

BENG INDIVIDUAL PROJECT

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

---

## Decentralized Financial Agents: From Peer-to-Peer Negotiation to Trustless Contract Execution

---

*Author:*  
Paris Mongkolkul

*Supervisor:*  
Dominik Harz  
Emil Lupu

June 26, 2020

## Abstract

In traditional finance, central authorities such as governments have issued the currencies that underpin our economics. This leads to the expectations of banks and other centralized institutions to manage and regulate the supply of currency in a safe manner. As the size and complexity of our economics grew, these central authorities gained more power as more trust must be placed in them. Subsequently, end users have very little to say about how their asset is being handled and simply lose control of their money. With this long-lasting problem comes Decentralized Finance. Decentralized Finance aims to create a system that is open to everyone while minimizing their need to trust and rely on central authorities.

With almost \$490 million USD locked inside its vault alone, The Maker protocol indicates a future for how distributed ledgers technology are being adopted into the world of finance. Maker, along with other Decentralized Finance projects, have utilized the capabilities of conditional transactions and smart contracts, thus allowing user's investment via their Ether currency. Although DeFi protocols promise a cheaper and more transparent transaction, they are considerably complex. So if we can make agents available for users, it will be easier to participate quickly in such investment.

With this in mind, a system of autonomous agents can be implemented to allow multiple users to collaborate and pool their resources together to maximize the outcome of their investment. We provide an overview of how the system is built in two parts. First, we implement how two agents can strictly communicate with each other and arrive at a mutual conclusion by designing an Agreement Calculus based on a process calculus. Agents should be able to discover each other, share their interests, and argue about the agreement they want to make, before encoding the agreement into a contract that does not require any central party to be enforced. Libp2p is used for the communication layer with the Contract Net Protocol as the communication protocol. Second, we implement the agreed smart contract on Ethereum in Solidity. The section covers the implementation and deployment of the contract along with how it interacts with Maker protocols. The project has achieved the negotiation of agents, the encoding of agreed terms into a smart contract, and the execution of those encoded terms without trusted third parties. Hence, our agents provide a new opportunity to bring a more resilient and transparent investment strategy to the world of finance.

### **Acknowledgements**

I would like to sincerely thank my supervisor, Dominik Harz, for offering continuous support, and invaluable feedback, even on the day before this paper is submitted. Without your kind guidance and weekly motivation, the project would, without a doubt be impossible.

I would like to thank Emil Lupu for the productive encouragement given in the interim meeting. It has always been the drive for me to keep a high work standard and the reminder for me to keep on pushing.

I would like to thank my family and friends for always taking care of me and carrying me through my clueless years at Imperial. This includes my Thai Society family for bringing the joy and making the years seem shorter.

Lastly, I would like to thank Prim for the love and for believing in me. You already know this.

I also would like to thank 4 for always being there for me. May you rest well in heaven.

# Contents

<b>1</b>	<b>Introduction</b>	<b>6</b>
1.1	Overview	6
1.2	Limitations	7
1.3	Objective	7
1.4	Contribution	7
<b>2</b>	<b>Background</b>	<b>9</b>
2.1	Multi-agent Systems	9
2.1.1	Overview	9
2.1.2	Intelligent Agents	9
2.1.3	Contract Net Protocol	10
2.1.4	FIPA protocols	12
2.2	Communication	12
2.2.1	IPFS	12
2.2.2	P2P	12
2.2.3	Libp2p	12
2.3	Agreements	15
2.3.1	Bitcoin and the rise of Ethereum	15
2.3.2	Smart Contracts	15
2.3.3	Tokens	15
2.3.4	Solidity	16
2.4	Dai Ecosystem	16
2.4.1	DApps and DeFi	16
2.4.2	MakerDAO	16
2.4.3	Dai	17
2.4.4	Collateralized Debt Position	17
2.4.5	MKR	17
2.4.6	Single-Collateral and Multi-Collateral Dai	18
<b>3</b>	<b>Design</b>	<b>19</b>
3.1	Agreement Calculus	19
3.1.1	Contract Advertisement	19
3.1.2	Contract Preconditions	19
3.1.3	Contracts	20
3.1.4	An example of the Calculus	21
3.2	Agent Negotiation Protocol	21
<b>4</b>	<b>Implementation</b>	<b>24</b>
4.1	Negotiation Protocol	24
4.2	Smart Contracts	27
4.2.1	DSPProxy	29
4.2.2	Interacting with Maker's Contracts	31
<b>5</b>	<b>Evaluation</b>	<b>34</b>
5.1	Applicability of Agreement Calculus	34
5.2	Negotiation Protocol	34
5.3	Testings	35

5.4	Challenges . . . . .	36
<b>6</b>	<b>Conclusion</b>	<b>41</b>
6.1	Future Work . . . . .	41
6.1.1	Integration with the Risk Modelling Project . . . . .	41
6.1.2	Agreement Compiler . . . . .	41
6.1.3	Contract Verification . . . . .	42
6.1.4	Integration with other DeFi protocols . . . . .	42
6.1.5	Agents . . . . .	42
6.1.6	Deployment to Mainnet . . . . .	42
<b>A</b>	<b>Online resources</b>	<b>43</b>
<b>B</b>	<b>Community Help</b>	<b>44</b>

# List of Figures

1.1	Project overview . . . . .	7
2.1	An agent in its environment [Russle and Norvig, 1995, p.32]. . . . .	10
2.2	AUML formalism of the Contract Net Protocol. . . . .	11
3.1	Interaction Flow . . . . .	22
4.1	Architecture . . . . .	29
4.2	Proxy architecture . . . . .	30
4.3	Context in delegatecall . . . . .	30
5.1	the Verified Contract on Etherscan . . . . .	39
5.2	The Transaction's Stack Trace from Tenderly . . . . .	40
B.1	DSPProxy Enquiry via chat.makerdao.com . . . . .	44
B.2	Custom Contract Interaction via Maker's Developers forum . . . . .	45
B.3	Forking off mainnet via StudyDeFi's Discord . . . . .	46
B.4	Libp2p's communication via SimpleAsWater's Discord . . . . .	46

# Listings

4.1	Some Bootstrapper PeerIds . . . . .	24
4.2	Peer Discovery Configuration . . . . .	25
4.3	Communication Encryption Configuration . . . . .	25
4.4	Dialer Peer . . . . .	26
4.5	Listener Peer . . . . .	26
4.6	Calculus Handler . . . . .	27
4.7	OpenAndLockETH function . . . . .	31
4.8	Contract Declaration and Agent Structure . . . . .	32
4.9	Deploying MyCustomVaultManager and DSPProxy . . . . .	32
4.10	Maker's Contract Addresses Release 1.0.7 (Wednesday, 03.06.2020) . . . . .	33
5.1	Error from Truffle . . . . .	35
5.2	Proxy Creation Test . . . . .	35
5.3	frob function in Vat.sol . . . . .	36
5.4	Error message from calling frob . . . . .	37
5.5	Error message from Etherscan's contract verification . . . . .	38

# Chapter 1

## Introduction

### 1.1 Overview

With the arrival of blockchain, today's digital economy is being transformed to be a more collaborative and transparent one. Blockchain, the technology at the heart of Bitcoin and other crypto currencies, promises a decentralized system that exists between all permitted parties. In 1994, Nick Szabo [1] realized that the decentralized property of a distributed ledger could be extended to conditional transactions or, as more commonly known today, smart contracts. Smart contracts are programs stored in a distributed blockchain network that can be invoked by credible transactions and allow the exchange of currency, property, shares, or anything of value in a transparent, conflict-free way. With this technology to collectively build and control a system without the need for trusted third parties (TTP), Decentralized Finance (DeFi) projects have swiftly emerged and changed the way we view ownership and material possessions. DeFi aims to give users non-custodial control over their digital assets, thus minimizing the risk one has to take compared to traditional, centralized finance. This is due to the advantages of decentralization e.g. anyone with internet can verify any and every transaction that occurs on the blockchain. Having attracted the attention of investors and developers rapidly, The total value locked in DeFi has massively increased by over 370% in the last year with the current value of \$1.12 billion USD.

DeFi protocols allow all sorts of agents to participate in financial products, including agents with *weak identities* [2]. For example, an agent can take a loan without the need to go through an identification procedure. To safeguard those loans, agents with weak identities have to provide financial collateral to take the loan. This principle allows the creation of different protocols including margin trading (dYdX), borrowing and lending protocols (Compound), and decentralized token exchanges (Uniswap). However, these protocols are dynamic: the potential returns change through exogenous influences (eg. price changes, actions of other agents), and endogenous influences (updates in the protocol through governance mechanisms or the operators of the protocol). With these varying instantaneous changes, humans might be too slow to react and adapt to the different scenario. A supportive system of autonomous agents could surely navigate the complexities of the protocols.

Furthermore, the agents hold private information used by the protocols to invest in, meaning that their actions can have an influence on other agent strategies [3]. If agents collaborate together, they can develop combined strategies which would even further maximize their portfolio. Nonetheless, agents could still change their mind or try to trick other agents into certain behaviour to gain individual advantages. These cheating behaviours must be prevented in a way that the agents do not need to trust each other. In addition, these agents would need to operate autonomously in a decentralized environment and enforce negotiated terms with other entities. Such agents can take the initiative and trade in a logical manner without the need to interact with them. With the advent of DeFi projects and advances in smart contracts technology, there seems to be an opportunity to introduce this system into today's digital economy.



## 1.2 Limitations

As DeFi protocols have different inherent risks, fluctuating payments, and diverging deposit requirements, the task of selecting the "best" protocol is not trivial. This would consist of constructing a model where the trade-off between risk and return for the user's portfolio is maximized by mathematical methods and machine learning solutions. This problem of finding optimal investment strategies for a single agent is the objective of another project, **not** the Peer-to-peer Negotiation project, and will not be covered here.

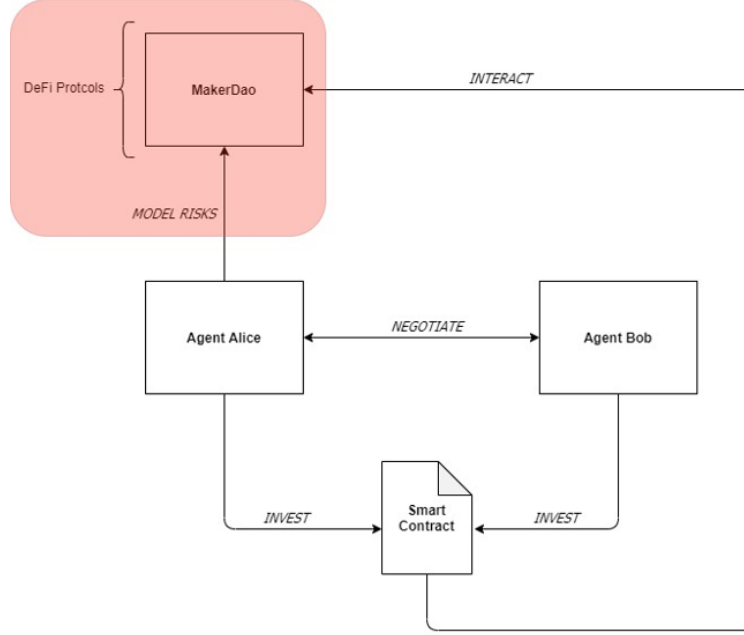


Figure 1.1: Project overview

As the Peer-to-peer Negotiation project is part of a larger DeFi agents project, We assume that the other project covers aspects such as in which protocols should the agents invest in, which currencies, and how much. The assumption is used as an input to the negotiation protocol. The outcome of the negotiation is a smart contract agreed by the negotiating parties. As seen in Figure 1.1, the assumption is highlighted in red while the Peer-to-peer Negotiation project covers the rest of the diagram.

## 1.3 Objective

The goal of the project is to develop a coordination system for autonomous agents in an open-world peer-to-peer setting to enable them to collaboratively invest in so-called DeFi protocols. Agents should be able to communicate directly with each other to negotiate the terms of the common agreement. For example, agents can decide how much currency to invest in a given DeFi protocol, for which amount of time, and the conditions to stop the investment.

The coordination system should allow multiple agents to:

1. negotiate the terms of their collaboration,
2. encode the agreed terms into a verifiable contract,
3. and execute the encoded terms without the need to trust any one agent in the system.

## 1.4 Contribution

The focus of the project is on the trustless encoding of the agreed intents between the agents. This is also the technical challenging aspect of the project as the contract terms have to be made

verifiable and that the smart contracts must be encoded in such a way that agents could understand its content. Along with this, the autonomous system agents will be the first to be deployed to the real world as none exist as of now. The logical system could arguably change how investors trade and impact the future of DeFi market as a whole.

1. **Negotiation Protocol**The negotiation of the terms is achieved through a modification of the Contract Net Protocol. Each agent is able negotiate terms to other agents through a decentralized messaging system implemented through libp2p. The basic actions of our negotiation protocol are to propose new terms, update a proposition, and accept or reject a proposition.
2. **Agreement Calculus** We propose an encoding format for the terms of an Agreement Calculus. Our format includes the amount, currency, time and exit criteria. The design is based on the BitML paper [4].
3. **Contract Encoding**We transform the encoded terms into a smart contract written in Solidity. Solidity is the smart contract language of the Ethereum platform. Encoding the terms into a smart contract allows us to deploy the contract to a blockchain, where the content of the contract are immutable. We provide a code template that translates our encoding format into the Solidity smart contract.
4. **Contract Execution**The contract includes any logic to execute the terms. For example, if two agents agreed to invest a certain amount of ETH into Maker and exit the protocol at a certain time, this would be encoded in the contract. By invoking a call to the contract the terms are fulfilled. Any agent is able to call the contract whereas no agent is able to change the logic of the contract once it is deployed on chain.

## Chapter 2

# Background

The project relies on two primary areas of research.

The study of agent negotiation concerns how agents can interact to reach an agreement. Agent negotiation is a subset of agent systems and lies within the field of distributed artificial intelligence. The focus of the research is enforcing how negotiations are carried out and the result of each investments, i.e. the agreed terms. Furthermore, studies of possible communication methods between two agents will be investigated as part of the negotiation process.

The second area concerns smart contracts and distributed ledger technology, which is used to eliminate the possibility of agents breaking their promise and acting in their own interest. The research focuses on Ethereum and decentralized finance projects, specifically MakerDAO. Decentralized finance is a subset of the wide-spread decentralized applications run on the Ethereum Virtual Machine. Tools for developing smart contracts are also discussed.

## 2.1 Multi-agent Systems

### 2.1.1 Overview

A multi-agent system (MAS) is a system composed of multiple interacting intelligent entities, known as agents. Agents are computer systems with two important capabilities. First, they are at least to some extent capable of autonomous action – of making the decision for what they need to do in order to satisfy their design objectives. Second, they are able to interact with other agents – not simply by exchanging data, but by engaging in analogues of the kind of social activities e.g. cooperation, coordination, negotiation.

Multi-agent systems have been studied as a field in their own right and has gained widespread recognition especially in the mid-1990s, when the Internet was becoming mainstream. Although they will undoubtedly play a pivotal role in exploiting the potential of the Internet and other distributed systems, there is a lot more to multi-agent systems than this. With the rapid growth of artificial intelligence and machine learning today, Multi-agent systems seem to be a natural foundation for understanding and building a wide range of artificial social systems [5].

