LEDGER
ledgerjournal.org

SSN 2379-5980 (online)
DOI 10.5195/LEDGER.2020.148

RESEARCH ARTICLE

Barter Machine: An Autonomous, Distributed Barter Exchange on the Ethereum Blockchain

Can Özturan*†

Abstract. Direct bartering allows for the trading of assets with or without the use of money. In this paper, we introduce a smart contract written in the Solidity language for the Ethereum blockchain that implements a distributed and autonomous direct barter exchange operated by crowds. Since ERC20 smart contract tokens are widely used for initial coin offerings (ICOs), our implementation provides autonomous bartering services for ERC20 tokens. The non-fungible ERC721 token, as well as Ethereum Name Service (ENS) name bartering, are also supported. Because finding a feasible bartering solution for bids involving multiple tokens is NP-hard in general, our exchange only provides escrow and swapping services. It assumes feasible solutions are submitted by bartering problem solvers from the crowd who run a solver locally on their machines. Bartering problem solvers are incentivized to submit solutions to the autonomous exchange by awarding them with excess tokens that are left over after granting the bidders the tokens they requested in their bids. Our system, called BarterMachine, can perhaps be used to facilitate a global barter economy. The BarterMachine prototype is available for testing on the Ethereum Ropsten network at `https://bartermachine.github.io/bartermachine/ropsten/`.

1. Introduction

In elementary schools, we were told by our teachers that pre-historic people would gather in an open square, and put down excess food or animals they had hunted and which they did not need, in order to form a collection of excess items. They would then pick from the excess collection the items which they needed. This simplistic exchange mechanism worked when the numbers of men and the items traded were small. As the worldwide population—and the number of items produced and consumed—increased to millions and then billions, this scheme no longer worked. The English economist William Stanley Jevons pointed to this problem using the now quite popular phrase "the difficulty of finding a double coincidence of wants" and made a strong argument for money as a solution to this problem. Specifically, in his book, Jevons stated, *"The first difficulty in barter is to find two persons whose disposable possessions mutually suit each other's wants. There may be many people wanting, and many possessing those things wanted;*

*19HiB4AvutzjWU6VbkDNTqWTK2x6hxMrM3
†Can Özturan (ozturaca@boun.edu.tr) is a faculty member in the Department of Computer Engineering at Bogazici University, Istanbul, Turkey.

but to allow of an act of barter there must be a double coincidence, which will rarely happen."[1] In the modern world, however, the difficulties mentioned by Jevons are easier to handle:

(1) We now have blockchain infrastructures that allow us to keep records of ownership of items and transfer them in a trustless manner across the globe. A blockchain can be viewed as an open square where the pre-historic person put forward all the items they owned and wanted to trade.
(2) There are fast optimization software packages (minimum cost flow solvers and integer programming solvers) that can give solutions to double (or multiple) coincidence of wants problems involving large number of people and items.[2–4] In general, finding multiple coincidences of wants of multiple items translates to finding hypercycles in directed hypergraphs and has been shown to be NP-complete.[5,6]
(3) The worldwide population and the number of items being traded are in the billions range. Hence, the likelihood of finding multiple coincidences of wants in smaller windows of time can be made higher by using AI-powered search engines.

We propose an autonomous system named BarterMachine, where almost any item—single or in combination with others, or in multiple quantities—can be directly bartered by representing them as tokens. Here are some examples of items that can potentially be bartered on the token barter exchange: houses, tickets (airline, bus, ship, concert tickets), company shares, services (worker hours), vehicles (new and used cars), electronic goods such as mobile phones, computers and tablets. Cross-country transfer of money by banks can also be implemented as a bartering process by treating, for example, euros in a bank branch in one country as tokens to be bartered with tokens representing euros in another bank branch in another country.

The Solidity language enables us to program smart contracts that implement autonomous systems running under the control of no one on the Ethereum blockchain.[7] Our token barter exchange is implemented as a Solidity contract and operated by crowds that are motivated to use it in two major ways:

(1) Traders can derive a greater utility by being able to trade one arbitrary set of tokens for another set of tokens directly. Currently, most of the exchanges provide token-pair trading. Bartering enables users to define their own trading patterns.
(2) In a bartering solution, providers can find a feasible solution to multiple coincidences of wants (requests) and submit it to the system. When the wants of the traders in the feasible solution are satisfied, and there are leftover tokens remaining, the leftover tokens can be transferred to the solution provider as an incentive.

In the rest of the paper, we first present an overview of related work in Section 2. The proposed autonomous token barter exchange model is covered in Section 3. Details of the implemented smart contract are given in Section 4. A web user interface based on MetaMask has also been developed to in order to make the BarterMachine prototype accessible to ordinary users.[8] It is presented in Section 5. Results of tests to determine (i) Ethereum gas consumption of critical functions in the smart contract and (ii) execution times of a Gurobi-based bartering problem solution finder are reported in Section 6.[4] Finally, the paper is concluded with a discussion and future work in Section 7.

SSN 2379-5980 (online)
DOI 10.5195/LEDGER.2020.148

2. Related Work

Bartering problems have been addressed in the last decade extensively in previous scholarship by the present author and others.[6,9–13] In these, the solution techniques focused on maximization of profits or the number of bids satisfied. The techniques were used by central authorities, polynomial minimum cost flow algorithms were used for restricted versions of bartering problems, and integer programming solvers were utilized to solve the more general NP-hard versions of the problems.

In this work, we focus on providing bartering services in a distributed setting with no central authority. In particular, we provide a blockchain-based autonomous barter exchange implementation that is operated by crowds. The bartering problem in a distributed setting is similar in spirit to Dijkstra's Banker's Problem.[14] Assuming crowds have access to bartering solver packages, we address the problem of how to implement an exchange service which provides escrow and multiple token-swapping services when feasible solutions are submitted.

