EXHIBIT A

US010025797B1

(12) **United States Patent**
Fonss

(10) **Patent No.: US 10,025,797 B1**
(45) **Date of Patent: Jul. 17, 2018**

(54) **METHOD AND SYSTEM FOR SEPARATING STORAGE AND PROCESS OF A COMPUTERIZED LEDGER FOR IMPROVED FUNCTION**

(71) Applicant: **True Return Systems LLC**, New Canaan, CT (US)

(72) Inventor: **Jack Fonss**, New Canaan, CT (US)

(73) Assignee: **True Return Systems LLC**, New Canaan, CT (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **15/923,317**

(22) Filed: **Mar. 16, 2018**

Related U.S. Application Data

(60) Provisional application No. 62/634,321, filed on Feb. 23, 2018.

(51) **Int. Cl.**
G06F 17/30 (2006.01)

(52) **U.S. Cl.**
CPC .. ***G06F 17/30194*** (2013.01); ***G06F 17/30227*** (2013.01)

(58) **Field of Classification Search**
CPC G06F 17/30227; G06F 17/30194
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

2012/0158558	A1*	6/2012	Hahn-Carlson	G06Q 10/10 705/30
2012/0203645	A1*	8/2012	Sutter	G06Q 30/04 705/19
2015/0244804	A1*	8/2015	Warfield	H04L 47/6295 709/219
2017/0046792	A1*	2/2017	Haldenby	G06Q 20/0655
2018/0095662	A1*	4/2018	Brennan	G06F 3/061
2018/0115428	A1*	4/2018	Lysenko	H04L 9/3247

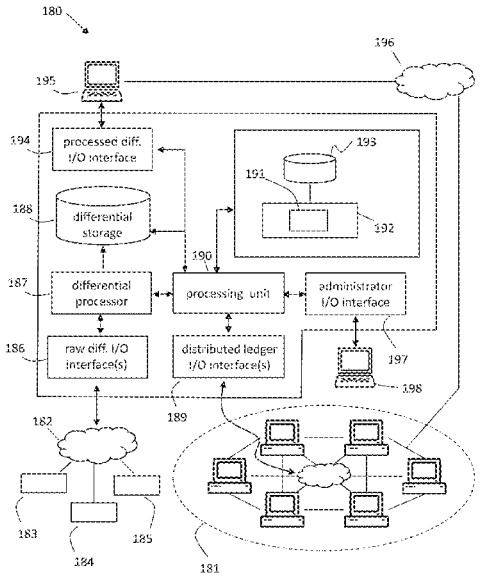
* cited by examiner

Primary Examiner — Leslie Wong
(74) *Attorney, Agent, or Firm* — Lerner, David, Littenberg, Krumholz & Mentlik, LLP

(57) **ABSTRACT**

A non-conventional method and system used with computerized ledgers provides advantages of computing efficiencies, data security, and universal use. The system, method, and computer readable storage medium for storing, creating, monitoring, managing, and modifying measurement, descriptive differences, and parameters of the records of distributed computerized ledgers works through a separation and linkage of stacked modular data storage and processing. Electronic transaction records reside on distributed ledgers and modifying measurement and descriptive differences reside in decentralized or centralized storage, where computers and related networks are improved with increased functionality through increased transaction speeds, decreased data transmissions, increased security, and improved modifiable functionality. The separation of parallel layered storage and modularity of design enable the system to perform a wide range of functionality while maintaining homogeneity with the distributed computerized ledger.

