Published with GitBook

Dharma: A Generic Protocol for Tokenized Debt Issuance

Version 2.0

Nadav Hollander -- nadav@dharma.io

B.S. in Computer Science -- Stanford University '17

Abstract

Dharma is a protocol that enables decentralized origination, underwriting, issuance, and administration of tokenized debt assets in a highly generic and unopinionated construction. The protocol aims to build a common informational interface by which exchanges, brokerages, and traders can reasonably price a tokenized debt's default risk without having to rely on a singular centralized data broker. The Dharma debt issuance scheme leverages two classes of utility players that compete in distinct marketplaces for compensatory fees -- underwriters and relayers. The former are trusted originators and assessors of debtor default risk, and the latter facilitate the funding and issuance of debts in a trustless manner. Both can be empirically evaluated on historical asset performance, and, as such, markets have lucid signals with which to evaluate the default risk of tokenized debts attested to by any given underwriter or relayer. The Dharma debt issuance process only requires one on-chain transaction to execute, and is heavily inspired by the mechanics of the 0x Protocol.

Introduction

Claim: An under-recognized advantage of blockchains is that they necessarily engender the creation of universal, permissionless standards for tokenized asset classes.

Token sale crowd-funds have, as of the time of this paper's writing, raised over $2B in 2017 alone. If this proves anything, it is that there is clearly an under-satisfied market demand for crowd-sale offerings that compensate retail investors with assets that have equity-like[1] risk profiles. Equity crowd-funding mechanisms, however, pre-date the ICO phenomenon significantly -- so how does one explain the sudden burst of interest? If equity crowd-sales have been technically feasibly and in production for years, what aspect of the token sale ecosystem did Ethereum uniquely enable from a technological perspective? I posit that the answer is jarringly simple: the ERC20 token standard created the common rails on top of which a diverse ecosystem of secondary markets for tokens could be built in a permission-less and interoperable manner. Judged on investor liquidity alone, token crowd-sales are a step-function improvement over the status quo of equity fundraising.

In the existing financial system, however, the sum total capital raised in equity fundraises is paltry in comparison to its big brother in the world of debt fundraising. Debt markets, however, remain opaque and proprietary; executing a debt fundraise, be it in a public offering or to private investors, is as bespoke and inefficient as executing an equity fundraise is. Consider the following: why not apply the token-sale model to debt fundraising?

As a toy example, a corporation could, hypothetically, issue a bond as an "ICO for debt," so to speak, and represent bond ownership with ERC20 tokens to be sold in a token crowd-fund. Ostensibly, a world where debt assets were represented by a permission-less, universal token standard would, similarly, be a step-function improvement over the status quo in terms of liquidity and transparency. In order for liquid secondary markets to crop up in a similarly permission-less manner, however, investors would need a standardized mechanism of pricing tokenized debt assets.

Whereas equity-like tokens are tied in value to branded protocols, projects, or entities, debt-like tokens are tied in value to empirical financial obligations from counter-parties that are often anonymized. The ERC20 standard, therefore, falls short of capturing the obligatory semantics of a debt asset insofar as it does not provide a means of:

1 Retrieving machine-readable debt-specific metadata (e.g. principal, interest rates) associated with the assets

2 Retrieving a history of payments between debtors and creditors in a debt asset's terms

3 Pricing default risk into the debt asset's value

Dharma protocol intends to bridge this gap and provide a permission-less, generic mechanism by which debt assets of flexible type can be issued, sold, administered, and priced without having to rely on centralized data brokers of any kind.

Architecture

Dharma protocol defines a procedure for issuing, funding, administering, and trading debt assets using a set of smart contracts, keeper marketplaces, and standardized interfaces elaborated on below. Dharma is heavily inspired in design by the 0x decentralized exchange protocol[2], using 0x Broadcast Order Messages as the blueprint on which we base Dharma Debt Orders, their analogous equivalent in Dharma protocol. This mechanism will be explicitly formalized further in the paper. The protocol is designed to support EVM blockchains, but could ostensibly be extended to support any blockchain with requisite generic smart contract functionality. First, we solidify some terminology.

Agents

We define agents as the end-consumers of the protocol -- i.e. entities looking to borrow or lend crypto-assets. Those entities, be they people, corporations, contracts, or automata, unsurprisingly fall into two categories:

1 **Debtor** - a party in a debt transaction who is borrowing an asset and owes a creditor some agreed upon value.

2 **Creditor** - a party in a debt transaction who is lending an asset is owed some agreed upon value by a debtor.

Keeper Marketplaces

We adopt the catchall term keepers[3] to encompass the utility players who provide value-added services to the network and compete in their respective marketplaces for compensatory fees.

