital representation of a volume of some wood-derived product. It stores, on chain, all data related to the location, ownership, FSC classification and product labelling of said volume. This is quite expensive, and every example we considered chose to instead store that data off-chain to save operational costs. However, this decision simplifies token management and exchange considerably, as ERC-721 tokens are designed to be safely traded, with all basic token functionality covered by widely used libraries like OpenZeppelin.

One of the most effective optimizations we used for *WoodTokens* is to limit the size of the records stored by an individual token. Our cursory evaluation of a recent model of Canadian forestry supply chains (Shahi, 2016) suggests that any non-cyclical path between source and end-customer contains less than ten different nodes. Nodes, for our purposes, can be understood as owners or locations. Therefore, it seems fair to assume that there will be no more than ten different owners or locations stored in any one token. This is actually a very conservative estimate, as it is extremely unlikely that a wood volume is exchanged more than twice without being split or transformed into newer tokens. With this in mind, we provide traceability of tokens

throughout their entire life-cycle by storing an additional field on each token: the immediate parents of the token (i.e. the tokens that were used to create said token). Thus, we can trace back a token to its source through a genealogical query. For reference, *WoodTokens* are defined in Section 1 of APPENDIX I, lines 23-31.

3.4 Recipes

Recipes are the mechanism that regulates the transformation of *WoodTokens*. Whenever the nature of a wood product is changed (through refinement, for example), there must be a corresponding recipe that describes and quantifies that change. *Recipes* are also stored on chain and contain the number of inputs required, the type of inputs required, the type of output produced and the overall efficiency of the transformation (i.e. the ratio by which one can multiply the input volumes to get an accurate estimate of the output volume). Conveniently, FSC-certified companies are already expected to keep records of their transformative processes (Council, 2016) that are entirely homologous to our *Recipes*, which would facilitate transition into this kind of system.

Similarly to *WoodTokens*, we limit the amount of different input types to ten. It is essential to understand that this limitation is directly tied to the maximum number of direct parents stored by a token. However, it is possible to increase or decrease the limit of both direct parents and recipe input types independently of ownership and location limits. *Recipes* are defined in Section 1 of APPENDIX I, lines 17-21. Functions related to *Recipes* can be found in lines 351-368, 370-383, 385-393, 395-399 and 401-405.

3.5 Transformations

As a rule, a transformation implicitly involves the minting of at least one new *WoodToken*, and may use a single or multiple inputs. During a transformation, the smart contract conducts the following operations, in order:

1. Verify that all inputs:
    - Consist only of the product types specified in the used recipe.
    - Are authorized to be modified by the account that initiated the transformation.
    - Are stored in the same location.
    - Represent a large enough volume to process adequately.
2. Determine the FSC category of the output given the FSC categories of the inputs.
3. Conduct the corresponding FSC credit or FSC percentage calculations, as needed.
4. Determine the volume(s) of the output(s).
5. Mint one or more new tokens based on the previous operations, as needed.

This functionality is one of the most resource-intensive aspects of the smart contract as it involves many on-chain calculations. Considering, in addition, the size of all the data that a *WoodToken* has to contain, it is remarkably easy to exceed the default deployment gas limits of the Truffle Suite (Ganache, 2020), which serve as a realistic boundary for the sake of our tests. Hence, our implementation required careful attention to gas costs throughout the development process. The transformation process can be read in detail in Section 1 of APPENDIX I, lines 425-547.

3.6 Implementation Details

As mentioned previously, our prototype was implemented and tested in an Ethereum-like context using Solidity as a development language. We chose Ethereum because it the largest and most mature general-purpose blockchain at the moment. Also, because it is open-source, it has extensive, well-written documentation, plenty of available tutorials online and a myriad of libraries available to speed up development. We were specifically interested in the OpenZeppelin libraries for the ERC-721 and ERC-1155 tokenization standards, which were covered in Section 1.4.

We relied heavily on a handful of core tools from the Truffle Suite (Truffle Teams, Ganache, Truffle and Drizzle). Truffle itself was our means of compiling and deploying our smart contracts,

with Ganache being the local blockchain to which our contracts were deployed. However, we did not use Drizzle since we did not make a front-end for our prototypes in order to save more development time and we did not use Truffle Teams since the code was developed by a single person. Other major libraries used in development include the Chai Assertion Library for testing, and various functions out of the Web3 API for general-purpose programming.

We decided to implement the whole system using a single smart contract. Had we split it into multiple smaller contracts (for example: one for recipes, one for WoodTokens), ongoing costs would have been greater because inter-contract calls cost more gas than intra-contact calls. This was not a problem for testing because Ganache allows us to set the maximum block size to any value, whereas in a production setting we may have to split the contract into smaller parts, as needed, depending on the maximum block size of the network it is being deployed to.

Architecturally speaking, LogLog is strictly a blockchain-based back-end. In order to have a fully-functioning service, one would have to implement either a web-based or an app-based front-end or even a desktop client with which to upload wood-volume related information to LogLog. This also factored into the decision of omitting a front-end, since there is a vast degree of variance between FSC entities in terms of operational equipment and standard operating procedures: one type of front-end may be completely inadequate for one entity and perfect for another. Furthermore, these front-ends do not require any novel functionality, they do not present any obvious implementation challenges and could be implemented in a number of common programming languages (such as Python or JavaScript), so they do not affect the viability of our solution.

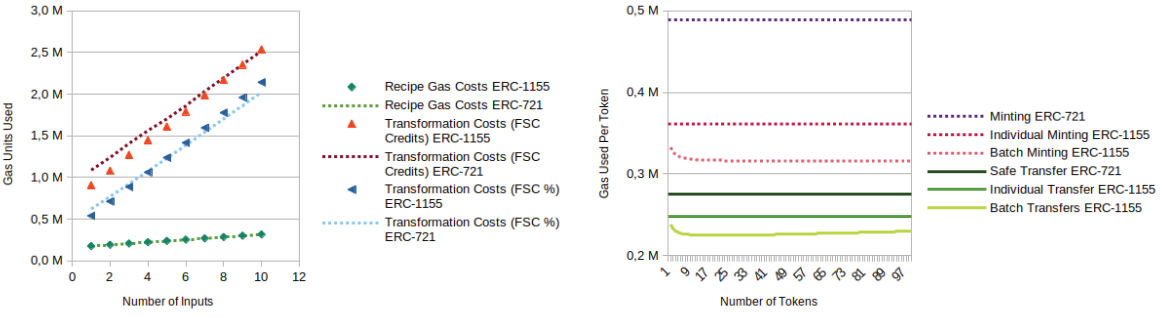
Within this context, it is possible to obtain realistic estimations of gas use. In a real-life deployment, latency varies depending on the gas cost that individual users are comfortable with. In contrast, we maintained relatively constant gas costs for each of our tests, with differences in latency being only relative to computational complexity.

3.7 Data Safeguards

When it comes to record-keeping, human error and malicious actors can generate sizeable costs. LogLog does indeed help prevent some forms of operational error. The first safeguard is protecting tokens from being modified or traded by unauthorized accounts: all operations on a token or its data must be called by the owner of the token. This same mechanism prevents transferring or modifying nonexistent *WoodTokens* as well. Overspending is also protected against: transformations and transactions both verify that sufficient *WoodTokens* are committed before conducting either operation. In instances were user error could not be prevented, such as when a wood volume is inaccurately measured prior to minting, all token metadata fields can be individually updated via functions on the smart contract. They can only be modified by authorized accounts, and records of the time and nature of the modification are kept for future reference, as is the case with all operations. These records make it trivial for a defrauded buyer, a vigilant manager or a skeptic auditor to prove who is responsible for any fraudulent operation.

CHAPTER 4

LOGLOG: THE RESULTS



a) Transformational Gas Performance

b) Transactional Gas Performance

Figure 4.1 Gas Cost Results

Both of our prototypes provided us with valuable gas data. However, calculating gas costs involves several external factors completely unrelated to the performance of either prototype. These external factors have a severe impact on the viability of our solution. Therefore, for our calculations, we assumed the following based on real-world data (Station, 2020):

- *Fast* transactions have a gas cost of 30 Gwei, and are executed within two minutes.
- *Standard* transactions have a gas cost of 26 Gwei, and are executed within five minutes.
- ETH 1.00 is equivalent to USD 240.00.

4.1 Gas Cost Analysis

Table 4.1 is based on those assumptions, illustrating the USD cost associated with some of the key operations that our system uses. Although the costs are modest, it is imperative that operations are performed in bulk whenever possible to minimize the expenses associated with running the system. Still, since the most recurring small-scale operation is a simple burn, which occurs between retailers (who are FSC certified) and end-consumers (who are not FSC certified), we believe that most entities should be able to absorb the costs of the system without difficulty. Naturally, it is possible to further reduce these costs by paying less per gas unit, increasing the

Table 4.1 Cost Analysis of ERC-721 and ERC-1155 Solutions

Token Standard	Operations	2-Input recipe	10-Input recipe	2-Input transformation (Credit)	10-Input transformation (Credit)	Deployment	Minting	Safe Transfer	Batch Minting (10 Tokens)	Batch Transfer (10 Tokens)
ERC-721	Gas Units Used	192710	316535	1241368	2519573	6009051	488558	275271	N/A	N/A
	Fast Cost (Gwei)	5781300	9496050	37241040	75587190	180271530	14656740	8258130	N/A	N/A
	Fast Cost (USD)	1.39	2.28	8.94	18.14	43.27	3.52	1.98	N/A	N/A
	Standard Cost (Gwei)	5010460	8229910	32275568	65508898	156235326	12702508	7157046	N/A	N/A
	Standard Cost (USD)	1.20	1.98	7.75	15.72	37.50	3.05	1.72	N/A	N/A
ERC-1155	Gas Units Used	193363	317188	1081716	2537330	8242331	361755	248564	3309222	2259728
	Fast Cost (Gwei)	5800890	9515640	32451480	76119900	247269930	10852650	7456920	99276660	67791840
	Fast Cost (USD)	1,39	2,28	7,79	18,27	59,34	2,60	1,79	23,83	16,27
	Standard Cost (Gwei)	5027438	8246888	28124616	65970580	214300606	9405630	6462664	86039772	58752928
	Standard Cost (USD)	1,21	1,98	6,75	15,83	51,43	2,26	1,55	20,65	14,10

latency in the process. However, there is a minimum safe price to pay per gas unit, below which any transaction is unlikely to be selected by miners (Wood, 2020). Users must keep this in mind even if they are comfortable with latencies greater than 30 minutes.

Figure 4.1a shows that all gas-related transformation costs scale linearly. This is to be expected since the operations described in 1.2, 1.2.1 and 1.2.2 have a computational complexity of O(n). On average, these costs are virtually identical for both prototypes since the underlying logic was left mostly unaltered. There is nonetheless a persisting discrepancy between the cost of a transformation that only involves FSC Percentage volumes and one that involves FSC Credit volumes. This is because FSC Credit transformations will practically always result in minting two tokens instead of one, since they have to account for the refuse (Controlled Wood) generated.

Due to the monolithic design of the prototype, considerable costs are entailed upon deployment. Table 4.1 shows that these one-time costs increased by roughly 37%. However, as shown in Figure 4.1b, ongoing transactional costs were reduced on average by 35% for minting and 17% for transferring. Furthermore, even with the additional costs of having a full ERC-1155-compliant

token, our implementation allowed us to include all metadata on the tokens themselves instead of having to store it off-chain.

To put our results into perspective, one of the previous hybridized designs (which used a maximum input cap five times greater than ours) reported costs of around 32K gas units for transferring, between 0.7M and 3.5M gas units for contract deployment and from less than 0.5M to 2M gas units for “batch creation" (Westerkamp *et al.*, 2020), a concept homologous to our transformation process. Surprisingly, even though the cited design took data off-chain to save costs and did not include any FSC certification logic, LogLog has cheaper transfer costs and similar transformational costs, which makes it more attractive for security-focused applications with smaller inputs. Still, we believe that with further optimization even more attractive results could be produced.

4.2 Yearly Cost Comparison with Traditional Certification Process

The inventories, and consequentially, trade volumes, observable throughout all members of a forestry supply chain fluctuate considerably between seasons due to the cyclical nature of timber production (D’Amours, Ouhimmou, Audy & Feng, 2017). These changes are compounded by the bull-whip effect and other economic forces. Accurately estimating the operating costs of our system is therefore quite difficult. We can nonetheless propose a ballpark figure for the yearly operation costs LogLog would incur upon a typical Lumber Mill. Under the assumptions (D’Amours *et al.*, 2017) that:

- There are 249 yearly workdays.
- The lumber mill operates on a spot purchase basis with all of its clients (distributors). In real life, this could also be done via a contract, which could in turn be automated by our system (not implemented).
- The average Lumber Mill makes between 5 and 10 sales daily, which corresponds with the number of daily shipments. Thus, we assume 8 transfers are made daily.
- The average Lumber Mill uses Standard Cost values presented in Table 4.1.