20 Claims, 9 Drawing Sheets



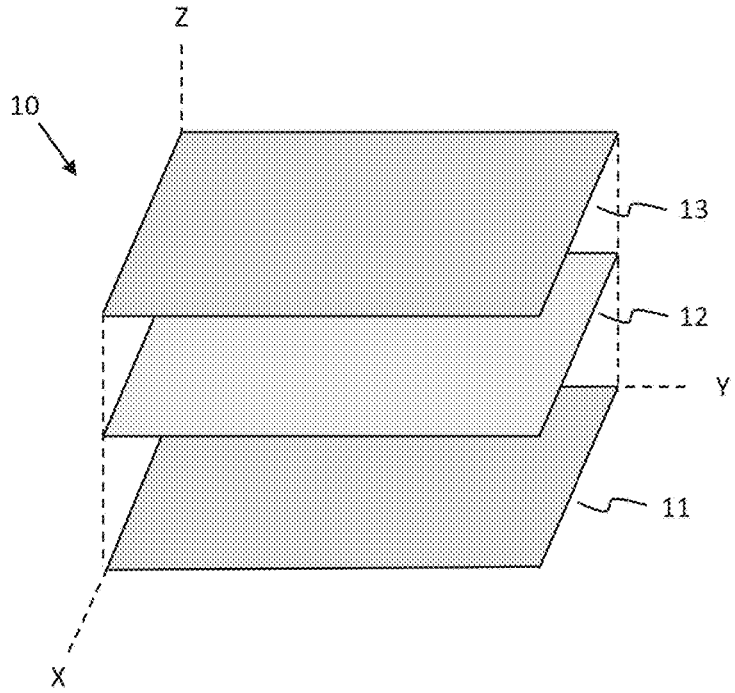

Z
10
13
12
Y
11
X

FIG. 1

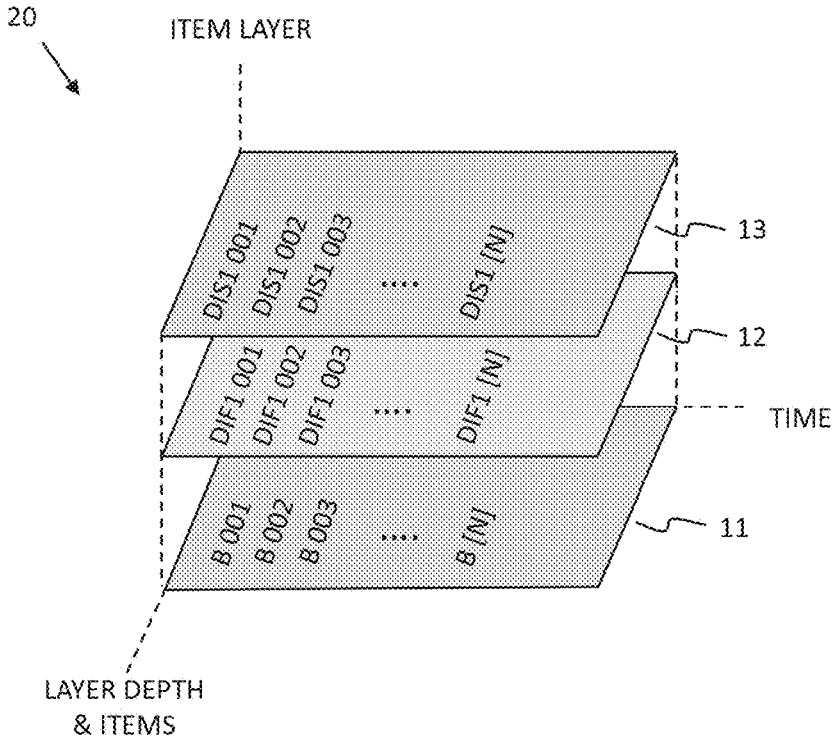

20
ITEM LAYER
DIS1 001
DIS1 002
DIS1 003
DIS1 [N]
13
DIF1 001
DIF1 002
DIF1 003
DIF1 [N]
12
TIME
B 001
B 002
B 003
B [N]
11
LAYER DEPTH
& ITEMS

FIG. 2

FIG. 3

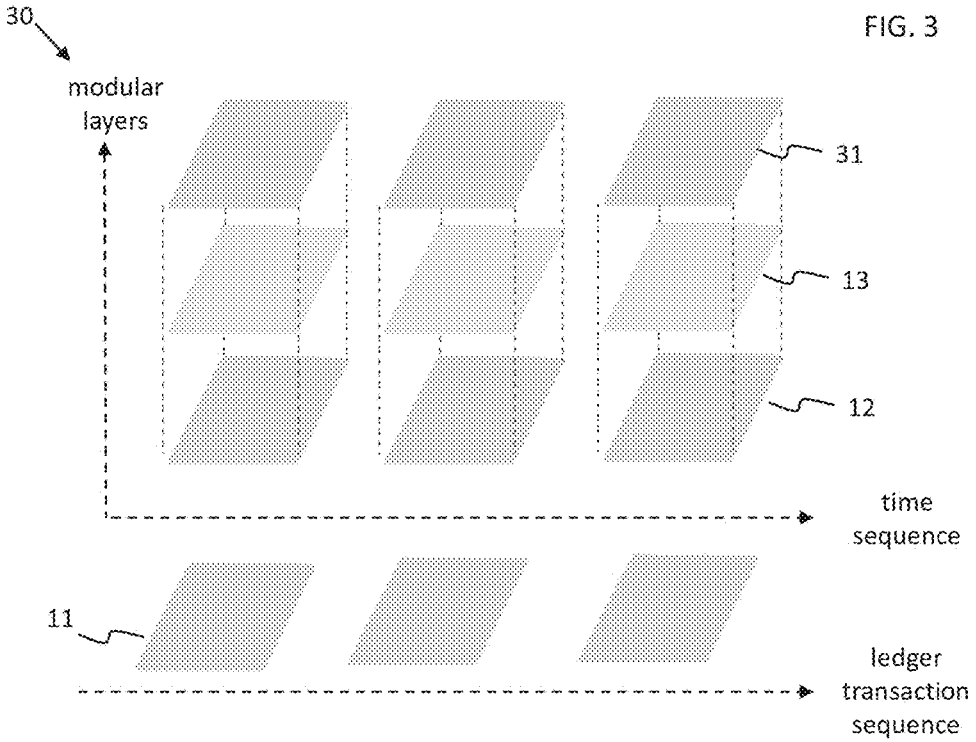
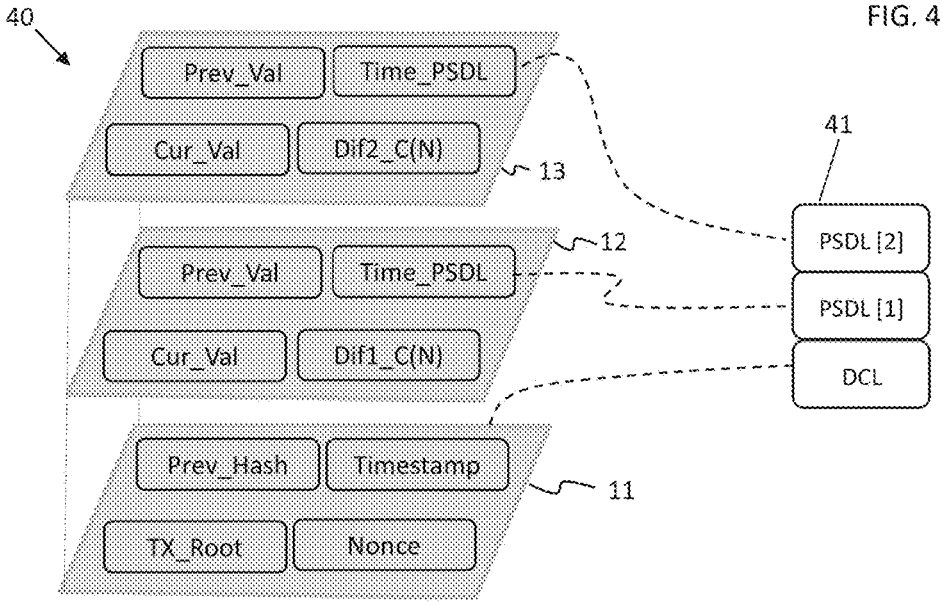