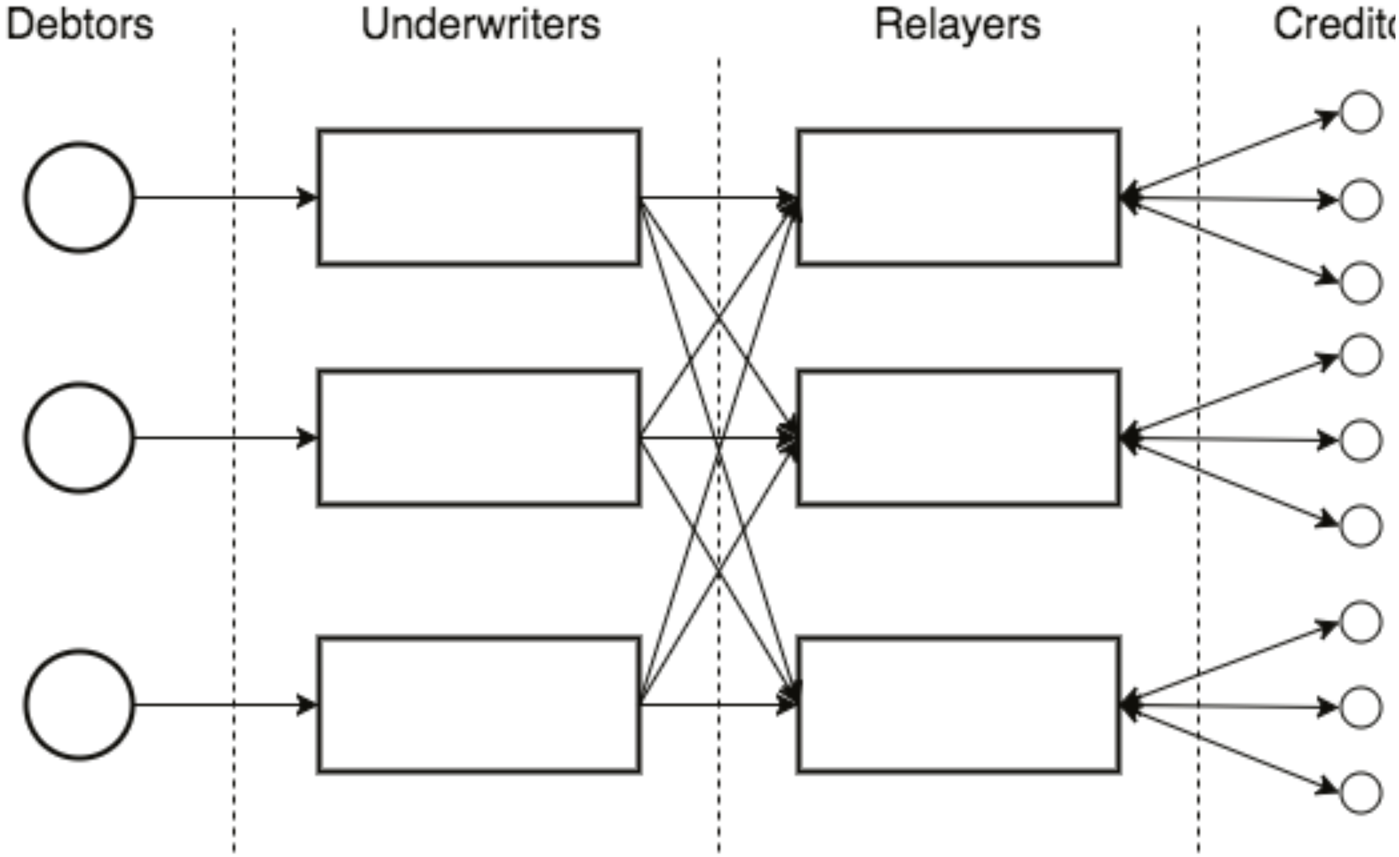


Figure 1: Flow of interactions between different agents and keepers

1 **Underwriters**[4]

In traditional debt markets, underwriters are entities that collect fees for administering the public issuance of debt and pricing borrower default risk into the asset. In Dharma protocol, this definition is expanded and formalized. **An underwriter is a *trusted* entity that collects market-determined fees for performing the following functions:**

- Originating a debt order from a borrower
- Determining and negotiating the terms of the debt (i.e. term length, interest, amortization) with the potential debtor
- Cryptographically committing to the likelihood they ascribe to that debt relationship ending in default (process described in detail under Specification)
- Administering the debt order's funding by forwarding it to any number of relayers.
- Servicing the debt -- i.e. doing everything in the underwriter's reasonable power to ensure timely repayment according to the agreed upon terms
- In the case of defaults or delinquencies, collecting on collateral (if debt is secured) or the individual's assets via legal mechanisms and passing collected proceeds to investors

This is not particularly out of band with what most online lenders do in their day-to-day underwriting and servicing operations. We foresee Dharma protocol facilitating an alternative, cheaper route for aspiring online lending platforms to bootstrap their operations and earn similar margins as they would in the status quo by becoming an underwriter -- all-the-while never holding balance sheet risk and avoiding the upfront time and capital costs associated with raising the requisite debt vehicles from traditional investors.

Example: Alice has a novel thesis on how to originate, underwrite, and service loans to aspiring ZCash miners who need significant upfront capital to buy GPUs on bulk. In lieu of knocking on the doors of traditional fixed-income investors, Alice decides to become an underwriter in Dharma protocol. She obtains the necessary lending licenses, sets up a website advertising lending services for miners, and drums up hype in the ZCash community for her credit product. When borrowers come to her site, their creditworthiness is automatically scored by Alice's proprietary technology and they are presented with the terms of the loan, as determined by Alice. Upon acceptance of the terms, Alice cryptographically attests to the borrower's likelihood of default, forwards the signed debt order to a relayer, and, upon the loan's funding, collects her desired fee. The entire flow of funds is transparently auditable on-chain, and Alice's competence in servicing and collecting on the debt can be empirically determined ex post facto.

1 **Relayers**

Relayers in Dharma protocol perform an analogous function to relayers in the 0x Protocol -- namely, relayers aggregate signed debt order messages and, for an agreed upon fee, host the messages in a centralized order book and provide retail investors with the ability to invest in the requested debt orders by filling the signed debt orders. Note that, similarly to the 0x relaying mechanism, Dharma Protocol relayers need not hold any agent's tokens -- they simply provide a mechanism for creditors to browse through aggregated signed debt order messages, which creditors can use to trustlessly issue themselves debt tokens in exchange for the requested principal via client-side contract interactions (this mechanism is specified later in this paper). The primary differences between relayers in Dharma protocol and 0x are:

1 Dharma protocol relayers are not hosting a secondary market order book, but rather, an order book containing requests for debts that have yet to be issued

2 Dharma protocol relayers provide creditors with signed debt-specific metadata associated with the debt order messages and their accompanying underwriter so that they can make informed investment decisions about the risk profile of a given debt order.

3 Dharma protocol relayers do not freely allow any anonymous party to publish signed debt orders on to their order book, and use their discretion to only accept signed debt orders from known, trusted underwriters.

Example: Bob wants to build a retail loan investor portal through which users can invest in a variety of debt assets -- a Kayak for peer-to-peer loans, if you will. Bob becomes a Dharma protocol relayer by setting up an online order book, building a retail investment platform, and allowing investors to browse through debt requests and examine associated data pertaining to the debtors' credit worthiness and the identity of the backing underwriters. Since Bob has seen that the empirical historical performance of Alice's attested assets has been in line with her predictions and knows that Alice's company is a publicly trusted and regulated entity, Bob allows Alice to broadcast signed debt orders onto his order book. When a debt order is filled on his platform, Bob is paid out a fee stipulated in the signed debt order.

Contracts

Dharma protocol leverages several contracts deployed on the Ethereum network. We highlight a few that are particularly relevant to understanding the prtocol's mechanics.

1 **Debt Kernel**

The debt kernel is a simple smart contract that governs all business logic associated with minting non-fungible debt tokens, maintaining mappings between debt tokens and their associated term contracts, routing repayments from debtors to creditors, and routing fees to underwriters and relayers. These mechanisms are easier to define within the context of the debt lifecycle, and are extensively elaborated on in the below specification.

1 **Terms Contract(s)**

Terms contracts are Ethereum smart contracts that are the means by which debtors and creditors agree upon a common, deterministically defined set of repayment terms. By extension, terms contracts expose a standard interface of methods for both registering debtor repayments, and programmatically querying the repayment status of the debt asset during and after the loan's term. A single terms contract can be reused for any number of debt agreements that adhere to its repayment terms -- for instance, a terms contract defining a simple compounded interest repayment scheme can be committed to by any number of debtors and creditors. The exact interface for this is defined within the specification below.