- 2-input FSC credit transformations are the standard. In this particular industry, transformations become considerably scarcer as the number of inputs increases. Naturally, a lumber mill that uses FSC percentage volumes predominantly would also save costs because of the discrepancy in gas performance previously discussed in this section.
- They only transform the tokens they need for each sale. This is actually extremely inefficient and unlikely, which makes for a very generous cost estimate. Normally, transforming entire volumes at once instead of incurring repeated transformation costs from transforming only subsets would be normal, but this makes estimation more difficult.
- All transactions are bulked together at the end of the day to save gas costs before uploading to the blockchain. Since there is no need for real-time tracking, and latency requirements are quite lax, this is a very effective way to cut system operating costs.
- Recipe costs are trivial and thus excluded. Unless the manufacturer creates new products repeatedly every year (very unlikely), recipes are only created once and even then they only cost about 2 dollars in the absolute worst case scenario.
- Deployment can be safely considered a one-time cost, but we will assume it is a yearly cost due to updates and changes in FSC requirements. Even then, it does not significantly impact cost estimates.

We propose the following estimate (in USD) for the yearly costs of running LogLog:

$$249 * 8 * 6.75 + 249 * 1 * 14.10 + 51.43 = 17008.33$$

Our lavish estimate falls well within the current FSC certification costs as discussed in Section 1.1, with all the added benefits that record digitization and certification automation provide. Additionally, yearly auditing costs should be substantially reduced.

It is worth noting that certification costs for a forest would be much lower than for a lumber mill or any other kind of manufacturer. Forests are not predominantly concerned with transforming wood volumes. Instead, they focus on minting and transacting them, which are much cheaper

operations. To understand just how much lower their costs would be, keep in mind that minting is about three times cheaper than transforming.

Our results make a strong case for implementing certification-specific DLT systems (hybridized or otherwise) as a means to reduce certification and auditing costs in the forestry industry.

CHAPTER 5

ZIPZAP: OVERVIEW

This chapter will provide a condensed explanation of ZipZap. Briefly put, ZipZap is an energy tokenization solution that is at least somewhat homologous to LogLog, despite the difference in context. Since ZipZapis a work in progress, it will not be covered as thoroughly as LogLog. Thus, many parts that would normally have their own chapters are found here instead as sections. With that in mind, this chapter begins with an overview of smart grid blockchain use cases and tokenization applications (Section 5.1). Section 5.2 briefly covers Ethereum-based smart-grid applications and systems similar to ZipZap. Section 5.3 explains the primary token used in our solution, whereas Section 5.4 explains the differences between the three prototypes produced. Finally, Section 5.5 broadly conveys some of the technical challenges encountered in the project and how they were handled.

5.1 Background

Integrating DLTs into smart grids has been the subject of considerable amounts of research. Many examples exist today of successful systems in this context, utilizing various blockchain platforms including Ethereum, Hyperledger and numerous others (Kuzlu, Sarp, Pipattanasomporn & Cali, 2020). Interest in DLTs has been driven by their capacity to improve existing services and create new use cases. What follows is a non-extensive list of these possibilities (Hassan, Yuen & Niyato, 2019b):

- Asset management: smart contracts allow for automatic distribution of costs and margins for energy-producing investments. For example, if multiple members of an apartment building choose to purchase a set of PV cells together, a smart contract can be used to distribute the energy fairly among them, to distribute the profits made from selling excess energy and to distribute the costs of ongoing maintenance and repairs, all based on their initial investment agreement.

- Grid monitoring: energy supply and consumption can be enriched with metadata via the tokenization of energy quantities, facilitating outage response, short-term decision-making and anticipatory planning.
- P2P energy exchanges: tokenizing energy quantities allows users to exchange energy directly among each other without necessarily having to use the energy company as an intermediary, guaranteeing payment and delivery by freezing funds allocated to the energy exchange contract, for example.
- Energy certification: just as with monitoring, tokenizing energy quantities allows the grid to append metadata to them and track their provenance (sustainable or non-sustainable source), as well a carbon credits or other accreditation incentives.

As evidenced by the short list above, energy tokenization is a prerequisite for some, if not all use cases that make DLTs attractive for smart grids. Thus, the primary goal of ZipZapat this stage of development is to provide the base energy tokenization functionality required for future extensions.

Just as with forestry-focused DLTs, Ethereum-based smart grid DLTs are always, to the best of our knowledge, never fully on-chain, opting to keep as much information off-chain as possible to save gas costs. That is why we followed a similar approach to LogLogin trying to develop a fully on-chain system with our first prototype. However, gas costs would force us to consider hybrid approaches for our following two prototypes.

Three different prototypes were developed for ZipZap, all with very different performance metrics. These differences can be entirely attributed to how much data each one of them stores and/or processes on-chain. The prototypes were named after weight categories, directly in proportion to their costs, with Heavyweight being the most expensive one and Featherweight being the least expensive one. They are all homologous in terms of functionality, but vary in terms of extendability given how vastly they differ in terms of data handling.

5.2 Related Works: Blockchain for Smart Energy Grids

There is an abundance of literature published in relation to DLT-enriched energy grids. Just as in the forestry context, the main value offerings of DLTs in a smart grid context are increased informational transparency and integrity, to name a few (Mylrea & Gourisetti, 2017). However, we found at least one interesting discrepancy between papers. To begin with, the previous comparative works cited, as well as many others, (Kuzlu *et al.*, 2020; Hassan *et al.*, 2019b) often showcase no small number of Ethereum-based solutions for several energy-related services. In a similar vein, there are also a number of other such papers that describe the different requirements that each energy use case has, as well as the features and limitations of many blockchain platforms (Mollah, Zhao, Niyato, Guan, Yuen, Sun, Lam & Koh, 2021; Hassan, Yuen & Niyato, 2019a). Both sets of papers show that Ethereum is a very popular, and, conservatively speaking, rather successful for commercial DLT software in this context.

In contrast with these observations, we found at several papers that highlight some of the key drawbacks of using Ethereum (and sometimes DLTs in general) in a smart energy grid context (Agung & Handayani, 2020; Hassan *et al.*, 2019a; Musleh, Yao & Muyeen, 2019). These drawbacks include, but are not limited to:

- Ethereum being a public blockchain means that additional efforts are required to anonymize customer data, which often means either using additional databases or having performance suffer from computational overheads caused by encryption.
- Ethereum is limited in how many transactions it can process per second. For small-scale systems this is not always problematic, but for a provincial or national-level power grid it could be a major hurdle to overcome.
- ETH fluctuates in value, which could make a very efficient system economically unsustainable almost overnight.

Our results do showcase some of these drawbacks, especially when it comes to gas costs, which leaves us wondering why Ethereum is such a widespread production-level platform for energy solutions instead of a prototype-focused one. We suspect that energy costs in countries where

successful instances were deployed may vary considerably from local energy costs, which has a noticeable impact on a system's viability.

5.3 Zap

A *Zap* is an ERC-1155 compatible token that represents a given amount of energy. Each *Zap* has the following metadata associated with it:

- A history of its geographical location.
- A history of its owner's accounts.
- The timestamp of its creation.
- The amount of energy it represents (in kWh).
- Power (in kW).
- Its estimated monetary value (in CAD).
- Its generator's ID number.
- The type of energy source exploited.

The most troublesome metadata fields are the two first ones listed, because they require, at a minimum, fixed-length arrays. Dynamic arrays and, to a much lesser extent, fixed-length arrays are quite costly in terms of gas. Heavyweight was able to save costs in this regard by limiting array size to five entries, since realistically it is extremely unlikely a *Zap* will ever be transferred more than once or twice at most given that the energy losses incurred during transportation and storage are so large that electricity is most often consumed right away.

Zaps are minted every five minutes, and reflect the amount of energy created by a generator during that time window. When a user consumer electricity, they consume *Zaps* until they have none left. In the event where their energy consumption out-scales their *Zap* collection, the system automatically requests additional *Zaps* from their local Transformer until the difference is covered.

5.4 Functional Overview of Prototypes

5.4.1 Heavyweight

Heavyweight was the first prototype developed. Practically speaking, this prototype is equivalent to the ERC-1155 version of LogLog because all of the metadata for each and every *Zap* is stored entirely on-chain. Consequentially, gas costs are quite steep. Attempts were made early in development to integrate not only *Zaps*, but also batteries and generators into the system, but gas costs quickly resulted in scoping out the two latter, since the additional arrays required were far too expensive. In the code provided in Section 1, we appended some of the planned battery and generator logic to showcase the issues with spatial complexity.

5.4.2 Featherweight

Featherweight is meant to represent the absolute lowest gas costs one could reasonably expect to get out of an Ethereum-based system with the same functionality as Heavyweight. A *Zap*'s metadata is stored off-chain, with only the hash of the aforementioned metadata being stored on-chain. This changes how transfers are handled. Since the smart contract cannot directly update the metadata after a transfer takes place, the new owners of the affected *Zaps* have to manually call a "modify *Zap*" function to update the metadata hash of the affected *Zaps* after a trade takes place. Therefore, the total cost of a transfer is the sum of the gas costs from the transfer itself and from the "modify *Zap*" function.

5.4.3 Lightweight

Featherweight also stores only *Zaps*' metadata. However, in contract with Lightweight, its functions require that the full *Zap* metadata be sent to the smart contract for all *Zap* related operations. The first step for any such operation is verifying the validity of the sent metadata using the stored hash. Once validated, if a change must take place in the metadata, the contract parses it and modifies it as needed before updating the stored hash and sending back the modified

metadata. Therefore, although it stores the same amount of information as Featherweight, Lightweight has a higher degree of automation and is more easily extended to include more complex operations.

5.5 Implementation Details

Just as with LogLog, we decided to use Ethereum as a platform because of its maturity and flexibility, but also because it has readily available tokenization libraries (notably from OpenZeppelin) that sped up development considerably. Therefore, all prototypes were developed using Solidity. They were compiled and tested with Truffle Suite, following much of the same methodology described in 3.6.

In developing ZipZap, the greatest hurdle to overcome was parsing the *Zaps*' metadata, although it was a challenged that only affected Lightweight. The problem arose from using a *bytes* array instead of a *string* array to handle metadata. The format of these *bytes* is often ambiguous because it is not standardized, which makes parsing them consistently very difficult. The solution to this first issue was to explicitly convert any and all inputs and outputs to *utf-8* prior to any manipulations (both by the front, or the back-end). However, parsing poses an additional problem. There are a number of libraries available for parsing bytes and strings in Solidity, but none of them are official, so they vary greatly in stability and functionality, to say the least. This forced us to tackle the bulk of the parsing manually, with minimal help from basic utilities like *BytesLib.sol* (Sá, 2021).

CHAPTER 6

ZIPZAP: THE RESULTS

This chapter will provide, for each prototype:

- their gas costs.
- an analysis of the aforementioned costs.
- some basic time-performance metrics and how they relate to the previous two points.

6.1 Heavyweight Gas Results

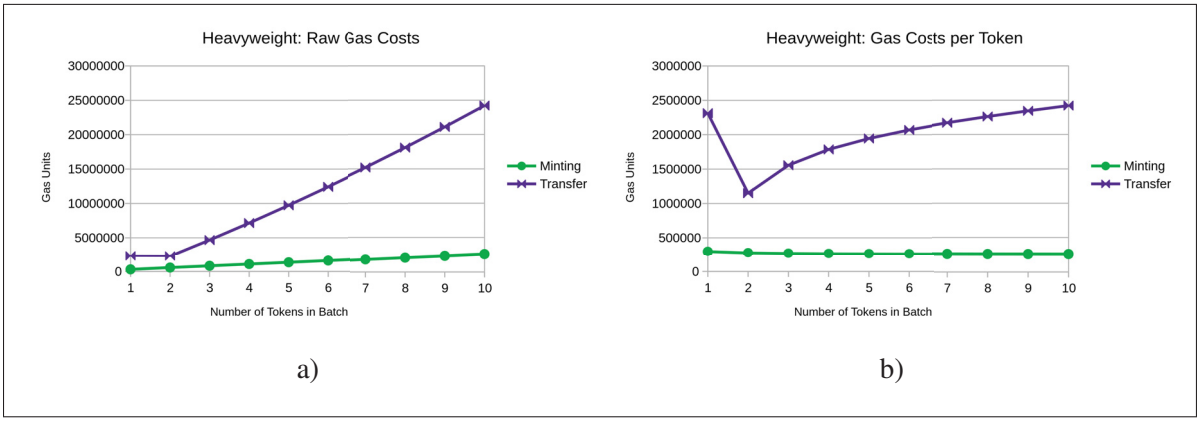


Figure 6.1 Heavyweight Gas Costs

Heavyweight's gas performance (as shown in Figures 6.1a and 6.1b) deviated from the norm in terms of bulk transformation savings. Whereas one would expect a constantly decreasing asymptotic curve (like the ones in Figure 6.3b), our results show a sharp decrease in gas costs, followed by a constantly increasing asymptotic curve. This is due to implicit on-chain operations out-scaling the gas savings associated with performing transactions in bulk. Namely, when any number of *Zaps* are transferred, the system has to update the metadata of each of them to reflect the change in ownership and address. This specific operation carries a computation cost of $O(N)$, bulk savings are approximately in the order of $1/O(log(N))$, as shown by the gas costs of Featherweight and Lightweight.