30
modular layers
31
13
12
time sequence
11
ledger transaction sequence

FIG. 4


40
Prev_Val
Time_PSDL
Cur_Val
Dif2_C(N)
13
41
12
Prev_Val
Time_PSDL
Cur_Val
Dif1_C(N)
PSDL [2]
PSDL [1]
DCL
Prev_Hash
Timestamp
11
TX_Root
Nonce

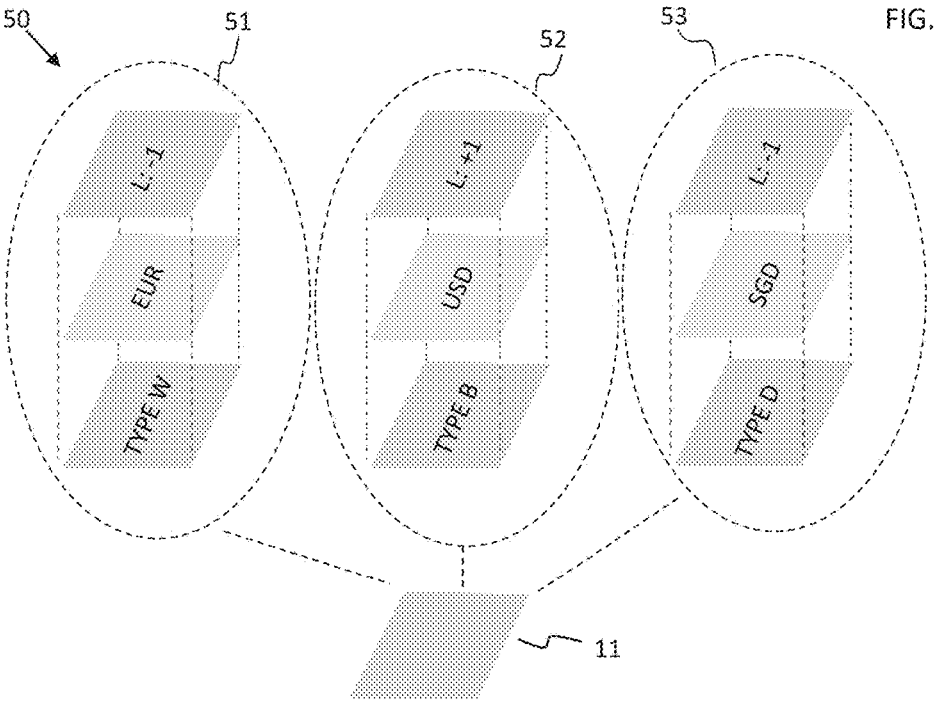

50
51
52
53
L: -1
EUR
TYPE W