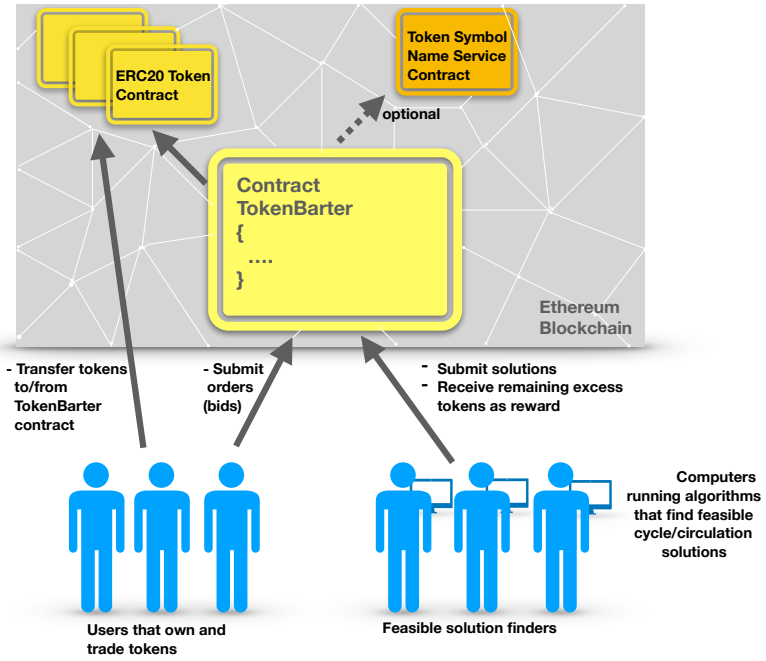


Fig. 1. Decentralized autonomous barter model

3. Token Barter Exchange Model

Our direct token barter exchange model is illustrated in Figure 1. Our autonomous token barter exchange is implemented as the BarterMachine TokenBarter contract. This contract accepts orders (bids) that are expressed as a set of tokens and their quantities to be traded for a set of tokens and their quantities. An example order is shown below:

```
A(3) + B(1) + C(2) => D(2) + E(3)
```


SSN 2379-5980 (online)
DOI 10.5195/LEDGER.2020.148

Here, a user that owns A, B, and C tokens places an order that indicates willingness to trade the set of 3 A, 1 B, and 2 C tokens on the left-hand side, for the 2 D and 3 E tokens on the right-hand side. A full bartering problem example is given in Figure 2. Here, we show the quantities of tokens owned and the orders placed by users. The tokens on the left-hand side of an order (*i.e.* tokens to be given away) are shown as positive numbers. The ones on the right-hand side (*i.e.* requested tokens) are shown as negative numbers. A feasible solution is a subset of orders that satisfies the requested (demanded) collection of tokens indicated by the right-hand side by the collection of the left-hand side tokens. If the token quantities in the vector corresponding to feasible solution orders are summed component-wise to get a vector (called *excess*), then each component in the excess vector will be a non-negative quantity. Formal formulation of constraints for feasible solution are given in "Resource Bartering in Data Grids" (Özturan, 2004).[6] After the tokens that appear on the right-hand side are given to the demanding users, there may be leftovers (*i.e.* positive quantities of tokens in the excess vector). These excess tokens can be given to the person who submits the feasible solution. In this way, we can form an incentive mechanism for solution finders who need to run time-consuming solution finding algorithms on their computers.

Storage and computation costs are expensive and restricted on the Ethereum blockchain. As a result, in order to avoid (i) exceeding the gas limit which leads to premature contract function terminations and (ii) large transaction costs which translate to heavy bills to the user, we make careful design decisions on what type of orders are provided and what is enforced as far as token balances are concerned on the contract. In our BarterMachine implementation, we assume the orders are *good until cancelled* by the user. We check sufficiency of token balances when new orders are placed. On the other hand, we do not enforce reservation of token balances for each order but rather provide the user with an option for voluntarily supplying a deadline until which he will not withdraw a certain amount of his tokens. There are actually valid reasons for not enforcing reservation of balances when orders are placed. For example, suppose a user places two bids: one for buying token A (*e.g.* representing a car, A) for 10 ether and one for buying token B (*e.g.* representing a car B) for 12 ether. Suppose the user has a balance of 12 ether and wants buy one of the cars but not both. Hence, our implementation allows the first solution that includes one of these bids that is submitted to withdraw the 10 or 12 ether. But later, if another solution that includes the other bid is submitted, it will not be accepted since there will not be enough balance. Hence, this mechanism allows us to realize exclusive OR (*i.e.*, capacitated) type bids.

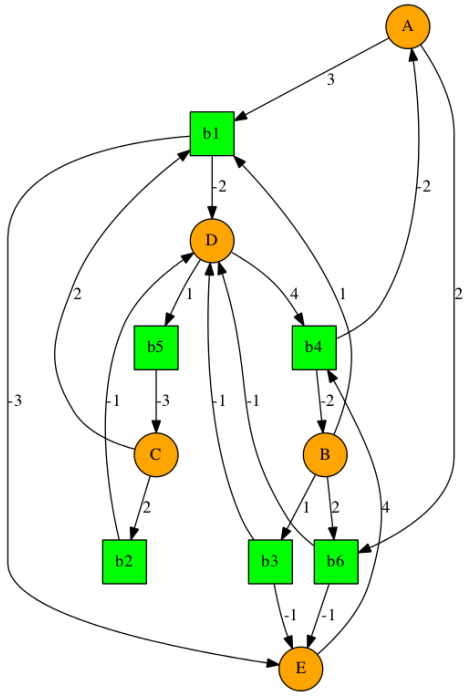
Note that the ability to cancel an order anytime may pose some concern to the solution finders. What if a solution finder spends some time for finding a solution that includes an order and yet the order is cancelled before the solution is submitted. Voluntary "no balance withdrawal" commitment by a user until some deadline can convince a solution finder to include this user's order in his solution. Hence, to prevent abuse of order cancellation, there are two mechanisms in place: (i) solution finders may skip processing of orders of users while finding solutions if they see that the user does not commit to no balance withdrawal and (ii) when one submits and cancels order by calling contract functions, transaction costs are charged to the user, *i.e.* there is some cost to the user. We also note that the no withdrawal commitment means just no withdrawal by the user himself: if an order of the user appears in a submitted solution, the token balance of a token can of course be reduced/increased, even if no token is withdrawn/deposited by the user.