Note: An alternative scheme for committing to loan terms would be to commit to a standardized schema of plaintext loan terms (a la Ricardian contracts[5]) on chain and assess loan repayment off-chain in client applications. We deliberately opt not to pursue this scheme for several reasons. Primarily, explicitly defining a universal schema for debt terms inherently limits the range of debt asset types that can be issued in the protocol, while a generic interface for terms contracts opens the door for an infinite array of debt term arrangements. Moreover, committing to a terms contract on-chain removes any ambiguity from the evaluation of a loan's repayment status -- the contract is a single, programmatic, and immutable source of truth that is queryable by both contracts and clients. Finally, having an on-chain provider of repayment status greatly simplifies *the mechanisms by which on-chain collateralized debt agreements can be structured and collected on in cases of default.*

1 **Repayment Router**

The repayment router contract is constructed to trustlessly route repayments from debtors to debt agreement beneficiaries (i.e. owners of the debt tokens). Additionally, the repayment router acts as a trusted oracle to the Terms Contract associated with any given debt agreement, reporting to it the exact details of each repayment as it occurs. This enables the terms contract to serve as a trustless interface for determining the default status of a debt.

Specification

Overview

The entire debt issuance process occurs synchronously in one on-chain transaction, when a signed debt order message is submitted to the Debt Kernel contract. If the message is valid as per the below specification, the following happen in one transaction:

1 The debtor's adherence to the chosen terms contract and the underwriter's prediction of default likelihood are committed to on-chain.

2 A non-fungible, non-divisible debt token is minted to the creditor and mapped to the above commitment.

3 The principal amount is transferred from the creditor to the debtor (minus fees) and any keepers' fees are similarly transferred from the creditor.

This process is detailed below. First, we formalize the the format of data packets in the protocol.

Message Types

Communications between the different agents and keepers in the protocol are comprised of data packets that we refer to as **Debt Orders**.

Debt Orders

Debt orders are data packets listed by relayers that are the fundamental primitive of Dharma protocol -- submitting a valid debt order to the Debt Kernel triggers the issuance of a debt token and its swap with the requested principal amount. Dharma protocol is agnostic to the means by which creditors, debtors, underwriters, and relayers communicate and transfer debt order packets between one another -- A debt order can have up to 3 ECDSA signatures attached to it -- a debtor's signature, a creditor's signature, and an underwriter's signature.

The payload they sign depends on their role in the transaction -- debtors and creditors are required to sign the hash of the debt order (i.e. the **debt order hash**), while underwriters are required to sign only a subset of the fields in the debt order, which we refer to as the **underwriter commitment**.

Moreover, not all 3 signatures must be attached to the debt order in order to submit it to the Debt Kernel -- if an agent or keeper is not involved in the transaction (i.e. their address in the debt order is null) **or** they *are* involved but are also the party submitting the Debt Order to the Debt Kernel contract, then their signature is not mandated.

The debt order is comprised of the following fields:

Name	**Data Type**	**Description**
`kernelVersion`	`address`	Address of the debt kernel contrac this address will be updated.
`issuanceVersion`	`address`	Address of the repayment router c commitment.
`principalAmount`	`uint256`	Total units of principal token reque
`principalToken`	`address`	Address of the ERC20 token used
`debtor`	`address`	Address of the debtor requesting t
`debtorFee`	`uint256`	Total units of principal token the th principal as fees. Note that the tota and creditor *must* equal the total a underwriter and relayer.
`creditorFee`	`uint256`	Total units of principal token the th principal amount. Note that the tota and creditor *must* equal the total a underwriter and relayer.
`relayer`	`address`	Address of the relayer listing the gi
`relayerFee`	`uint256`	Total units of principal token the re kernel when the debtor-creditor rel
`underwriter`	`address`	Address of the underwriter wishing asset.
`underwriterFee`	`uint256`	Total units of principal token the u kernel when the debtor-creditor rel

`underwriterRiskRating`	`uint32`	The underwriter's assessment of th unit-of-value the debtor is expecte be a value between 0 and 1, encoc understood to have 9 decimal poin represented as `500000000`)
`termsContract`	`address`	Address of the Terms Contract Inte defines the repayment terms of the
`termsContractParameters`	`bytes32`	Data packet of parameters ingeste specific values relevant to the repa rate, etc.)
`expirationTimestamp`	`uint256`	Unix timestamp of the time at whic unfilled.
`salt`	`uint`	A pseudo-random salt value used orders with identical parameters.

Debt Issuance Commitments

A debt issuance commitment is a subset of the debt order data packet that we consider separately in order to define a canonical, unique identifier for any given debt agreement. The debt issuance commitment indicates the debtor's (and underwriter's) desire to mint a non-fungible debt token, where that debt token is to be immutably associated with a pairing `(TC, P)`, `TC` being the address of a deployed terms contract adhering to the Terms Contract Interface (see below) and `P` representing the set of parameters ingested by the contract at `TC`. Moreover, the underwriter commits to a value `R` representing the underwriter's assessment of the average likelihood that the debtor will repay any given unit-of-value he is expected to, as defined by `(TC, P)`. The hash of of this data packet is known as the **issuance hash**, which is used throughout the protocol as the canonical unique identifier of a debt agreement.

A sample schema for a debt issuance commitment follows:

Name	**Data Type**	**Description**
`issuanceVersion`	`address`	Address of the repayment router co commitment.
`debtor`	`address`	Address of the debtor wishing to m
`underwriter`	`address`	Address of the underwriter wishing asset.
`underwriterRiskRating`	`uint32`	The underwriter's assessment of th unit-of-value the debtor is expecte

		be a value between 0 and 1, encoc understood to have 9 decimal poin represented as `500000000`)
`termsContract`	`address`	Address of the Terms Contract Inte defines the repayment terms of the
`termsContractParameters`	`string`	Data packet of parameters ingeste specific values relevant to the repa rate, etc.)
`salt`	`uint`	A pseudo-random salt value used orders with identical parameters.

Debtor/Creditor Commitment Hash

The debtor/creditor commitment hash is the payload signed by a debtor or creditor in order to indicate her consent to the parameters of the debt order. It is comprised of the Keccak 256 hash of the the following subset of the debt order parameters:

Name	Data Type	Description
`kernelVersion`	`address`	Address of the debt kernel contract. Wh address will be updated.
`issuanceHash`	`bytes32`	Hash of the debt issuance commitment UID for the debt agreement.
`principalAmount`	`uint256`	Total units of principal token requested
`principalToken`	`address`	Address of the ERC20 token used for p
`debtorFee`	`uint256`	Total units of principal token the that wi principal as fees.
`creditorFee`	`uint256`	Total units of principal token the the cre amount.
`relayer`	`address`	Address of the relayer listing the given c
`relayerFee`	`uint256`	Total units of principal token the relayer when the debtor-creditor relationship is
`underwriterFee`	`uint256`	Total units of principal token the underv kernel when the debtor-creditor relation
`expirationTimestamp`	`uint256`	Unix timestamp of the time at which thi unfilled.