6.2 Featherweight Gas Results

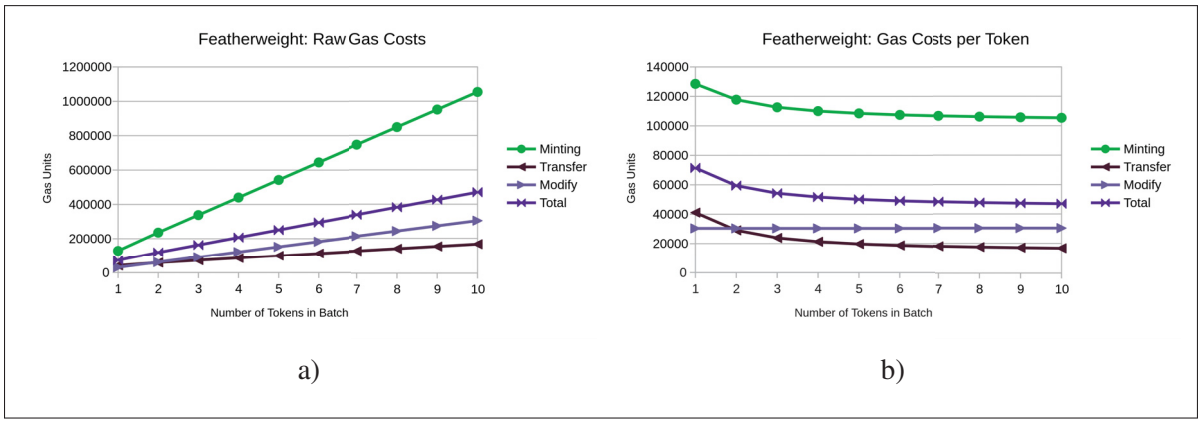


Figure 6.2 Featherweight Gas Costs

Featherweight is an interesting edge case. As noted in Section 5.4.2, the total cost of a transfer in this context is the sum of the gas costs from the transfer itself and from the "modify *Zap*" function, hence the additional lines in Figures 6.2a and 6.2b. Beyond being the lightest prototype, the main peculiarity of Featherweight is that it is the only prototype where minting is more expensive than transferring. This is because, despite having to add the costs of modification, all transfer-related sub-operations (like updating a *Zap*'s information) have to be manually triggered, so there are far fewer checks, validations and automatic triggering of functions for every transfer. This results in considerable gas savings for transferring tokens, since the operation is simplified.

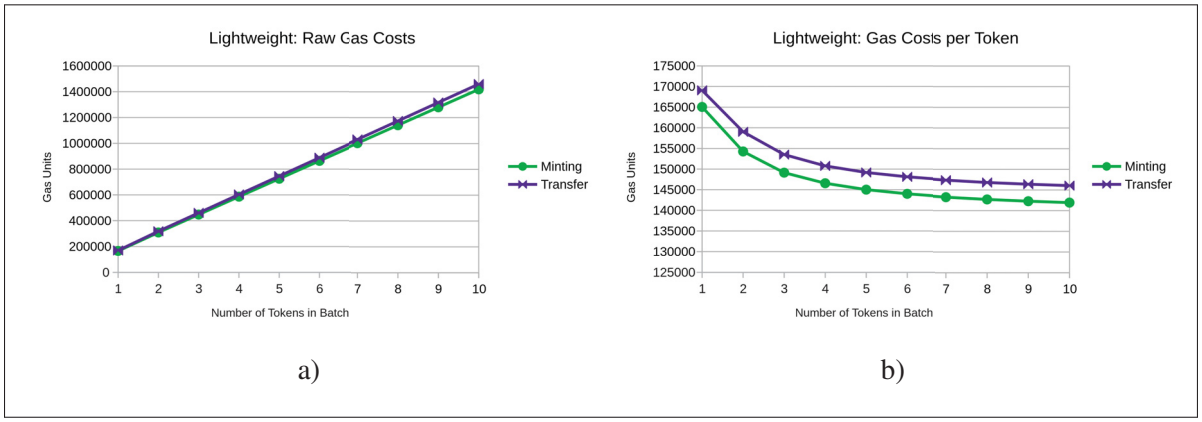


Figure 6.3 Lightweight Gas Costs

6.3 Lightweight Gas Results

Lightweight is a happy medium between Featherweight and Heavyweight, handling metadata in a similar manner to most blockchain applications. It is the most viable of all three prototypes, not strictly because of its gas metrics, which are slightly inferior to those of Featherweight, but because its processing of metadata allows it to be easily extended in terms of functionality. In other words, when compared to Featherweight, it would be almost trivial to extend Lightweight to include more features because it actually reads and modifies the metadata related to each token even if it only stores the hash. Heavyweight, on the other hand, has all the information it would need for many future use cases one could consider, but is limited by the already disproportionate costs of storing all that data.

Figure 6.4 shows how substantial the gas cost reductions are. All of the graphs shown therein are normalized in relation to Heavyweight's results. From these graphs, we can infer that the gas reductions attributed to storing metadata off-chain vary between approximately 40 to 30 percent when it comes to deployment. However, the biggest, most impressive savings come from recurring operations, where the cost associated with minting tokens is reduced between 50 and 60 percent and, incredibly, between 97 to 90 percent when it comes to transferring tokens.

6.4 Time Performance

ZipZaphas much lower latency requirements than LogLog, and a significantly higher rate of operations per day. This is because the power counting equipment currently installed throughout Quebec operates on five minute intervals. Therefore, all households must mint and/or transfer *Zaps*within that same five minute window. Assuming every household in the system generates some form of energy, this translates to, at a minimum, 288 mintings per household, per day. This wouldn't be a problem if they could be handled in bulk, but because they are spaced out in time and on a per-household basis, it is difficult to make the most out of the gas savings provided by ERC-1155 tokenization. Regardless, all three prototypes manage to meet the five minute requirement, with massive latency reductions recorded for both Featherweight and Lightweight.

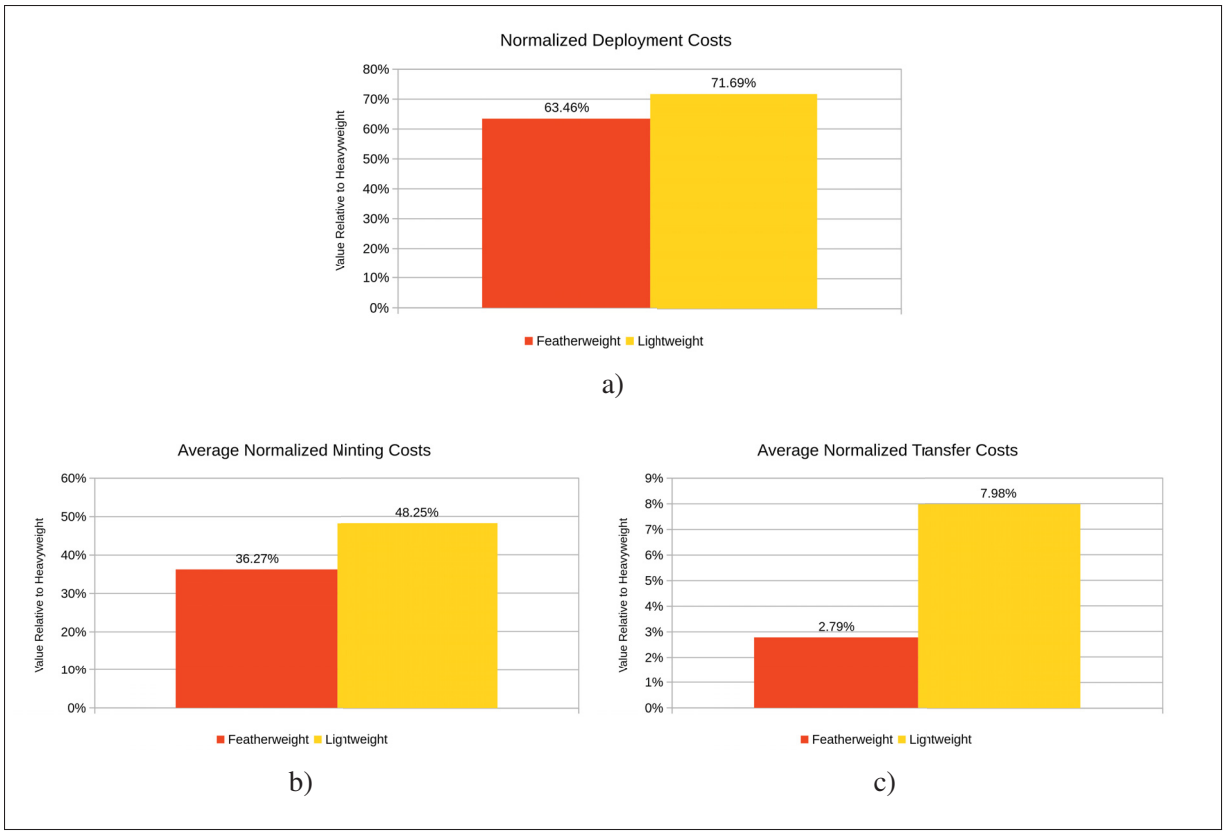


Figure 6.4 Gas Cost Comparison

6.5 Cost Viability Analysis

In this section we make the same assumptions related to real-world gas costs (Station, 2020) that we made in Section 4, but we also assumed that the energy cost for a single kWh hovered around 30 cents. Under these assumptions, the gas results obtained for both Lightweight and Featherweight were processed to produce Table 6.1. The table shows that despite our promising results, even the lightest of our prototypes can only be economically viable if the amount of energy stored by each *Zap* exceeds one kWh by several orders of magnitude. This is of course hardly reasonable since each generator in the system must emit a *Zap*every five minutes as per our requirements, and we are dealing with small generators that supply a single-household instead of provincial hydroelectric power plants. With this in mind, we aim to create a future-market-based implementation of ZipZap, so that *Zaps* can be transferred all at once at fixed time intervals.

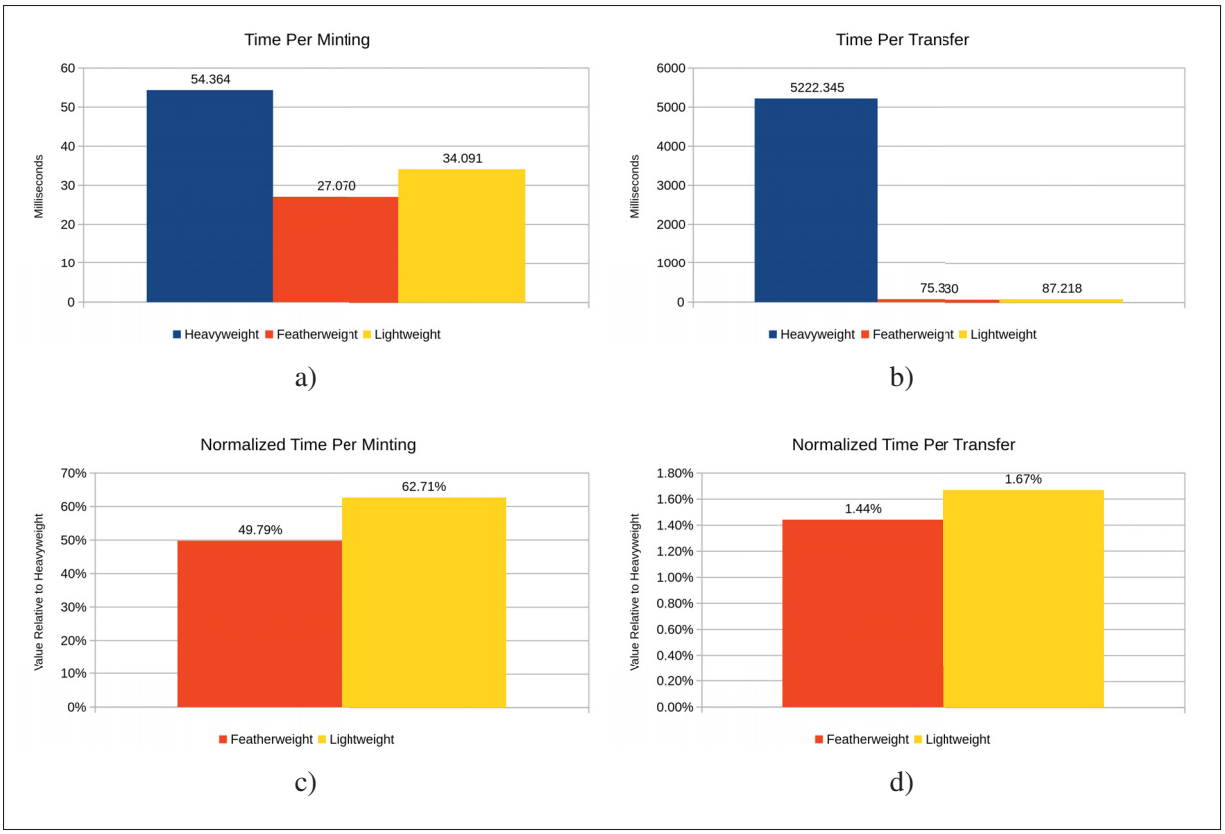


Figure 6.5 Time Performance Metrics

Table 6.1 Cost Analysis of ZipZap's Featherweight and Lightweight Versions

Prototype	Operations	Deployment	Minting	Safe Transfer	Batch Minting (10 Tokens)	Batch Transfer (10 Tokens)
Featherweight	**Gas Units Used**	3702977	158483	71456	1053673	469711
	Fast Cost (Gwei)	111089310	4754490	2143680	31610190	14091330
	Fast Cost (USD)	26.66	1.14	0.51	7.59	3.38
	Standard Cost (Gwei)	96277402	4120558	1857856	12212486	12212486
	Standard Cost (USD)	23.11	0.99	0.45	2.93	2.93
Lightweight	**Gas Units Used**	4592780	165052	169077	1419443	1460545
	Fast Cost (Gwei)	137783400	4951560	5072310	42583290	43816350
	Fast Cost (USD)	33.07	1.19	1.22	10.22	10.52
	Standard Cost (Gwei)	119412280	4291352	4396002	37974170	37974170
	Standard Cost (USD)	28.66	1.03	1.06	9.11	9.11

Another alternative would be to use a private blockchain like Hyperledger, which would not have any gas costs associated with it, but would have recurring equipment, labor and maintenance costs.