All the information in the BarterMachine contract is public. User balances as well as orders

SSN 2379-5980 (online)
DOI 10.5195/LEDGER.2020.148

	Quantities owned by users				Orders(bids)						A soln.
	u_1	u_2	u_3	u_4	u_1	u_2	u_1	u_3	u_3	u_4	b_1, b_3, b_4
Token	u_1	u_2	u_3	u_4	b_1	b_2	b_3	b_4	b_5	b_6	(excess)
A	5			3	3			-2		2	1
B	5			3	1		1	-2		2	0
C	5	3			2	2			-3		2
D			8		-2	-1	-1	4	1	-1	1
E			8		-3		-1	4		-1	0

(a)



(b)

Fig. 2. A token bartering example: tokens, user balances, bids and solution (a) and the corresponding bid graph (b)

SSN 2379-5980 (online)
DOI 10.5195/LEDGER.2020.148

can be queried using the contract functions. Solution finders need this collective information because they need to form a global optimization problem and solve it, for example, by using an integer programming package. Solution finders can check whether sufficient quantities of tokens are present in user balances before selecting an order for inclusion in the feasible solution. Such an approach, in effect, shifts the costly balance sufficiency check computation on the smart contract to the computer of the solution finder. In this way, gas costs of the functions on the smart contract will be lower.

Currently, when an ERC20, ERC721 token(s), or ENS name are deposited for the first time to a BarterMachine contract by any user, a token structure is created internally in the contract and an integer token number is assigned to it. Within our BarterMachine contract, the integer token number is the ID of the token. No token symbols are stored in our BarterMachine contract, since ERC20 token contracts can be set to have the same symbol name by different token contract deployers. Optionally, a separate token symbol server contract can also be implemented in the future that acts as an authority to register token symbol names as shown in Figure 1.

Finally, we note that our BarterMachine prototype currently supports an open solution submission policy meaning any user can support a solution. Whoever's solution transaction is first entered into a block, those solution bids will be satisfied; hence, solution finders run the risk of wasting their computational efforts due to being late. This is the disadvantage you get in return for offering the advantage of being open to anyone. We have a similar situation in the Bitcoin and Ethereum mining processes, where miners solve a hash puzzle. The networks are open and anyone can mine (*i.e.* try to solve the hash puzzle), but whoever solves the hash puzzle first gets the mining reward. In order to save on computational efforts, other policies can be considered in the future. For example, we can have ERC721 tokens representing time intervals; then, only the person who buys the token representing an interval can submit solutions in that time interval.

4. Smart Contract Implementation

The BarterMachine contract provides several functions that broadly serve one of these purposes:

(1) *Deposit:* Ether, ERC20 and ERC721 tokens, and ENS names can be deposited as well as consumable dummy tokens be created.
(2) *Withdrawal:* Ether, ERC20 and ERC721 tokens, and ENS names can be withdrawn and commitment to no-withdrawal for a number of blocks can be made.
(3) *Transfer:* Ether, ERC20 and ERC721 tokens, and ENS names (which are all viewed as tokens inside the BarterMachine) can be internally transferred from one account to another.
(4) *Get Info:* Various information on orders, tokens, balances, and users can be queried.
(5) *Orders (Bids):* Orders can entered, cancelled, and listed.
(6) *Solution:* Feasible bartering solution locking, submission, and validation can be done.

Descriptions of the most of the BarterMachine functions are given in Table 1.

When depositing tokens, the user is required to call the approve function of ERC20 and ERC721 contracts and then afterwards invoke the BarterMachine WithdrawToken function to transfer the ownership of the tokens to BarterMachine. The EnterOrder function is used to enter a bid to the contract. This is done by passing an array of token numbers and the corresponding array of token quantities. For example, bid b_1 in the bartering example given in Figure 2 can

SSN 2379-5980 (online)
DOI 10.5195/LEDGER.2020.148

be be passed as `[2,3,4,5,6]`, `[3,1,2,-2,-3]`. Note that we reserve the token number 1 for representing the main currency of the blockchain, *i.e.* 1 represents the ether.