Underwriter Commitment Hash

The underwriter commitment hash is the payload signed by a underwriter in order to indicate her consent to the parameters of the debt order. The underwriter is given a different subset of parameters to sign so that debtors will not need to request new signatures from the underwriter for each relayer with whom they broadcast their debt order -- we will elaborate on this further below. The underwriter commitment is comprised of the Keccak 256 hash of the the following subset of the debt order parameters:

Name	Data Type	Description
`kernelVersion`	`address`	Address of the debt kernel contract. Wh address will be updated.
`issuanceHash`	`bytes32`	Hash of the debt issuance commitment
`principalAmount`	`uint256`	Total units of principal token requested
`principalToken`	`address`	Address of the ERC20 token used for p
`underwriterFee`	`uint256`	Total units of principal token the underv kernel when the debtor-creditor relation
`expirationTimestamp`	`uint256`	Unix timestamp of the time at which thi unfilled.

All together, a debt order is considered ready for submission if attached to it are:

1 The debtor's ECDSA signature of the debtor/creditor commitment hash. (required unless the debtor is the address submitting the order to the Dharma smart contracts)

2 The underwriter's ECDSA signature of the underwriter commitment hash (if no underwriter is present, or the underwriter is submitting the order to the Dharma smart contracts, this is not required)

3 The creditor's ECDSA signature of the debtor/creditor commitment hash. (required unless the creditor is the address submitting the order to the Dharma smart contracts)

Debt Issuance Process

Any party that possesses a valid debt order with the requisite attached signatures can fill the order by submitting it to the Dharma smart contracts. Submitting the debt order to the Dharma smart contracts kick starts a mechanism, described below, in which a debt agreement token is minted and synchronously swapped with the principal amount. We will refer to arrangements in which a debtor fills a complete debt order as **Debtor-Filler Order Submissions** and arrangements in which a creditor fills a complete order as **Creditor-Filler Order Submissions**.

Creditor-Filler Order Submissions

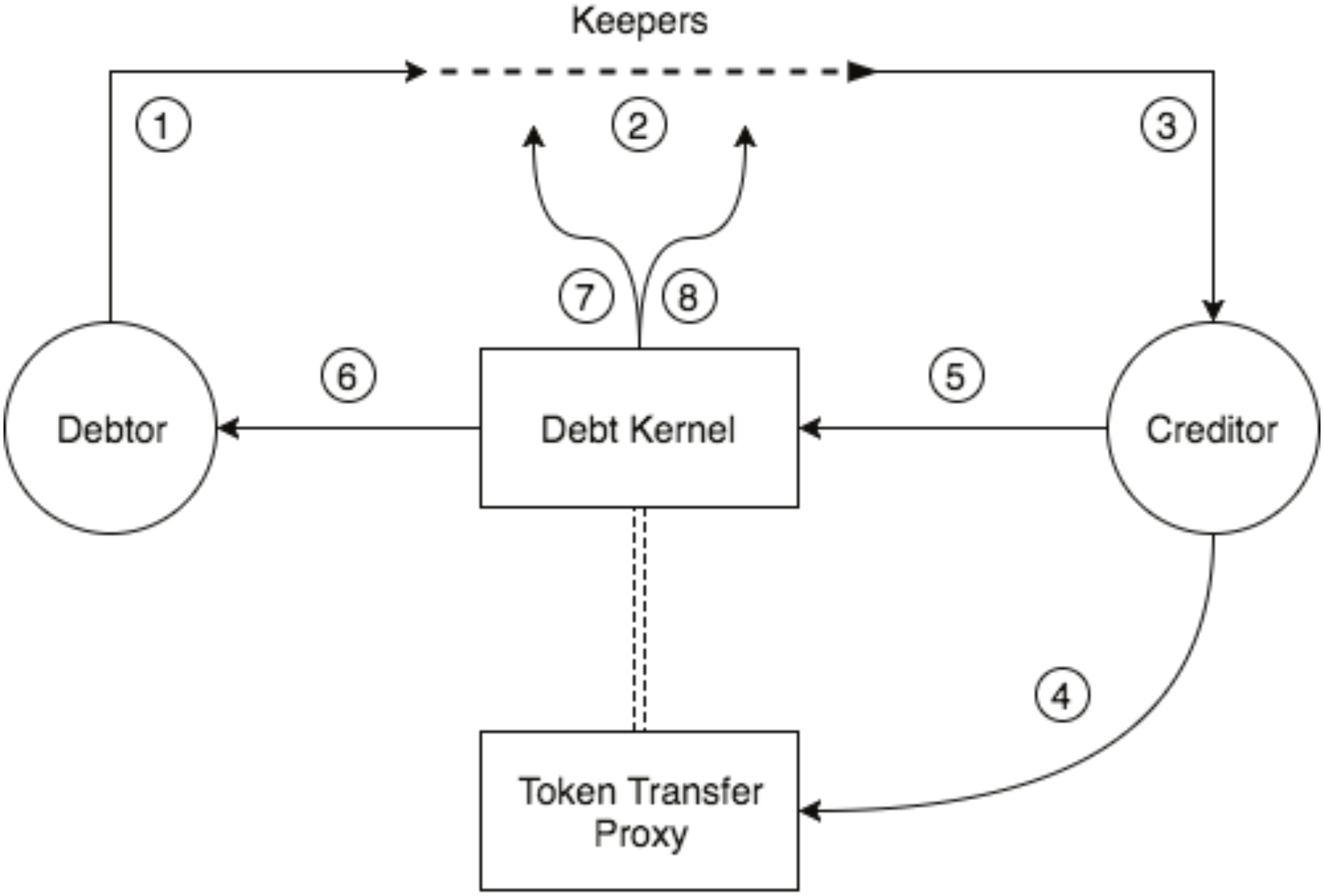


Figure 2: The process of Creditor-Filler Order Submission

The following steps correspond to the circled numbers in the above diagram:

1 Debtor requests loan from an underwriter.

2 Debt Order Handshake (described in detail further) occurs between the debtor, underwriter, and relayer(s), resulting in the relayer listing a valid, complete debt order.

3 Creditor evaluates the terms of the Debt Order on a relayer's public order book.

4 If the creditor wants to fill the order, he first grants the token transfer proxy an approval for transferring an amount of tokens greater than or equal to `principal + creditorFee` (i.e. using the ERC20 `approve` method). Note that this step need not be repeated for every order the creditor fills -- a creditor can grant an `approval` to the token transfer proxy once for a very large number of tokens knowing that

the contract will only withdraw from his account if he consents to it via his submission or signature of a debt order.

5 The creditor then submits it directly to the Debt Kernel contract. Note that his signature is not required in this scenario, given that his submission of the order to the kernel implicates his consent to its parameters. Debt Kernel then issues to the creditor a non-fungible, non-divisible token representing the debtor's commitment to the terms contract and associated parameters.

6 The Debt Kernel transfers an amount of `principal - debtorFee` from the creditor to the debtor.

7 The Debt Kernel transfers the underwriter her allotment of the fee, as defined by the Debt Order.

8 The Debt Kernel transfers the relayer his allotment of the fee, as defined by the Debt Order.

The Debtor-Maker scheme is advantageous for scenarios in which there are many potential creditors, and the debtor does not care to control the precise moment at which the debt is eventually issued. For most use cases, the Debtor-Maker scheme would likely be most efficient.