CHAPTER 7

LESSONS LEARNED

Designing and implementing all five of these prototypes provided invaluable information relative to the development of smart contracts, mostly in relation to gas expenses.

7.1 Fully On-Chain Systems

Whereas DLTs are almost exclusively used in conjunction with other types of databases, we decided to explore the performance of a fully on-chain system. Common sense dictates that on-chain systems are more expensive from a gas point of view, but that does not mean that one such system is altogether not viable. With LogLog, we managed to showcase at least one combination of circumstances where an on-chain system is comparable in term of ongoing monetary expenses to a currently used system and boasts significant functional and security advantages. A fully on-chain system might be viable wherever and whenever:

1. The need for data security and integrity is paramount. This is the key one advantage on-chain systems have over hybridized ones.
2. Latency requirements are relaxed. High latency helps reduce gas costs since prioritizing transactions normally entails paying a premium to have them processed more quickly.
3. The number and size of different data structures is low. Less complex systems are more likely to be viable than more complex ones. This is a direct consequence of the computational and spatial complexity metrics of the smart contract.

Furthermore, LogLog also showed that tokenization standards make sizeable impacts to the viability of on-chain systems. Additional developments in this area may easily widen the context in which on-chain systems are viable.

7.2 Computational Complexity and Spatial Complexity

A painful lesson to learn when it comes to optimizing smart contracts is that spatial complexity must drive the reasoning for most design decisions. As far as both LogLog prototypes are concerned, spatial complexity incurred much greater costs than computational complexity. The number of dynamic arrays in an object and the number of objects with dynamic arrays were the single greatest factors affecting the gas costs throughout development. Whenever possible, on-chain arrays have to be fixed-size, and the overall number of objects used in a system has to be as low as possible, especially when those objects use dynamic arrays.

These observations were also consistent in ZipZap, where the inclusion of batteries and generators in Heavyweight incurred substantial costs due both data structures requiring the addition of at least three dynamic arrays in total: one to keep track of tokens stored in a battery, one to keep track of all batteries and one to keep track of all generators.

If a dynamic array is necessary, then one must select the absolute smallest variable type to save costs whenever possible (for example, one should favor *uint8* over *int256*). The impacts of array type consideration became even more glaring when developing ZipZap, where we stopped using *string* arrays altogether to favor *byte* arrays regardless of the heavy parsing involved. The gas savings were so substantial than the results from ZipZap warrant developing a new LogLog prototype that does not use *strings*.

We believe that although spatial complexity had a much greater impact in our results compared to computational complexity, it is difficult to ascertain which one of these most directly affects performance. In our case, we only dealt, at most, with $O(N)$ computational complexities. On the other had, we had at least one instance where spatial complexity could have been $O(N^2)$ had we used dynamic arrays (for the FSC matrix, for example).

CHAPTER 8

FUTURE WORK

In the future, we aim to further optimize this LogLog through additional development iterations to decrease gas costs and increase maintainability. The following sections cover some of these possibilities. Please note that, since ZipZap is still in active development, we abstain from proposing future work in relation to any of its existing prototypes.

8.1 LogLog: Multi-token Approach

Using multiple types of WoodToken as opposed to our current, monolithic version may have some attractive advantages. Each WoodToken type could represent a different FSC category. This would make the code much more maintainable and extensible as it would allow for the the FSC matrix conversion table to be replaced entirely with polymorphic code. This would also remove the need for the various if-else blocks found mainly in the transformation function.

8.2 LogLog: Yearly-Agreement Extension

At larger scales, our current prototypes' spot-price functionality is not as realistic. In such cases, it is more common for supply chain entities to establish quarterly or yearly contract-based price agreements for wood volume exchanges. Naturally, this also presents an interest opportunity to implement an additional smart contract to manage this type of agreement and adjust WoodToken prices accordingly. In such a context, we would no longer follow the single-smart-contract approach because each trade agreement could potentially be complex enough to warrant its own compartmentalized logic and requires the ability to be modified as necessary. There is also the possibility that this approach becomes its own stand-alone solution with very different performance metrics.

8.3 LogLog: Data Restructuring

We could consider another alternative implementation that arranges each token's metadata into a JSON object, transforms that object into bytes and store only that sequence of bytes on a per-token basis. It could potentially save a great amount of gas, but the implementation of the system would be much more difficult and its maintainability would suffer since any and all changes to the metadata would involve non-standardised parsing. We implemented a similar system for ZipZap's Lightweight version, though it does not store the transformed JSON object so it is not entirely representative of expected gas costs. It does nonetheless serve to show that gas costs do not suffer as substantially from computing as they do from storing large amounts of data.

CONCLUSION

Our latest LogLog prototype demonstrates the adequacy of an entirely on-chain solution for tracking wood volumes and enforcing FSC standards. Our results show that blockchain technology can produce competitive single-database systems whenever the circumstances described in Section 7.1 take place.

In such a context, LogLog has the following benefits over a hybridized system:

- Greater availability: LogLog uses a single distributed database. Hybridized approaches need to keep at least two different databases online at all times, with at least one of them being distributed.
- Greater traceability: LogLog tracks all volume exchanges and modifications. Hybridized approaches are limited in their ability to capture these changes due to the informational gap between on-chain data and off-chain data.
- Greater data integrity: LogLog-tracked volume data has a high degree of immutability since it is wholly on-chain. Hybridized approaches use off-chain data that is more susceptible to tampering.

As for ZipZap, our prototypes demonstrate that in order for an Ethereum-based system to be a viable solution for energy tokenization, time-related requirements must be relaxed either by increasing the allowed latency per operation, or by reducing the number of daily operations. Its current requirements violate the second and third optimal circumstances for a fully on-chain system described in Section 7.1, so we can see that they also apply to hybridized systems, albeit in a less radical manner. On a similar note, although ZipZapis currently not economically viable at a small scale, large-scale energy suppliers may still find the solution very attractive due to the much larger energy quantities generated and transferred at any time interval.

We believe that many provenance-based certifications (for forestry, energy, agriculture, or other industries), may benefit greatly from similar approaches and encourage others to pursue research in alternative contexts.