Table 1. BarterMachine Contract Functions (partial list)

Category	*Function*	*Description*
Deposit	`DepositEther`	Receives ether to msg.sender's BarterMachine account
	`DepositERC20Token`	Transfers ERC20 token to msg.sender's BarterMachine account
	`DepositERC721Token`	Transfers ERC701 token to msg.sender's BarterMachine account
	`DepositENSName`	Transfers ENS name to msg.sender's BarterMachine account
	`CreateDummyToken`	Creates an internal consumable dummy token and returns its token number
Withdrawal	`WithdrawEther`	Sends ether to msg.sender's address
	`WithdrawERC20Token`	Transfers ERC20 token(s) to msg.sender's ERC20 contract account
	`WithdrawERC721Token`	Transfers ERC721 token to msg.sender's ERC721 contract account
	`WithdrawENSName`	Transfers ownership of ens name to msg.sender
	`CommitNoWithdraw`	Given a token number, amount of tokens and number of blocks, commits no withdrawal of these tokens for the given number of blocks (note that the tokens are still available for bartering)
	`GetCommitNoWithdrawInfo`	Given a token and an account numbers returns the no-withdraw details
Transfer	`TransferInternalToAcc`	Given a token number, account number and amount of tokens, transfers the ownership of tokens to the given user internally
Orders (Bids)	`EnterOrder`	Given order details (an array of token numbers and an array of quantities), enters it as an order and returns the order number
	`CancelOrder`	Given an order number, cancels the order, *i.e.* makes it not live
	`GetNoOfMyOrders`	Returns user's number of orders both live and not live orders
	`GetTotalNoOfOrders`	Returns total number of orders of all users both live and not live orders
	`GetOrder`	Given an order number, returns details of the order
	`GetAccLastOrder`	Given an account number, returns the last order number of the account
	`GetAccPrevOrder`	Given an order number, returns the order number of the previous order
Get Info	`GetAccNo`	Given Ethereum address, returns the account number
	`GetNoOfAccs`	Returns number of accounts
	`GetMyBalance`	Given an account number, returns the token balance of the calling user
	`GetAccBalance`	Given token and account numbers, returns the token balance
	`GetNoOfTokens`	Returns number of tokens registered
	`GetTokenInfo`	Given token number, returns the information about the token
	`GetTokenNo`	Given token identification returns the token number
	`GetTokenTotalBalance`	Given token number returns the total token balance of all accounts
Solution	`LockSolnHash`	Locks a solution
	`UnlockSolnHash`	Unlocks a solution
	`Hash`	Takes hash of a solution
	`SubmitSoln`	Submits a solution

We focus on the details of the SubmitSoln function since it is the most complex function in the contract, needing to verify the feasibility of the submitted solution, the sufficiency of user token balances, collective transfer of tokens among users whose orders make up the solution, as well as the payment of the incentive to the solution finder. The function steps are given in Algorithm 1. The symbols used in the algorithm are given in Table 2. The solution is submitted to the SubmitSoln function as a *sorted* array of bid numbers in ascending order. For the bartering example given in Figure 2, the solution is passed as `[1,3,4]` representing the sequence of bids b_1, b_3, b_4. Optionally, before submitting a solution, a hash of the solution sequence can be passed to the LockSolnHash function to lock the sequence and hence prevent others from submitting the same solution sequence. Note that this only locks the specific submitted sequence and not the orders. For example, even if `[1,3,4]` is locked, others can still lock and/or submit a solution like `[4,6]`. Lines 3-5 check this hash lock. We assume Solidity maps return 0 when a map does

SSN 2379-5980 (online)
DOI 10.5195/LEDGER.2020.148

not contain a key. A solution can also be submitted without locking it.

SubmitSoln makes two passes over the orders making up the submitted solution. The first pass (lines 7-16) checks the sufficiency of the user token balances and computes the excess vector. Line 8 checks the sorted ordering of the solution. Lines 11-13 check the left-hand side of a bid and make sure that the user has at least that many number of tokens in his balance. If he has, the offered number of tokens on the left-hand side is deducted from his balance on line 14. If he does not, the require statement on line 13 aborts execution and reverts the state changes. Line 15 updates the excess vector.

Table 2. Symbols used in SubmitSoln algorithm

Symbol	Meaning
s_i	ith order (bid) number in submitted solution.
$L[]$	Map of solution hashes.
a_s	Address of sender (msg.sender).
$U[]$	map of user records indexed by addresses.
p	Previous order number.
O[]	Array of orders with live bit, array I beginning index and owner fields $O^{live}, O^{beg}, O^{own}$.
I[]	Array of token numbers appearing in left and right sides of bids with quantity and token symbol number fields I^{quan}, I^{t}.
B[]	Map of (user,token t) pair balances, *i.e.* how many token t user has inside BarterMachine contract. A pairing function is used for (user,token t) pairs.
T[]	Array of token records. The field $T^{ex}[t]$ denotes the excess vector component value computed for a token t.
require(cond)	abort execution and revert state changes if cond is false. This function is provided in Solidity.

The second pass (lines 17-26) checks the feasibility of the solution by making sure excess vector components are non-negative. While the second pass proceeds, transfer of tokens among users whose orders make up the solution are also carried out. Feasibility is checked by the require statement at line 19 which will revert all the contract state changes so far and abort execution if a negative excess value is encountered. If excess value is non-negative, the user will be delivered the tokens that he requested on the right side of his order. It is important to focus on the values assigned to the excess vector component since it triggers token transfer to the bidders and the solution providers. A non-negative (0 or positive) excess value enables token transfer to the bidder after the require statement at lines 19-22. A positive excess value enables transfer of leftover tokens to the solution provider as incentive at lines 23-26. It is possible that the positive excess value is contributed to by multiple bids in the solution. The positive excess of a token t will be transferred to solution provider (line 25) when it is first referenced in the loop over solution bids. At that moment, excess value will be set to 0 at line 26. In subsequent iterations, if other bids that contribute to the positive excess value of the token t are encountered, the contract

SSN 2379-5980 (online)
DOI 10.5195/LEDGER.2020.148

will still make token t payment to the user whose bid was selected in the solution (since we check for 0 or positive value at line 19). But it will not make token t payment to the solution provider, since it was already made previously. On entry to SubmitSoln all excess values are 0. On successful return from the SubmitSoln, all excess values are guaranteed to be 0. If negative values are computed within SubmitSoln, the require statement will cause reverting of all contract state changes and, hence, all excess values will again be zero.