Debtor-Filler Orders

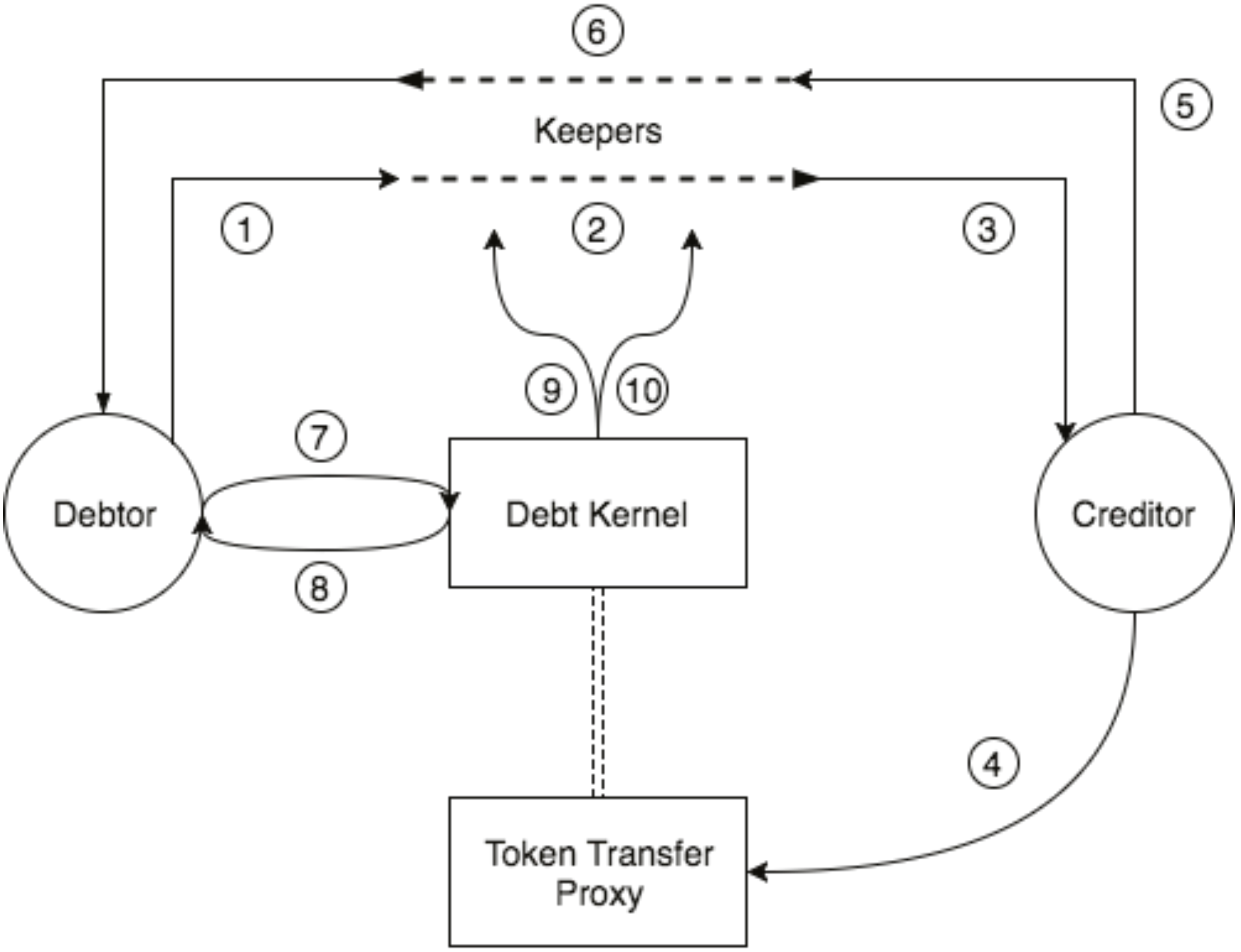


Figure 3: The process of Debtor-Filler Order Submission

The following steps correspond to the circled numbers in the above diagram:

1 Debtor requests loan from an underwriter.

2 Debt Order Handshake (described in detail further) occurs between the debtor, underwriter, and relayer(s), resulting in the relayer listing a valid, complete debt order.

3 Creditor evaluates the terms of the Debt Order on a relayer's order book.

4 If the creditor wishes to participate in the transaction, he first grants the token transfer proxy an approval for transferring an amount of tokens greater than or equal to `principal + creditorFee` (i.e. using the ERC20 `approve` method). He then attaches his ECDSA signature of the debtor/creditor commitment hash to the debt order.

5 The creditor submits the signed Debt Order to the relayer

6 The relayer forwards the signed Debt Order to the debtor.

7 At his convenience, the debtor submits the signed debt order to the Debt Kernel contract, which triggers the minting of a unique, non-fungible debt token to the creditor acting as a signatory of the debt order.

8 The Debt Kernel transfers an amount of `principal – debtorFee` from the creditor to the debtor.

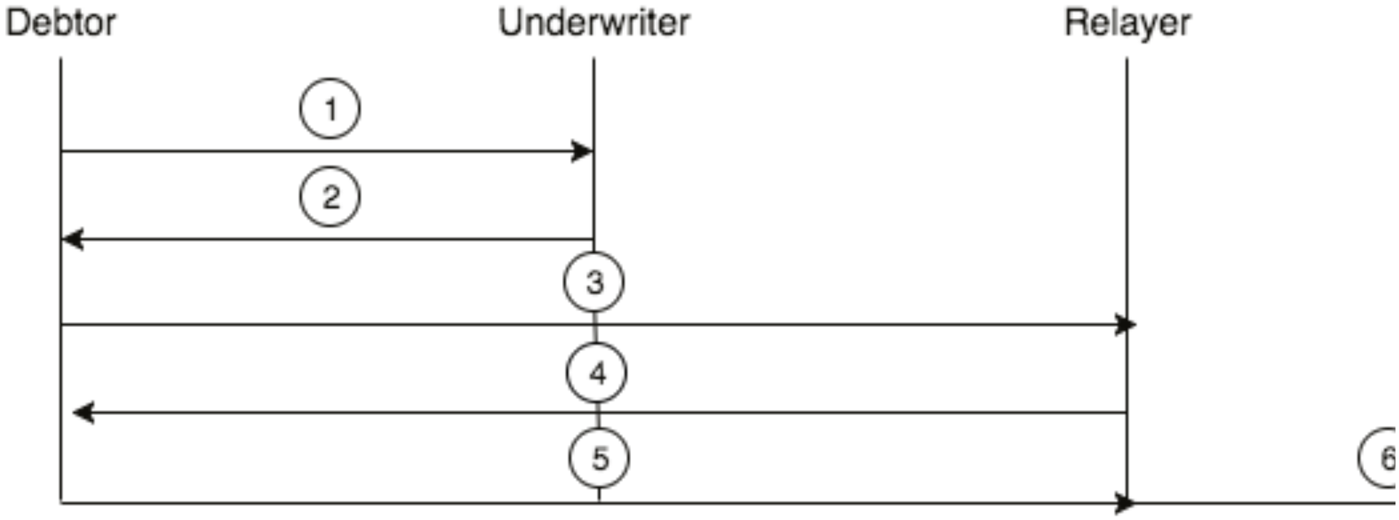
9 The Debt Kernel transfers the underwriter her allotment of the fee, as defined by the Debt Order.