APPENDIX I

LOGLOG

1. Full Smart Contract for ERC-1155 LogLog

Listing A I-1: Code for ERC-1155 LogLog

```
pragma solidity ^0.6.xx;
pragma experimental ABIEncoderV2;

import "@openzeppelin/contracts/token/ERC1155/ERC1155.sol";
import "@openzeppelin/contracts/utils/Counters.sol";

contract WoodTokenFactory is ERC1155 {
using Counters for Counters.Counter;
Counters.Counter private wood_token_ids;
Counters.Counter private recipe_ids;

enum FSC_CATEGORY {
FSC100, FSCMIXC, FSCMIXP, FSCRECYCLEDC, FSCRECYCLEDP, FSCRECYCLED100
    ,
FSCCONTROLLED, NOCLAIMS, FSCMIX100
}

struct Recipe {
uint transformation_ratio;
string[10] input_product_types;
string output_product_type;
}

struct WoodToken {
string product_type;
```

```
25 string[10] location_history;
26 address[10] owner_history;
27 uint[10] token_ancestry; //only direct parents
28 uint volume;
29 FSC_CATEGORY FSC_category;
30 uint percentage;
31 }
32
33 event WoodTokenCreated(uint[] newTokenIds);
34 event RecipeCreated(uint recipeId);
35 event ApprovalCheck(address sender, address approved);
36 event NumberCheck(uint somenum, uint someothernum);
37
38 WoodToken[] private wood_tokens;
39 mapping(uint256 => uint256) private wood_tokens_index;
40
41 Recipe[] private recipes;
42 mapping(uint256 => uint256) private recipes_index;
43
44 //trying to save as much space as possible
45 FSC_CATEGORY[7][7] private FSC_MATRIX;
46
47 constructor () public
48 ERC1155('https://woodtoken.dapp/tokens/{id}')
49 {
50 address[10] memory empty_address;
51 uint[10] memory empty_int;
52 string[10] memory empty_string;
53 wood_tokens.push(
54   WoodToken(
55     "zero",
```

```
56      empty_string,
57      empty_address,
58      empty_int,
59      0,
60      FSC_CATEGORY.NOCLAIMS,
61      0
62    )
63 );
64 recipes.push(Recipe(0, empty_string, "zero"));
65
66 //loads all the FSC-matrix stuff
67
68 FSC_MATRIX = [
69 [FSC_CATEGORY.FSC100, FSC_CATEGORY.FSCMIXC, FSC_CATEGORY.FSCMIXP,
70 FSC_CATEGORY.FSCMIXC, FSC_CATEGORY.FSCMIXP, FSC_CATEGORY.FSCMIX100,
71 FSC_CATEGORY.FSCCONTROLLED],
72 [FSC_CATEGORY.FSCMIXC, FSC_CATEGORY.FSCMIXC, FSC_CATEGORY.FSCMIXP,
73 FSC_CATEGORY.FSCMIXC, FSC_CATEGORY.FSCMIXP, FSC_CATEGORY.FSCMIXC,
74 FSC_CATEGORY.FSCCONTROLLED],
75 [FSC_CATEGORY.FSCMIXP, FSC_CATEGORY.FSCMIXP, FSC_CATEGORY.FSCMIXP,
76 FSC_CATEGORY.FSCMIXP, FSC_CATEGORY.FSCMIXP, FSC_CATEGORY.FSCMIXP,
77 FSC_CATEGORY.FSCCONTROLLED],
78 [FSC_CATEGORY.FSCMIXC, FSC_CATEGORY.FSCMIXC, FSC_CATEGORY.FSCMIXP,
79 FSC_CATEGORY.FSCRECYCLEDC, FSC_CATEGORY.FSCRECYCLEDP,
80 FSC_CATEGORY.FSCRECYCLEDC, FSC_CATEGORY.NOCLAIMS],
81 [FSC_CATEGORY.FSCMIXP, FSC_CATEGORY.FSCMIXP, FSC_CATEGORY.FSCMIXP,
82 FSC_CATEGORY.FSCRECYCLEDP, FSC_CATEGORY.FSCRECYCLEDP,
83 FSC_CATEGORY.FSCRECYCLEDP, FSC_CATEGORY.NOCLAIMS],
84 [FSC_CATEGORY.FSCMIX100, FSC_CATEGORY.FSCMIXC, FSC_CATEGORY.FSCMIXP,
85 FSC_CATEGORY.FSCRECYCLEDC, FSC_CATEGORY.FSCRECYCLEDP,
86 FSC_CATEGORY.FSCRECYCLED100, FSC_CATEGORY.NOCLAIMS],
```

```
87 [FSC_CATEGORY.FSCCONTROLLED, FSC_CATEGORY.FSCCONTROLLED,
88 FSC_CATEGORY.FSCCONTROLLED, FSC_CATEGORY.NOCLAIMS,
89 FSC_CATEGORY.NOCLAIMS, FSC_CATEGORY.NOCLAIMS,
90 FSC_CATEGORY.FSCCONTROLLED]
91 ];
92
93 }
94
95 //All volumes will be handled in square centimeters to avoid
      decimals.
96
97 function mintUniqueTokensTo(
98   address _to, string[] memory _product_types,
99   string[] memory _locations,
100   uint[] memory _volumes, uint[] memory _percentages,
101  FSC_CATEGORY[] memory _FSC_categories)
102 public returns (uint256[] memory _tokenIds){
103 require(
104     _product_types.length == _locations.length &&
105     _locations.length == _volumes.length &&
106     _volumes.length ==  _percentages.length &&
107     _percentages.length == _FSC_categories.length,
108     "Mismatched list lengths."
109 );
110
111 //prepping required info for ERC1155: removing off-chain data,
112 //forcing unique (non-fungible) tokens by setting amounts to 1
113 bytes memory _data = '';
114
115 uint j = 0;
116 string[10] memory _locationhistory;
```

```
117 address[10] memory _owners;
118 _owners[0] = _to;
119 uint256[10] memory _token_ancestry;
120
121 uint256[] memory _amounts = new uint256[](_product_types.length);
122 _tokenIds = new uint256[](_product_types.length);
123
124 for(j; j < _product_types.length; j++){
125 _amounts[j]=1;
126
127 wood_token_ids.increment();
128 _tokenIds[j] = wood_token_ids.current();
129
130 _locationhistory[0] = _locations[j];
131 wood_tokens_index[_tokenIds[j]] = wood_tokens.length;
132 wood_tokens.push(
133   WoodToken(
134     _product_types[j],
135     _locationhistory,
136     _owners,
137     _token_ancestry,
138     _volumes[j],
139     _FSC_categories[j],
140     _percentages[j]
141   )
142 );
143 }
144
145 super._mintBatch(_to, _tokenIds, _amounts, _data);
146 emit WoodTokenCreated(_tokenIds);
147 return _tokenIds;
```

```
148 }
149
150 function totalSupply() public view returns (uint256 _supply){
151 //not going to count the empty "origin/zero" token, hence the -1
152 _supply = wood_tokens.length-1;
153 return _supply;
154 }
155
156 function mintUniqueTokenTo(
157   address _to, string memory _product_type,
158   string memory _location, uint _volume, uint _percentage,
159  FSC_CATEGORY _FSC_category)
160 public returns (uint256 _tokenId){
161
162 string[] memory _product_types = new string[](1);
163 string[] memory _locations = new string[](1);
164 uint[] memory _volumes = new uint[](1);
165 uint[] memory _percentages = new uint[](1);
166 FSC_CATEGORY[] memory _FSC_categories = new FSC_CATEGORY[](1);
167
168 _product_types[0] = _product_type;
169 _locations[0] =_location;
170 _volumes[0] = _volume;
171 _percentages[0] = _percentage;
172 _FSC_categories[0] = _FSC_category;
173
174 return mintUniqueTokensTo(_to, _product_types, _locations, _volumes,
175 _percentages, _FSC_categories)[0];
176 }
177
178 function safeWoodTransferFrom(
```

```
179     address from, address to,
180     uint256 tokenId, string memory destination
181 ) public{
182 
183 setTokenCurrentLocation(tokenId, destination);
184 setTokenCurrentOwner(tokenId, to);
185 super.safeTransferFrom(from, to, tokenId, 1, '');
186 
187 }
188 
189 function safeBulkWoodTransferFrom(
190   address from, address to, uint256[] memory tokenIds,
191   string[] memory destinations, uint256[] memory amounts)
192 public{
193 
194 require(
195   tokenIds.length == destinations.length,
196   "ID list size does not match destination list size"
197 );
198 uint i = 0;
199 while(i < tokenIds.length){
200 setTokenCurrentLocation(tokenIds[i], destinations[i]);
201 setTokenCurrentOwner(tokenIds[i], to);
202 i++;
203 }
204 emit NumberCheck(tokenIds.length, amounts.length);
205 super.safeBatchTransferFrom(from, to, tokenIds, amounts, '');
206 }
207 function exists(uint256 _tokenId) public view returns (bool) {
208 return wood_tokens_index[_tokenId] > 0;
209 }
```

```
210
211 function setTokenCurrentLocation(
212   uint256 _tokenId, string memory _location
213 ) public{
214
215 require(
216   balanceOf(msg.sender, _tokenId) == 1,
217   "Access to one or more listed tokens not allowed."
218 );
219 string[10] memory history = getTokenLocationHistory(_tokenId);
220 uint i = 0;
221 for(i; i < history.length; i++){
222 if(keccak256(abi.encodePacked((history[i]))) ==
223   keccak256(abi.encodePacked((""))){
224 history[i] = _location;
225 break;
226 }
227 }
228 require(
229   keccak256(abi.encodePacked((history[i]))) ==
230   keccak256(abi.encodePacked((_location))),
231   "location array full, cannot add current location."
232 );
233 wood_tokens[wood_tokens_index[_tokenId]].location_history = history;
234 }
235
236
237 function setTokenCurrentOwner(
238     uint256 _tokenId, address _new_owner
239 ) public{
240 require(
```

```
241   balanceOf(msg.sender, _tokenId) == 1,
242   "Access to one or more listed tokens not allowed."
243 );
244 address[10] memory history = getTokenOwnerHistory(_tokenId);
245 address temp;
246 uint i = 0;
247 for(i; i < history.length; i++){
248 if(history[i] == temp){
249 history[i] = _new_owner;
250 break;
251 }
252 }
253 require(
254   history[i] == _new_owner,
255   "owner array full, cannot add current owner."
256 );
257 wood_tokens[wood_tokens_index[_tokenId]].owner_history = history;
258 }
259
260 function setTokenVolume(uint256 _tokenId, uint _volume) private{
261 require(exists(_tokenId), "Token does not exist.");
262 wood_tokens[wood_tokens_index[_tokenId]].volume = _volume;
263 }
264
265 function getToken(uint256 _tokenId) private view returns (
266 string memory _product_type,
267 string[10] memory _location_history,
268 address[10] memory _owner_history,
269 uint[10] memory _token_ancestry, //only direct parents
270 uint _volume,
271 FSC_CATEGORY _FSC_category,
```

```
272 uint _percentage
273 ) {
274 require(exists(_tokenId), "Token does not exist.");
275 _product_type = wood_tokens[wood_tokens_index[_tokenId]].
      product_type;
276 _location_history =
277     wood_tokens[wood_tokens_index[_tokenId]].location_history;
278 _owner_history =
279     wood_tokens[wood_tokens_index[_tokenId]].owner_history;
280 _token_ancestry =
281     wood_tokens[wood_tokens_index[_tokenId]].token_ancestry;
282 _volume = wood_tokens[wood_tokens_index[_tokenId]].volume;
283 _FSC_category = wood_tokens[wood_tokens_index[_tokenId]].
      FSC_category;
284 _percentage = wood_tokens[wood_tokens_index[_tokenId]].percentage;
285 }
286
287 function getTokenProductType(
288   uint256 _tokenId
289 ) public view returns (string memory a){
290 (a, , , , , , ) = getToken(_tokenId);
291 }
292
293 function getTokenLocationHistory(
294   uint256 _tokenId
295 ) public view returns (string[10] memory a){
296 ( , a, , , , , ) = getToken(_tokenId);
297 }
298
299 function getTokenLocation(
300   uint256 _tokenId
```

```
301 ) public view returns (string memory a){
302 string[10] memory history = getTokenLocationHistory(_tokenId);
303 for(uint i = 0; i< history.length; i++){
304 if(keccak256(abi.encodePacked((history[i]))) !=
305   keccak256(abi.encodePacked(("")))
306 ){
307 a = history[i];
308 }
309 }
310 }
311
312 function getTokenOwnerHistory(
313   uint256 _tokenId
314 ) public view returns (address[10] memory a){
315 ( , , a, , , , ) = getToken(_tokenId);
316 }
317
318 function getTokenOwner(
319     uint256 _tokenId
320 ) public view returns (address a){
321 address[10] memory owners = getTokenOwnerHistory(_tokenId);
322 for(uint i = 0; i < owners.length; i++){
323 if(owners[i] != address(0)){
324 a = owners[i];
325 }
326 }
327 }
328
329 function getTokenAncestry(
330   uint256 _tokenId
331 ) public view returns (uint[10] memory a){
```

```
332 ( , , , a, , , ) = getToken(_tokenId);
333 }
334
335 function getTokenVolume(uint256 _tokenId) public view returns (uint
      a){
336 ( , , , , a, , ) = getToken(_tokenId);
337 }
338
339 function getTokenFSCCategory(
340   uint256 _tokenId
341 ) public view returns (FSC_CATEGORY a){
342 ( , , , , , a, ) = getToken(_tokenId);
343 }
344
345 function getTokenPercentage(
346   uint256 _tokenId
347 ) public view returns (uint a){
348 ( , , , , , , a) = getToken(_tokenId);
349 }
350
351 function writeRecipe(
352   uint _transformation_ratio,
353   string[10] memory _input_product_types,
354   string memory _output_product_type
355 ) public returns (uint256 _recipe_id){
356
357 recipe_ids.increment();
358 _recipe_id = recipe_ids.current();
359 recipes_index[_recipe_id] = recipes.length;
360 recipes.push(
361   Recipe(
```

```
362       _transformation_ratio,
363       _input_product_types,
364       _output_product_type
365     )
366 );
367 emit RecipeCreated(_recipe_id);
368 }
369
370 function getRecipe(uint256 _recipe_id) private view returns (
371 uint _transformation_ratio,
372 string[10] memory _input_product_types,
373 string memory _output_product_type
374 ) {
375 require(recipeExists(_recipe_id), "Recipe does not exist.");
376
377 _transformation_ratio =
378     recipes[recipes_index[_recipe_id]].transformation_ratio;
379 _input_product_types =
380     recipes[recipes_index[_recipe_id]].input_product_types;
381 _output_product_type =
382     recipes[recipes_index[_recipe_id]].output_product_type;
383 }
384
385 function recipeExists(uint256 _recipe_id) public view returns (bool)
      {
386 return recipes_index[_recipe_id] > 0;
387 }
388
389 function getRecipeTransformationRatio(
390   uint256 _recipe_id
391 ) public view returns (uint a){
```

```
392 (a, , ) = getRecipe(_recipe_id);
393 }
394
395 function getRecipeInputProductTypes(
396   uint256 _recipe_id
397 ) public view returns (string[10] memory a){
398 (, a, ) = getRecipe(_recipe_id);
399 }
400
401 function getRecipeOutputProductType(
402   uint256 _recipe_id
403 ) public view returns (string memory a){
404 (, , a) = getRecipe(_recipe_id);
405 }
406
407 function subtractVolumeFromToken(
408   uint256 _tokenId, uint subvolume
409 ) private{
410 uint tokenVolume = getTokenVolume(_tokenId);
411 require(tokenVolume >= subvolume);
412 tokenVolume = tokenVolume - subvolume;
413 setTokenVolume(_tokenId, tokenVolume);
414 if (tokenVolume == 0){
415   _burn(_tokenId);
416 }
417 }
418
419 function findCorrespondingCategory(
420   FSC_CATEGORY a, FSC_CATEGORY b
421 ) private view returns (FSC_CATEGORY c){
422 c = FSC_MATRIX[uint(a)][uint(b)];
```

```
423 }
424
425 function transformToken(
426   uint256 _recipe_id, uint256[10] memory _tokenIds,
427   uint[10] memory _input_volumes
428 ) public{
429 uint arrayLength = _input_volumes.length;
430 require(recipeExists(_recipe_id), "Invalid recipe.");
431 string[10] memory recipe_prod_types =
432     getRecipeInputProductTypes(_recipe_id);
433
434
435 uint total_input_volumes = 0;
436 uint i=0;
437
438 uint256 final_percentage = 0;
439 FSC_CATEGORY final_category;
440 uint256 total_fsc_volume = 0;
441
442 for (i; i<arrayLength; i++) {
443 if(keccak256(abi.encodePacked((recipe_prod_types[i]))) ==
444   keccak256(abi.encodePacked((""))))
445 ){
446 break;
447 }
448 //check that input tokens have the needed input volumes
449 //check if caller of function is the owner of all tokens
450 require(
451   balanceOf(msg.sender, _tokenIds[i]) == 1,
452   "Access to one or more listed tokens not allowed."
453 );
```

```
454 require(
455   getTokenVolume(_tokenIds[i]) >= _input_volumes[i],
456   "Insufficient volume."
457 );
458
459 //check they are all in the same place
460 if(i+1 < arrayLength &&
461   keccak256(abi.encodePacked((recipe_prod_types[i+1]))) !=
462   keccak256(abi.encodePacked((""))))
463 ){
464 require(
465   keccak256(abi.encodePacked((getTokenLocation(_tokenIds[i])))) ==
466   keccak256(abi.encodePacked((getTokenLocation(_tokenIds[i+1])))),
467   "Volumes not at same physical location."
468 );
469 }
470 total_input_volumes = total_input_volumes + _input_volumes[i];
471
472 //iterate through input types, make sure we have all of them
473 require(
474   keccak256(abi.encodePacked((getTokenProductType(_tokenIds[i]))))
     ==
475   keccak256(abi.encodePacked((recipe_prod_types[i]))),
476   "Expected product types does not match recipe (order matters)."
477 );
478
479 if(exists(_tokenIds[i])){
480 if(i==0){
481 final_category = getTokenFSCCategory(_tokenIds[i]);
482 }
483 else{
```

```
484 final_category = findCorrespondingCategory(
485   final_category,getTokenFSCCategory(_tokenIds[i])
486   );
487 }
488 subtractVolumeFromToken(_tokenIds[i], _input_volumes[i]);
489 total_fsc_volume += _input_volumes[i]*getTokenPercentage(_tokenIds[i
      ]);
490 }
491 }
492
493 //apply ratio (2 steps for safety)
494 uint output_volume = total_input_volumes *
495   getRecipeTransformationRatio(_recipe_id);
496
497 output_volume /= 100;
498
499 if(final_category==FSC_CATEGORY.FSCCONTROLLED ||
500   final_category==FSC_CATEGORY.NOCLAIMS
501 ){
502 final_percentage = 0;
503 }
504 else if(final_category==FSC_CATEGORY.FSC100 ||
505   final_category==FSC_CATEGORY.FSCMIX100 ||
506   final_category==FSC_CATEGORY.FSCRECYCLED100
507 ){
508 final_percentage = 100;
509 }
510 else if(final_category==FSC_CATEGORY.FSCMIXC ||
511   final_category==FSC_CATEGORY.FSCRECYCLEDC
512 ){
513 //minting controlled portion
```

```
514 total_fsc_volume /= 100;
515 output_volume = (total_input_volumes-total_fsc_volume) *
516   getRecipeTransformationRatio(_recipe_id);
517 output_volume /= 100;
518 mintUniqueTokenTo(
519   msg.sender,
520   getRecipeOutputProductType(_recipe_id),
521   getTokenLocation(_tokenIds[0]),
522   output_volume,
523   0,
524   FSC_CATEGORY.FSCCONTROLLED
525 );
526
527 //no division by 100: show percentages as 100, 50, etc.
528 output_volume = total_fsc_volume *
529     getRecipeTransformationRatio(_recipe_id);
530 output_volume /= 100;
531 //removed the controlled portion from the volume: remainder is 100
532 final_percentage = 100;
533 }
534 else{ //FSC percentage categories: FSCMIXP and FSCRECYCLEDP
535 final_percentage = total_fsc_volume/total_input_volumes;
536 }
537
538 mintUniqueTokenTo(
539   msg.sender,
540   getRecipeOutputProductType(_recipe_id),
541   getTokenLocation(_tokenIds[0]),
542   output_volume,
543   final_percentage,
544   final_category
```

```
545 ) ;
546 }
547 }
```

Note: we only list the code for the ERC-1155 version because there are no substantial differences between it and the ERC-721 version beyond some function signatures to accommodate for the change in library.