L: +1
USD
TYPE B
L: -1
SGD
TYPE D
11

FIG. 5

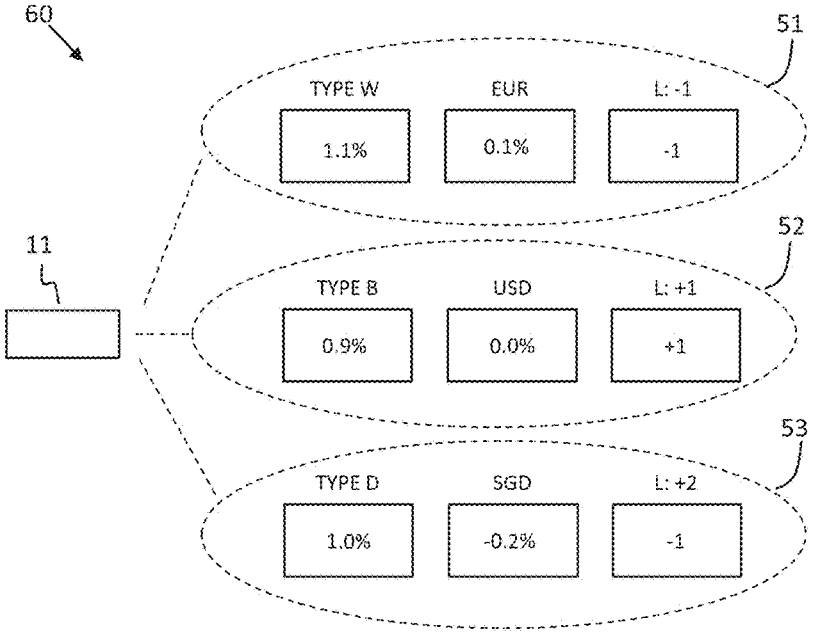

60
11
51
TYPE W
EUR
L: -1
1.1%
0.1%
-1
52
TYPE B
USD
L: +1
0.9%
0.0%
+1
53
TYPE D
SGD
L: +2
1.0%
-0.2%
-1

FIG. 6

FIG. 7

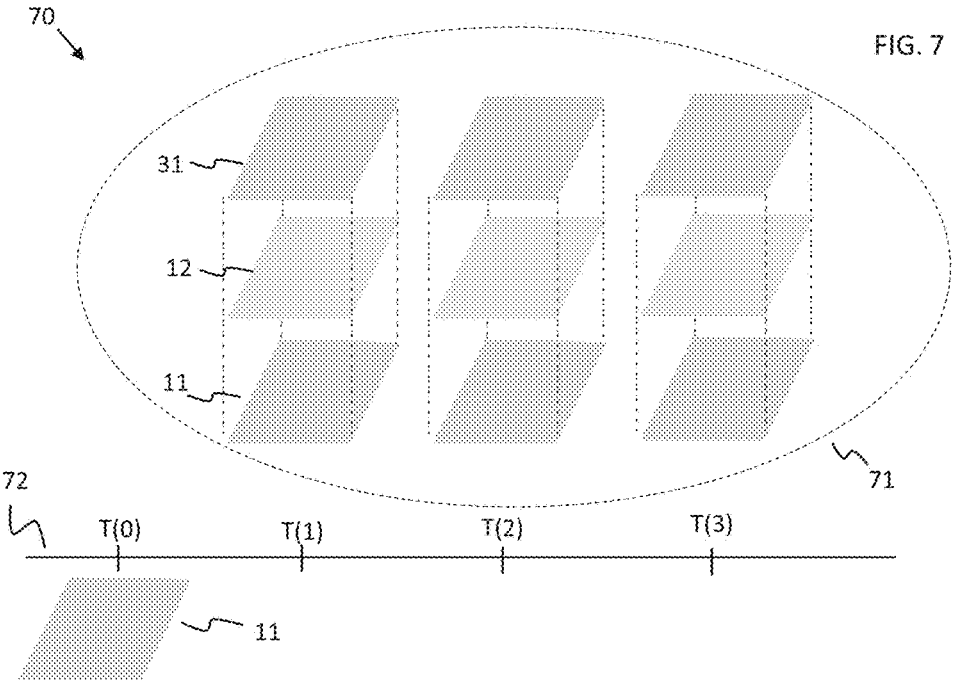
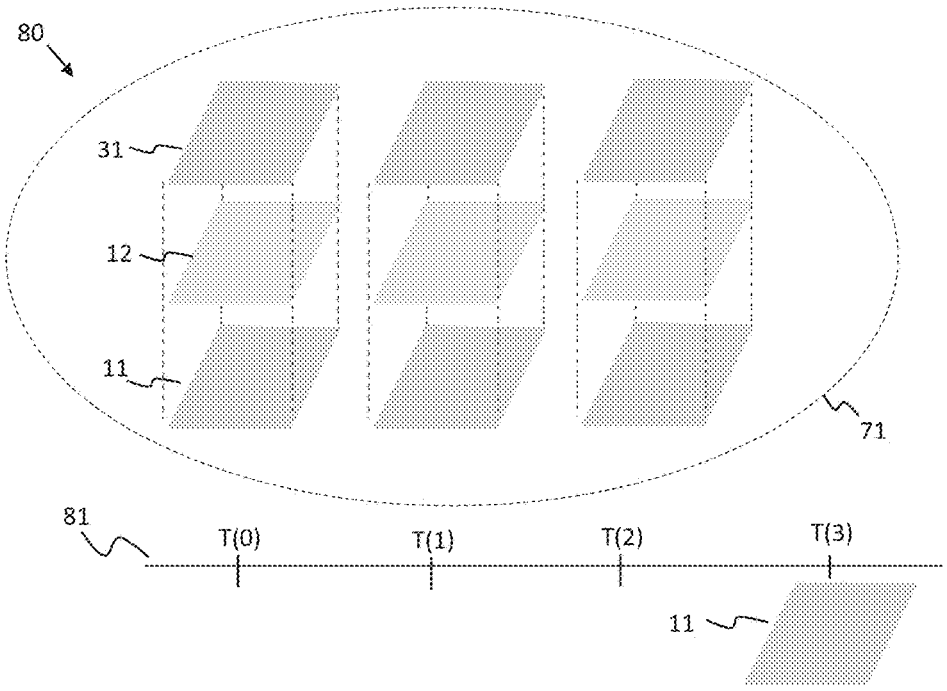