10 The Debt Kernel transfers the relayer his allotment of the fee, as defined by the Debt Order.

The Debtor-Filler scheme is advantageous for scenarios in which the debtor wants to synchronously borrow tokens as part of another, broader transaction. For instance, if a smart contract requires a user pay a certain amount of storage-specific tokens (e.g. FileCoin, Storj, etc.) in order to make use of it, the user could include a valid, signed Debt Order obtained through the above scheme as an argument to the smart contract function call. The smart contract could then submit the order on the debtor's behalf to the Debt Kernel, synchronously lending the debtor the necessary storage tokens and then debiting them to the smart contract in one transaction. This greatly reduces the friction around executing transactions in virtually any context on borrowed credit.

Debt Order Handshake

The Debt Order Handshake alluded to above is formalized as follows:



1 Debtor requests an underwritten debt from a given underwriter, enumerating his desired loan terms (i.e. principle, term length).

2 Underwriter assesses the debtor's default risk using her proprietary risk

models, constructs an underwriter commitment (which, in turn, entails constructing a debt issuance commitment), attaches her ECDSA signature to the hash of the underwriter commitment, and sends the debt issuance commitment, underwriter commitment, and ECDSA signature to the debtor.

3 If the parameters are in line with the debtor's desired terms, the Debtor now has all the requisite parameters to construct a complete debt order. If he wishes to have his order relayed to potential creditors, however, the debtor will request a relayer's fee schedule and address.

4 The relayer responds to the debtor's request with his fee schedule and address.

5 If the relayer's fees are in line with the debtor's wishes, the debtor will construct a complete debt order using the parameters supplied by both the underwriter and relayer, and send the order to the relayer.

6 The relayer lists the completed Debt Order on their order book. Their signature is not necessary, given that their consent to the parameters of the order is tacitly affirmed by nature of the fact that they are voluntarily listing the order.

Debt Repayment Process

In order for the repayment status of any debt asset to be empirically evaluated on-chain, we define a repayment process in which repayments are facilitated by the Repayment Router contract and immutably recorded. When a debtor wishes to make a repayment, they do the following:

1 The Debtor grants the Token Transfer Proxy a transfer allowance (i.e. via the ERC20 `approve` method) greater than or equal to the desired repayment amount.

2 The Debtor sends a transaction to the Repayment Router calling the `repay` function with parameters stipulating the desired repayment amount. The Repayment Router then retrieves the address of the current beneficiary of the debt agreement (i.e. the owner of the debt token) and transfers the desired repayment amount from the Debtor's account to the beneficiary's account, and registers the repayment by calling the `registerRepayment` method of debt's associated Terms Contract.

Note: we could construct a trivially simpler scheme in which Debtors send Creditors repayments directly without leveraging the Debt Kernel contract.

It is necessary, however, for the Debt Kernel to facilitate the repayment process in order to ensure that, when a repayment is registered with a given Terms Contract, it empirically corresponds to a repayment transaction.

Terms Contract Interface

We require that any Debt issued via Dharma protocol commit to a smart contract, referred to as a Terms Contract. The purpose of the Terms Contract is to provide an immutable and programmatically queryable source-of-truth revealing the repayment status of the debt. This allows us to empirically and unambiguously both define the terms repayment scheme in the Debt Issuance process and evaluate the debt's repayment status during the debt's lifecycle both on and off-chain. The interface of required functionality is as follows:

```
interface TermsContract {
     /// When called, the registerRepayment function records the
debtor's
     ///  repayment, as well as any auxiliary metadata needed by
the contract
     ///  to determine ex post facto the value repaid (e.g. current
USD
     ///  exchange rate)
     /// @param  agreementId bytes32. The agreement id (issuance
hash) of the debt agreement to which this pertains.
     /// @param  payer address. The address of the payer.
     /// @param  beneficiary address. The address of the payment's
beneficiary.
     /// @param  unitsOfRepayment uint. The units-of-value repaid
in the transaction.
     /// @param  tokenAddress address. The address of the token
with which the repayment transaction was executed.
    function registerRepayment(
        bytes32 agreementId,
        address payer,
        address beneficiary,
        uint256 unitsOfRepayment,
        address tokenAddress
    ) public returns (bool _success);

    /// A variant of the registerRepayment function that records
the debtor's
    /// repayment in non-fungible tokens (i.e. ERC721), as well as
any auxiliary metadata needed by the contract
    /// to determine ex post facto the value repaid (e.g. current
USD
```

```
    /// exchange rate)
    /// @param  agreementId bytes32. The agreement id (issuance
hash) of the debt agreement to which this pertains.
    /// @param  payer address. The address of the payer.
    /// @param  beneficiary address. The address of the payment's
beneficiary.
    /// @param  tokenId The tokenId of the NFT transferred in the
repayment transaction
    /// @param  tokenAddress The address of the token with which
the repayment transaction was executed.
    function registerNFTRepayment(
        bytes32 agreementId,
        address payer,
        address beneficiary,
        uint256 tokenId,
        address tokenAddress
    ) public returns (bool _success);

     /// Returns the cumulative units-of-value expected to be
repaid by any given blockNumber.
     ///  Note this is not a constant function -- this value can
vary on basis of any number of
     ///  conditions (e.g. interest rates can be renegotiated if
repayments are delinquent).
     /// @param  agreementId bytes32. The agreement id (issuance
hash) of the debt agreement to which this pertains.
     /// @param  blockNumber uint. The block number for which
repayment expectation is being queried.
     /// @return uint256 The cumulative units-of-value expected to
be repaid by the time the given blockNumber lapses.
    function getExpectedRepaymentValue(
        bytes32 agreementId,
        uint256 blockNumber
    ) public view returns (uint256);

     /// Returns the cumulative units-of-value repaid by the point
at which a given blockNumber has lapsed.
     /// @param  agreementId bytes32. The agreement id (issuance
hash) of the debt agreement to which this pertains.
     /// @param blockNumber uint. The block number for which
repayment value is being queried.
     /// @return uint256 The cumulative units-of-value repaid by
the time the given blockNumber lapsed.
    function getValueRepaid(
        bytes32 agreementId,
        uint256 blockNumber
    ) public view returns (uint256);
}
```

Note that in the `getExpectedRepaymentValue` and `getValueRepaid` functions, repayments are defined abstractly in terms of 'units-of-value'. The units by which repayments are measured are intentionally left undefined -- this gives debt issuers the flexibility to, say, denominate the expected repayment values in fiat currencies whilst executing the actual transactions in tokens.