APPENDIX II

ZIPZAP

1. Full Smart Contract for Heavyweight

Listing A II-1: Code for Heavyweight

```
pragma solidity ^0.6.xx;
pragma experimental ABIEncoderV2;

import "@openzeppelin/contracts/token/ERC1155/ERC1155.sol";
import "@openzeppelin/contracts/utils/Counters.sol";

contract ZapFactory is ERC1155 {
using Counters for Counters.Counter;
Counters.Counter private zap_ids;
Counters.Counter private generator_ids;
Counters.Counter private battery_ids;

enum ENERGY_SOURCE_TYPE {HYDROELECTRIC, PHOTOVOLTAIC, EOLIC,
    GEOTHERMAL, BIODIESEL, DIESEL, GAS, CHARCOAL}

struct Generator {
ENERGY_SOURCE_TYPE e_type;
address owner;
}

struct Battery {
uint capacity;
address owner;
}

```

```
26 struct Zap {
27 string[5] location_history;
28 address[5] owner_history;
29 uint creation_timestamp;
30 uint power; //in watts
31 uint kilowatt_hours;
32 uint kwh_spent;
33 uint dollar_value; //in cents
34 uint generator_id;
35 ENERGY_SOURCE_TYPE e_type;
36 }
37
38 //need to know HQ standard power rate
39
40 event ZapCreated(uint[] newTokenIds);
41 event ZapList(uint[] TokenIds);
42 event GeneratorCreated(uint newGenId);
43 event BatteryCreated(uint newBatteryId);
44 event ApprovalCheck(address sender, address approved);
45 event NumberCheck(uint somenum, uint someothernum);
46
47 Generator[] private generators;
48 //mapping generator ids to generators
49 mapping(uint256 => uint256) private generators_index;
50 //mapping owner addresses to generators
51 mapping(address => uint256[]) private owned_generators;
52
53 Battery[] private batteries;
54 //mapping battery ids to batteries
55 mapping(uint256 => uint256) private batteries_index;
56 //mapping owner addresses to batteries
```

```
57 mapping(address => uint256[]) private owned_batteries;
58
59 Zap[] private zap_tokens;
60 //mapping zap ids to zaps
61 mapping(uint256 => uint256) private zap_tokens_index;
62 //mapping owner addresses to zaps
63 mapping(address => uint256[]) private owned_zaps;
64
65 constructor () public
66 ERC1155('https://Zap.dapp/tokens/{id}')
67 {
68 address[5] memory empty_address;
69 string[5] memory empty_string;
70 zap_tokens.push(
71     Zap(
72         empty_string,
73         empty_address,
74         0,
75         0,
76         0,
77         0,
78         0,
79         0,
80         ENERGY_SOURCE_TYPE.HYDROELECTRIC
81     )
82 );
83 generators.push(
84     Generator(
85         ENERGY_SOURCE_TYPE.HYDROELECTRIC,
86         address(0)
87     )
```

```
88 );
89 batteries.push(Battery(0, address(0)));
90 //load energy rates and other stuff here
91 }
92
93 //All volumes will be handled in watthours to avoid decimals.
94
95 function mintUniqueTokensTo(
96     address _to, string[] memory _locations,
97     uint[] memory _timestamps,
98     uint[] memory _powers,
99     uint[] memory _kwhs,
100    uint[] memory _dollars,
101    uint[] memory _generator_ids,
102    ENERGY_SOURCE_TYPE[] memory _e_types)
103    public returns (uint256[] memory _zapIds){
104 require(_locations.length ==  _timestamps.length &&
105    _timestamps.length == _kwhs.length &&
106    _kwhs.length == _powers.length &&
107    _kwhs.length == _dollars.length &&
108    _dollars.length == _generator_ids.length &&
109    _generator_ids.length == _e_types.length,
110    "Mismatched list lengths.");
111
112 //prepping required info for ERC1155: removing off-chain data,
113 //forcing unique (non-fungible) tokens by setting amounts to 1
114 bytes memory _data = '';
115
116 uint j = 0;
117 string[5] memory _locationhistory;
118 address[5] memory _owners;
```

```
119 _owners[0] = _to;
120
121 uint256[] memory _amounts = new uint256[](_locations.length);
122 _zapIds = new uint256[](_locations.length);
123
124 for(j; j < _locations.length; j++){
125 _amounts[j]=1;
126
127 zap_ids.increment();
128 _zapIds[j] = zap_ids.current();
129
130 _locationhistory[0] = _locations[j];
131 zap_tokens_index[_zapIds[j]] = zap_tokens.length;
132 zap_tokens.push(
133     Zap(
134         _locationhistory,
135         _owners,
136         _timestamps[j],
137         _powers[j],
138         _kwhs[j],
139         0,
140         _dollars[j],
141         _generator_ids[j],
142         _e_types[j]
143     )
144 );
145 owned_zaps[_to].push(zap_ids.current());
146 }
147
148 super._mintBatch(_to, _zapIds, _amounts, _data);
149 emit ZapCreated(_zapIds);
```

```
150 return _zapIds;
151 }
152
153 function totalSupply() public view returns (uint256 _supply){
154 //not going to count the empty "origin/zero" token, hence the -1
155 _supply = zap_tokens.length-1;
156 return _supply;
157 }
158
159 function mintUniqueTokenTo(
160     address _to,
161     string memory _location,
162     uint _timestamp,
163     uint _power,
164     uint _kwh,
165     uint _dollar,
166     uint _generator_id,
167     ENERGY_SOURCE_TYPE _type)
168     public returns (uint256 _zapId){
169 string[] memory _locations = new string[](1);
170 _locations[0] = _location;
171
172 ENERGY_SOURCE_TYPE[] memory _types = new ENERGY_SOURCE_TYPE[](1);
173 _types[0] = _type;
174
175 /*
176 _timestamps[0] = _timestamp;
177 _powers[0] = _power;
178 _kwhs[0] = _kwh;
179 _dollars[0] = _dollar;
180 _genids[0] = _generator_id;
```

```
181 _types[0] = _type;
182 */
183 return mintUniqueTokensTo(
184     _to, _locations, returnUintArray(_timestamp),
185     returnUintArray(_power), returnUintArray(_kwh),
186     returnUintArray(_dollar), returnUintArray(_generator_id),
187     _types)[0];
188 }
189
190 function returnUintArray(uint a)
191     private returns (uint[] memory b){
192 b = new uint[](1);
193 b[0] = a;
194 }
195
196 function safeZapTransferFrom(
197     address from, address to, uint256 tokenId,
198     string memory destination) public{
199 setTokenCurrentLocation(tokenId, destination);
200 setTokenCurrentOwner(tokenId, to);
201 super.safeTransferFrom(from, to, tokenId, 1, '');
202 }
203
204 function safeBulkZapTransferFrom(
205     address from, address to, uint256[] memory tokenIds,
206     string[] memory destinations,
207     uint256[] memory amounts) public{
208 require(tokenIds.length == destinations.length,
209     "ID list size does not match destination list size");
210 uint i = 0;
211 while(i < tokenIds.length){
```

```
212 setTokenCurrentLocation(tokenIds[i], destinations[i]);
213 setTokenCurrentOwner(tokenIds[i], to);
214 i++;
215 }
216 emit NumberCheck(tokenIds.length, amounts.length);
217 super.safeBatchTransferFrom(from, to, tokenIds, amounts, '');
218 }
219
220 function exists(uint256 _zapId) public view returns (bool) {
221 return zap_tokens_index[_zapId] > 0;
222 }
223
224 function setTokenCurrentLocation(
225     uint256 _zapId, string memory _location) public{
226 require(balanceOf(msg.sender, _zapId) == 1,
227     "Access to one or more listed tokens not allowed.");
228 string[5] memory history = getZapLocationHistory(_zapId);
229 uint i = 0;
230 for(i; i < history.length; i++){
231 if(keccak256(abi.encodePacked((history[i]))) ==
232     keccak256(abi.encodePacked(("")))){
233 history[i] = _location;
234 break;
235 }
236 }
237 require(keccak256(abi.encodePacked((history[i])))
238     == keccak256(abi.encodePacked((_location))),
239     "location array full, cannot add current location.");
240 zap_tokens[zap_tokens_index[_zapId]].location_history = history;
241 }
242
```

```
243
244 function setTokenCurrentOwner(
245     uint256 _zapId, address _new_owner) public{
246 require(balanceOf(msg.sender, _zapId) == 1,
247     "Access to one or more listed tokens not allowed.");
248 address[5] memory history = getZapOwnerHistory(_zapId);
249 address temp;
250 uint i = 0;
251 for(i; i < history.length; i++){
252 if(history[i] == temp){
253 history[i] = _new_owner;
254 break;
255 }
256 }
257 uint j = 0;
258 for(j; j < owned_zaps[getZapOwner(_zapId)].length; j++){
259 if (owned_zaps[getZapOwner(_zapId)][j] == _zapId){
260 delete owned_zaps[getZapOwner(_zapId)][j];
261 }
262 }
263 owned_zaps[_new_owner].push(_zapId);
264 require(history[i] == _new_owner,
265     "owner array full, cannot add current owner.");
266 zap_tokens[zap_tokens_index[_zapId]].owner_history = history;
267 }
268
269 function setTokenKWH(uint256 _zapId, uint _kwh) private{
270 require(exists(_zapId), "Token does not exist.");
271 zap_tokens[zap_tokens_index[_zapId]].kilowatt_hours = _kwh;
272 }
273
```

```
274 function getZap(uint256 _zapId) private view returns (
275 string[5] memory _location_history,
276 address[5] memory _owner_history,
277 uint _timestamp,
278 uint _power,
279 uint _kilowatt_hours,
280 uint _kwh_spent,
281 uint _dollar_value,
282 uint _generator_id,
283 ENERGY_SOURCE_TYPE _e_type
284 ) {
285 require(exists(_zapId), "Token does not exist.");
286 _location_history =
287     zap_tokens[zap_tokens_index[_zapId]].location_history;
288 _owner_history = zap_tokens[zap_tokens_index[_zapId]].owner_history;
289 _timestamp = zap_tokens[zap_tokens_index[_zapId]].creation_timestamp
      ;
290 _power = zap_tokens[zap_tokens_index[_zapId]].power;
291 _kilowatt_hours = zap_tokens[zap_tokens_index[_zapId]].
      kilowatt_hours;
292 _kwh_spent = zap_tokens[zap_tokens_index[_zapId]].kwh_spent;
293 _dollar_value = zap_tokens[zap_tokens_index[_zapId]].dollar_value;
294 _generator_id = zap_tokens[zap_tokens_index[_zapId]].generator_id;
295 _e_type = zap_tokens[zap_tokens_index[_zapId]].e_type;
296 }
297
298 function getZapLocationHistory(
299     uint256 _zapId) public view
300     returns (string[5] memory a){
301 (a, , , , , , , , ) = getZap(_zapId);
302 }
```

```
303
304 function getZapLocation(
305     uint256 _zapId) public view
306     returns (string memory a){
307 string[5] memory history = getZapLocationHistory(_zapId);
308 for(uint i = 0; i< history.length; i++){
309 if(keccak256(abi.encodePacked((history[i]))) !=
310     keccak256(abi.encodePacked((""))))){
311 a = history[i];
312 }
313 }
314 }
315
316 function getZapOwnerHistory(
317     uint256 _zapId) public view
318     returns (address[5] memory a){
319 ( , a, , , , , , , ) = getZap(_zapId);
320 }
321
322 function getZapOwner(
323     uint256 _zapId) public view
324     returns (address a){
325 address[5] memory owners = getZapOwnerHistory(_zapId);
326 for(uint i = 0; i < owners.length; i++){
327 if(owners[i] != address(0)){
328 a = owners[i];
329 }
330 }
331 }
332
333 function getZapPreviousOwner(
```

```
334     uint256 _zapId) public view
335     returns (address a){
336 address[5] memory owners = getZapOwnerHistory(_zapId);
337 uint i = 0;
338 for(i; i < owners.length; i++){
339 if(owners[i] == address(0)){
340 break;
341 }
342 }
343 if(i-2 >= 0){
344 a = owners[i-2];
345 }
346 else{
347 a = address(0);
348 }
349 }
350
351 function getZapTimestamp(
352     uint256 _zapId) public view
353     returns (uint a){
354 ( , , a, , , , , , ) = getZap(_zapId);
355 }
356
357 function getZapPower(
358     uint256 _zapId) public view
359     returns (uint a){
360 ( , , , a, , , , , ) = getZap(_zapId);
361 }
362
363 function getZapKWH(
364     uint256 _zapId) public view
```

```
365     returns (uint a){
366 ( , , , , a, , , , ) = getZap(_zapId);
367 }
368
369 function getZapSpentKWH(
370     uint256 _zapId) public view
371     returns (uint a){
372 ( , , , , , a, , , ) = getZap(_zapId);
373 }
374
375 function getZapDollarValue(
376     uint256 _zapId) public view
377     returns (uint a){
378 ( , , , , , , a, , ) = getZap(_zapId);
379 }
380
381 function getZapGenId(
382     uint256 _zapId) public view
383     returns (uint a){
384 ( , , , , , , , a, ) = getZap(_zapId);
385 }
386
387 function getZapEnergyType(
388     uint256 _zapId) public view
389     returns (ENERGY_SOURCE_TYPE a){
390 ( , , , , , , , , a) = getZap(_zapId);
391 }
392
393 function registerGenerator(
394     ENERGY_SOURCE_TYPE _e_type, address _owner) public
395     returns (uint256 _generator_id){
```

```
396 generator_ids.increment();
397 _generator_id = generator_ids.current();
398 generators_index[_generator_id] = generators.length;
399 generators.push(Generator(_e_type, _owner));
400 emit GeneratorCreated(_generator_id);
401 }
402
403 function getGenerator(
404     uint256 _generator_id)
405     private view returns (
406 ENERGY_SOURCE_TYPE _e_type,
407 address _owner
408 ) {
409 require(generatorExists(_generator_id),
410     "Generator does not exist.");
411 _owner = generators[generators_index[_generator_id]].owner;
412 _e_type = generators[generators_index[_generator_id]].e_type;
413 }
414
415 function generatorExists
416     (uint256 _generator_id) public view
417     returns (bool){
418 return generators_index[_generator_id] > 0;
419 }
420
421 function getGeneratorOwner(
422     uint256 _generator_id) public view
423     returns (address a){
424 ( , a) = getGenerator(_generator_id);
425 }
426
```

```
427 function getGeneratorEnergyType(
428     uint256 _generator_id) public view
429     returns (ENERGY_SOURCE_TYPE a){
430 (a, ) = getGenerator(_generator_id);
431 }
432
433 function registerBattery(
434     uint _capacity, address _owner) public
435     returns (uint256 _battery_id){
436 battery_ids.increment();
437 _battery_id = battery_ids.current();
438 batteries_index[_battery_id] = batteries.length;
439 batteries.push(Battery(_capacity, _owner));
440 emit BatteryCreated(_battery_id);
441 }
442
443 function getBattery(
444     uint256 _battery_id) private view
445     returns (
446 uint _capacity,
447 address _owner
448 ) {
449 require(batteryExists(_battery_id), "Battery does not exist.");
450 _owner = batteries[batteries_index[_battery_id]].owner;
451 _capacity = batteries[batteries_index[_battery_id]].capacity;
452 }
453
454 function batteryExists(
455     uint256 _battery_id) public view
456     returns (bool){
457 return batteries_index[_battery_id] > 0;
```

```
458 }
459
460 function getBatteryOwner(
461     uint256 _battery_id) public view
462     returns (address a){
463 ( , a) = getBattery(_battery_id);
464 }
465
466 function getBatteryCapacity(
467     uint256 _battery_id) public view
468     returns (uint a){
469 (a, ) = getBattery(_battery_id);
470 }
471
472 function consumeEnergy(address customer, uint subkwh) public{
473 uint256[] memory zapIds = owned_zaps[customer];
474 uint256[] memory zapsKwh;
475 uint256 currentKwhCount = 0;
476 uint i = 0;
477 for(i; i<zapIds.length; i++){
478 zapsKwh[i] = getZapKWH(zapIds[i])-getZapSpentKWH(zapIds[i]);
479 currentKwhCount += zapsKwh[i];
480 if (currentKwhCount > subkwh){
481 spendKWHFromToken(zapIds[i], currentKwhCount-subkwh);
482 break;
483 }
484 spendKWHFromToken(zapIds[i], zapsKwh[i]);
485 }
486 if (currentKwhCount < subkwh){
487 //HQ provides the difference when no tokens are available.
488 uint256 HQToken = mintUniqueTokenTo(
```

```
489     address(0), "X1X1X1", 111111, 10,
490     subkwh - currentKwhCount,
491     (subkwh - currentKwhCount)*7,
492     0, ENERGY_SOURCE_TYPE.HYDROELECTRIC);
493 safeZapTransferFrom(
494     address(0), customer, HQToken, getZapLocation(zapIds[i]));
495 spendKWHFromToken(HQToken, subkwh - currentKwhCount);
496 }
497 }
498
499 function spendKWHFromToken(
500     uint256 _tokenId, uint256 _kwh) private{
501 require(getZapKWH(_tokenId) >= _kwh && _kwh > 0);
502 zap_tokens[zap_tokens_index[_tokenId]].kwh_spent += _kwh;
503 }
504
505 function listZapsOwnedBy(
506     address customer) public
507     returns (uint256[] memory){
508 emit ZapList(owned_zaps[customer]);
509 return owned_zaps[customer];
510 }
511 }
```