The importance of negotiation in MAS has been addressed in [6] and [7]. The authors discuss different approaches to achieve this specific architecture for agents to cooperate with one another, and that some domains simply require the technology – in particular, when different organizations with different, possibly conflicting, goals and proprietary information. Even if each organization wants to model its internal affairs with a single system, the organizations will not give authority to any single party to build a system that represents them all. Different entities will need their own systems that reflect their capabilities and priorities.

### 2.1.2 Intelligent Agents

Expanding the term agent in section 2.1.1, an agent is a computer system that is situated in some environment, and that is capable of autonomous action in the environment in order to meet its

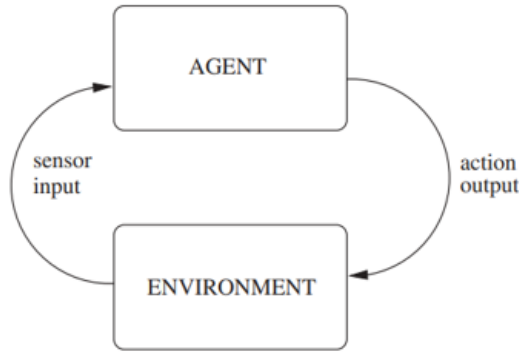


Figure 2.1: An agent in its environment [Russle and Norvig, 1995, p.32].

design objectives. As seen in figure 2.1, the agent takes sensory input from the environment, and produces an output actions that affect it. The interaction is usually an ongoing, non-terminating one. In most domains, an agent will not have complete control over its environment, but partial control – that it can influence the environment. This is the case for DeFi projects since an agent will need to deposit or withdraw a surprisingly enormous amount of asset to fluctuate the currency price.

The following is some properties of agents [8]:

- **Autonomy.** An agent possesses individual goals, resources, and competences. Without direct human or other intervention, it must have some degree of adaptability and control over its actions and internal states, i.e. how to invest and the value held.
- **Sociability.** An agent can interact with other agents via some kind of *agent communication language*. This allows agents to ask and provide help as well as refusing an action.
- **Reactivity.** An agent perceives and acts in a timely fashion according to changes that occur in the close environment.
- **Pro-activeness.** An agent may be able to exhibit goal-directed behavior by taking the initiative to invoke other agents without having to wait for a stimulation.

As a computational paradigm, multi-agent systems can be used to solve problems which are difficult or impossible for an individual agent or monolithic system to solve. Hence, multi-agent systems seem to offer a promising solution to the DeFi agents negotiation project. With the objective to gain the most possible profit through investments, agents have to find common interests and potentially pool resources to achieve a better outcome. This is not necessarily bilateral, i.e. one-to-one, since an agent is not limited to an interaction with only one other agent. For this scenario, a multilateral negotiation protocol is needed [9]. Multilateral protocols allow one-to-many and many-many negotiations. One of these is the contract net protocol.

### 2.1.3 Contract Net Protocol

The contract net protocol (CNP) is a many-to-many negotiation protocol that allows agent to find negotiation partners and conduct negotiations with them.

In [10], the authors developed the contract net protocol using a distributed problem solver. A distributed problem solver is comprised of “nodes” that can be viewed as agents. Each agent has a set of tasks to accomplish and is limited in terms of its capabilities for achieving the task. Due to this limitation, an agent must find other agents in the system to accomplish its tasks. Like our investment scenario, the agents are peers; so their objectives never conflict one another. Hence, their objectives must be mutually agreed upon.

Each agent in the contract net protocol takes on one of two roles: *manager* or *contractor*. A manger monitors the execution of a task, while a contractor is responsible for the actual execution of the task. Any node can take on either role dynamically during the course of the negotiation.

The following is an overview of the key stages in the process of negotiation using the contract net protocol [9]:

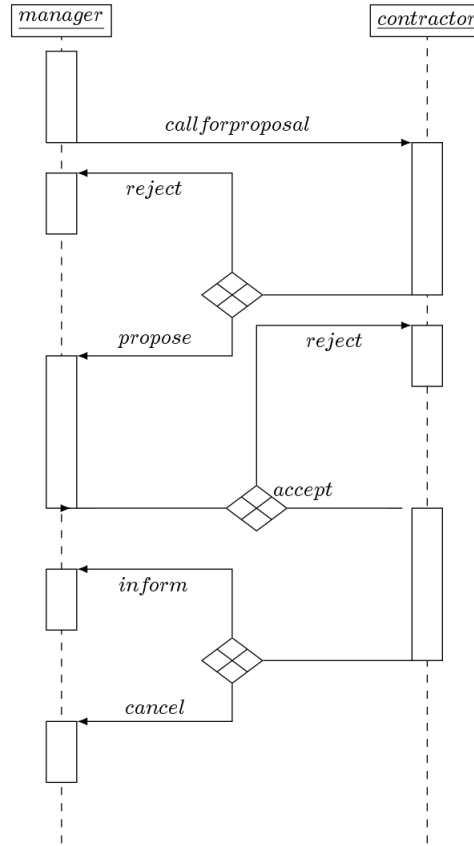


Figure 2.2: AUML formalism of the Contract Net Protocol.

1. **Recognition and announcement.** The manager starts of the negotiation process by recognizing the need to find a contractor to execute their task. The manager then announces an advertisement for the task. In the absence of information about the other agent's capabilities, A broadcast is necessary to reach every node. If this information is available, the *task announcement* can be directed to relevant agents instead.
2. **Bidding.** The agents listen to announcements and evaluate whether their capabilities fit the requirements. If a suitable announcement is picked up, the agent sends a *proposal* message to the manager.
3. **Bid selection and awarding.** The manager receives proposals for an announcement and chooses among them the one that suits it best. The manager then allocates the task to the agent that submitted the chosen proposal by sending an *accept* message. The agent that receives this message becomes the contractor for the task. The manager also sends *reject* messages to other agents to inform them of the decision.
4. **Reporting results.** When a contractor accomplishes an awarded task, an *inform* message is sent to the manager. If the contractor cannot fulfill its engagement, it notifies the manager through a *cancel* message.

In many situations, the agents may be self-interested. In this case, an agent must convince another to do agree to the agent's proposal. Furthermore, if a potential contractor can execute tasks with different levels of effort, the corresponding manager then needs a method to make sure that the contractor puts in a desired effort. The issue of incentive contracting has been a popular research area for game theorists [11]. Since our investment agents act solely in their individual interest, we will have to consider when the possibility of this.

### 2.1.4 FIPA protocols

The Foundation for Intelligent Physical Agents (FIPA) is a body for developing and setting computer software standards for heterogeneous and interacting agents and agent-based systems. The most widely adopted of the FIPA standards is the Agent Communication Language (ACL) which acts as a gateway of communication between different types of agents regardless of the platform used to create them [12]. This ACL defines an ‘template’ language for messages including 20 *performatives* e.g. inform, agree, request. The syntax aims to define the intended interpretation of messages without mandating any specific language for message content.

One issue that has emerged out of FIPA-ALC concerns ontologies. If two agents are to communicate about some domain, then it is necessary for them to agree on the terminology that they use to describe this domain. For example, an agent is buying a cup of coffee from another agent: the buyer needs to unambiguously specify to the seller the properties of the item, such as its size and type. The agents need to agree both on what ‘size’ means, and also what terms like ‘ounces’ or ‘cappuccino’ mean. An ontology is thus a formal definition of a body of knowledge that involves a structural component. In an attempt to solve this problem, work has begun on several languages and tools – notably the Darpa Agent Markup Language (DAML) [13] which is based on XML. However, these protocols have not really taken off in terms of real life adoption.

## 2.2 Communication

### 2.2.1 IPFS

In the 1960s, most of the communication infrastructure consists of human operators connecting wires to facilitate country-wide communication. Fast-forward to today’s world, most of the internet organizations have centralized servers hosted in data centers. Hence, the internet today is still full of services similar to the bank of human operators which are highly centralized and fragile, e.g. any technical or natural disaster case significant damage. The problem mainly comes from the network design problems, most notably the location-addressing and client-server model.

Inter Planetary File System (IPFS), an open-source project, is introduced in 2015 to address the deficiencies of the client-server model. IPFS acts as a protocol and peer-to-peer network storing and sharing data in a distributed file system. Built around a decentralized system, IPFS uses content-addressing to uniquely identify each file in a global namespace connecting all computing devices. This is done with the study and implementation of distributed hash tables, block exchanges, and Merkle DAG. However, the key technology in IPFS is the peer-to-peer networking fundamental.

### 2.2.2 P2P

A peer-to-peer (P2P) network is one in which nodes, or peers, in the system communicate with one another directly. Even though, this does not necessarily mean that all nodes are identical, they are assumed to be in our investment agents’ scenario. P2P network does not require a privileged set of “clients” and “servers”, breaking away from the client-server paradigm. Because the definition of P2P is not specific, many different kinds of systems have been built and fall under the umbrella of “peer-to-peer”. The most prominent examples are the file sharing networks like bitorrent, and, more recently, the proliferation of blockchain networks that communicate in a peer-to-peer fashion.

### 2.2.3 Libp2p

Libp2p is a modular system of protocols, specifications and libraries that enable the development of peer-to-peer network applications. It strangely emerges in the process of overcoming P2P challenges while building IPFS. Having the original aim to be IPFS, libp2p does not require or depend on IPFS.

The foundation of libp2p is the transport layer, which is responsible for the actual transmission and receipt of data from one peer to another. Libp2p provides a simple interface that can be adapted to support existing and future protocols, allowing libp2p applications to operate in many different runtime and networking environments. The security aspect of this medium is also addressed by using public key cryptography as the basis of peer identity. This gives each peer a globally unique

'name', or *PeerId*, which allows anyone to retrieve the public key for the identified peer. The communication between peers thus become more secure this way - i.e. no third-party can read conversation or alter it mid-flight.

Like IPFS, Libp2p provides a content routing interface. This allows contents to be passed around without caring who sent them since the system can verify its integrity. As address of a peer is not used to track where they are, a way to locate them on the network is needed. This is solved by peer routing which is the process of discovering peer addresses by leveraging the knowledge of other peers through *PeerId*. For example, if we want to send a message to Alice, we could contact a random peer, Bob. Bob can either give us the address we need if they have it, or else send our inquiry to another peer who is more likely to have the answer. As we contact more peers, we not only increase our chances of finding Alice, we build more complete view of the network in our routing tables, which enables us to answer routing queries from others. These routing tables are distributed hash tables and the current implementation uses a Kademlia routing algorithm [14].

**Transport.** When a connection is made from your computer to another machine on the internet, it is most likely that TCP/IP (Transmission Control Protocol) is used to handle addressing and delivery of data packets. In some cases, a TCP adds too much overhead, so applications might use alternatives such as UDP, a much simpler protocol with no guarantees about reliability or ordering of packets. In libp2p, these foundational protocols are known as transports and being transport agnostic is one of libp2p's core requirements. This simply means that developers get to decide which transport protocol to use and how many, since one application can support many different transports at the same time.

**Listening and Dialing.** Transports consist of two core operations, *listening* and *dialing*. A listener accepts incoming connections from other peers, using whatever facility is provided by the transport implementation. A dialer opens an outgoing connection to a listening peer. When listening, the listener gives the transport the address they would like to listen on, and when dialing, the dialer provides the address to dial to. These addresses are known as multiaddresses.

**Addresses.** In order to dial a peer and open a connection, Libp2p uses multiaddress to find out how to reach them across a wide variety of networks. Multiaddress is a convention for encoding multiple layers of addressing information into a single path structure. It takes into account human-readability and machine-obtainability of common transport and overlay protocols that allow many layers of addressing scheme to be combined and used together.

The following is an example of a multiaddress for a TCP/IP transport:

`/ip4/7.7.7.7/udp/6543`

Although, this is equivalent to the more familiar 7.7.7.7:6542 convention, it has the advantage of being explicit about the protocols that are being described. One could easily observe that the 7.7.7.7 address belongs to the IPv4 protocol, and that we want to send UDP packets to port 6543.

In addition, the multiaddress can include the *PeerId* of the destination peer when dialing a remote peer. This allows libp2p to establish a secure communication channel and prevents impersonation through public key verification.

An example multiaddress that includes a *PeerId*:

`/ip4/1.2.3.4/tcp/4321/p2p/QmcEPrat8ShnCph8WjkREzt5CPXF2RwhYxYBALDcLC1iV6`

The `/p2p/QmcEPrat8ShnCph8WjkREzt5CPXF2RwhYxYBALDcLC1iV6` component uniquely identifies the remote peer using the hash of its public key. Combining "location multiaddress" (IP and port) with "identity multiaddress" (*PeerId*), produces a new multiaddress containing both key pieces of information. If peer routing is enabled, only the presence of *PeerId* is suffice, without the need to know the peer's transport addresses.

Libp2p applications also support multiple transports at once, however since the investment agents run their service as a long-running daemon processes via TCP, this functionality is not needed.

**Nat Traversal** The internet is composed of countless networks, bound together into shared address spaces by transport protocols. As traffic moves between network boundaries, Network Address

Translation (NAT) maps an address from one address space to another. Since libp2p applications should be able to run everywhere, not just in data centers or on machines with stable public IP addresses, there are many approaches to NAT traversal available to enable this. [STUN]

One of libp2p's core protocols is the identify protocol. The protocol allows peers to inform each other about their observed network addresses while requesting some identifying information. When sending over their public key and other useful data, the peer being identified includes the set of addresses that it has observed for the peer asking the question. This discovery mechanism serves the same role as STUN, but without the need for a set of STUN servers. Without this identify protocol, communication across NATs would be impenetrable.

Since not all networks allow incoming connections on the same port used for dialing out, peers can attempt to dial other peers at their observed addresses. For example, Alice dials Bob at his address. If this succeeds, Bob can rely on other peers being able to dial Bob as well and can start advertising his listening address. This service is available through a libp2p protocol called *AutoNAT*, which lets peers request dial-backs from peers.

**Circuit Relay** In many cases, peers will not be able to communicate without a secondary protocol. The problems could occur from peers being unable to transverse their NAT in a way that makes them publicly accessible, or not sharing common transport protocols that would allow them to converse directly. To enable peer-to-peer architectures in the face of connectivity barriers, libp2p introduces p2p-circuit. When a peer is unable to listen on a public address, the p2p-circuit protocol allows the peer to dial out to a relay peer, which will keep a long-lived connection open. Using a p2p-circuit address, this relay peer acts as a middleman - being able to forward traffic to the original peer when a dial is received. Relay connections between the relay node and the source node are also end-to-end encrypted, which means the relay peer is unable to read or tamper with any traffic that flows through the connection.

To prevent users from hard coding a list of well-known relays into their application, which acts as a point of centralization in the architecture, a feature called *Autorelay* is introduced. This Autorelay service is responsible for:

1. Discovering relay nodes around the world.
2. Establishing long-lived connections to them.
3. Advertising known relay-enabled addresses to peers, thus making the node routable through delegated routing.

The foundation of the circuit relay protocol originates from *TURN* [REFERENCE]

**Protocols** Libp2p has many available protocols, each defining a specific application and providing its core functionality. Libp2p protocols have unique string identifier, or *Protocol Ids*, which are used in the protocol negotiation process when connections are first opened. With the aim to be developer-friendly, and easy version-matching, protocols ids have a path-like structure, including a version number as the last component:

*/defi-agents/invest-protocol/1.0.1*

In order to accept connections, a libp2p application needs to register handler functions for specific protocols using their protocol id. These handlers will then later be invoked when an incoming stream is tagged with the registered protocol id.