Algorithm 1: Submit Solution

1 **Input:** Sorted order numbers of the solution $[s_0, s_1, \ldots, s_{n-1}]$
2 **Output:** Returns true/false or the transaction state reverts if requirements are not met

`// check hash lock only if it was used`
3 $h \leftarrow hash(s_0 s_1 \ldots s_{n-1})$
4 **if** $((L[h] \neq 0) \wedge (L[h] \neq a_s)) \vee (U[a_s] = 0)$ **then**
5 **return** *(false)*
end
6 $p \leftarrow 0$
7 **foreach** $i \in [0, \ldots, n-1]$ **do** `// first pass`
8 **require** $(O^{live}[s_i] \wedge (s_i > p))$
9 $p \leftarrow s_i$
10 **foreach** $j \in [O^{beg}[s_i - 1], \ldots, O^{beg}[s_i]$ **do**
11 **if** $I^{quan}[j] > 0$ **then** `// bid left side (positive)`
12 $k \leftarrow pairing(O^{own}[s_i], I^t[j])$
13 **require** $(B[k] \geq I^{quan}[j])$
14 $B[k] \leftarrow B[k] - I^{quan}[j]$
end
15 $T^{ex}[I^t[j]] \leftarrow T^{ex}[I^t[j]] + I^{quan}[j]$
end
16 $O^{live}[s_i] \leftarrow false$
end
17 **foreach** $i \in [0, \ldots, n-1]$ **do** `// second pass`
18 **foreach** $j \in [O^{beg}[s_i - 1], \ldots, O^{beg}[s_i]]$ **do**
19 **require** $(T^{ex}[I^t[j]] \geq 0)$
20 **if** $I^{quan}[j] < 0$ **then** `// bid right side (negative)`
21 $k \leftarrow pairing(O^{own}[s_i], I^t[j])$
22 $B[k] \leftarrow B[k] - I^{quan}[j]$
end
23 **if** $T^{ex}[I^t[j]] > 0$ **then** `// positive excess`
24 $k \leftarrow pairing(U[a_s], I^t[j])$
25 $B[k] \leftarrow B[k] + T^{ex}[I^t[j]]$
26 $T^{ex}[I^t[j]] \leftarrow 0$
end
end
end
27 **return** *(true)*

Consumable dummy tokens enable a user to allow a specified number of his bids to be eligible to be satisfied. Only the creator of a dummy token can use his own dummy token in a bid he places. For example, if you have two bids `A(1) => ETH(1)` and `B(1) => ETH(1)` and you want at most one to be satisfied, you can create a dummy token and enter the bids as: `A(1) => DUMMY(1)`, `B(1) => DUMMY(1)` and `DUMMY(1) => ETH(1)`. This will have the

SSN 2379-5980 (online)
DOI 10.5195/LEDGER.2020.148

effect of wanting to sell either A or B, but not both, for 1 ether. Note that excess dummy tokens are not transferred to anyone at all in SubmitSoln. Once they are used in a satisfied bid as part of a solution, they should be deducted from the account balance and vanish, since their sole function is to implement exclusive OR type bid functionality. Hence, they are called consumable tokens.

Finally, we also note that rather than using a map of maps in Solidity, we used only a map and computed keys by using Szudzik's "elegant pairing function" at lines 12, 21, and 24.[15] This lowered gas usage a little when accessing map elements.

5. BarterMachine Web User Interface Based on MetaMask

Interacting with the BarterMachine contract by calling contract functions directly can be difficult for ordinary users. Even running a node and syncing the whole Ethereum blockchain data can be non-trivial and take a long time. Therefore, in order to make BarterMachine easy to use, a web-based prototype system that utilizes MetaMask was developed that would allow ordinary users to immediately use the BarterMachine through their browsers.[8] MetaMask is a browser extension that acts as a bridge between the browser and the Ethereum blockchain. It enables users to run Ethereum distributed applications in the browser without running a full Ethereum node. It also provides a secure account vault and a user interface to manage accounts and sign blockchain transactions.

Figure 3 shows screendumps of the BarterMachine web interface. The menus and the forms provided by the web interface allow the user to provide data that go into the smart contract functions given in Table 1. Note that the BarterMachine web interface does not need to run on a centralized server. BarterMachine is a truly autonomous decentralized exchange (DEX): it stores nothing on a centralized server. All data is stored on the blockchain; hence, anyone can download the HTML web page and the Javascript files from Github and run it on their local machine.

A BarterMachine contract is currently deployed on the Ropsten Ethereum test network for academic research purposes. In addition to providing a user-friendly interface to the contract functions, the BarterMachine web interface also provides an ERC20 token faucet so that users can get test tokens and be able to immediately start using the system. The web interface also includes a bid graph drawing system based on vis.js, as shown in Figure 3(c).[16]

6. Tests and Function Gas Consumption

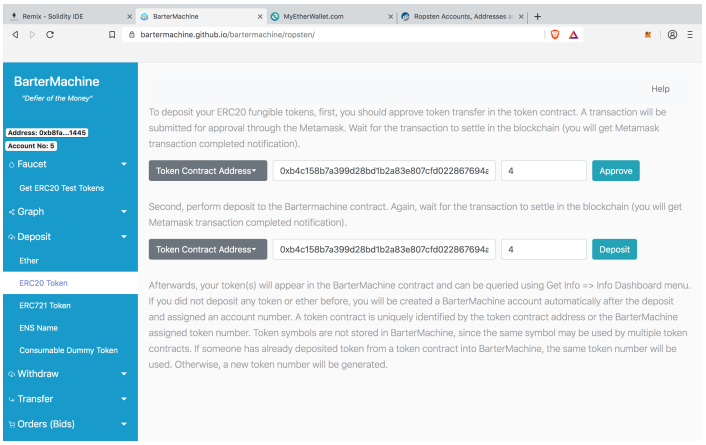
We tested our BarterMachine contract using version v2.3.5-stable of the Parity Ethereum client and version 0.5.1 of the Remix Solidity browser compiler. The testing was done in Parity's development mode (*i.e.* using –config dev option) privately on a Macbook Pro notebook equipped with a 2.6 GHz Intel Core i5 processor and 8 GB memory.