2. Full Smart Contract for Featherweight

Listing A II-2: Code for Featherweight

```
1 pragma solidity ^0.6.xx;
2 pragma experimental ABIEncoderV2;
3
4 import "@openzeppelin/contracts/token/ERC1155/ERC1155.sol";
```

```
5 import "@openzeppelin/contracts/utils/Counters.sol";
6
7 contract ZapFactoryHybrid is ERC1155 {
8 using Counters for Counters.Counter;
9 Counters.Counter private zap_ids;
10 Counters.Counter private generator_ids;
11 Counters.Counter private battery_ids;
12
13 enum ENERGY_SOURCE_TYPE {HYDROELECTRIC, PHOTOVOLTAIC,
14     EOLIC, GEOTHERMAL, BIODIESEL, DIESEL, GAS, CHARCOAL}
15
16 struct Zap {
17 bytes32 metadatahash;
18 }
19
20 event ZapCreated(uint[] newTokenIds);
21 event SingleZapCreated(uint newTokenId);
22 event ZapList(uint[] TokenIds);
23 event ApprovalCheck(address sender, address approved);
24 event NumberCheck(uint somenum, uint someothernum);
25
26 Zap[] private zap_tokens;
27 //mapping zap ids to zaps
28 mapping(uint256 => uint256) private zap_tokens_index;
29 //mapping owner addresses to zaps
30 mapping(address => uint256[]) private owned_zaps;
31
32 constructor () public
33 ERC1155('https://zap.dapp/tokens/{id}')
34 {
35 zap_tokens.push(Zap(''));
```

```
36 //load energy rates and other stuff here
37 }
38
39 //All volumes will be handled in watthours? to avoid decimals.
40
41 function mintUniqueTokenTo(
42     address _to, bytes32 _hash) public returns (uint256 _zapId){
43 bytes memory _data = '';
44
45 zap_ids.increment();
46 _zapId = zap_ids.current();
47
48 zap_tokens_index[_zapId] = zap_tokens.length;
49 zap_tokens.push(Zap(_hash));
50 owned_zaps[_to].push(zap_ids.current());
51
52 super._mint(_to, _zapId, 1, _data);
53 emit SingleZapCreated(_zapId);
54 return _zapId;
55 }
56
57 function mintUniqueTokensTo(
58     address _to, bytes32[] memory _hashes) public
59     returns (uint256[] memory _zapIds){
60 uint j = 0;
61
62 uint256[] memory _amounts = new uint256[](_hashes.length);
63
64 _zapIds = new uint256[](_hashes.length);
65
66 for(j; j < _hashes.length; j++){
```

```
67 _amounts[j]=1;
68
69 zap_ids.increment();
70 _zapIds[j] = zap_ids.current();
71
72 zap_tokens_index[_zapIds[j]] = zap_tokens.length;
73 zap_tokens.push(Zap(_hashes[j]));
74 owned_zaps[_to].push(zap_ids.current());
75 }
76
77 super._mintBatch(_to, _zapIds, _amounts, '');
78 emit ZapCreated(_zapIds);
79 return _zapIds;
80 }
81
82 function totalSupply() public view returns (uint256 _supply){
83 //not going to count the empty "origin/zero" token, hence the -1
84 _supply = zap_tokens.length-1;
85 return _supply;
86 }
87
88 function exists(uint256 _zapId) public view returns (bool) {
89 return zap_tokens_index[_zapId] > 0;
90 }
91
92 //getZap now just returns the hash of the Zap,
93 //if the Zap exists. Replaces getZapHash(uint256 _zapId).
94 function getZap(uint256 _zapId) public view returns (
95 bytes32 _metadatahash
96 ) {
97 require(exists(_zapId), "Token does not exist.");
```

```
98 _metadatahash = zap_tokens[zap_tokens_index[_zapId]].metadatahash;
99 }
100
101 function modifyZapHash(
102     uint256 _zapId, bytes32 _newhash) public {
103 require(balanceOf(msg.sender, _zapId) == 1,
104     "Access to one or more listed tokens not allowed.");
105 zap_tokens[zap_tokens_index[_zapId]].metadatahash = _newhash;
106 }
107
108 function safeZapTransferFrom(
109     address from, address to,
110     uint256 tokenId, uint256 amount) public{
111 super.safeTransferFrom(from, to, tokenId, amount, '');
112 }
113
114 function safeBulkZapTransferFrom(
115     address from, address to, uint256[] memory tokenIds,
116     uint256[] memory amounts) public{
117 require(tokenIds.length == amounts.length,
118     "ID list size does not match amount list size");
119 super.safeBatchTransferFrom(from, to, tokenIds, amounts, '');
120 }
121 }
```

3. Full Smart Contract for Lightweight

Listing A II-3: Code for Lightweight

```
1 pragma solidity ^0.6.xx;
2 pragma experimental ABIEncoderV2;
3
```

```
4 import "@openzeppelin/contracts/token/ERC1155/ERC1155.sol";
5 import "@openzeppelin/contracts/utils/Counters.sol";
6 import "installed_contracts/bytes/contracts/BytesLib.sol";
7
8 contract ZapFactoryHybrid2 is ERC1155 {
9 using Counters for Counters.Counter;
10 using BytesLib for bytes;
11 Counters.Counter private zap_ids;
12 Counters.Counter private generator_ids;
13 Counters.Counter private battery_ids;
14
15 enum ENERGY_SOURCE_TYPE {HYDROELECTRIC, PHOTOVOLTAIC,
16     EOLIC, GEOTHERMAL, BIODIESEL, DIESEL, GAS, CHARCOAL}
17
18 struct Zap {
19 bytes32 metadatahash;
20 }
21
22 event ZapCreated(uint[] newTokenIds);
23 event SingleZapCreated(uint newTokenId);
24 event ZapList(uint[] TokenIds);
25 event ApprovalCheck(address sender, address approved);
26 event NumberCheck(uint somenum, uint someothernum);
27 event NewMetaInHex(bytes data);
28
29
30 Zap[] private zap_tokens;
31 //mapping zap ids to zaps
32 mapping(uint256 => uint256) private zap_tokens_index;
33 //mapping owner addresses to zaps
34 mapping(address => uint256[]) private owned_zaps;
```

```
35
36 constructor () public
37 ERC1155('https://zap.dapp/tokens/{id}')
38 {
39 zap_tokens.push(Zap(''));
40 //load energy rates and other stuff here
41 }
42
43 //All volumes will be handled in watthours to avoid decimals.
44
45 function mintUniqueTokenTo(
46     address _to, bytes memory _data) public
47     returns (uint256 _zapId){
48
49 zap_ids.increment();
50 _zapId = zap_ids.current();
51
52 zap_tokens_index[_zapId] = zap_tokens.length;
53 zap_tokens.push(Zap(keccak256(_data)));
54 owned_zaps[_to].push(zap_ids.current());
55
56 super._mint(_to, _zapId, 1, '');
57 emit SingleZapCreated(_zapId);
58 return _zapId;
59 }
60
61
62 function mintUniqueTokensTo(
63     address _to, bytes[] memory _data) public
64     returns (uint256[] memory _zapIds){
65
```

```
66 uint j = 0;
67
68 uint256[] memory _amounts = new uint256[](_data.length);
69
70 _zapIds = new uint256[](_data.length);
71
72 for(j; j < _data.length; j++){
73 _amounts[j]=1;
74
75 zap_ids.increment();
76 _zapIds[j] = zap_ids.current();
77
78 zap_tokens_index[_zapIds[j]] = zap_tokens.length;
79
80 zap_tokens.push(Zap(keccak256(_data[j])));
81 owned_zaps[_to].push(zap_ids.current());
82 }
83
84 super._mintBatch(_to, _zapIds, _amounts, '');
85 emit ZapCreated(_zapIds);
86 return _zapIds;
87 }
88
89 function stringToBytes32(
90     string memory source) public pure
91     returns (bytes32 result) {
92 bytes memory tempEmptyStringTest = bytes(source);
93 if (tempEmptyStringTest.length == 0) {
94 return 0x0;
95 }
96
```

```
97 assembly {
98 result := mload(add(source, 32))
99 }
100 }
101
102 function toAsciiString(
103     address x) internal view
104     returns (string memory) {
105 bytes memory s = new bytes(40);
106 for (uint i = 0; i < 20; i++) {
107 bytes1 b = bytes1(uint8(uint(uint160(x)) / (2**(8*(19 - i)))));
108 bytes1 hi = bytes1(uint8(b) / 16);
109 bytes1 lo = bytes1(uint8(b) - 16 * uint8(hi));
110 s[2*i] = char(hi);
111 s[2*i+1] = char(lo);
112 }
113 return string(s);
114 }
115
116 function char(
117     bytes1 b) internal view
118     returns (bytes1 c) {
119 if (uint8(b) < 10) return bytes1(uint8(b) + 0x30);
120 else return bytes1(uint8(b) + 0x57);
121 }
122
123 function findFirst(
124     bytes memory query, byte character,
125     uint i) internal pure returns (uint256 position){
126 uint size = query.length;
127 position = 7777777;
```

```
128 while(i < size){
129 if(query[i] == character){
130 position = i;
131 break;
132 }
133 i++;
134 }
135 require(position < 7777777, "Invalid metadata.");
136 return position;
137 }
138
139 function totalSupply() public view returns (uint256 _supply){
140 //not going to count the empty "origin/zero" token, hence the -1
141 _supply = zap_tokens.length-1;
142 return _supply;
143 }
144
145 function exists(uint256 _zapId) public view returns (bool) {
146 return zap_tokens_index[_zapId] > 0;
147 }
148
149 //getZap now just returns the hash of the Zap,
150 //if the Zap exists. Replaces getZapHash(uint256 _zapId).
151 function getZap(uint256 _zapId) private view returns (
152 bytes32 _metadatahash
153 ) {
154 require(exists(_zapId), "Token does not exist.");
155 _metadatahash = zap_tokens[zap_tokens_index[_zapId]].metadatahash;
156 }
157
158 function modifyZapHash(
```

```
159     uint256 _zapId, bytes32 _newhash) public {
160 require(balanceOf(msg.sender, _zapId) == 1,
161     "Access to one or more listed tokens not allowed.");
162 zap_tokens[zap_tokens_index[_zapId]].metadatahash = _newhash;
163 }
164
165 function safeZapTransferFrom(
166     address from, address to, uint256 tokenId,
167     bytes memory json, string memory postalcode) public{
168
169 bytes memory postal = bytes(postalcode);
170
171 uint addressposition = findFirst(json, "]", 0);
172 //second param takes length of substring,
173 //NOT THE END CHARACTER POSITION
174 bytes memory a = json.slice(0, addressposition-1);
175 bytes memory b = json.slice(addressposition-1,
176     json.length-addressposition+1);
177 bytes memory c = a.concat(abi.encodePacked("\",
178     \"", postal)).concat(b);
179
180 addressposition = findFirst(c, "]", 0);
181 addressposition = findFirst(c, "]", addressposition+1);
182 a = c.slice(0, addressposition-1);
183 //+5 because of the quotes, the comma,
184 //the space, the quotes and the 0-indexing
185 b = c.slice(addressposition-1,
186     json.length-addressposition+5+postal.length);
187 c = a.concat(
188     abi.encodePacked("\", \"", "0x", toAsciiString(to))).concat(b);
189 modifyZapHash(tokenId, keccak256(c));
```

```
190 emit NewMetaInHex(c);
191
192 super.safeTransferFrom(from, to, tokenId, 1, '');
193 }
194
195 function safeBulkZapTransferFrom(
196     address from, address to, uint256[] memory tokenIds,
197     uint256[] memory amounts, bytes[] memory jsonarray,
198     string[] memory postals) public{
199 require(tokenIds.length == amounts.length &&
200     amounts.length == jsonarray.length &&
201     jsonarray.length == postals.length,
202     "List sizes for bulk transfer do not match");
203 bytes memory postal = '';
204 uint i = 0;
205 uint addressposition = 0;
206 bytes memory a = '';
207 bytes memory b = '';
208 bytes memory c = '';
209
210 while(i<jsonarray.length){
211
212 postal = bytes(postals[i]);
213 addressposition = findFirst(jsonarray[i], "]", 0);
214 a = jsonarray[i].slice(0, addressposition-1);
215 b = jsonarray[i].slice(addressposition-1,
216     jsonarray[i].length-addressposition+1);
217 c = a.concat(abi.encodePacked("\", \"", postals[i])).concat(b);
218
219 addressposition = findFirst(c, "]", 0);
220 addressposition = findFirst(c, "]", addressposition+1);
```

```
221 a = c.slice(0, addressposition-1);
222 //+5 because of the quotes, the comma,
223 //the space, the quotes and the 0-indexing
224 b = c.slice(addressposition-1,
225     jsonarray[i].length-addressposition+5+bytes(postals[i]).length);
226 c = a.concat(abi.encodePacked("\", \"",
227     "0x", toAsciiString(to))).concat(b);
228 modifyZapHash(tokenIds[i], keccak256(c));
229 emit NewMetaInHex(c);
230 i++;
231 }
232 super.safeBatchTransferFrom(from, to, tokenIds, amounts, '');
233 }
234 }
```