For example, when a peer dials out to open a new connection, libp2p includes the selected protocol id *"/chat/1.2.0"* with the message. The listening peer on the other end will check the incoming protocol id against the registered protocol handlers. If no configuration for *"/chat/1.2.0"* is present, the listening peer will end the stream, and the dialing peer can make another attempt with a different protocol, or possibly a fallback version of the initially requested protocol e.g. *"/chat/1.0.0"*. If the protocol is supported, the listening peer will echo back the protocol id as an acknowledgement that future data will use the agreed protocol semantics. This process of reaching agreement about what protocol to use for a connection is called protocol negotiation.

In addition to user-defined protocols, libp2p has defined several foundational protocols that are used for core features. The following are some of these protocols:



- **/Ping.** The ping protocol is a simple check that peers can use to quickly see if another peer is online. After the establishing a connection, the dialing peer sends 32 bytes of random binary data and wait for the data to be echoed back from the listening peer. Verification of the returned data and latency between request and response could then be carried out by the dialing peer.
- **/Identify.** The identify allows peers to exchange information about each other, in particular their public keys and known network addresses. The dialing peer sends an identify message consisting of their public key and other information about themselves for the listening peer to derive their PeerId. This includes a list of observed addresses containing the multiaddresses that the peer observed the request coming in on. The observed multiaddresses here allow peers to determine their NAT status, letting them see what other peers observe as their public address and cross-check it with their own view of the network.

## 2.3 Agreements

### 2.3.1 Bitcoin and the rise of Ethereum

Launched in 2009, Bitcoin (BTC) establishes the first peer-to-peer digital currency to allow opening an account and sending toamoney without any trusted intermediaries. Bitcoin uses the Unspent Transaction Output (UTXO) model which represents a chain of ownership. However, limitations of Bitcoin and UTXO were later discovered [15] as it facilitates a weak version of a concept of “smart contracts”. For example, Bitcoin’s underlying scripting language does not support loops or recursions resulting in difficulties in building certain types of systems. UTXO can only be used to build simple, one-off contracts and not more complex stateful ones. In other words, the way Bitcoins transactions work under the hood makes creating new applications on the Bitcoin’s blockchain very complicated.

Ethereum was introduced in 2015 and addressed several limitations of the Bitcoin’s scripting language. The aim is to create an alternative protocol to create decentralized applications and to provide rapid code development time with security features. This is possible since Ethereum’s programming language supports loops and other Turing-Complete features by executing on Ethereum Virtual Machine (EVM). This tectonic shift in the blockchain world attracts developers to build applications on top of the Ethereum blockchain with Ether (ETH) as the network’s crypto-fuel. Ether is now the second largest digital currency by market cap after Bitcoin.

### 2.3.2 Smart Contracts

Business logic coded as software had been automating and revolutionizing the digital market long before Bitcoin came along. With the advent of smart contracts, there is a rapid ramping up of this trend. Smart contracts are decentralized, computerized contracts that include a complex set of rules embedded in code. Because the actual contract itself is a protocol that can be verified, signed, and deployed to a blockchain, there is no need for middlemen in contract construction execution, and enforcement. In business terms, smart contracts increase the speed at which agreement takes place, lower transaction risk, and could replace lawyers altogether.

As an alternative cryptocurrency network, Ethereum is not perfect. One concern is the requirement of Ether to fuel any computation, which introduces out-of-gas errors that does not exist in Bitcoin. Another trade-off is in the matter of security. In [16], the researchers focus their attention on properties of smart contracts vulnerabilities: finding contracts that either lock funds indefinitely, leak them carelessly, or can be killed by anyone. Out of nearly one million contracts, 34,200 of those are discovered to be vulnerable. Thus, MAIAN was implemented to be the first analysis tool for precisely specifying and reasoning about contract properties. With more than 1,000,000 contracts deployed and 100,000,000 ETH held on Ethereum, blockchain developers need sophisticated tools to design [17] and keep their contracts secure and safe from malicious activities.

### 2.3.3 Tokens

Tokens are not a new concept and have existed long before the emergence of Ethereum. They can represent a form of economic value and have some inbuilt anti-counterfeiting measure in order to

prevent malicious users from cheating the system. In the blockchain space, tokens can combine two concepts: access rights to some underlying economic value or a permission to access the service of someone else. Hence, cryptographic tokens represent a set of rules, encoded in a smart contract, called the token contract. A token contract can represent physical objects, another monetary value, and the holder's reputation. One popular type of these token contract is an ERC-20 token contract. An ERC-20 token contract is defined by the contract's address and the total supply of tokens available to it, along with a number of optional information for users to digest e.g. token's name, symbol, and decimals. The major difference between ERC20 tokens and other cryptocurrencies is that ERC20 tokens are created and hosted on the Ethereum network, whereas others such as Bitcoin are the native currencies of their respective blockchains. All of the different tokens in this project have been implemented as ERC20 tokens.

### 2.3.4 Solidity

A fairly new language, Solidity was created to run on the Ethereum Virtual Machine and allow users who use the Ethereum digital transaction ledger to develop smart contracts. As Solidity is a contract-oriented programming (COP) language [18], a controversy has risen - that Solidity was not required since object-oriented programming languages like C and C++ were able to handle smart contraction. However, COP offer additional functionalities which optimize these contracts to a higher level.

One advantage which Solidity borrow from OOP languages is the support for inheritance properties. Through inheritance, developers can create abstract classes, use hierarchical mappings, and utilize member variable within objects and classes. In addition, Solidity introduces interface specifications such as preconditions, post-conditions, and invariants which make smart contracts generation easier and more understandable.

Being a young language, Solidity suffers from the lack of libraries, documentation, and reference materials which may also not contain as much information concerning its structures and various components. Another disadvantage concerns bugs and changes. Unlike other more popular languages, Solidity does not have a large number of developers to resolve bugs. This also means that a line of code could break after a new patch has been released as updates are flooding in frequently.

## 2.4 Dai Ecosystem

### 2.4.1 DApps and DeFi

A new innovation for building massively scalable and decentralized applications has emerged. Ethereum has paved the way with its cryptographically stored ledger, transparency, and peer-to-peer technology. These properties provide a foundation for building a decentralized applications (DApps) with smart contracts serving as the blueprint. With no central point of failure, the platform allows developers to build far more sophisticated functionality than simply sending and receiving cryptocurrency. The most popular DApps are decentralized finance projects.

Decentralized Finance (DeFi) is the movement that leverages decentralized networks to transform old financial products into transparent protocols that run without TTP. As of 2019, More than 58% of DApps are DeFi projects. This allows users to create new assets, take out loans, build financial instruments, and implement automated, advanced investment strategies.

### 2.4.2 MakerDAO

According to financial metrics platforms like CoinMarketCap and CoinGecko, the total number of cryptocurrencies today is just shy of 6,000. Although this seems like a lot, most of these cryptocurrencies are attached to projects that were either abandoned or are created for scamming purpose. Not all crypto projects can or will last forever.

Thus, the following is an overview of the tokens relevant to this project:

- *BTC* – an alternative to national currencies and aspires to be a medium of exchange and store of value.

- *ETH* - used to facilitate and monetize the operation of the Ethereum smart contracts and decentralized applications.
- *WETH* - section [2.4.6](#)
- *PETH* - section [2.4.6](#)
- *DAI* - section [2.4.3](#)
- *MKR* - section [2.4.5](#)

Price fluctuations in popular cryptocurrencies such as Bitcoin (BTC) and Ether (ETH) has been historically unpredictable. The volatile nature of these digital assets prevents them from being a good medium of exchange. Having risen and fallen by as much as 25% in a single day and occasionally rising over 300% in a month, the Dai stablecoin system is introduced to solve this problem. Dai is one of the components in the MakerDAO project (or Maker), a decentralized credit platform and the leading DApps in today's Ethereum ecosystem with 58% dominance in the DeFi market.

### 2.4.3 Dai

The Dai stablecoin system is a set of blockchain smart contracts designed to issue a collateral-backed token, DAI, whose value is stable relative to the US Dollar - 1 USD equals 1 DAI. Dai has been implemented as an ERC20 token and, unlike other stablecoins, Dai subjects its price to a decentralized stability mechanism. Most stablecoins allow fiat to interact with the world of digital assets [19]. For example, a user deposits USD to mint tokens at a 1:1 ratio. Dai works the other way: it allows the value of cryptocurrencies to interact with assets in the real world. This is done through the Collateralized Debt Position.

### 2.4.4 Collateralized Debt Position

[20] Dai is obtained and exchanged through a dynamic system of Collateralized Debt Positions (CDPs). Through a CDP, anyone can leverage their Ethereum assets to generate Dai which can be used in the manner as any other cryptocurrency: For example, It can be used in crypto trading, as payments for good or services, or held as long-term savings. As Dai is generated, the active CDP locks up the collateral assets deposited until the owner pays back an equivalent amount of Dai, at which point the withdraw of the collateral is permitted. Through this process, users can get liquidity by generating DAI stablecoin without giving up ownership of their collateral.

As a procedure, the value of the locked up Ethereum collateral must always be more than 150% of the amount of DAI that you generate. However, if the value of the locked up collateral falls below 150%, the corresponding CDP will be liquidated. The system will then sell the collateral to cover the value of the generated DAI and will return any leftover collateral to the CDP owner. Here is a simplified formula [20] to determine how far the value of a collateral must fall in order to trigger a settlement:

$$StabilityDebt \times \frac{LiquidationRatio}{Collateral \times PETH/ETHratio} = LiquidationPrice$$

There is also a stability fee on the generated Dai of 9% per year, which is taken into the total account when the user pays back the Dai. If the CDP becomes liquidated, a 13% liquidation penalty then will be subtracted when the locked collateral is sold.

### 2.4.5 MKR

Dai debt incurs a stability fee in MKR, a governance token used by stakeholders to maintain the system and manage Dai. The compound stability fee is calculated continuously against the total amount of Dai drawn on the CDP and must be paid upon repayment of borrowed Dai.

### 2.4.6 Single-Collateral and Multi-Collateral Dai

In Single-Collateral Dai (SCD), Maker supports PETH (Pool Ether) as its only collateral type [21]. In November 2019, A new version of the Maker Protocol, Multi Collateral Dai (MCD), has been released on the main Ethereum network.

The following is the stages of the CDP creation for Single-Collateral Dai:

1. Creation of proxy (proxy will be discussed in details in Section 4.2.1)
2. Creation of CDP
3. Wrap ETH to WETH
4. Convert WETH to PETH
5. CDP collateralized with PETH
6. DAI generated
7. DAI transferred to wallet

In Maker, ETH itself is not directly used. Users who wish to open a CDP and generate Dai must first transform their ETH into an ERC-20 token called WETH (Wrapped Ether). An ERC-20 token is a type of digital asset that represents a financial value, and can be exchanged directly for other tokens in a decentralized manner unlike ETH.

Once the user deposits their WETH to Maker, the WETH is then automatically converted to PETH (Pooled Ether). It is this pooling system that collateralizes Dai. In comparison, the US government can show that they have X trillions of gold ‘pooled’ in Nevada, such that people can trust the USD they issue against it. In the same way, Maker uses their X millions of PETH pooled in their system to build the trust in Dai. Pooled Ether is the temporary mechanism for Single-Collateral Dai.

# Chapter 3

## Design

### 3.1 Agreement Calculus

We first formally describe how agents interact and are connected to each other. The semantics, syntax, and design is inspired by the BitML paper [4] and the  $\pi$ -calculus.

Our contracts allow participants to interact according to the following workflow. First a participant, or agent  $A$ , broadcasts a contract advertisement  $\{G\}C$ . The component  $G$  is a set of preconditions to its execution. For example, a precondition could require participants to deposit some ETH upfront in order to successfully engage in the contract. The component  $C$  is the contract, which specifies the rules to transfer ETH among participants. Once  $\{G\}C$  has been published, participants can choose whether to accept the advertisement, or not. When all the involved participants have responded, either accepting with all the preconditions satisfied or rejecting, the contract  $C$  becomes stipulated. At this points, accepted participants can then transfer the deposited funds by acting as prescribed by  $C$ .  $C$  then starts its execution with a balance, the sum of all deposits in its advertisement. After  $C$  has been successfully executed, the resulting funds will be transferred back to the corresponding participants.

The calculus is based on the assumption that agents are economically rational. A rational agent is one that has clear preferences, models uncertainty from expected values, and always chooses to act with the optimal expected outcome for itself among all feasible actions. With the capability of goal directed behavior, agents can estimate future benefits and the chances of success of their actions [22]. Nonetheless, The possible abuse of the system is discussed in Section 5.2.

#### 3.1.1 Contract Advertisement

A contract advertisement  $\{G\}C$  must satisfy the follow conditions:

1. the names in  $G$  are distinct
2. each name in  $C$  occurs in  $G$
3. the names in *put*  $\vec{x}$  &  $\vec{a}$  if  $p$  *reveal* are distinct, and each  $p$  occurs in
4. each  $A$  in  $\{G\}C$  has a persistent deposit in  $G$

The last condition (4) will be used to guarantee that the contract is stipulated only if all the involved participants give their authorizations. Requiring exactly the authorizations to spend persistent deposits in the symbolic semantics is key to implement contract stipulation in our investment protocol.

#### 3.1.2 Contract Preconditions

$G ::= A : ! @ x$	persistent deposit of $v$ ETH, expected from $A$
$  A : secret a$	committed secret by $A$
$  A : identify key$	$A$ identifies public $key$
$  G   G$	composition

The preconditions in  $G$  specify what is required to participate in a contract  $C$  and are composable, i.e. we can combine multiple of these. For example, this can include a certain deposit  $x$  that has to be made into the contract or restricts the agent's by their account id. If agent Alice and Bob agree that they want to invest together into a CDP with each at least 1 ETH, then the  $G$  would basically say: "only allow Alice's and Bob's public key to deposit money into  $C$  and both need to at least deposit 1 ETH into  $C$ ". Specifically, The precondition  $A : ! @ x$  requires  $A$  to own  $v$ ETH in a deposit  $x$ , and to spend it for stipulating a contract  $C$ .

The call into the contract must come from a certain public key  $key$ . For example, Alice and Bob know each other's Ethereum address and don't allow others to join the contract.

Finally, calling the contract requires agents to know the secret to a hash pre-image  $a$ . So, for example, if we want to prevent arbitrary agents to call our contract, we could require that the agents know the pre-image from a hash. We could store that hash in the smart contract and the agent needs to provide the pre-image when calling the contract. The contract then checks if the pre-image hashes to the stored hash and would allow the call. The secret  $a$  needs to be committed before  $C$  starts and  $A$  can choose whether to reveal  $a$  to the other participants during the execution of  $C$ .

### 3.1.3 Contracts

$C ::= \sum_{i \in I} D_i$	contract
$D ::=$	guarded contract
$  A : D$	wait for $A$ 's authorization
$  A \text{ Lending } E \ \& \ \bar{a} \text{ if } p \text{ reveal}$	$A$ calls <i>Lending</i> function
$\text{Lending} ::=$	CDP function calls
$\text{deposit}$	deposit $E$ to CDP
$  \text{withdraw}$	withdraw $E$ from CDP
$  \text{payback}$	payback $E$ to CDP, only in DAI or MKR
$  \text{borrow}$	borrow $E$ from CDP only in DAI
$  \text{create}$	open CDP
$  \text{exit}$	close CDP
$p ::=$	predicate
$\text{true}$	will pass condition
$  \text{false}$	will fail condition
$  \neg p$	negation
$  E = E$	equality
$  E < E$	less than
$  E \leq E$	less than or equal to
$  E > E$	greater than
$  E \geq E$	greater than or equal to
$  \text{in } N \text{ time}$	will trigger in the next N time
$  p \ \&\& \ p$	and operation
$  p \    \ p$	or operation
$E ::= N \text{ cur}$	expression
$N ::= [0 - 9]^*$	number expression
$\text{cur} ::=$	currency
$\text{ETH}$	ETH
$  \text{DAI}$	DAI token
$  \text{MKR}$	MKR token
$\text{time} ::= \text{blocktime} \   \ \text{blocknumber}$	blocktime

A contract  $C$  encodes the actual actions that are being executed once  $G$  is fulfilled. So in our example above,  $C$  would encode that once Alice and Bob had made their deposit the 2 ETH in the contract would be used to open a CDP in Maker. Further,  $C$  could encode that with that 2 ETH a certain amount of Dai is created. The contract could for example specify that a fixed amount of Dai is created, say 150. Or the contract could specify that all ETH is used to draw a maximum amount of Dai while having a 200% collateralization rate. These are the "terms" of the contract.