For testing, we used 260 ERC20 token contracts.[17] 500 user Ethereum accounts were created, randomly assigned ERC20 token balances, and assigned new accounts inside the BarterMachine contract by making deposits. Each user submitted 5 randomly-generated orders. Two rounds of tests were done which varied the number of tokens on the left and right sides of a bid as follows:

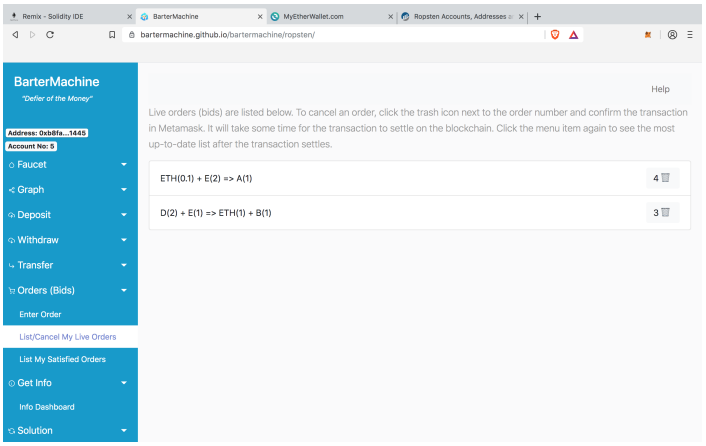
- R1: Up to 2 tokens were used in each side of bid (*i.e.* left and right sides). This implied that up to 8 integers were submitted when a bid was submitted (up to 4 token IDs and the corresponding up to 4 token quantities).
- R2: Up to 20 tokens were used in each side of bid meaning up to 80 integers were

SSN 2379-5980 (online)
DOI 10.5195/LEDGER.2020.148

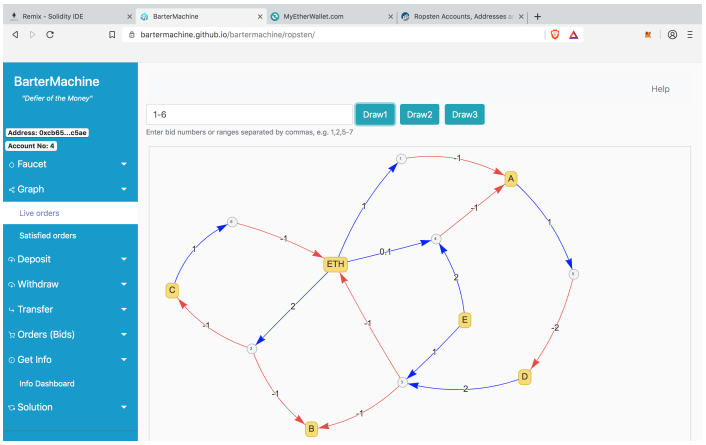
submitted when a bid was submitted.



(a)



(b)



(c)

Fig. 3. BarterMachine web user interface ; deposit (a), orders listing (b), and graph (c) screens.

SSN 2379-5980 (online)
DOI 10.5195/LEDGER.2020.148

The solver was also invoked to get solutions by specifying a number that indicated the maximum number of bids in the solution.

On Ethereum networks, one must pay for storage and computation. These costs are specified in terms of gas. There is also a gas limit that is imposed on Ethereum. A contract's functions cannot be executed if they exceed the gas limit. As of March 14th, 2019:

- The gas limit on the Parity based private development node, the Ropsten test network and the real Ethereum Mainnet is 8M.
- The standard gas price taken from ETH Gas Station is 2 Gwei and the price of 1 ether (ETH) is 133 USD.[18]

Apart from testing for the correctness of the BarterMachine contract, we also spent a lot of effort to reduce the gas costs of contract deployment and function. To reduce deployment costs, our data structure definitions and functions were put in a library contract. The BarterMachine contract simply called these library functions. When EnterOrder and SubmitSoln functions are called, integer lists are passed as parameters which are then iterated in loops. Therefore, the gas usages of these two functions are critical. We investigated the relative gas usage of these two functions.

The gas usage of the EnterOrder function is shown in Figure 4. The plot shows the average gas and USD costs of entering orders of various sizes. We refer to the summation of the size of left- and right-hand sides of a bid as the size of an order (bid). For the example in Figure 2, size of order b_1 is 5 and size of b_2 is 2. Currently, the commercial exchanges provide token pair trading (*i.e.* size of order is 2). Our contract allows arbitrarily-sized bid patterns to be submitted. As Figure 4 shows, entering an order even with a size of 20 consumes 1.25M gas and does not exceed the gas limits.