FIG. 8



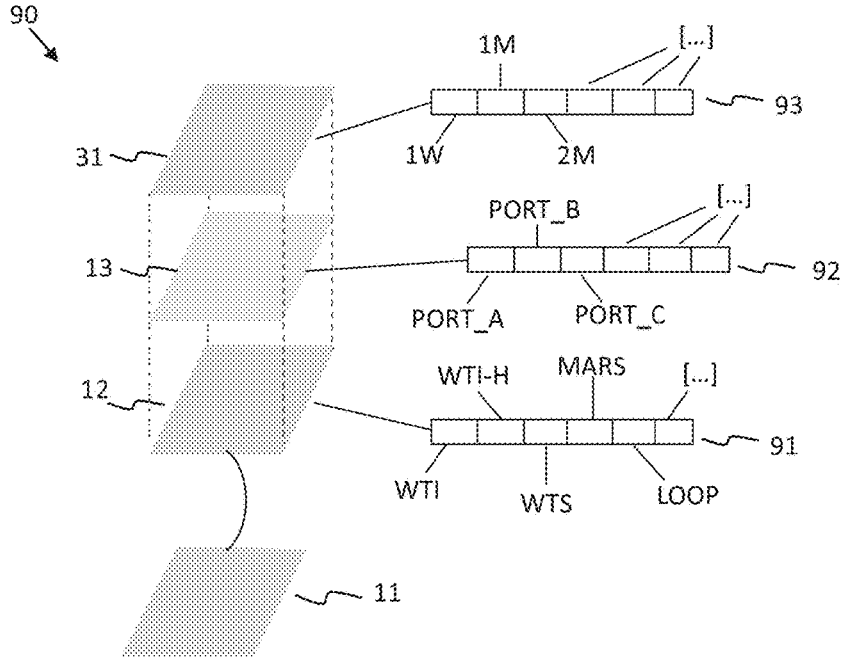

90
1M
[...]
93
1W
2M
31
PORT_B
[...]
13
92
PORT_A
PORT_C
WTI-H
MARS
[...]
12
91
WTI
WTS
LOOP
11

FIG. 9

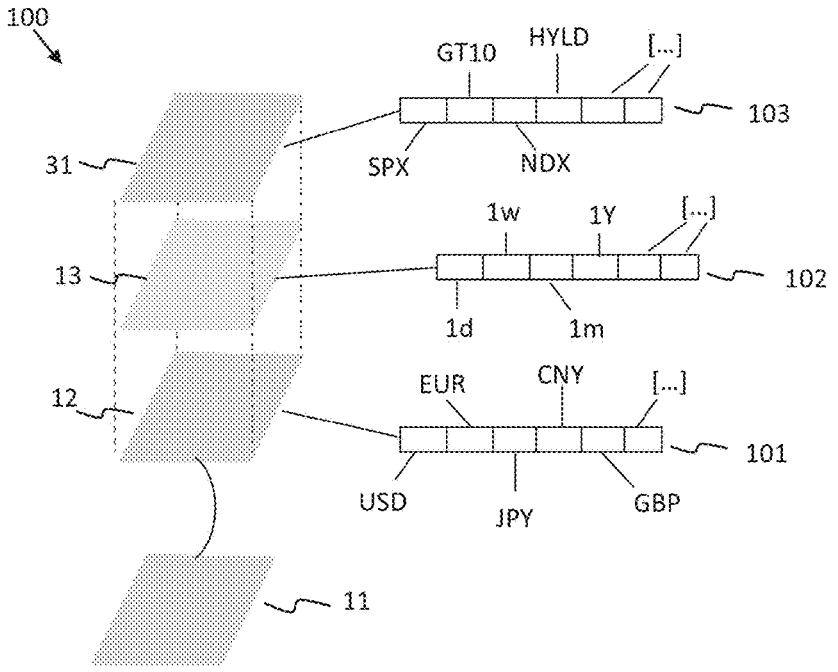

100
GT10
HYLD
[...]
103
31
SPX
NDX
1w
1Y
[...]
13
102
1d
1m
EUR
CNY
[...]
12
101
USD
JPY
GBP
11

FIG. 10

FIG. 11

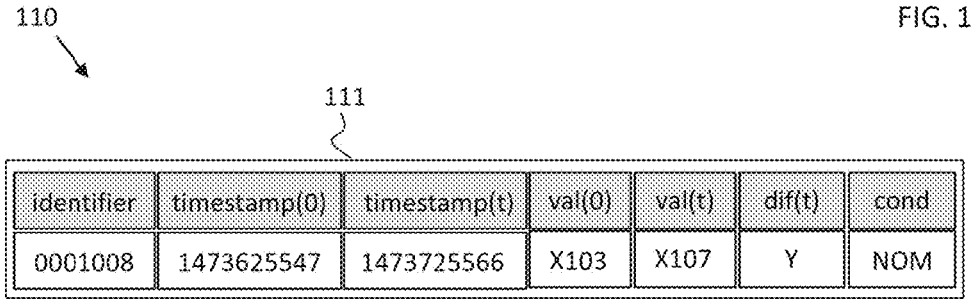

110
111
identifier
timestamp(0)
timestamp(t)
val(0)
val(t)
dif(t)
cond
0001008
1473625547
1473725566
X103
X107
Y
NOM

FIG. 12

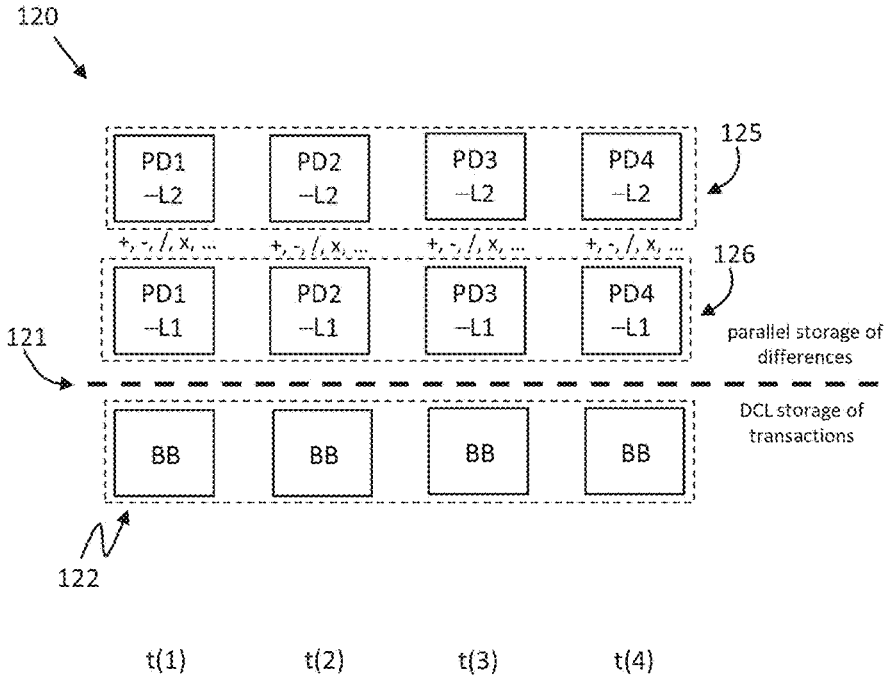

120
125
PD1 –L2
PD2 –L2
PD3 –L2
PD4 –L2
+, -, /, x, ...
126
PD1 –L1
PD2 –L1
PD3 –L1
PD4 –L1
121
parallel storage of differences
DCL storage of transactions
BB
BB
BB
BB
122
t(1)
t(2)
t(3)
t(4)