A contract  $C$  further denotes a choice among branches,  $D$ s, each representing an action. To restrict who can execute a branch and when, one can use the decoration  $A : D$ , which requires the authorization of  $A$  in order to move forward with the contract. The main branch to be used by the agents involves a *Lending* function which denotes a call to the CDP of the first participant  $A$  that broadcasts the contract advertisement, or the manager. This consists of mainly four functions: (i) *deposit* the initial Ethereum asset  $E$  to the CDP, (ii) *withdraw* Ethereum asset  $E$  from the CDP, (iii) *borrow*  $E$  DAI from the CDP, and (iv) *payback*  $E$  DAI to the CDP. A complete investment will need to call all four functions at least once to complete the Maker cycle. In addition, (v) *open* opens up a position in another contract e.g. creates a new, empty CDP in Maker, and (vi) *exit* withdraws all the coins from the invested position e.g. Alice decides the contract is not beneficial any longer for her.

$p$  denotes general predicates that are either true or false, which for example, can be used to implement a condition of an exit. Either Alice or Bob could execute the *exit* function if the profits are above a certain level or if losses are above a certain level. This criteria further allows time-based rules as an option. For example, Alice or Bob could both exit the contract after a certain time period has passed. The time metrics is only allowed in block-time for simplicity. Predicates can be combined.

### 3.1.4 An example of the Calculus

The example below follows a simple Maker CDP contract between two agents, Alice and Bob.

- **Contract advertisement  $\{G\}C$ :** Alice proposed to Bob that each of them deposit 1 ETH into a contract  $C$  where only the two of them can join. The contract says that the combined 2 ETH are used to draw 150 Dai from Maker and place the 150 Dai into the Maker contract to earn interest via the Dai savings rate. When they have earned 5 Dai through their 150 Dai deposit, either of them can close the CDP position and they can exit the contract.
- **Preconditions  $G$ :** Both Alice and Bob have to provide a "persistent deposit" of 1 ETH and it is bound to their "identity".
- **Contract  $C$ :** "Create" a CDP in Maker and "Enter" the CDP position with 2 ETH and draw 150 Dai from it. "Enter" the Maker Dai savings rate contract with the 150 Dai. "Exit" the Dai savings rate when there is a profit of 5 Dai, "Exit" the CDP position and "Withdraw" the deposited collateral.

## 3.2 Agent Negotiation Protocol

To provide a strong foundation for our agents implementation, we properly outline our Negotiation Protocol for agents to strictly follow. The proposed protocol has been designed to facilitate a flexible cooperation with multiple agents and to support automated negotiations in competitive domains. The protocol is based on FIPA standards described in Section 2.1.4 and extended the Contract Net Protocol to be made more suitable for our agents. Figure 3.1 portrays the flow of interaction between two agents, Alice and Bob.

The interaction can be broken down into the following four phases.

1. **Broadcast Phase** The first phase of the protocol allows the initiator, or the manager, to broadcast their proposal which contains investment information such as the amount of ETH they are willing to invest, conditions for the investment, and the deadline to reply by. This resembles the *Call-For-Proposals* stage of the CNP.

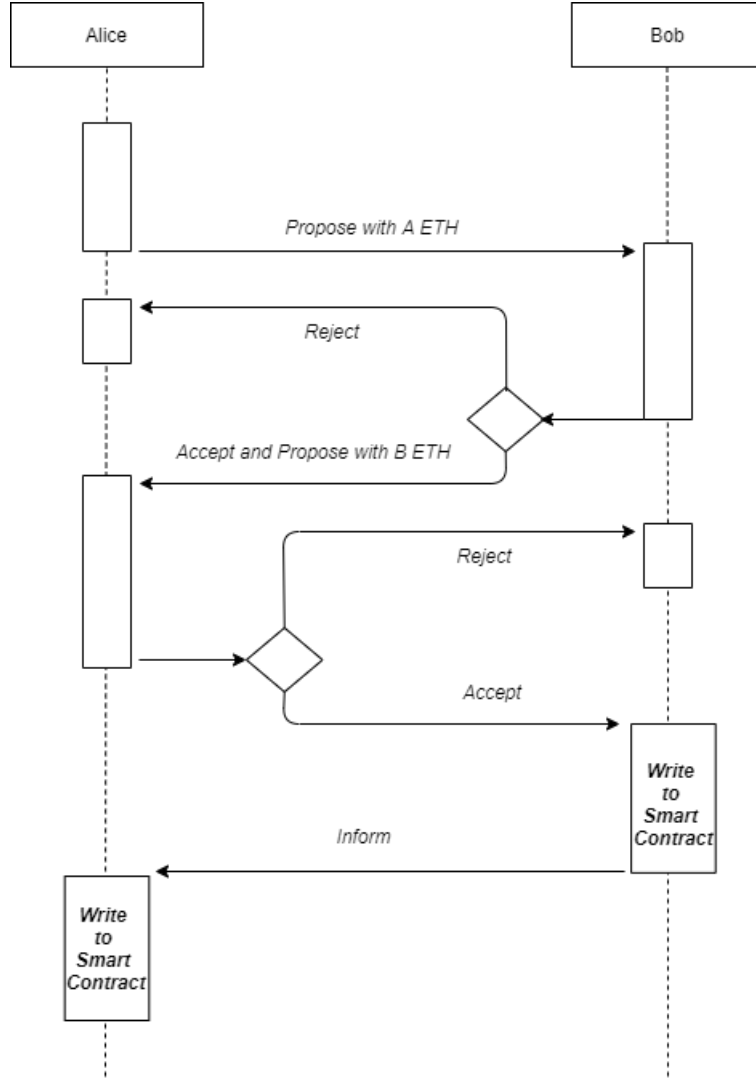


Figure 3.1: Interaction Flow

2. **Bidding Phase** The initiated agent evaluate whether the receives proposal matches their preference. In an advanced agent, an analysis of the proposed information can be carried out in this stage. An estimation of the how intellect the initiator is can be made and compared to own investment model. Thus, agents can keep historic record of others' strategy to form a inventive models. Once a decision is made, the agent responds back by either rejecting the proposal, or accepting it along their own conditions.
3. **Bid Selection Phase** Similarly, the initiator evaluates the condition from the response and make a decision about whether to move forward with the investment or not. After a set amount of time, the agent performs an analysis of all the responses it has acquired from the Bidding Phase, and move forward with the most suitable offer. Alternatively, the agent can reject and terminate the negotiation if none of the responses satisfy it. This stage lies the final decision to be made and no changes can be applied after this.
4. **Acknowledgement Phase** If the response from the Bid Selection Phase is positive, the initiated agent writes to the smart contract and deposits their share of the agreed investment. Once the execution is completed, an inform message will be sent to the initiator, which in turn will deposit the remaining share of the agreed amount. Unlike CNP, there does not exist an option for the initiated agent to cancel their task since smart contracts guarantee conditional transactions, ensuring that their agreement will be taken place one way or another.

For the Negotiation Protocol to work, some assumptions must be made concerning the agents and



their environment. First, we assume that users are running these agents on their online device, thus each agent has an internet connection. In particular, they are able to access their Ethereum wallet and connect to the live blockchain. This is crucial since, although negotiation can be taken place in the same LAN (e.g. Alice and Bob are operating on the same computer), no conclusion will ever be reached. This is due to the fact that the offline environment forbids our contract to interact with any DeFi protocol. The matter will be explained in Section 5.2. Second, agents are assumed to have at least some money in their Ethereum wallet, otherwise they cannot participate. This is a straightforward regulation since these money-less agents can be viewed as malicious attackers, attempting to scam or harm legitimate users.

## Chapter 4

# Implementation

Our development has been carried out in two main phases: implementing the multi-agent negotiations between agents and building the smart contracts to interact with DeFi protocols.

### 4.1 Negotiation Protocol

The main language the agent has been developed in is *JavaScript* with Node runtime environment. This has been chosen since Node offers many helpful libraries for the project, in particular Libp2p, a modular networking framework, and Dai.js, a JavaScript library to build applications on top of MakerDAO's platform of smart contracts. The latter is only used in early stages to get familiarized with Maker architecture before moving on to implementing Solidity code. Python and Rust were considered but since their Libp2p's modules are recent and unstable, we decided to not go through with them.

**Libp2p** is used to implement the negotiation between the agents before writing to the contract. To communicate over a multiplexed, secure channel, we have to configure a Transport, Crypto, and Stream Multiplexer module in our project. Each node needs to have a listen address for the given transport (TCP), so that it can be reached by other nodes in the network. There is no limits to the number of nodes or agents that can exist in the network at a single time. Thus, the agents would continuously broadcast its contract until another agent accepts or a timeout is reached. To achieve this, we use Behavior Driven Development (BDD) to split up the functionalities of the agent for easier implementation. This includes the following:

#### Peer Creation

#### Peer Discovery

A Peer Discover mechanism enables our agent to find other agent to connect to in the same network. There are two approaches to achieve this. First, an agent node can have a set of nodes to always connect on boot, or *bootstrapper nodes*. As shown in Listing 4.1, this is done by pre-configuring array listing PeerIds of the agents to connect on boot.

```
1 const bootstrappers = [  
2   '/ip4/104.131.131.82/tcp/4001/p2p/QmaCpDMGvV2BGHeYERUEnRQAwe3N8SzbUtfsmvsgQLuvuJ '  
3   ',  
4   '/ip4/104.236.176.52/tcp/4001/p2p/QmSoLnSGccFuZQJzRadHn95W2CrSfMzuTdDWP8HXaHca9z '  
5   ',  
6   '/ip4/104.236.179.241/tcp/4001/p2p/QmSoLPppuBtQSGwKDZT2M73ULpjvfd3aZ6ha4oFGL1KrGM '  
7   ',  
8   '/ip4/162.243.248.213/tcp/4001/p2p/QmSoLuer4xBeUby9WZ9xGUUxunbKWcrNFTDAadQJmocrnWm '  
9   ',  
10  '/ip4/128.199.219.111/tcp/4001/p2p/QmSoLSafTMBsPKadTEgaXctDQVcqN88CNLHXMkTNwMKPnu '  
11  ',  
12  '/ip4/104.236.76.40/tcp/4001/p2p/QmSoLV4Bbm51jM9C4gDYZQ9Cy3U6aXMJDAbzgu2fzaDs64 '  
13  ',  
14  '/ip4/178.62.158.247/tcp/4001/p2p/QmSoLer265NRgSp2LA3dPaeykiS1J6DifTC88f5uVQKNAd '  
15  ',  
16  '/ip4/178.62.61.185/tcp/4001/p2p/QmSoLMeWqB7YGVVLJN3pNLQpmmEk35v6wYtsMGLzSr5QBU3 '  
17  ',  
18  '/ip4/104.236.151.122/tcp/4001/p2p/QmSoLju6m7xTh3DuokvT3886QRYqxAzb1kShaanJgW36yx '  
19  ']
```

Listing 4.1: Some Bootstrapper PeerIds

The method provides a simple but controlled environment so once the node is created and started, we can listen for events such as `peer:discovery` and `peer:connect`. These events tell us to independently find the input peers, and if found, dial to them accordingly. However, this puts a limit on the number of peers that our agent can connect to and also prevent us from spawning new agent on-the-go without having to manually adding their `PeerId`. Hence, we use the second approach which is to use MulticastDNS to discover nodes through locality. This way, once new peers are discovered, their known data is stored in the agent's `PeerStore`. This is implemented in Listing 4.2.

```

1  const MulticastDNS = require('libp2p-mdns')
2
3  const createNode = () => {
4    return Libp2p.create({
5      addresses: {
6        listen: ['/ip4/0.0.0.0/tcp/0']
7      }
8      modules: {
9        peerDiscovery: [ MulticastDNS ],
10       ...
11     },
12     config: {
13       peerDiscovery: {
14         mdns: {
15           interval: 20e3,
16           enabled: true
17         }
18       }
19     }
20   })
21 }
22
23 const [node1, node2] = await Promise.all([
24   createNode(),
25   createNode()
26 ])
27
28 node1.on('peer:discovery', (peer) => console.log('Discovered:', peer.id.toB58String()))
29 node2.on('peer:discovery', (peer) => console.log('Discovered:', peer.id.toB58String()))

```

Listing 4.2: Peer Discovery Configuration

## Security

We can further leverage the encrypted communications from the transports libp2p uses. To ensure that every connection is encrypted, independently of how it was set up, libp2p also supports a set of modules that encrypt every communication established. We call this usage a connection upgrade where given a connection between agent Alice to agent Bob, a protocol handshake can be performed that gives that connection new properties. A byproduct of having these encrypted communications modules is that we can authenticate the peers we are dialing to. This is done through the use of agents' `PeerId` at the end, for example:

```
/ip4/127.0.0.1/tcp/89765/p2p/QmWCbVw1XZ8hiYBwwshPce2yaTDYTqTaP7GCHGpry3ykWb)
```

This `PeerId` is generated by hashing the Public Key of the peer. With this, we can create a crypto challenge when dialing to another peer and prove that peer is the owner of a PrivateKey that matches the Public Key we know. We adopt the module on top of the agent creation in Listing 4.3.

```

1  const Libp2p = require('libp2p')
2  const { NOISE } = require('libp2p-noise')
3  const SECIO = require('libp2p-secio')
4
5  const createNode = () => {
6    return Libp2p.create({
7      modules: {

```

```

8     transport: [ TCP ],
9     streamMuxer: [ Mplex ],
10    // Attach secio as the crypto channel to use
11    connEncryption: [ NOISE, SECIO ]
12  }
13 })
14 }

```

Listing 4.3: Communication Encryption Configuration

## Communication

Now that we have the basic building blocks of transport, multiplexing, and security in place, we implement the communication mechanism. For two agents to communicate with each other, one acts as the dialer node with the other being the listener node. Listing 4.4 allows the dialer to reach the listener on protocol `/echo/1.0.0`. "it-pipe" npm package is used to allow streaming of data between agents.

```

1  async function startAndDial() {
2    const [dialerId, listenerId] = await Promise.all([
3      PeerId.createFromJSON(require('./id-d')),
4      PeerId.createFromJSON(require('./id-l'))
5    ])
6
7    // Dialer
8    const dialerNode = new Node({
9      addresses: {
10        listen: ['/ip4/0.0.0.0/tcp/0']
11      },
12      peerId: dialerId
13    })
14
15    // Add peer to Dial (the listener) into the PeerStore
16    const listenerMultiaddr = '/ip4/127.0.0.1/tcp/10333/p2p/' + listenerId.
      toB58String()
17
18    // Start the dialer libp2p node
19    await dialerNode.start()
20
21    console.log('Dialer ready, listening on:')
22    dialerNode.multiaddrs.forEach((ma) => console.log(ma.toString() +
23      '/p2p/' + dialerId.toB58String()))
24
25    // Dial the listener node
26    console.log('Dialing to peer:', listenerMultiaddr)
27    const { stream } = await dialerNode.dialProtocol(listenerMultiaddr, '/echo/1.0.0'
28      )
29
30    console.log('nodeA dialed to nodeB on protocol: /echo/1.0.0')
31
32    pipe(
33      // Source data
34      ['hey'],
35      // Write to the stream, and pass its output to the next function
36      stream,
37      // Sink function
38      async function (source) {
39        // For each chunk of data
40        for await (const data of source) {
41          // Output the data
42          console.log('received echo:', data.toString())
43        }
44      }
45    )