BIBLIOGRAPHY

Agung, A. A. G. & Handayani, R. (2020). Blockchain for smart grid. *Journal of King Saud University - Computer and Information Sciences*. doi: https://doi.org/10.1016/j.jksuci.2020.01.002.

Andoni, M., Robu, V., Flynn, D., Abram, S., Geach, D., Jenkins, D. P., McCallum, P. & Peacock, A. (2019). Blockchain technology in the energy sector: A systematic review of challenges and opportunities. *Renewable and Sustainable Energy Reviews*, 100, 143-174. Consulted at https://researchportal.hw.ac.uk/en/publications/blockchain-technology-in-the-energy-sector-a-systematic-review-of.

Authority, F. C. (2019, October, 2). Cryptoassets: our work [HTML/web-page]. Consulted at https://www.fca.org.uk/firms/cryptoassets.

Cashore, B. & Gale, F. (2006). Introduction: Forest Certification in Analytical and Historical Perspective. *Confronting Sustainability: Forest Certification in Developing and Transitioning Countries*.

Council, F. S. (2016). *Chain of Custody Certification: FSC-STD-40-004 V3-0* (ed. 3). Adenauerallee 134, 53113, Bonn, Germany: Forest Stewardship Council.

Cueva-Sánchez, J. J., Coyco-Ordemar, A. J. & Ugarte, W. (2020). A blockchain-based technological solution to ensure data transparency of the wood supply chain. *2020 IEEE ANDESCON*, pp. 1-6. doi: 10.1109/ANDESCON50619.2020.9272176.

D'Amours, S., Ouhimmou, M., Audy, J.-F. & Feng, Y. (2017). *Forest value chain optimization and sustainability* (ed. 1). 6255 Cantay Road, L5R 3Z4, Mississauga, ON, Canada: CRC Press.

Düdder, B. & Ross, O. (2020). Timber Tracking: Reducing Complexity of Due Diligence by using Blockchain Technology (Position Paper). *Department of Computer Science, University of Copenhagen*.

Entriken, W., Shirley, D., Evans, J. & Sachs, N. (2018, June, 2). EIP 721: ERC-721 Non-Fungible Token Standard [HTML/web-page]. Consulted at https://eips.ethereum.org/EIPS/eip-721.

Etherscan. (2020a, June, 2). Non-Fungible (ERC-721) Tokens (NFT) Token Tracker | Etherscan [HTML/web-page]. Consulted at https://etherscan.io/tokens-nft.

Etherscan. (2020b, June, 2). Token Tracker | Etherscan [HTML/web-page]. Consulted at https://etherscan.io/tokens.

Figorilli, S., Antonucci, F., Costa, C., Pallottino, F., Raso, L., Castiglione, M. & Pinci, E. (2018). A blockchain implementation prototype for the electronic open source traceability of wood along the whole supply chain. *Sensors*, 18(9).

FSC. (2020a, July, 2). Chain of Custody Certification [HTML/web-page]. Consulted at https://ca.fsc.org/en-ca/certification/chain-of-custody-certification.

FSC. (2020b, July, 2). Facts and Figures [HTML/web-page]. Consulted at https://us.fsc.org/en-us/what-we-do/facts-figures.

FSC. (2020c, February, 2). FSC Labels [HTML/web-page]. Consulted at https://fsc.org/en/fsc-labels.

Gallersdörfer, U. & Matthes, F. (2018). Tamper-Proof Volume Tracking in Supply Chains with Smart Contracts. *International European Conference on Parallel and Distributed Computing*.

Gan, J., Cerutti, P., Masiero, M., Pettenella, D., Andrighetto, N. & Dawson, T. (2016). In Kleinschmit, D., Mansourian, S., Wildburger, C. & Purret, A. (Eds.), *Quantifying Illegal Logging and Related Timber Trade* (ed. 1, vol. 35, ch. 3, pp. 37-60). Vienna: International Union of Forest Research Organizations (IUFRO).

Ganache. (2020, June, 2). Ganache | Ganache Settings | Documentation | Truffle Suite [HTML/web-page]. Consulted at https://www.trufflesuite.com/docs/ganache/reference/ganache-settings.

Hassan, N. U., Yuen, C. & Niyato, D. (2019a). Blockchain Technologies for Smart Energy Systems: Fundamentals, Challenges, and Solutions. *IEEE Industrial Electronics Magazine*, 13(4), 106-118. doi: 10.1109/MIE.2019.2940335.

Hassan, N. U., Yuen, C. & Niyato, D. (2019b). Blockchain Technologies for Smart Energy Systems: Fundamentals, Challenges, and Solutions. *IEEE Industrial Electronics Magazine*, 13(4), 106-118.

Helo, P. & Hao, Y. (2019). Blockchains in operations and supply chains: a model and reference implementation. *Computers and Industrial Engineering*.

Howson, P., Oakes, S., Baynham-Herd, Z. & Swords, J. (2019). Cryptocarbon: The promises and pitfalls of forest protection on a blockchain. *Geoforum*, 100, 1-9. doi: https://doi.org/10.1016/j.geoforum.2019.02.011.

IBM. (2020, February, 2). About IBM Food Trust [HTML/web-page]. Consulted at https://www.ibm.com/downloads/cas/8QABQBDR.

Kuzlu, M., Sarp, S., Pipattanasomporn, M. & Cali, U. (2020). Realizing the Potential of Blockchain Technology in Smart Grid Applications. *2020 IEEE Power and Energy Society Innovative Smart Grid Technologies Conference (ISGT)*. doi: 10.1109/ISGT45199.2020.9087677.

Lee, H., Padmanabhan, V. & Whang, S. (2004). Information Distortion in a Supply Chain: The Bullwhip Effect. *Management Science*, 43, 546-558. doi: 10.1287/mnsc.1040.0266.

Mollah, M. B., Zhao, J., Niyato, D., Guan, Y. L., Yuen, C., Sun, S., Lam, K.-Y. & Koh, L. H. (2021). Blockchain for the Internet of Vehicles Towards Intelligent Transportation Systems: A Survey. *Internet of Things Journal IEEE*, 8(6), pp. 4157-4185.

Musleh, A. S., Yao, G. & Muyeen, S. M. (2019). Blockchain Applications in Smart Grid–Review and Frameworks. *IEEE Access*, 7, 86746-86757. doi: 10.1109/ACCESS.2019.2920682.

Mylrea, M. & Gourisetti, S. N. G. (2017). Blockchain for smart grid resilience: Exchanging distributed energy at speed, scale and security. *2017 Resilience Week (RWS)*, pp. 18-23. doi: 10.1109/RWEEK.2017.8088642.

Nikolakis, W., John, L. & Krishnan, H. (2018). How Blockchain Can Shape Sustainable Global Value Chains: An Evidence, Verifiability, and Enforceability (EVE) Framework. *Multidisciplinary Digital Publishing Institute*.

OpenZeppelin. (2020a, February, 2). ERC 1155 [HTML/web-page]. Consulted at https://docs.openzeppelin.com/contracts/3.x/api/token/erc1155.

OpenZeppelin. (2020b, February, 2). ERC 721 [HTML/web-page]. Consulted at https://docs.openzeppelin.com/contracts/2.x/api/token/erc721.

Radomski, W., Cooke, A., Castonguay, P., Therien, J., Binet, E. & Sandford, R. (2018, June, 2). EIP 1155: ERC-1155 Multi Token Standard [HTML/web-page]. Consulted at https://eips.ethereum.org/EIPS/eip-1155.

Rafael, G. C., Fonseca, A. & Jacovine, L. A. G. (2018). Non-conformities to the Forest Stewardship Council (FSC) standards: Empirical evidence and implications for policy-making in Brazil. *Forest Policy and Economics*, (88).

Shahi, S. K. (2016). Supply chain management of the Canadian Forest Products industry under supply and demand uncertainties: a simulation-based optimization approach.

Sheng, S. W. & Wicha, S. (2021). The Proposed of a Smart Traceability System for Teak Supply Chain Based on Blockchain Technology. *2021 Joint International Conference on Digital Arts, Media and Technology with ECTI Northern Section Conference on Electrical, Electronics, Computer and Telecommunication Engineering*, pp. 59-64. doi: 10.1109/ECTI-

DAMTNCON51128.2021.9425780.

Station, E. G. (2020, June, 2). Consumer oriented metrics for the Ethereum gas market [HTML/web-page]. Consulted at https://ethgasstation.info/index.php.

Sugiura, K. & Oki, Y. (2018). Reasons for choosing Forest Stewardship Council (FSC) and Sustainable Green Ecosystem Council (SGEC) schemes and the effects of certification acquisition by forestry enterprises in Japan. *Forests*, 9(4).

Sá, G. (2021, March, 18). Solidity Bytes Arrays Utils [HTML/web-page]. Consulted at https://github.com/GNSPS/solidity-bytes-utils.

Vilkov, A. & Tian, G. (2019). Blockchain as a solution to the problem of illegal timber trade between Russia and China: SWOT analysis. *International Forestry Review*.

Westerkamp, M., Victor, F. & Küpper, A. (2020). Tracing manufacturing processes using blockchain-based token compositions. *Digital Communications and Networks*, 6(2).

Wood, G. (2020). Ethereum: A secure decentralised generalised transaction ledger. Consulted at https://ethereum.github.io/yellowpaper/paper.pdf.