Defaults and Collections

Dharma protocol is agnostic to the means by which underwriters deter defaults and go about collecting on debts. Ostensibly, some underwriters could issue legally binding lending agreements with debtors off chain and collect on debts by leveraging the courts. Alternatively, others could use on-chain collateralization schemes that leverage the functions exposed by a given debt's committed terms contract to release collateral to creditors in a trustless manner whenever the units-of-value repaid fall short of the expected units-of-value repaid.

Innumerable other schemes could be constructed to disincentivize defaults -- **Dharma protocol doesn't advocate or design for any particular solution, but rather aims to provide a standard mechanism by which underwriters can be empirically evaluated for the performance of the debt assets they've attested to in the past**. The market ought to gravitate towards rewarding underwriters whose past performance has been strong, and vice versa in punishing underwriters whose past performance has been weak. The metric by which an underwriter's past performance can be evaluated is what we'll refer to as the

F_{β}

Fβ

metric, a function that borrows from the statistical analysis of binary classification in order to classify how accurate an underwriter's default predictions are:

Let

$$x \in 1,...,n$$

x

$\in 1, . . . , n$ be a debt in the underwriter’s portfolio of the n debts he has attested to

Let α_x be the total expected repayment value the borrower of x is liable for at the end of x's term

Let γ_x be the amount paid back by the borrower in actuality at the end of x

x

's term

Let

δ

x

δ

x

be the probability of

$$x$$

x

defaulting, as predicted in the underwriter's attestation.

Let

$$\beta$$

β

be a hyperparameter we use to weight the importance of recall vs. precision

$$p=$$

Σ

x

$$min($$

α

x

$$-$$

γ

x

$$,$$

δ

x

α

x

$$)$$

Σ

x

δ

x

α

x

$p = \sum x \min(\alpha x - \gamma x, \delta x \alpha x) \sum x \delta x \alpha x$

$r=$

Σ

x

$min($

α

x

$-$

γ

x

$,$

δ

x

α

x

$)$

Σ

x

$($

α

x

$-$

γ

x

$)$

r

=

Σ

x

m

i

n

(

α

x

$-$

γ

x

,
δ
x
α
x
)
Σ
x
(
α
x
$-$
γ
x
)

F
β

$=$

(1+

β
2

$)pr$

β
2

$p+r$

F
β
$=$
(
1

+

β

2

)

p

r

β

2

p

+

r

It is crucial to emphasize that this is NOT a trustless, all-encompassing metric by which we evaluate an underwriter's performance -- fraudulent underwriters can game this metric in a variety of manners (see Attacks). Rather, this is an empirical signal by which good-faith, trusted underwriters can be transparently evaluated. The metric is a valuable performance signal for the market *only* insofar as the underwriter is a trusted actor.

Use Cases

Debt is a massively varied asset class, and, in theory, virtually any type of debt agreement could be issued and underwritten via Dharma protocol. An online lender of *any* class of fiat loans could port their back office onto Dharma protocol by acting as both the debtor and underwriter in every transaction, converting principle payments into fiat on receipt, and conducting all borrower payments & repayments using fiat, thus obfuscating the on-chain nature of the loan funding process away entirely from the end user. We think that, eventually, this will provide an attractive alternative route for online lenders seeking debt capital.

In the short term, however, the more compelling use cases for on-chain debt issuance will be those that are uniquely enabled or augmented by an on-chain implementation. We highlight a few of those below.

Initial Debt Offerings / Tokenized SAFTs

Putting together an ICO is a surprisingly expensive endeavor, and projects in the space are beginning to raise larger and larger pre-ICO rounds in order to finance their eventual crowd-sale. This has led to the creation of the SAFT6: a legal instrument which effectively functions like a convertible debt agreement -- an investor contributes `$X` and, in the event of a future utility token-sale, expects to receive `$X` worth of tokens at a discounted price.

We propose that the pre-ICO financing of a cryptographic protocol can be issued, crowd-funded, and tokenized via Dharma protocol, under terms and conditions that are more flexible than status quo pre-sales.

In the most simple of examples, the debt terms contract can stipulate a vanilla discounted pre-sale agreement akin to a SAFT -- the principle lent would be the `$X` raised, and the repayment expected would be `$X` worth of the eventually deployed utility tokens at a discounted price. Thus, investors would be given a convertible debt token indicating their discounted ownership in the eventual utility tokens. However, the benefit in this arrangement over simply selling tokens in a proxy token that will eventually become the utility token are marginal.

The arrangements become more useful and interesting insofar as the terms contract can encode any variety of investor compensation schemes. Consider, for instance, a pre-sale which the debtor, at the time of the eventual ICO, owes 90% of the raised `$X`, paid out in Ether, at a `Y%` interest rate, with the remaining 10% paid out in discounted utility tokens. This is akin to venture debt fundraising in the traditional financial system, and would give investors a floor on downside-risk (i.e. `90%` of the `$X` are betting on the success of the ICO) with a more muted up-side potential (i.e. `10%` of the `$X` are betting on the eventual use of the utility token). This can further be augmented by creating a contract that acts as one of the multi-sig holders of the contract collecting proceeds in the eventual ICO, locking the proceeds as collateral until the debt terms contract indicates that the pre-sale investors have been properly compensated. Infinitely more pre-sale arrangements can be structured using debt terms contracts in Dharma protocol (e.g. variable discount rates based on how long it takes the debtor to reach their eventual ICO, refunding investors if an ICO is never achieved, etc.).

Decentralized Margin Lending

Decentralized exchanges relying on off-chain order books (e.g. 0x Protocol) are an important and necessary piece of infrastructure in the token ecosystem. If they are to match centralized exchanges in utility and user experience, however, they will need to support margin trading in some capacity.

We can construct a decentralized margin lending system by issuing margin loans via Dharma protocol and having the terms contract associated with a given debt act as a gating function for allowing agents to close margin positions locked into a smart contract. In such a scheme, an underwriter would require a debtor to lock collateral into a margin account contract according to a desired margin ratio as a prerequisite for attesting to the borrower's debt issuance commitment. The debtor would then execute a trade on margin by synchronously filling a Debt Order and a 0x Broadcast Order in the same transaction, with the receiving account being the locked margin account. During the position's term, the underwriter would act as a trusted price oracle to the terms contract. Closing the account would be within the purview of the debtor until the terms contract indicates that either the margin account is in a state of default (as per the agreed upon margin ratio) or the position has expired, at which point the margin account contract will allow the creditor to close the account and earn her desired interest.

Thus, we can leverage the protocol to build a margin lending system that fits natively into the order books of relayer-like entities in existing decentralized exchanges.