```

Listing 4.4: Dialer Peer

As expected, the listener code in Listing 4.5 waits for a connection to be established from a dialer and logs the incoming stream to console.

```

1  async function startAndListen() {
2    const listenerId = await PeerId.createFromJSON(require('./id-l'))

```

```

3
4 // Listener libp2p node
5 const listenerNode = new Node({
6   addresses: {
7     listen: ['/ip4/0.0.0.0/tcp/10333']
8   },
9   peerId: listenerId
10 })
11
12 // Log a message when we receive a connection
13 listenerNode.connectionManager.on('peer:connect', (connection) => {
14   console.log('received dial to me from:', connection.remotePeer.toB58String())
15 })
16
17 // Handle incoming connections for the protocol by piping from the stream
18 // back to itself (an echo)
19 await listenerNode.handle('/echo/1.0.0', ({ stream }) => pipe(stream.source,
20   stream.sink))
21
22 // Start listening
23 await listenerNode.start()
24
25 console.log('Listener ready, listening on:')
26 listenerNode.multiaddrs.forEach((ma) => {
27   console.log(ma.toString() + '/p2p/' + listenerId.toB58String())
28 })

```

Listing 4.5: Listener Peer

**Tokenizr**, an npm package for flexible string tokenization functionality, is implemented to handle the Agreement Calculus used by agents to communicate. The program acts as the underlying lexical scanner and syntax parser for the agents. Our Agreement Calculus has been integrated into the library to detect terminology mismatch and violation of our syntax. This can be seen in Listing 4.6.

```

1 const Tokenizr = require("tokenizr");
2 const lexer = new Tokenizr();
3
4 lexer.rule(/[a-zA-Z_][a-zA-Z0-9_]*/, (ctx, match) => {
5   ctx.accept("command");
6 });
7 lexer.rule(/[+-]?[0-9]+/, (ctx, match) => {
8   ctx.accept("number", parseInt(match[0]));
9 });
10 lexer.rule(/"((?:\\\"|[^\\r\\n])*)"/, (ctx, match) => {
11   ctx.accept("string", match[1].replace(/\\\"/g, '\"'));
12 });
13 lexer.rule(/\\/\\/[^\r\n]*\r?\n/, (ctx, match) => {
14   ctx.ignore();
15 });
16 lexer.rule(/[ \t\r\n]+/, (ctx, match) => {
17   ctx.ignore();
18 });
19
20 lexer.rule(/./, (ctx, match) => {
21   ctx.accept("char");
22 });

```

Listing 4.6: Calculus Handler

## 4.2 Smart Contracts

After a mutual agreement is reached via our Negotiation Protocol, the terms of agreement will be encoded to a shared smart contract via our implementation of a calculus handler shown in Listing 4.7, and a function template. The next step is to write and deploy the agreed contract to a live network for agents to deposit their ETH. The following describes core functions implemented in the contract not including utility functions.

- **deposit()**: Called by participants in order to deposit their share of investment. Some requirements are need to be passed to ensure validity of the agent e.g. if amount matches what has been agreed, if sender address matches the address used in the Negotiation Protocol.
- **openAndLockETH()**: Invoked when deposits are made from two agents and the sum matches what has be agreed upon. The function opens a new CDP and lock the combined balance inside the Maker Vault.
- **exit()**: Invoked when the agreed condition is met. The function terminate the contract and transfer the ETH back to each agents respectively.

Throughout the project, smart contracts are used extensively to interact with the MakerDAO protocol for the agents to complete a successful invest cycle. Hence, smart contracts were implemented alongside the agents in order to execute and test the agents' functionality. Many frameworks and Ethereum tools were experimented with and applied throughout the project including Truffle, Ganache, Ethers, Infura, Tenderly, and Buidler.

**Buidler** is a fairly new task runner that facilitates building Ethereum smart contracts, which stands out through its automation of recurring tasks, i.e. compiling and testing. The bulk of Buidler's functionality comes from the concepts of tasks and plugins which can connect directly to their provider, in particular @nomiclabs/buidler-ethers, which allows our contracts to interact with the Ethereum blockchain through ether.js. With the capability to work with other developer tools, rather than replacing them, Buidler offers unique flexibility, interoperability, and debugging functionality for our contracts to be built upon. In early development, @nomiclabs/buidler-truffle4 plugin was used to allow tests and scripts written for Truffle to work with Buidler but was replaced entirely with Buidler's fashion code.

Buidler also comes built-in with **Buidler EVM**, a local Ethereum network to deploy our contracts, run tests and debug. The EVM is backed by the ethereumjs-vm EVM implementation, which is based upon by various popular Ethereum frameworks, and can be configured to connect to live blockchains. However, the most beneficial aspect of Buidler to the project is due to its capacity to debug. Since Buidler can spin up an local testing node through its command line, we can use this option to print logging messages and contract variables from our Solidity code. The functionality is not provided by Truffle or other frameworks. However, the debugging feature is only exclusive to nodes spawned by Builder EVM. Thus, it is not available when developing Maker related contracts since these contracts only run on live blockchains.

**Tenderly** is a development tool that provides readable stack traces to simplify Ethereum debugging. Its visual debugger is used to detect bugs mainly in how our contract interacts with the Maker contracts. This is crucial since transactions could be successful with its underlying logic being faulty. However, being a new considerably new tool in the Ethereum ecosystem, Tenderly does not support Buidler environment. Thus, the contract must be deployed to a live network, in particular Kovan, before being imported to Tenderly's dashboard through Etherscan, a "Block Explorer" used to track and navigate smart contracts. Note that the contract must also be verified and published once it is deployed. This is to ensure that the contract code is exactly what is being deployed onto the blockchain and also allows the public audit and read the contract.

**Infura**, a development suite providing access to the Ethereum networks, is used to host Ethereum node cluster for our contracts to be deployed upon. This is done via forking off Mainnet with local test chain so we have access to the state and protocol that reside on Mainnet, while being able to develop locally. Infura works hand in hand with Ganache, a test blockchain that can be run locally and will insta-mine transactions. The Infura API key used is 7d0d81d0919f4f05b9ab6634be01ee73 and can be obtained publicly from Ether.js source code. Now, we have a running Ganache instance at <http://localhost:8545> (default network), and can be used to interact with Maker's Mainnet contracts.

For agents to successfully make an investment with Maker, they have to atomically perform transactions across multiple contracts through **DSPProxy** as seen in 4.1. The proxy contains contract interfaces, proxies, and aliases to functions necessary for both Vault management and Maker governance. Developing core smart contracts with this pattern in mind increase our smart contracts security without sacrificing usability, while reducing overall complexity, and preserving atomicity. The Maker architecture is fairly complex so we integrated the Money-Legos library to build our

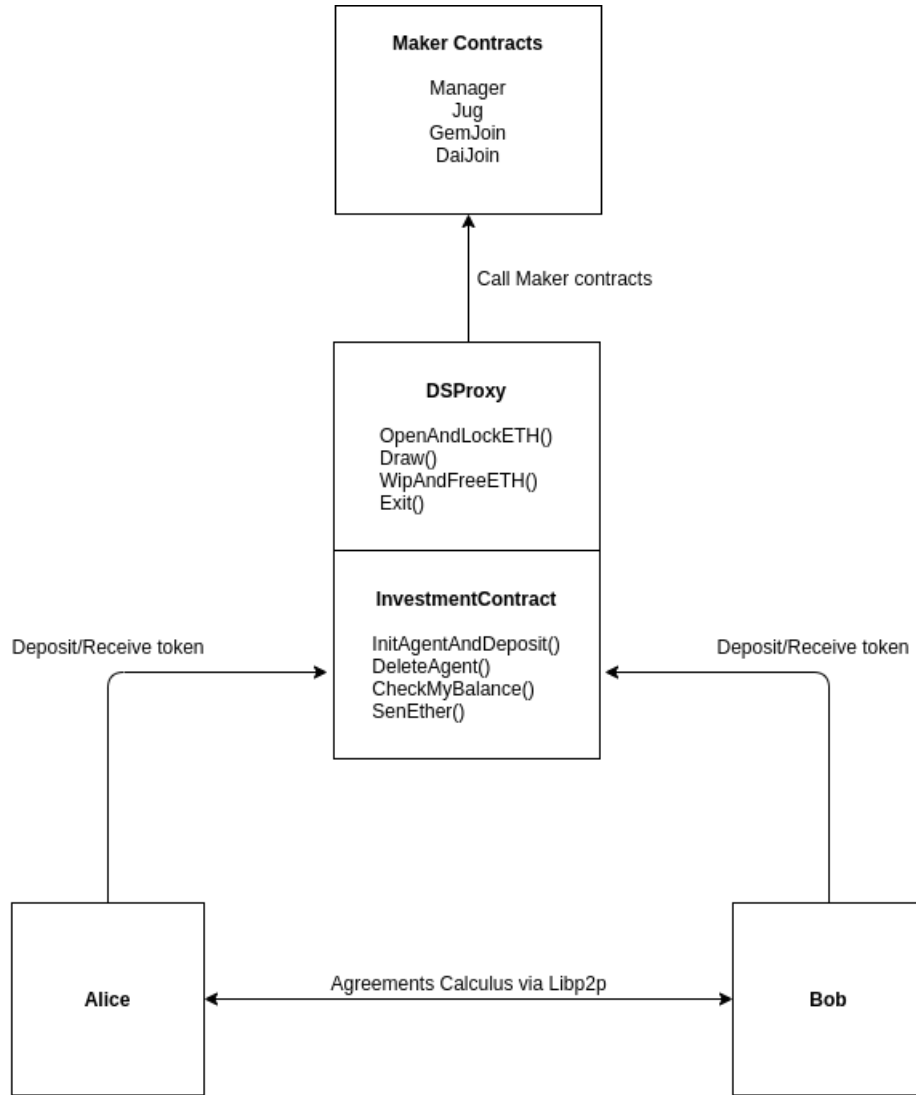


Figure 4.1: Architecture

contracts.

**Money-Legos** is an NPM package that provides the mainnet addresses, ABIs, and Solidity interfaces for popular DeFi protocols. It supports both JavaScript and Solidity integration of relevant protocols, including MakerDAO, ERC20, and Compound. Although, being an exceptionally new library, Money-Legos heavily lacks documentations and user's resources, but compensates with its responsive and helpful community through its Discord channel.

#### 4.2.1 DSPProxy

When an agent creates a CDP, a proxy contract is deployed at the same time and tied to their Ethereum account. This proxy contract is technically the owner of a CDP, and each CDP owner has one of their own to keep track of users' vaults. This contract is called a DSPProxy. Hence, we have to perform a *delegate calls* to Maker's "managers", which are contracts where the underlying logic lives. For our contracts to interact with these managers, there are several methods to achieve this goal in Solidity. Without knowing the target contract ABI, we could not directly use function signature to call the target contract's functions. Thus, we perform a delegate call which is a calling mechanism of how caller contract calls target contract function but when the target contract executed its logic, the context (`msg.sender`, `msg.value`, etc.) is not on the user who execute caller contract but on caller contract. Note that when `delegatecall` to target, the context is on caller contract, all state change logics reflect on caller's storage as seen in 4.3. In our case, when the agent

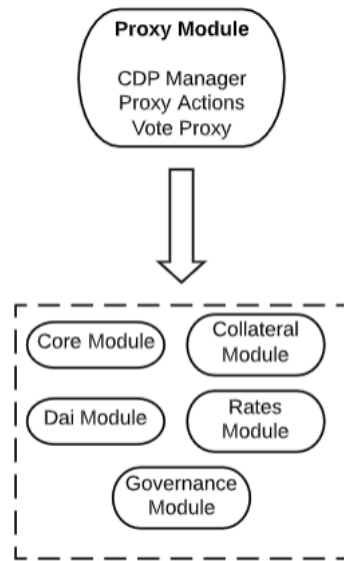


Figure 4.2: Proxy architecture

calls Proxy contract, Proxy contract will delegatecall to managers contract and function would be executed. But all state changes will be reflected Proxy contract storage, not the managers contract.

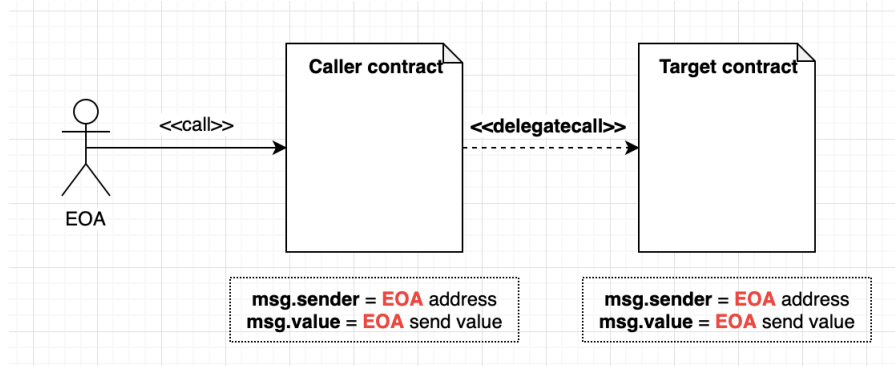


Figure 4.3: Context in delegatecall

There are two main advantages in taking this DSProxy approach. The first is to allow actions to be executed through the proxy identity. This can be very useful for securing complex applications. Because delegatecall retains msg.sender and msg.value properties, internal functions can be set to only accept calls coming from the proxy through an ownership model like ds-auth. In this manner as long as the proxy is not compromised, the internal system is protected from outsider access. Should the owner of the internal calls ever need to be changed, this is as simple as updating the owner of ds-proxy rather than manually updating each individual internal function call, making it much more secure and adaptable. Second, DSProxy executes a sequence of actions atomically. Due to restrictions in the EVM instruction set such as being unable to be nested dynamically sized types and arguments, 1 transaction could be done at a time. Since ds-proxy takes in bytecode of a contract, rather than relying on a pre-deployed contract, customized script contracts can be used. These script contracts share a very important property in that they enable a sequence of actions to be executed atomically (all or nothing). This prevents having to manually rollback writes to contracts when a single transaction fails in a set of transactions. As such, we needed to first create a proxy on Maker's Proxy Registry before we can start automating our vaults.



### 4.2.2 Interacting with Maker's Contracts

Once a common Proxy Registry is used to deploy a DSProxy contract tied our agent's Ethereum account, we can start interacting with the Maker protocol. Nonetheless, due to the way Maker has constructed its architecture, we had to re-implement some logic into our contract ourselves. Unfortunately, Maker has a tendency to name things unintuitively and the primitives heavily lack comments in the MakerDAO's source code. For example, Maker's token is called a 'gem', and the act of selling those gems is called a 'flap'. With these terms being inscrutable and unclear of their functionality, the followings are short descriptions of some terminology that we used to implement our contract.