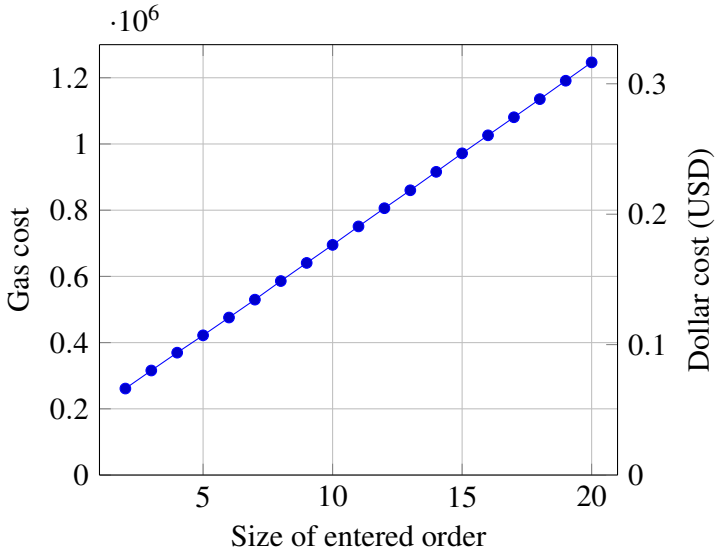


Fig. 4. Gas and dollar costs of entering orders with different sizes (*i.e.* total number of tokens in the left and right sides of order)

To investigate SubmitSoln's gas consumption, we submitted solutions with various numbers of bids in them. The sizes of the solutions and their gas usages are reported in Table 3. In SubmitSoln, two passes are made over the tokens that appear in the left and right sides of bids in

SSN 2379-5980 (online)
DOI 10.5195/LEDGER.2020.148

the solution. As a result, the actual number of iterations depend not on the number of bids that form the submitted solution, but rather on the summation of sizes (total left and right sides) of bids in the solution. As seen in Table 3, when this summation of sizes reaches around 230, the 8M gas limit is reached. A gas limit greater than 8M on the Ethereum networks will allow bigger number of bids and hence larger sized cycles and hypercycles to be entered as solutions.

Table 3. Gas and USD costs of the SubmitSoln Function

Test Round	No. of bids in Solution	Summation of left and right sizes of bids	Gas Used by SubmitSoln	Cost in USD as of March 14, 2019
R1	20	60	1971238	0.52
	30	88	2768302	0.74
	40	120	5176580	1.38
	80	234	9901705*	2.63
	160	481	18987328*	5.05
R2	10	59	1812913	0.48
	13	83	2484672	0.66
	20	126	4627089	1.23
	40	251	10110622*	2.69
	80	529	18741627*	4.99
	160	1102	34522770*	9.18

* exceeds gas limit 8M on Ethereum Mainnet, Ropsten network and the private development node.

In order to find solutions, we employ a Gurobi-based integer programming solver.[4] The integer programming formulation given in "Resource Bartering in Data Grids" is used. [6] The constraints are augmented with an additional constraint that limits the number of bids in the solution to some specified number. This is done so as to find solutions with small sizes which in turn will keep the gas expenditure in the SubmitSoln function small, and thereby avoid exceeding the gas limit. As an objective function, we use a function that maximizes the total number of tokens in the excess vector. It is possible to use other objective functions also: for example, the numbers of tokens in the excess vector can be multiplied by the weights or prices of tokens. Table 4 shows how long the solver takes to solve some problem instances that are randomly generated. We used a time limit of 20 minutes on the Gurobi solver. In tests T5 and T9, the sign X means that Gurobi was not able to find the *optimal* solution within 20 minutes. Even though we are solving an NP-hard problem, we note that the performance of the commercial integer programming solver Gurobi is impressive considering that we have the number of variables (no. of bids) in the thousands to million range.

Our smart contract has been deployed on the Ropsten test network and can be used by visiting the web site:

`https://bartermachine.github.io/bartermachine/ropsten/`

SSN 2379-5980 (online)
DOI 10.5195/LEDGER.2020.148

Table 4. Gurobi solution times for some problem instances with time limit of 20 minutes

Test Case	No of tokens	No. of users	No. of bids per user	No. of bids	Max left hand size of bid	Max right hand size of bid	Solution size	Soln. Time (secs)
T1	677	1000	6	6000	4	4	10	8.7
T2	677	10000	6	60000	4	4	10	13.8
T3	677	10000	6	60000	3	3	40	25.4
T4	2000	50000	6	300000	3	3	40	126.2
T5	10000	10000	10	100000	10	10	40	X
T6	2000	100000	10	1000000	4	4	40	447.7
T7	2000	100000	10	1000000	3	3	40	213.5
T8	5000	100000	10	1000000	3	3	40	870.1
T9	10000	200000	10	2000000	3	3	40	X

7. Discussion and Conclusion

One of the goals of this paper is to draw attention to the advantages that direct bartering can have over transacting with money alone. Whereas money can act as a store of value and medium of exchange, it also introduces complications and hinders trading in situations where the buyers do not have money, and yet, a cycle of trading pattern is possible. A simple example from the domain name markets can illustrate this. Suppose A, B, and C are domain names owned by three persons. A cycle of trading pattern `A => B => C => A` may be possible because each person wants to sell his domain name and get the other one which is owned by the other person. If they post their domain names for sale asking 10,000 dollars, they can wait for a long time and not be able to sell their domain names or use the proceeds to buy the domain name they want. Direct bartering can allow them to trade their domain names without the need to have money.

The Barter Economy is a superset of the Monetary Economy in the sense that it provides richer trading patterns to be discovered. Hence, collective trading patterns that may be difficult to efficiently coordinate in Monetary Economy may be possible in the Barter Economy which in turn may provide greater benefit to the society. The catch, however, is the problem of finding these circular trading patterns (*i.e.* cycles, circulations, hypercycles and hypercirculations). Fast optimization packages like Gurobi are making it possible to find these trading patterns easily. What was missing was an autonomous exchange that would collect the bids and the solutions and carry out the transfer of ownership of the tokens collectively and atomically. Our SubmitSoln algorithm solves this problem.