FIG. 13

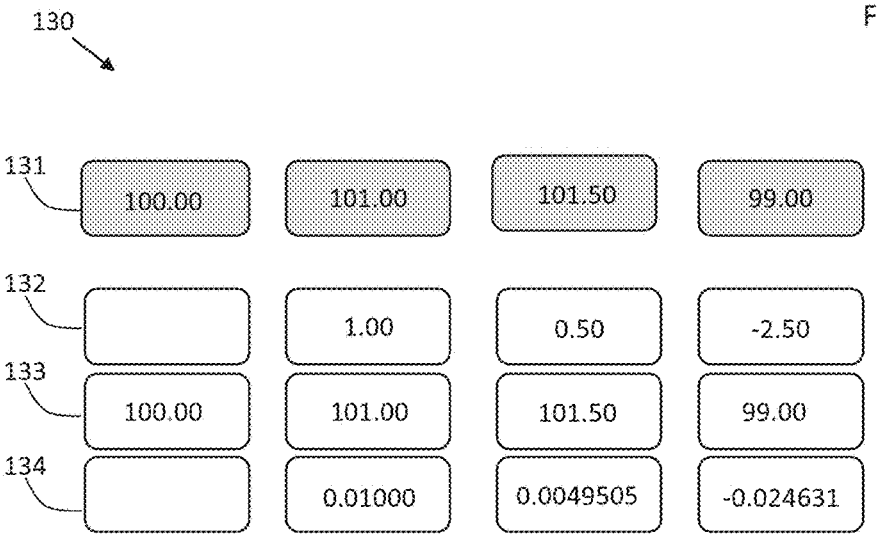


FIG. 14

140

VAL	DIF1	DIF2
USD	USD	10000
EUR	EUR	01000
JPY	JPY	00100
CNY	CNY	00010
GBP	GBP	00001

FIG. 15

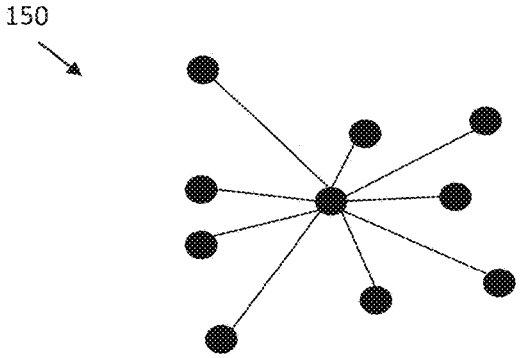


FIG. 16

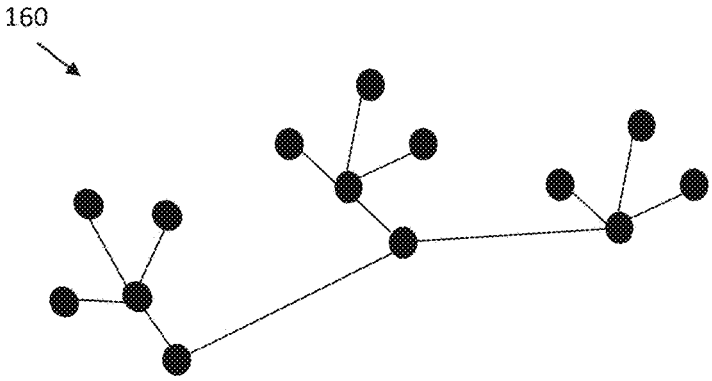


FIG. 17

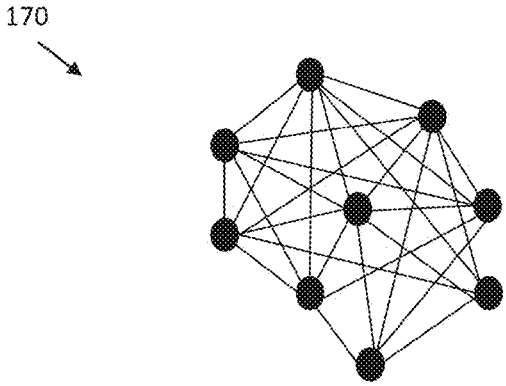
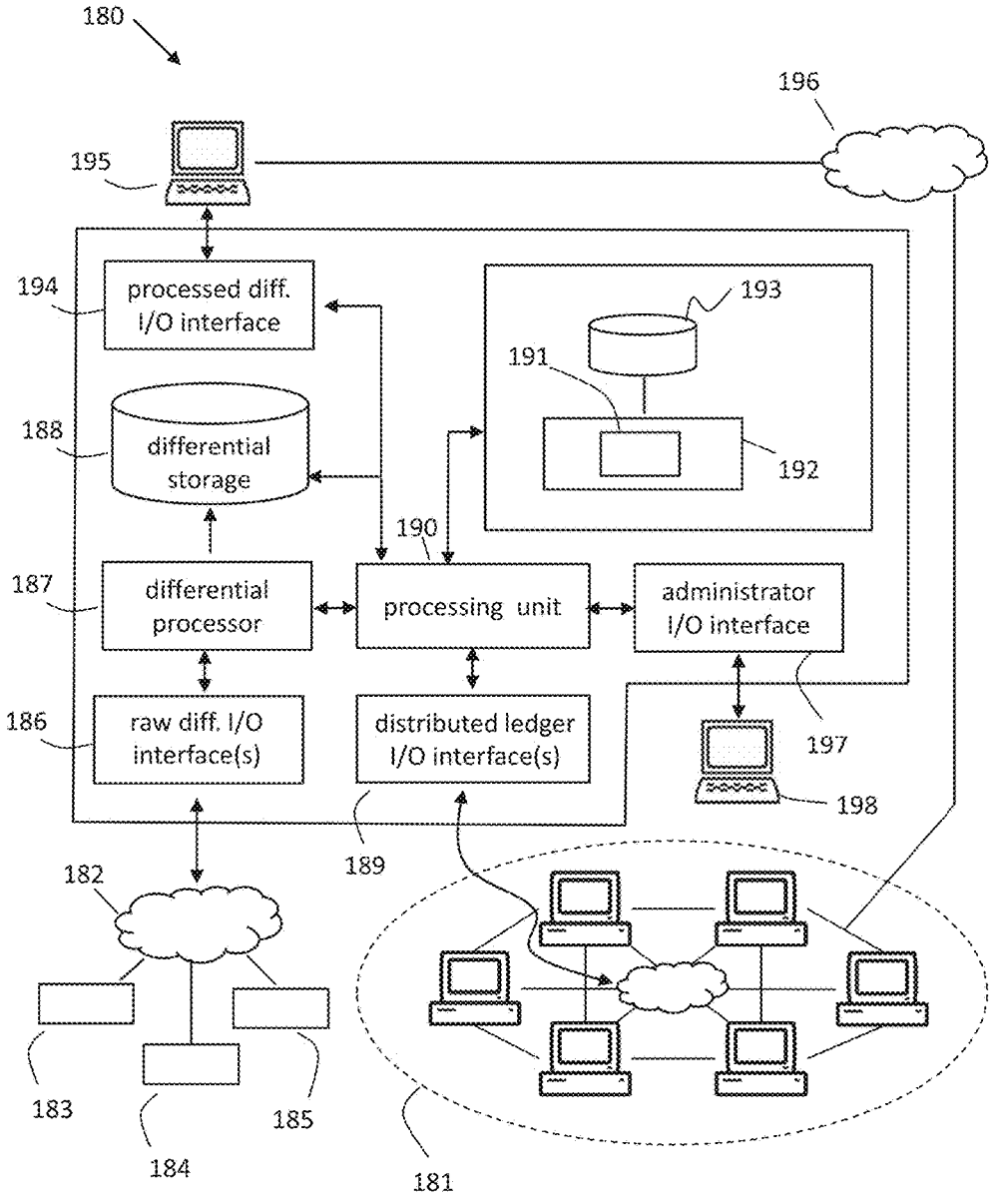