- *gem*: collateral tokens, can be transferred to any address by its owner. This is the original ETH of the agent in our case.
- *dai*: stablecoin tokens, can only be moved or transferred to any address with the consent of its owner.
- *ilk*: collateral type, each has its own set of risk parameters used to calculate price feed and debt rate.
- *urn*: a specific vault, an address can control one urn per collateral type.
- *vat*: the storage of the Vat contract's address. This cannot be changed since Vat is the core of the mcd system, housing the public interface for Vault management, allowing vault owners to adjust their vault state balances.
- *join*: the adapter used to deposit or withdraw unlocked collateral into the Vat.

Sitting between our agents and the Maker protocol, our contract was written to include the main functionality necessary for interacting with the Maker contracts, including: `OpenAndLockETH()`, `Draw()`, `Wipe()`, `Exit()`. In addition to numerical parameters, these functions need to take in addresses of the maker contracts to be called once the function is executed. An example is shown in Listing 4.7.

```
1 function openAndLockETH(  
2     address manager,  
3     address jug,  
4     address ethJoin,  
5     address daiJoin,  
6     uint256 wadD  
7 ) public payable {  
8     // Opens ETH-A CDP  
9     bytes32 ilk = bytes32("ETH-A");  
10    uint256 cdp = open(manager, ilk, address(this));  
11  
12    address urn = ManagerLike(manager).urns(cdp);  
13    address vat = ManagerLike(manager).vat();  
14  
15    // Receives ETH amount, converts it to WETH and joins it into the vat  
16    ethJoin_join(ethJoin, urn);  
17  
18    // Locks WETH amount into the CDP and generates debt  
19    frob(  
20        manager,  
21        cdp,  
22        toInt(msg.value),  
23        _getDrawDart(vat, jug, urn, ilk, wadD)  
24    );  
25  
26    // Moves the DAI amount (balance in the vat in rad) to proxy's address  
27    move(manager, cdp, address(this), toRad(wadD));  
28  
29    // Allows adapter to access to proxy's DAI balance in the vat  
30    if (VatLike(vat).can(address(this), address(daiJoin)) == 0) {  
31        VatLike(vat).hope(daiJoin);  
32    }  
33 }
```

Listing 4.7: OpenAndLockETH function

Here, all four addresses of the Maker contracts (manager, jug, ethJoin, daiJoin) must be provided. These can be found manually through Maker or, much simpler, can be referenced through the Money-Legos library.

In order for the contract to fulfil our agent negotiation's Calculus, an Agent mapping is built and stored in our contract, as seen in Listing 4.8. This is used to keep tracks of the agents participating in the contract including their addresses, names, deposits, and validity. In addition, a set of simple CRUD functions is implemented for the agents to easily manage themselves on the contract. Note that anyone has the ability to deploy the DSPProxy contract, whether it's Alice, Bob, or some other entity that might not participate in the negotiation at all. Only the address and the ABI of the deployed contract must be available for the agents in order for them to start investing.

```

1 contract MyCustomVaultManager is DssProxyActionsBase {
2     int256 agentNum = 0;
3     address[] agentAddress;
4     string[] agentName;
5
6     struct Agent {
7         string name;
8         uint256 balance;
9         bool valid;
10    }
11
12    mapping(address => Agent) Agents;

```

Listing 4.8: Contract Declaration and Agent Structure

To deploy the agreed smart contract, we execute our Node's script using the caller's address. Note that we do not need to redeploy the DSPProxy contract if it has already been done and can build its instance from its address. Listing 4.9 shows the deployment of our custom contract (MyCustomVaultManager) and the approach to deploy a new proxy registry. As seen in line 72, the deployment requires a considerable amount of gas as it is an expensive transaction. We once needed as high as 3000000 to deploy the contract on Kovan via Remix, an on-the-web Ethereum IDE.

```

1 const maker = require("@studydefi/money-legos/maker");
2 const dappsys = require("@studydefi/money-legos/dappsys");
3 const erc20 = require("@studydefi/money-legos/erc20");
4 const proxy_registry_abi = require("../kovan_artifacts/Proxy_Registry.json");
5 const abi = require("../artifacts/MyCustomVaultManager.json");
6
7 async function main() {
8     // Deploy and create MyCustomVaultManager instance
9     const MyCustomVaultManager = await ethers.getContractFactory(
10         "MyCustomVaultManager"
11     );
12     let myCustomVaultManager = await MyCustomVaultManager.deploy();
13     await myCustomVaultManager.deployed();
14
15     // Build Manager Instance and connect it to the contract
16     const provider = new ethers.providers.JsonRpcProvider();
17     const bobprivateKey =
18         "0x8a9d16d5aee4cc35c090f2d0fe6e6c8e57cf658048ce631dbbea0fe550bc77cd";
19     const bobwallet = new ethers.Wallet(bobprivateKey, provider);
20     myCustomVaultManager = myCustomVaultManager.connect(bobwallet);
21
22     // We need some account with some gas to deploy the proxy initially - this won't
23     // matter later.
24     const privateKey =
25         "0x8a9d16d5aee4cc35c090f2d0fe6e6c8e57cf658048ce631dbbea0fe550bc77cd";
26
27     const wallet = new ethers.Wallet(privateKey, provider);
28     const daiContract = new ethers.Contract(
29         erc20.dai.address,
30         erc20.abi,
31         wallet
32     );
33
34     const proxyRegistry = new ethers.Contract(
35         maker.proxyRegistry.address,
36         maker.proxyRegistry.abi,

```

```

36     wallet
37   );
38
39   // Build Maker proxy if we don't have one
40   let proxyAddress = await proxyRegistry.proxies(wallet.address);
41   if (proxyAddress === "0x0000000000000000000000000000000000") {
42     console.log("building proxy...");
43     await proxyRegistry.build({
44       gasLimit: 1500000,
45     });
46     proxyAddress = await proxyRegistry.proxies(wallet.address);
47   }
48
49   // Build Contract instance
50   const proxyContract = new ethers.Contract(
51     proxyAddress,
52     dappsys.dsProxy.abi,
53     wallet
54   );
55
56   // Prepare data for delegate call
57   const ProxyActions = new ethers.utils.Interface(
58     myCustomVaultManager.interface.abi
59   );
60
61   // Encode parameters
62   const _data = ProxyActions.functions.someFunction.encode([
63     maker.dssCdpManager.address,
64     maker.jug.address,
65     maker.ethAJoin.address,
66     maker.daiJoin.address,
67     ethers.utils.parseUnits("25", ERC20.dai.decimals),
68   ]);
69
70   // Call a function via proxy
71   await proxyContract.execute(myCustomVaultManager.address, _data, {
72     gasLimit: 2500000,
73   });
74 }

```

Listing 4.9: Deploying MyCustomVaultManager and DSProxy

Here, Listing 4.9 is executing on a local forked mainnet as seen in line 16. To make real transaction and properly test our contract, we need to run the script on a live network. In our case, Kovan testnet is used due to Maker's availability. First, we need to create a live node on infura.io and import the newly generated project url and private key into our build.config.js. Then we reassign our provider in line 16 to `let provider = ethers.getDefaultProvider("kovan");`. Lastly, we replace the addresses of all the contracts we use to their corresponding Kovan addresses. For example, rather than importing the Maker addresses via Money-Legos on line 1, we use the addresses in Listing 4.10 instead.

```

1 maker.proxyRegistry.address = "0x64A436ae831C1672AE81F674CAb8B6775df3475C";
2 maker.dssCdpManager.address = "0x1476483dD8C35F25e568113C5f70249D3976ba21";
3 maker.jug.address = "0xcbB7718c9F39d05aEEDe1c472ca8Bf804b2f1EaD";
4 maker.ethAJoin.address = "0x775787933e92b709f2a3C70aa87999696e74A9F8";
5 maker.daiJoin.address = "0x5AA71a3ae1C0bd6ac27A1f28e1415fFFB6F15B8c";
6 maker.proxyRegistry.abi = proxy_registry_abi;

```

Listing 4.10: Maker's Contract Addresses Release 1.0.7 (Wednesday, 03.06.2020)

Maker's latest address can be found on <https://changelog.makerdao.com>. Note that the addresses we use need to be up to date with the latest release of their protocol, else a "contract not found" error might be invoked.

All smart contracts are written in Solidity 0.5.1.

## Chapter 5

# Evaluation

The finished project consist of a successful negotiation between agents along with the successful process of investment through the agents' interaction with the Maker, the largest DeFi protocol by market share. Specifically, the following is what we have acheived:

1. Our negotiation protocol is able to handle multiple agents.
2. The encoding of the terms is sufficient to encode the terms required for the Maker protocol and further support Compound, Aave, Synthetx, and other major DeFi protocols. Thereby, we estimate the Agreement Calculus to cover over 90% of the DeFi market share.
3. Translation of the encoded terms to Solidity smart contracts.
4. Deploy the protocol in the real world to allow agents to invest on the Kovan testnetwork.

The result of their investment will not determine the outcome of the project since we are on the assumption of the risk modelling project.

### 5.1 Applicability of Agreement Calculus

Apart from being able to formally encode the terms required for the Maker protocol, our Agreement Calculus can also be applied to other DeFi projects as well. *Compound* is one of the compatible projects since it allows users to take out loan by supplying their cryptoassets (not necessarily limited to only Ethers) to the market. Thus, agents can negotiate over the Compound protocol using our Agreement Calculus, particularly, the Lending terms (*deposit, withdraw, payback, borrow*). A small extension of adding new currencies into the *cur* term must be made for our Agreement Calculus to support other type of collateral e.g. *WBTC* for Wrapped Bitcoin, *BAT* for Basic Attention Token. Similar to Compound, Aave offers a DeFi lending protocol and the exclusive functionality to support uncollateralized loans and unique collateral types. Synths also operates in a similar manner, but rather than issuing DAI, it uses Synths. Maker, Compound, Synthetix, and Aave are the biggest DeFi protocols as of now, and have the combined locked value of \$1.3 billion (USD) against the total value of \$1.4 billion (USD). With these four protocols alone, our Agreement Calculus is ensured to cover more then 90% of the market share.

### 5.2 Negotiation Protocol

Evaluating our negotiation mechanism, we have achieved successful communication between multiple agents and thus reaches one of the project's objectives. However, discussions can be made about the limitations of our agents and the communication layer they live on. First, agents are able to negotiate in an offline network, in particular, broadcasting their proposals and accepting others' proposals. Nevertheless, they will be unable to write and deploy to the agreed contract since a DSProxy is needed to be deployed via the Maker's Proxy Registry. A "contract not found" error will be invoked from running the agents offline. As expected, agents cannot perform a complete investment cycle without connecting to the internet.

With transactions of digital assets involved, it is crucial that the privacy of the messages must be reliable. This is one of the strengths of our negotiation protocol. We use Libp2p-Secio, a component of the libp2p project, to provide a secure transport channel for agents. The module enforces an initial plaintext handshake on communication which encrypts all data exchanged between agents and protects the core content from being eavesdropped and being attacked by man-in-the-middle. Nevertheless, it does leak some metadata which resides in the shared plaintext, and thus a passive eavesdropper can see who is talking to who. However, this is merely a minor drawback since attackers are still unable to penetrate the channel.

Finally, a possibility exists where one attacker agent intentionally tries to lead into a 'bad' contract. The attacker agent can claim to have a great investment strategy while having an underperforming one and will actually lose the victim agent a lot of money. This malicious act can also be carried out in a way that does not affect the attacker as much as the victim. For example, the attacker proposes to invest 1 ETH while the victim naively agrees with the proposal of 100 ETH. This is a clear abuse of the system and is a disadvantage of our protocol.

## 5.3 Testings

In order to evaluate the effectiveness of the agent's negotiation protocol, a set of success metrics need to be defined. If we get two protocols integrated, we can simulate agents with different preferences and test how they behave. We can then evaluate the utility functions for each agent and gauge their result. For example, how much money can an agent make, the ratio of success to fail investments. Observation can be made depending on the environment we put the agents in such as their initial money and time to invest. By testing individual agents, we can then compare that to a baseline where each agent interacts by itself without cooperating with each other. In theory, the multi-agent systems will result in a better outcome since agents have better options to reach their designed objective. However, with many agents come more risk in each investment. Unfortunately, the experiment is not possible without the risk modelling project which provides the inputs to our agents and give them their investment behaviour. Due to unexpected timing issues and the difficulty in the integration, the two projects can not be executed as a whole.

However, the investment mechanism of trustless contract execution can be tested to ensure that an agent can actually perform a successful collaboration with other agents. The testing is carried out using a locally spawned Ethereum node via Buidler EVM and @nomiclabs/buidler-ganache. This way, we can debug our smart contracts in a friendly manner since Buidler generates combined JavaScript and Solidity stack traces. For example in Listing 5.1, we can observe a failing transaction and that the error originates from our JavaScript calling to the contract, and hence gives a "function selection was not recognized" error in the Solidity call stack.

```

1 Error: Transaction reverted: function selector was not recognized and there's no
  fallback function
2   at ERC721Mock.<unrecognized-selector> (contracts/mocks/ERC721Mock.sol:9)
3   at ERC721Mock._checkOnERC721Received (contracts/token/ERC721/ERC721.sol:334)
4   at ERC721Mock._safeTransferFrom (contracts/token/ERC721/ERC721.sol:196)
5   at ERC721Mock.safeTransferFrom (contracts/token/ERC721/ERC721.sol:179)
6   at ERC721Mock.safeTransferFrom (contracts/token/ERC721/ERC721.sol:162)
7   at TruffleContract.safeTransferFrom (node_modules/@nomiclabs/truffle-contract/lib
  /execute.js:157:24)
8   at Context.<anonymous> (test/token/ERC721/ERC721.behavior.js:321:26)

```

Listing 5.1: Error from Truffle

Along with Buidler, we use Mocha and Chai as our testing tools. **Mocha** is a Node testing framework that allows asynchronous testing to be carried out. **Chai** is a behaviour drive development and test driven development assertion library for node that provides several interfaces, well known for its expressive language and readable style. Listing 5.2 shows one of the features we tested.

```

1 const { expect } = require("chai");
2 const maker = require("@studydefi/money-legos/maker");
3 const { Wallet } = require("ethers");
4
5 describe("agent", function () {
6   it("create a proxy on Maker", async () => {
7     const provider = new ethers.providers.JsonRpcProvider();

```

```

8     const privateKey =
9       "0x3ff355e2f5de0366747d2818919f36f061514ff079315c3f39717ef82ec7b219";
10    const wallet = new ethers.Wallet(privateKey, provider);
11
12    const proxyRegistry = new ethers.Contract(
13      maker.proxyRegistry.address,
14      maker.proxyRegistry.abi,
15      wallet
16    );
17
18    // Build proxy if we don't have one
19    let before = await proxyRegistry.proxies(wallet.address);
20    await proxyRegistry.build({ gasLimit: 1500000 });
21    const after = await proxyRegistry.proxies(wallet.address);
22
23    expect(before).to.equal("0x0000000000000000000000000000000000000000");
24    expect(after).not.to.equal("0x0000000000000000000000000000000000000000");
25  });
26 }

```

Listing 5.2: Proxy Creation Test

As its name says, the function builds and deploys a personal DSPProxy on Maker's Proxy Registry. We test this by comparing if the proxy's address before execution is equal to a blank address, and that it has been assigned to some new address after the execution. Note that other packages, such as @studleydefi/money-legos/maker and ethers are used here as well for Ethereum and Maker related functionality.