As the bids accumulate, very large sized instances of the most general NP-hard bartering problem will be challenging to solve; however, there are a few approaches that may be pursued in future works to reduce the problem sizes and possibly get, not necessarily optimal, but feasible and profitable solutions:

(1) Barter-connected components of the whole bid graph can be computed using the linear

SSN 2379-5980 (online)
DOI 10.5195/LEDGER.2020.148

time algorithm described in "Resource Bartering in Data Grids".[6] The optimization problem can then be solved on each smaller component in parallel. Also, due to the geographical proximity of users bartering such things as tickets, houses, and cars, we can expect localized components corresponding to towns and cities to be present. Even if the whole graph is connected, a heuristic that employs some graph partitioning software such as Metis can be used to break up the bid graph into multiple parts with small cut sizes and hence producing smaller-sized components on which the Gurobi solver can be applied in parallel.

(2) The simpler versions of the problem, *e.g.*, differential barter auction with unrestricted solution size described in "Used Car Salesman Problem," (Özturan, 2005) and "Digraphs: Theory, Algorithms and Applications" (Bang-Jensen and Gutin, 2009) can be solved by polynomial time minimum cost network flow algorithms. [2,9,10] Even though the bid pattern is simpler in this case, (*i.e.* $item1 + money => item2$ or $item1 => item2 + money$, or $item => money$, or $money => item$), it may appeal to people when bartering their cars, tickets, or houses with differential money amounts. Components involving only these bids can be extracted and solved fast. Also, the users can be informed that if they submit bids in differential form, their bids are more likely to be picked up by solution finders employing fast polynomial time minimum cost solvers.

We designed our functions (in particular the EnterOrder and SubmitSoln functions) in such a way that they use storage and computation that is proportional to the sizes of the bids that are passed as parameters. The bartering problem solver may produce solutions that have many bids in them. We modified our solver and added constraints so as to produce solutions with small numbers of bids. This enabled us to avoid the problem of exceeding the gas limit in some cases. However, there may be cases where small sized solutions may not exist but large sized ones exist. Entering of solutions with many bids (*i.e.* very long cycles/hypercycles) can perhaps be made by multiple function calls. This approach, however, may require additional costly contract storage to store temporary values and hence and not be as economical as the single call approach that uses a single excess array. Our future work will focus on addressing these challenges.

Specifically, our future plans include:

- Deployment on the real Ethereum and Ethereum Classic main networks after making large scale stress tests on the Ropsten test network and after country regulations regarding distributed autonomous exchanges become clearer,
- Introducing bids with deadlines,
- Addressing the problem of submitting long sized solutions.

Notes and References

[1] Jevons, W. S. *Money and the Mechanism of Exchange*. New York: D. Appleton and Co. (1876).

[2] Goldberg, A. V. "An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm." *Journal of Algorithms* **22.1** 1–29 (1997).

[3] Dezső, B., Jüttner, A., Kovács, P. "LEMON–An Open Source C++ Graph Template Library." *Electronic Notes in Theoretical Computer Science* **264.5** 23–45 (2011).

[4] No Author. "Gurobi Optimization." *Gurobi* (accessed 21 May 2020) `http://www.gurobi.com/`.

[5] Özturan, C. "On Finding Hypercycles in Chemical Reaction Networks." *Applied Mathematics Letters* **21.9** 881–884 (2008).

SSN 2379-5980 (online)
DOI 10.5195/LEDGER.2020.148

[6] Özturan, C. "Resource Bartering in Data Grids." *Scientific Programming* **12.3** 155–168 (2004).

[7] No Author. "Solidity Documentation." *Solidity* (accessed 21 May 2020) `https://solidity.readthedocs.io`.

[8] No Author. "MetaMask." *MetaMask* (accessed 21 May 2020) `https://metamask.io/`.

[9] Özturan, C. "Used Car Salesman Problem: A Differential Auction–Barter Market." *Annals of Mathematics and Artificial Intelligence* **44.3** 255–267 (2005).

[10] Bang-Jensen, J., Gutin, G. *Digraphs: Theory, Algorithms and Applications, Second Edition*. Springer 683–684 (2009).

[11] Abraham, D. J., Blum, A., Sandholm, T. "Clearing Algorithms for Barter Exchange Markets: Enabling Nationwide Kidney Exchanges." In *Proceedings of the 8th ACM Conference on Electronic Commerce* ACM 295–304 (2007) .

[12] Özer, A. H., Özturan, C. "A Direct Barter Model for Course Add/Drop Process." *Discrete Applied Mathematics* **159.8** 812–825 (2011).

[13] Glorie, K. M., van de Klundert, J. J., Wagelmans, A. P. "Kidney Exchange with Long Chains: An Efficient Pricing Algorithm for Clearing Barter Exchanges with Branch-and-Price." *Manufacturing & Service Operations Management* **16.4** 498–512 (2014).

[14] Dijkstra, E. W. "The Mathematics Behind the Banker's Algorithm." In *Selected Writings on Computing: A Personal Perspective* Springer 308–312 (1982).

[15] Szudzik, M. "An Elegant Pairing Function." In *NKS2006 Conference* (2006) `http://www.szudzik.com/ElegantPairing.pdf`.

[16] No Author. "vis.js." *vis.js* (accessed 21 May 2020) `https://visjs.org/`.

[17] Vogelsteller, F., Buterin, V. "ERC-20 Token Standard." *Github* (accessed 21 May 2020) `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md`.

[18] Councourse Open Community. "Recommended Gas Prices in Gwei." *ETH Gas Station* (accessed 21 May 2020) `https://ethgasstation.info/`.

SSN 2379-5980 (online)
DOI 10.5195/LEDGER.2020.148