FIG. 18


180
196
195
194
processed diff. I/O interface
193
191
188
differential storage
192
190
187
differential processor
processing unit
administrator I/O interface
186
raw diff. I/O interface(s)
distributed ledger I/O interface(s)
197
198
189
182
183
184
185
181

METHOD AND SYSTEM FOR SEPARATING STORAGE AND PROCESS OF A COMPUTERIZED LEDGER FOR IMPROVED FUNCTION

CROSS-REFERENCE TO RELATED APPLICATION

The present application claims the benefit of the filing date of U.S. Provisional Patent Application No. 62/634,321, filed Feb. 23, 2018, the disclosure of which is hereby incorporated herein by reference.

FIELD OF USE

A novel computer system, network connectivity, and data storage software architecture improves the efficiency, utility and security of a computerized ledger. The disclosed system, method, and computer readable storage medium improve and may be utilized with a wide range of computerized ledgers, including distributed ledgers, decentralized ledgers, and centralized ledgers, where computerized ledgers store and report encrypted, or otherwise secured, electronic transactions, and provide a universal solution to improve the efficiency, utility, and security of computerized ledgers.

BACKGROUND

Generally, computerized ledgers are databases operated on one or more servers by a specialized computer, or operated on a specialized network and controlled by separate computers. A computerized ledger records encrypted or otherwise secured records of transactions, and a computerized ledger can be, among other things, centralized, decentralized, or distributed. Briefly a centralized computerized ledger system is where all nodes connect to a central hub. The management and modifications to the computerized ledger in a centralized environment are generally performed by a centralized computer system and there is usually only one official (or consensus) copy of the computerized ledger. A distributed computerized ledger (DCL) system is where all nodes are independently connected to each other, and the management and modifications to the computerized ledger in a distributed environment are generally performed by separate computers and each computer usually stores its own official copy of the computerized ledger which is proofed for accuracy by a consensus system running on the decentralized network.

The use of distributed computerized ledgers is gaining acceptance and popularity in a number of industrial uses including health care, international trade, and electronic (or crypto) currencies. Distributed ledgers are believed to have a number of advantages over other storage and transaction recording systems. Among the advantages of distributed ledgers are the ability to perform simultaneous updates across multiple fully independent nodes, decreased risk of data loss and corruption through widely distributed consensus-proofed copies, and the ability to create peer-to-peer environments where network validated transactions can be executed with or without a central intermediary. In theory, removing central intermediaries and more directly connecting counterparties through an instantaneous updating and tamper-proof ledger has the promises of improved speed, transparency, and efficiency in related computer systems and networks.

Through supporting systems and internet connectivity, computerized ledgers typically write, encrypt, store, access,

and transmit stored and modified records to specialized computers or specialized networks of computers. DCLs are expected to deliver a number of benefits over alternative storage and access systems including, high levels of security, immutability of transaction records, automated integrity processing, and concurrent read/write capability across multiple nodes. While implementations are currently limited, industry forecasters continue to expect DCLs to store and process transactions relating to commercial goods, health records, tangible property, financial instruments, and other items.

Developers of DCL technology face a number of competing tradeoffs and challenges in function and practical implementation. For example, some of these competing tradeoffs and challenges include secrecy of data, privacy of transactions, speed of recording transactions, speed in updating records, speed in storage and transmission, and full security of the transactions record trail. Typically DCLs engage in redundant movement of transaction data on a peer-to-peer basis such that there is independent processing at every relevant node to facilitate different forms of consensus control and audit, often without the services of a central administrator.

One of the most common data structure formats for DCLs is a block format, in which transactions are aggregated and processed within distinct computer timestamp measured periods of time, and where each aggregation of properly authenticated transactions is written to the DCL in the form of an appended block or comparable structure.