## 5.4 Challenges

One of the major challenges of the project has always been with the Maker protocol. Although, Maker without a doubt is one of the most open, transparent financial system in the world, the protocol has suffered immensely from its language being written in a way that is nearly inaccessible. From choosing to rename common concepts to 3-4 variable names to having almost no comments in their source code, Maker's repositories are a nightmare for developers to delve into. The issue has been widely known and discussed in the DeFi community, and has been addressed in <https://github.com/alexvansande/MuchClearerDAI>, a project aiming to clarify Maker's logic.

The following is an example of issues that we encountered during our project's implementation. Listing 5.3 shows an internal logic within the Maker Vat's repository with useful comments from the MuchClearerDai repository.

```

1  // --- CDP Manipulation ---
2  function frob(bytes32 i, address u, address v, address w, int dink, int dart)
3    external note {
4      // system is isLive
5      require(isLive == 1, "Vault/not-isLive");
6
7      Urn memory urn = urns[i][u];
8      CollateralType memory collateralType = collateralTypes[i];
9
10     // collateralType has been initialised
11     require(collateralType.rate != 0, "Vault/collateralType-not-init");
12
13     urn.lockedCollateral = add(urn.lockedCollateral, dink);
14     urn.normalisedDebt = add(urn.normalisedDebt, dart);
15     collateralType.TotalDebt = add(collateralType.TotalDebt, dart);
16
17     int dtab = mul(collateralType.rate, dart);
18     uint tab = mul(collateralType.rate, urn.normalisedDebt);
19     totalDaiIssued = add(totalDaiIssued, dtab);
20
21     // either totalDaiIssued has decreased, or totalDaiIssued ceilings are not
22     // exceeded
23     require(either(dart <= 0, both(mul(collateralType.TotalDebt, collateralType
24       .rate) <= collateralType.debtCeiling, totalDaiIssued <=
25       TotalDebtCeiling)), "Vault/ceiling-exceeded");
26
27     // urn is either less risky than before, or it is safe

```

```

24     require(either(both(dart <= 0, dink >= 0), tab <= mul(urn.lockedCollateral,
25         collateralType.maxDAIPerCollateral)), "Vault/not-safe");
26
27     // urn is either more safe, or the owner consents
28     require(either(both(dart <= 0, dink >= 0), wish(u, msg.sender)), "Vault/not
29         -allowed-u");
30
31     // collateral src consents
32     require(either(dink <= 0, wish(v, msg.sender)), "Vault/not-allowed-v");
33     // totalDAIissued dst consents
34     require(either(dart >= 0, wish(w, msg.sender)), "Vault/not-allowed-w");
35
36     // urn has no totalDAIissued, or a non-dusty amount
37     require(either(urn.normalisedDebt == 0, tab >= collateralType.debtFloor), "
38         Vault/debtFloor");
39
40     tokenCollateral[i][v] = sub(tokenCollateral[i][v], dink);
41     dai[w] = add(dai[w], dtab);
42
43     urns[i][u] = urn;
44     collateralTypes[i] = collateralType;
45 }

```

Listing 5.3: frob function in Vat.sol

The frob function is used to modify a Vault. We can notice that the issue has risen in the function declaration's parameters: address u, address v, and address w. The naming of these variables are nearly arbitrary, hence the logic is very difficult to follow along. Having spent a considerable amount of time, we discover that the Vault belongs to user u, is using *gem* from user v, and is creating DAI for user w. In addition, int dink and int dart are as nearly as confusing. Fortunately, some vocabularies are roughly described in the MakerDAO's glossary (<https://docs.makerdao.com/smart-contract-modules/core-module/vat-detailed-documentation>) with dink meaning "change in collateral" and dart meaning "change in debt". With this in mind, a more serious problem emerges when an error in the transaction has occurred but no useful logs are presented in the response message. Listing 5.4 shows one of the mentioned error message.

```

1 Error: VM Exception while processing transaction: revert
2 at getResult (/home/paris/DeFi/my-contract/node_modules/ethers/providers/json-rpc-
3 provider.js:40:21)
4 at exports.XMLHttpRequest.request.onreadystatechange (/home/paris/DeFi/my-contract/
5 node_modules/ethers/units/web.js:111:30)
6 at exports.XMLHttpRequest.dispatchEvent (/home/paris/DeFi/my-contract/node_modules/
7 xmlhttprequest/lib/XMLHttpRequest.js:591:25)
8 at setState (/home/paris/DeFi/my-contract/node_modules/xmlhttprequest/lib/
9 XMLHttpRequest.js:610:14)
10 at IncomingMessage.<anonymous> (/home/paris/DeFi/my-contract/node_modules/
11 xmlhttprequest/lib/XMLHttpRequest.js:447:13)
12 at IncomingMessage.emit (events.js:228:7)
13 at endReadableNT (_stream_readable.js:1185:12)
14 at processTicksAndRejections (internal/process/task_queues.js:81:21) {
15   code: -32000,
16   data: {
17     '0x96402dfdcbb396ddfd876975cef72b891b176fc4a7c5f7af83eedbddd6898f79': { error: '
18       revert', program_counter: 948, return: '0x' },
19     stack: 'o: VM Exception while processing transaction: revert\n' +
20       ' at Function.o.fromResults (/home/paris/.npm-global/lib/node_modules/ganache-cli/
21       build/ganache-core.node.cli.js:2:149830)\n' +
22       ' at v.A.processBlock (/home/paris/.npm-global/lib/node_modules/ganache-cli/build/
23       ganache-core.node.cli.js:17:130171)\n' +
24       ' at runMicrotasks (<anonymous>)\n' +
25       ' at processTicksAndRejections (internal/process/task_queues.js:94:5)',
26     name: 'o'
27   },
28   url: 'http://localhost:8545',
29   body: '{"method":"eth_sendRawTransaction","params":["0
30     xf9018b028504a817c800832625a094464eccc1f92f2d3b4f622e9a493070ed04a3cee380b901241cfff79cd0000000000
31     ","id":59,"jsonrpc":"2.0"]}',
32   responseText: '{"id":59,"jsonrpc":"2.0","result":"0
33     x96402dfdcbb396ddfd876975cef72b891b176fc4a7c5f7af83eedbddd6898f79","error":{"
34     message":"VM Exception while processing transaction: revert","code":-32000,"
35     data":{"0x96402dfdcbb396ddfd876975cef72b891b176fc4a7c5f7af83eedbddd6898f79":{"

```



```
error":"revert","program_counter":948,"return":"","stack":""o: VM Exception  
while processing transaction: revert\\n at Function.o.fromResults (/home/paris/.npm-global/lib/node_modules/ganache-cli/build/ganache-core.node.cli.js  
:2:149830)\\n at v.A.processBlock (/home/paris/.npm-global/lib/node_modules/  
ganache-cli/build/ganache-core.node.cli.js:17:130171)\\n at runMicrotasks (<  
anonymous>)\\n at processTicksAndRejections (internal/process/task_queues.js  
:94:5)","name":"'o'"}'}',  
  
transaction: {  
nonce: 2,  
gasPrice: BigNumber { _hex: '0x04a817c800' },  
gasLimit: BigNumber { _hex: '0x2625a0' },  
to: '0x464EccC1F92F2D3B4f622e9a493070ED04a3cee3',  
value: BigNumber { _hex: '0x00' },  
data: '0  
x1cff79cd000000000000000000000005de461a0f2b0de4cceaf2b607cfe083782aa11170000000000000000000000'  
,  
chainId: 1,  
v: 37,  
r: '0xf067e5eeefde4db9a366167c6d36750e4fad1be41a5ad1d597a16d2ac38629c1a',  
s: '0x552d7b2b4e0d7acb9e9fa56b6c9ae18e37f658a2980514ede6d1ecce6f0baab7',  
from: '0x110Ae5e13b67ad64B20131DD9f7F00f6bdcd98Ca',  
hash: '0x96402dfdcbb396ddfd876975cef72b891b176fc4a7c5f7af83eedbdd6898f79',  
},  
transactionHash: '0  
x96402dfdcbb396ddfd876975cef72b891b176fc4a7c5f7af83eedbdd6898f79'  
}  
}
```

Listing 5.4: Error message from calling frob

We must find another way to debug this and one of the approaches we adopt is via Tenderly. Unfortunately, this leads to our second problem which is the conflict in the tools we used. Since Buidler is a new tool in the ecosystem, Tenderly is still in the process of integrating the framework into its stack. This means that we cannot simply import our contracts onto the debugger, but instead we must import the deployed contracts via Etherscan. The conflict adds a lot of load to our workflow since we not only have to deploy the contract to Kovan, but also have to verify and publish said contracts. The verification process yet gives us another error shown in Listing 5.5.

```

1 Compiler debug log:
2 Error! Unable to generate Contract ByteCode and ABI
3 Found the following ContractName(s) in source code: DssProxyActionsBase.sol:
    DssProxyActionsBase, IDssProxyActions.sol:DaiJoinLike, IDssProxyActions.sol:
    DssProxyActionsCommon, IDssProxyActions.sol:EndLike, IDssProxyActions.sol:
    GNTJoinLike, IDssProxyActions.sol:GemJoinLike, IDssProxyActions.sol:GemLike,
    IDssProxyActions.sol:HopeLike, IDssProxyActions.sol:IDssProxyActions,
    IDssProxyActions.sol:JugLike, IDssProxyActions.sol:ManagerLike,
    IDssProxyActions.sol:PotLike, IDssProxyActions.sol:ProxyLike, IDssProxyActions.
    sol:ProxyRegistryLike, IDssProxyActions.sol:VatLike, MyCustomVaultManager.sol:
    MyCustomVaultManager
4 But we were unable to locate a matching bytecode (err_code_2)

```

Listing 5.5: Error message from Etherscan’s contract verification

This is an uncommon error since the same source code is being used to generate the contract's bytecode and ABI. So we solve this by deploying the contract via Remix, an on-the-web Ethereum IDE. This, again, gave us another error with Remix not being able to compile our Solidity code. We later found out that there is a new bug in deploying contracts via Chrome, so alternatively we switched to Firefox and deploy the contract shown in Figure 5.1. With all of this done, we can finally import the contracts onto Tenderly debugger and observe the error message as shown in Figure 5.2 .

Going through all of this, each step is very time consuming, especially, in deploying a new DSProxy. This is due to the fact that we cannot merely debug on a locally spawned Ethereum network or even on a forked network. We need to connect with a live network, deploy and verify our contracts online, and then import the transaction back to see its result. Each step can take up to 8 minutes and the entire debugging process can take up to more than 20 minutes to view an error message from one small change.

Our last challenge lies with the agent negotiation via Libp2p. Since Libp2p’s main target language is Go, the JavaScript module, being a secondary priority, has suffered from the lack of accurate



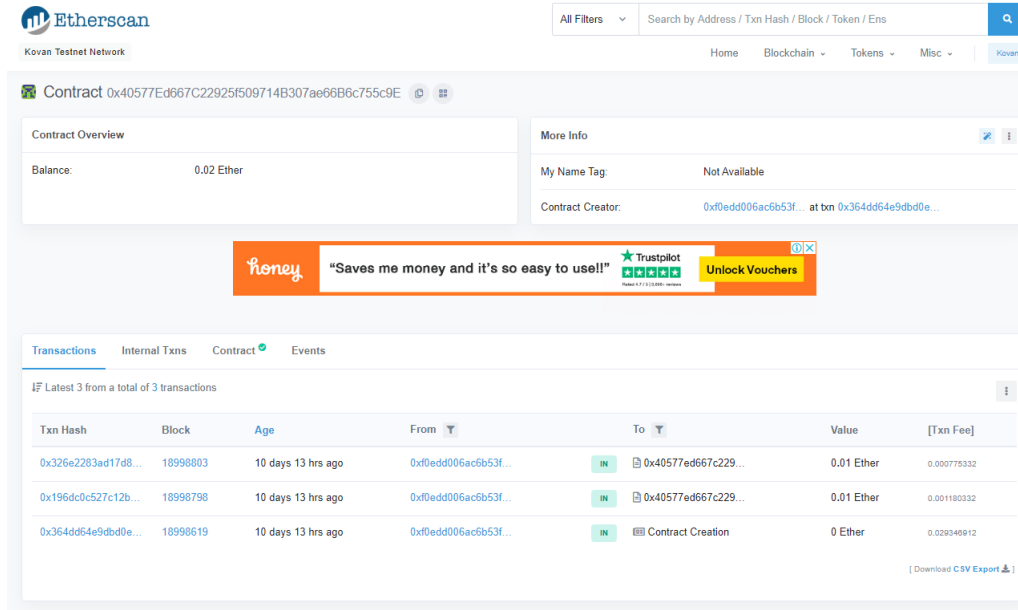


Figure 5.1: the Verified Contract on Etherscan

documentation and examples. For example, Libp2p has removed a functionality for peer to be created using a json object entirely and has frequently changed the way asynchronous execution is used within its dial mechanism. As a result, the package has turned out to be more time consuming than what we expected.

Note that, throughout the project, we have received several help from the DeFi community. Their technical forums, RocketChat, and Discord channels are very active and friendly. Some of their supports are shown in Appendix B.

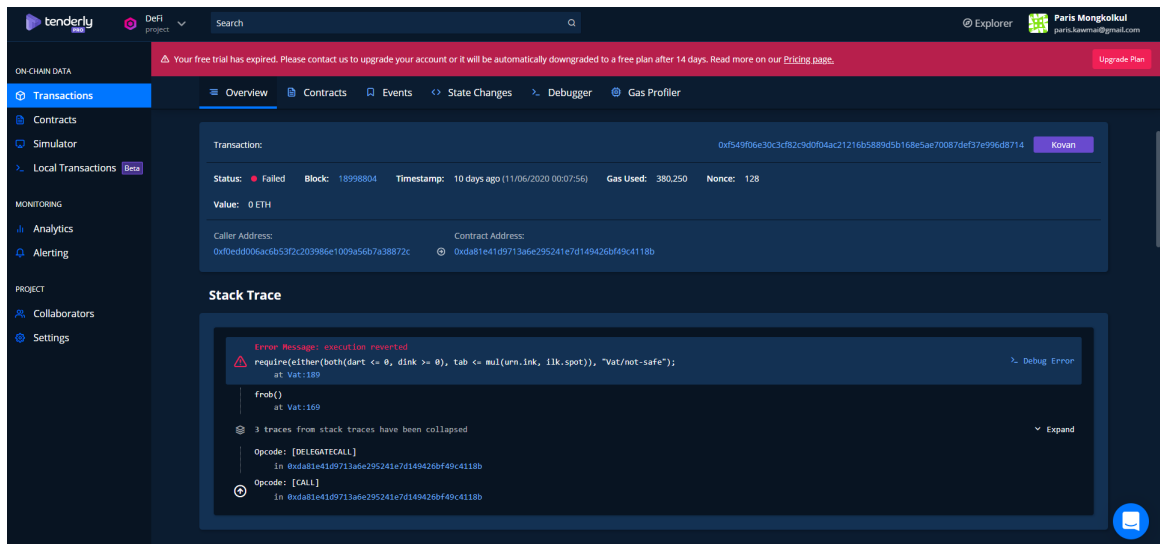


Figure 5.2: The Transaction's Stack Trace from Tenderly

## Chapter 6

# Conclusion

The project introduced a modern and better way for anyone looking to invest in a safe, transparent, and decentralized manner. With more and more data being passed around in the financial world, investors struggle to keep up with the latest news, whether it is a fluctuation in currency or an unexpected pandemic. We have utilized the new-found DeFi protocols to take advantage of the conditional transaction property of smart contracts and the possibility of Ethereum.