Attacks & Limitations

Fraudulent Underwriters

Underwriters operating in a fraudulent manner can deceive investors into believing that they are originating prime debt from borrowers, when in reality they are in fact acting as borrowers themselves. Conceivably, a fraudulent underwriter could build up an extremely positive on-chain attestation performance by lending to themselves using many assumed public keys, and then execute a large exit scam by deceiving creditors into investing in Debt Orders they've attested to in which they are the actual borrower and defaulting entirely on those debts. **It is for this reason that we emphasize that Dharma is *not* a fully trustless protocol -- borrowing and lending under this scheme is only safe insofar as underwriters are known, trusted entities, and not pseudonymized public keys.** It is an underwriter's responsibility to go to whichever lengths are possible to build creditors' trust in their legitimacy -- be that by publicly corroborating the identities of the entities' whose creditworthiness they're attesting to (e.g. if the debtor is a business or corporation), or by cultivating a publicly trusted and known relationship with regulators (e.g. as would be the case with most existing online lenders leveraging the protocol). The sort of trust an underwriter must build with the investment community is not unlike that of a protocol's development team in the run up to a token-sale -- both are facilitating a public investment offering, and, as such, both have an onus of asserting their legitimacy and integrity.

Sybil Attacks

In the context of a lending protocol, a sybil attack involves generating a large number of fraudulent identities, taking on multiple unsecured debts at a time, and defaulting on all at once with little consequence. Largely, defending against sybil attacks falls within the purview of an underwriter's responsibilities -- if an underwriter's identification and KYC procedures are not robust enough to detect sybil behavior, the empirical performance of debts they've attested to will reflect it. However, sybil attackers could ostensibly solicit debt orders from several underwriters at a time, meaning that data sharing amongst underwriters operating in similar debt verticals would likely become a pertinent need.

Debtor Griefing

In scenarios in which a creditor is acting as the market maker in a Debt Order transaction, the creditor could ostensibly engage in what we call "Debtor Griefing" -- the act of front-running the debtor's attempts to fill the given Debt Order by transferring the principle tokens away from the address in which they were originally held for an impending atomic swap. In order to execute the attack, the creditor would facilitate the completion of a Debt Order as the market maker in the handshake process, await the submission of the debtor's filling transaction by listening to the pending transaction pool of a local Ethereum node, and submit a transaction in the same block moving his tokens away from the address at which the principle tokens were located, causing the debtor's transaction to fail and throw. The attack would result in a debtor's fill order throwing an exception, and waste a small amount of gas on the debtor's behalf. In one-off situations, the wasted gas is a nuisance at worst. However, in a systematically executed context, a creditor could ostensibly use this mechanism to perform a denial-of-service attack on a relayer's entire order book by acting as a market maker under a variety of continually generated addresses to the entire span of Debt Orders in the order book, consistently griefing the debtors and recycling the funds for usage in further griefing attacks. The gas costs of executing such an attack at large scale are non-negligible, but, assuming an attacker is motivated by some sort of extortionary bounty levied from a relayer, the attack could prove to be profitable if an attacker possesses the requisite liquidity. The best line of defense against such attacks is the relayer herself -- relayers that are plagued by such attacks have a variety of options to temper denial-of-service attacks from malicious creditors, ranging from simple IP blocking to more aggressive policies that force creditors to go through KYC walls.

Side Deals

We refer to "side deal attacks" as attempts by agents in any given debt transaction to avoid paying fees to the keepers involved in the transaction's structuring. Consider the following scheme -- a third-party maintains an online registry in which potential debtors voluntarily associate their public keys with their email addresses. Given that creditors can inspect Debt Orders in their client browser on a relayer's order book, a creditor could leverage the third-party's registry to communicate with debtors whose Debt Orders are listed on the relayer's order book. The creditors and debtors could then coordinate on a separate channel to construct a point-to-point Debt Order in which neither the underwriter nor relayer are compensated for their services, saving the agents fees that would otherwise go to the agents.

Underwriters are likely less sensitive to this vulnerability insofar as creditors rely on their services throughout the debt's term. With relayers, however, creditors have nothing to lose in conducting side deals, meaning it's feasible for an adversary to systematically sabotage a relayer's operations by constructing a publicly usable matching engine that ingests Debt Orders from the relayer's order book and matches participating debtors and creditors such that they are in a position to construct a side deal.

We contend that, in practicality, participating in most systemic side deal matching constructions is a fairly high friction process for agents, in that successful side deal matching would require agents to engage in yet another Debt Order Handshake, which, under highly asynchronous conditions, could be too time consuming to merit the fee saved. But even if we construct side deal matching arrangements which are frictionless enough to merit their usage, relayers are not without recourse. A particularly targeted relayer could obfuscate debtors' addresses via a Commit-Execute construction that retains a trustless relationship between the relayer and creditor[7]:

1 In lieu of listing Debt Orders in the intended manner, relayers list Debt Orders modified such that all fields containing the debtor's address are zero'd out -- nullifying the validity of Debt Order messages and their associated signatures, but providing creditors with the data points necessary to reason about investing in any given debt offering.

2 **Commit**: Creditors commit to filling an order by submitting the invalid and incomplete Debt Order to a purpose-built smart contract that acts as an intermediary cog in the transaction. Additionally, creditors grant the smart contract token transfer allowances[8] equivalent to the token allowances necessary to fill the Debt Order's attached 0x Broadcast Order.

3 **Execute**: The relayer executes the issuance and swap by submitting the complete, valid Debt Order to the aforementioned contract. The contract then verifies that all fields in the submitted Debt Order match their counterparts in the creditor's committed, incomplete Debt Order (with the exception of fields containing the debtor's address), and then, in sequence, transfers itself the requisite principle tokens from the creditor, submits the complete Debt Order to the debt kernel contract in its capacity as an intermediary creditor, and finally transfers the debt token it newly possesses to the creditor. If the submitted debt order is invalid or mismatched with the Debt Order committed to by the creditor, the contract throws and the creditor maintains ownership of her tokens.

Thus, relayers can leverage the above Commit-Execute construction in order give creditors a means of filling obfuscated Debt Orders in a trustless manner, if circumstances necessitate its usage.

Footnotes

[1] I emphasize *equity-like* insofar as protocol tokens are, in theory, **not** equity, but, in terms of their risk profile and the class of speculative interest they attract, behave exactly like equity.↩

[2] Warren, Will, and Amir Bandeali. “0x: An Open Protocol for Decentralized Exchange on the Ethereum Blockchain.” Https://0xproject.Com, 0xproject.com/pdfs/0x_white_paper.pdf.↩

[3] Zurrer, Ryan. “Keepers - Workers That Maintain Blockchain Networks.” Medium, Medium, 5 Aug. 2017, medium.com/@rzurrer/keepers-workers-that-maintain-blockchain-networks-a40182615b66.↩

[4] In previous versions of this white paper, these entities were referred to as Risk Assessment Attesters, or RAAs. We've moved away from this terminology, given that 'underwriters' are more intuitively digestible as a broad class.↩

[5] Grigg, Ian. “The Ricardian Contract.” Iang.org, 2000, iang.org/papers/ricardian_contract.html↩

[6] Batiz-Benet, Juan, et al. “The SAFT Project: Toward a Compliant Token Sale Framework.” saftproject.com/static/SAFT-Project-Whitepaper.pdf.↩

[7] Note that this construction only works for Debtor-Maker Debt Orders.↩

[8] i.e. via the ERC20 `approve` method. ↩