With our decentralized financial agents, one can collaborate with others instantaneously without needing any trusted third parties. The negotiation takes place inside the boundary of our Agreement Calculus before their mutual agreement gets encoded into a smart contract and deploys on a live network. This way, the agents can pool their resources together to achieve a better result while assuring that their agreement will not be violated by anyone.

We advocate that any future negotiation system should be implemented through smart contracts, since they are self-executing and exist across a distributed, decentralised blockchain network. These properties allow anyone with an internet to verify any transactions. In addition, since Distributed Ledger Technology is a fairly new invention, there is a significant lack of documentation and user examples. Having reached out to Maker, StudyDefi, Tenderly, and Libp2p, we truly advice DeFi developers to get involved in the community whether via an online channel, a forum, or a simple email.

## 6.1 Future Work

Only [6.1.1](#) was the original extension of this project. Other extensions are beyond the scope of the time we have and can be expected as a future work.

### 6.1.1 Integration with the Risk Modelling Project

As mentioned in [1.2](#), there exists another project that adopts machine learning and mathematical approaches to build an investment model. The model will produce inputs for our agents to tell them how to behave e.g. how much and when to invest. The integration of the investment strategies into our agents would surely make them more intelligent and it will definitely be interesting to see how they interact with each other. In addition, it will be quite revolutionary if these agents can learn or trade their investment strategies among each other. The process will allow the best investment strategy to become popular while eliminating underperforming ones. This functionality will be the perfect solution to the possible abuse of our system mentioned in [Section 5.2](#).

### 6.1.2 Agreement Compiler

Currently, a code template is used to translate our Agreement Calculus to a Solidity smart contract. A compiler, interpreter, or some other tool would surely improve the stability of the process. This would provide a faster, more efficient encoding of the contract while minimizing opportunities for failures.

### 6.1.3 Contract Verification

We should be able to allow agents to verify the terms of the contract through a de-compilation. This ensures that, semantically, each agent can verify that the terms of the agreement are correctly encoded. Further, this allows agents to purely rely on the deployed contract without the need to trust each other. This would require a study in context-free grammar and its application [23].

### 6.1.4 Integration with other DeFi protocols

The only protocol our agent supports is Maker. However, it should not be difficult to further integrate other DeFi protocols because our Agreement Calculus can be used extensively with the majority of protocols out there. Thus, little to no effort will needed to be made on the negotiation side. The target protocol has to be studied in order to understand its logic and how it operates before implementing the smart contracts e.g. Does it use a proxy? How do you interact with its contracts? Can transaction be triggered automatically without users interaction?

### 6.1.5 Agents

Even though our agents can successfully collaborate and invest in Maker protocol, there are limitations in what the agents can do. First, future work can be carried out to better improve the living environment of the agents. Rather than having the agents operating locally, we can deploy the agents online or on a more durable machine/vm. So in the case that our machine goes down unexpectedly, there will be no negative consequences to the agents. Second, our agents can be extended with the functionality to communicate with other agents across networks. This will be a more realistic usage of our system since each agent will belong to different users and on different devices. In addition, we can implement some sort of frontend user interface for users to view the agent's status, strategy, and historical data e.g. how much profit/loss it made, frequency of investments, who are the collaborators. A user verification functionality can be implemented, so a user has to grant access to received proposal before agreement is made. This will somewhat defeat the purpose of the agent being autonomous and instantaneous, but it will grant users more control over their asset, providing a more friendly process.

### 6.1.6 Deployment to Mainnet

Throughout the project, our environment has been either a locally spawned node, a forked public network, or the Kovan testnet. It will be a great achievement to deploy these agents on Mainnet which is where actual transactions take place on a distributed ledger. With real money and assets involved, a considerable amount of effort and testings must be put in to ensure that our agents are stable and secure. In particular, the agreed contract has to be verified and published on the network so anyone can verify any and every transaction. Backup protocol will also have to be implemented to cover unexpected events e.g. the corresponding DeFi protocol shuts down, what to do with the money deposited in the smart contracts if one agent decides to sign off.

# Appendix A

## Online resources

The following provides locations of online papers, documentations, and articles that have extensively contributed to the project.

**Working with DSProxy:** <https://github.com/makerdao/developerguides/blob/master/devtools/working-with-dsproxy/working-with-dsproxy.md>

**StudyDefi documentation:** <https://studydefi.com/>

**Maker documentation:** <https://docs.makerdao.com/>


**Maker whitepaper:** <https://makerdao.com/en/whitepaper/abstract>


**MuchClearerDai:** <https://github.com/alexvansande/MuchClearerDAI/tree/master/src>

**SimpleAsWater, blockchain community for Libp2p help:** <https://simpleaswater.com/>

**IPFS secure channel:** <https://github.com/auditdrivencrypto/secure-channel/blob/master/prior-art.md#ipfs-secure-channel>


# Community Help

 **gbalabasquer** @online  
I can see that execution error happend here <https://kovan.etherscan.io/bx/0xf549f06e30c3cf82c9d0f04ac21216b5889d5b168e5ae70087def37e996d8714>

 **Paris** 12:16 AM  
so yes, the failure is in calling `[vat.frob]`: <https://kovan.etherscan.io/vmtrace?txhash=0xf549f06e30c3cf82c9d0f04ac21216b5889d5b168e5ae70087def37e996d8714&type=parity>

Error Message: execution reverted  
`require(either(both(dart <= 0, dink >= 0), tab <= mul(urm.lnk, ilk.spot)), "Vat/not-safe");`

I'm getting this from Tenderly debugger

 **gbalabasquer** 12:17 AM  
basically you do not have enough collateral locked

if you analyze the calldata of the subcall that is failing:


```

0x76088703 (vat.frob)
4554482d4100000000000000000000000000000000000000000000000000000000
000000000000000000000000aa360b89fdd37c4fbcee99f819b1459f077c999
000000000000000000000000aa360b89fdd37c4fbcee99f819b1459f077c999
000000000000000000000000aa360b89fdd37c4fbcee99f819b1459f077c999
000000000000000000000000aa360b89fdd37c4fbcee99f819b1459f077c999
0000000000000000000000000000000000000000000000000000000000000000 -> collateral
0000000000000000000000000000000000000000000000000000000000000000 -> dai
    
```


so basically there is not collateral being added

I do not see your proxy: `0xda81e41d9713a6e295241e7d149426bf49c4118b`  
has ETH balance

that is why is not locking anything

 **Paris** 12:23 AM  
My guess is that the `ethJoin_join` function doesn't work

I can see that the contract has some balance in it from the initial deposit

 **gbalabasquer** 12:24 AM  
the proxy doesn't have ETH balance

ah you mean you are sending ETH in the same tx

let me check

hmmm no

44



Do you want live notifications when people reply to your posts? [Enable Notifications](#)

## Error Calling frob using own proxy

■ Developers



Paris

16d

Hi,

I've run into a problem when I tried calling my custom function on my deployed proxy contract. The function is supposed to open, lock ETH, and draw Dai via the Maker contracts.

Instead of directly transferring the ETH from the caller to the Maker Contract, my custom function is supposed to use the balance of the proxy contract (which I've paid beforehand).

```
function openAndLockETH(
  address manager,
  address jug,
  address ethJoin,
  address daiJoin,
  uint256 wadD
) public payable {
  // Opens ETH-A CDP
  bytes32 ilk = bytes32("ETH-A");
  uint256 cdp = open(manager, ilk, address(this));

  address urn = ManagerLike(manager).urns(cdp);
  address vat = ManagerLike(manager).vat();
  // Receives ETH amount, converts it to WETH and joins it into the vat
  ethJoin_join(ethJoin, urn);
  // Locks WETH amount into the CDP and generates debt
  frob(
    manager,
    cdp,
    toInt(address(this).balance),
    _getDrawDart(vat, jug, urn, ilk, wadD)
  );
}
```

I can see that the code breaks on the frob call. The only thing I changed is its parameter from

Figure B.2: Custom Contract Interaction via Maker's Developers forum

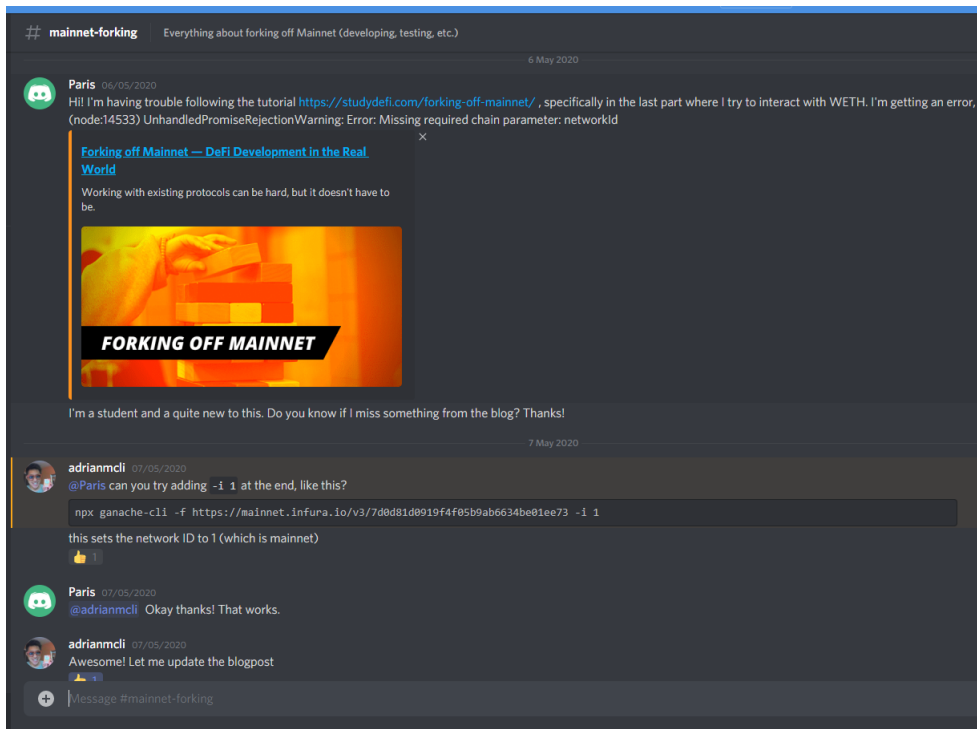


Figure B.3: Forking off mainnet via StudyDeFi's Discord

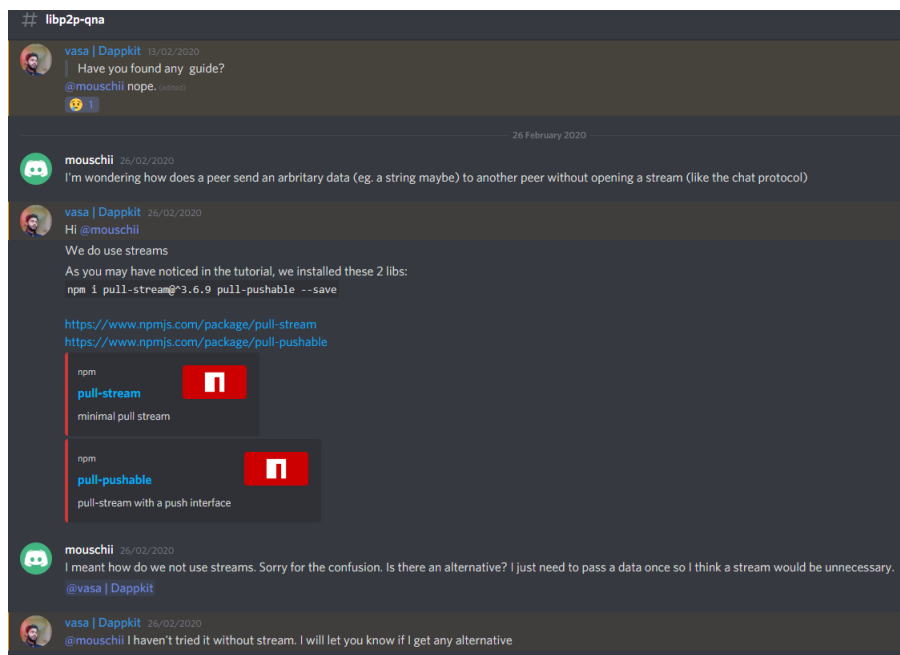


Figure B.4: Libp2p's communication via SimpleAsWater's Discord



# Bibliography

- [1] Christian Sillaber and Bernhard Watzl. Life cycle of smart contracts in blockchain ecosystems. *Datenschutz und Datensicherheit-DuD*, 41(8):497–500, 2017.
- [2] Lewis Gudgeon, Daniel Perez, Dominik Harz, Arthur Gervais, and Benjamin Livshits. The decentralized financial crisis: Attacking defi, 2020.
- [3] Dominik Harz, Lewis Gudgeon, Arthur Gervais, and William J Knottenbelt. Balance: Dynamic adjustment of cryptocurrency deposits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1485–1502, 2019.
- [4] Massimo Bartoletti and Roberto Zunino. Bitml: a calculus for bitcoin smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 83–100, 2018.
- [5] Michael Wooldridge. *An introduction to multiagent systems*. John Wiley & Sons, 2009.
- [6] Nicholas R Jennings, Peyman Faratin, Alessio R Lomuscio, Simon Parsons, Carles Sierra, and Michael Wooldridge. Automated negotiation: prospects, methods and challenges. *International Journal of Group Decision and Negotiation*, 10(2):199–215, 2001.
- [7] Michael R Genesereth. Software agents michael r. genesereth logic group computer science department stanford university. 1994.
- [8] Katia Potiron, Amal El Fallah Seghrouchni, and Patrick Taillibert. Multi-agent system properties. In *From Fault Classification to Fault Tolerance for Multi-Agent Systems*, pages 5–10. Springer, 2013.
- [9] Shaheen Fatima, Sarit Kraus, and Michael Wooldridge. *Principles of automated negotiation*. Cambridge University Press, 2014.
- [10] Reid G Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on computers*, (12):1104–1113, 1980.
- [11] Martin L Weitzman. Efficient incentive contracts. *The Quarterly Journal of Economics*, 94(4):719–730, 1980.
- [12] Federico Bergenti, Giovanni Rimassa, Matteo Somacher, and Luis Miguel Botelho. A fipa compliant goal delegation protocol. In *Communication in Multiagent Systems*, pages 223–238. Springer, 2003.
- [13] James Hendler and Deborah L McGuinness. The darpa agent markup language. *IEEE Intelligent systems*, 15(6):67–73, 2000.
- [14] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.
- [15] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37), 2014.
- [16] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 653–663, 2018.

- [17] Anastasia Mavridou and Aron Laszka. Designing secure ethereum smart contracts: A finite state machine based approach. In *International Conference on Financial Cryptography and Data Security*, pages 523–540. Springer, 2018.
- [18] Chris Dannen. *Introducing Ethereum and Solidity*, volume 1. Springer, 2017.
- [19] Aleksander Berentsen and Fabian Schär. Stablecoins: The quest for a low-volatility cryptocurrency. 2019.
- [20] The Maker Team. The dai stablecoin system. 2017.
- [21] Mikael Brockman Nikolai Mushegian, Daniel Brockman. Maker reference implementation. 2018.
- [22] Martin J Osborne and Ariel Rubinstein. *A course in game theory*. MIT press, 1994.
- [23] Sophia Drossopoulou, James Noble, Julian Mackay, and Susan Eisenbach. Holistic specifications for robust programs, 2020.