Where a DCL relies on multimode consensus, audit trails and sequencing control may include a range of cryptographic techniques including so-called mining processes based on cryptography or processing power (also known as "proof of work") or proof of stake processes based on holders and holdings within the records providing some validation. Even in some of the least data rich DCLs, such as the block chain implementations of cryptocurrencies (including Bitcoin, Ethereum and the like), the computational burden of basic transactional data in DCLs is slowing networks and jeopardizing recordkeeping, accuracy and potential growth. Cryptocurrencies typically contain only the data necessary to maintain transaction records; as industry attempts to expand the types of DCL applications, higher data requirements are certain to further frustrate processing and transmission speeds.

Most decentralized electronic ledgers (including those used for electronic currencies) are limited in functionality in that their representational blocks are homogenous and their use of timestamped sequencing is limited to curing the "double spend" problem; that is, the transacting of a ledger item which has already been transacted. The most promising known solutions to higher functionality involve pushing more data or computer code through already limited blocked data arrangements.

The promise of DCLs is big, but the industry is still challenged by the barebones data requirements of cryptocurrencies; using known techniques including colored coins and smart contracts to put real estate, health records, commercial transactions, and financial instruments on DCLs is likely to exacerbate current speed and block size challenges. The addition of smart contracts is already introducing serious security concerns.

Expanded implementation of DCLs, for example beyond homogeneous block cryptocurrencies, has been slower than many professionals in computer science, government, and commerce had anticipated. The simplified homogeneous blocks of electronic currencies are already proving difficult

to transact, transmit, and secure; news reports regularly cite problems including delays in validations and settlements, and excessive transaction costs. Proposed extensions of DCLs are generally directed at techniques such as colored coins and smart contracts, however these types of implementations also have many drawbacks including they will: (i) demand continuously revised and customized systems, (ii) add additional pressure to networks and computer systems relating to processing, storage, and transmission, and (iii) introduce vulnerabilities where operative code or descriptors is openly accessible or widely distributed.

Data heavy DCLs (including colored coins and smart contracts) will have a number of drawbacks including: (i) the need for purpose built architecture, operations, and interfaces for new applications and implementations, (ii) the need to coordinate the storage of application specific data and the operations of that data with all possible contributors and users, and (iii) the technological limitations relating to increasing file and block sizes which hampers processing efficiency, decreases practical applications for many users, and severely limits a universal application approach. For example, bitcoin's architecture of 1 gigabyte block sizing and 10-minute block-creation intervals has created an aggregate block chain size of approximately 150 gigabytes, and transaction frequencies limited to fewer than 10 per second; these limitations inherent in the known DCL architecture preclude or severely limit its use in high frequency applications such as retail sales and financial markets. Some systems designers have proposed "trusted systems" for speeding up transactions in which a parallel transaction settlement system is run in conjunction with the block chain; in these systems a trusted intermediary executes rapid transaction settlements on a centralized network, and then the intermediary's transactions are released in bulk to the distributed ledger.

However, these types of parallel transaction settlement systems are known to have many drawbacks. For example, one drawback is that running two systems in parallel demands twice the resources to accomplish the same work as a single system. Another drawback is that, since input errors are always a possibility, the probability of an error increases because the amount of data being input doubles.

Thus there still remains a need in the art for a system and method that provides the advantages of current such computer ledger systems without the above drawbacks. Furthermore there also remains a need in the art for a system and method that is compatible with current technology.

SUMMARY

The present invention solves the problems of current state of the art and provides many more benefits. The disclosed improves the storage efficiency, computational processing, transmission speeds, and functional utility of distributed computerized ledgers (DCLs). Included in the disclosed embodiment is a multiple, parallel, and modular system of storage and processing used to improve the function of the computers running on a network with a DCL. The DCL industry is trying to achieve the reality of a peer-to-peer distributed ledger with immutable records of transactions, commercial practicality, rapid communication and transaction times, and universal techniques to operate over a wide range of applications. Known DCL systems have largely been directed at very narrow applications of electronic currencies (or cryptocurrencies), and those electronic currency applications have highlighted many limitations in transmission and processing speeds in known DCL formats. The proposed system and method is universally compatible with current technologies as well as compatible with current alternative settlement arrangements.

In addition to the data storage and requirements for customization, the disclosed has an important advantage over smart contract and similar coding solutions relating to security and tampering. Because the known methods of smart contracts and colored coins are based on distributed computer code, they are more subject to error, loss of security, and hacking. The networks over Ethereum block chain implementation based on smart contracts were recently hacked, and implementations based on distributed code are particularly vulnerable (see "The Ether Thief", Bloomberg, Matthew Leising, Jun. 30, 2017). The disclosed eliminates the risks relating to distributed code, and eliminates the customization required to create, store, and operate new functionality.

The disclosed system, method and computer readable storage medium also solves certain internet based challenges not previously addressed in the industry, including expanding the capabilities of computerized ledgers and repurposing existing narrowly specified computerized ledgers through at least one processing engine capable of running and storing results in either a parallel or integrated storage architecture of automated system entries for purposes including the creation of new electronic property types, and improving the speed and security of computer based transactional networks. The disclosed embodiment utilizes unconventional architecture and processes not routinely integrated in computerized ledger systems.

Known and deployed DCLs have been largely limited to cryptocurrencies and limited networks to track the shipment of goods. Importantly, known cryptocurrency DCLs carry no data other than cryptographic markers (sometimes operating as unique block indicators) and transaction records; the unit counts (or ledger entries) are treated as the item of value and there is no other value attribution, linkage, and generally no convertibility into tangible property or items. DCLs directed at trade and shipping reside on very limited networks where a small number of permissioned parties perform simultaneous write and read operations to a shared ledger; these limited networks are generally specially purposed and have limited scalability. While there have been high expectation for banks and financial exchanges to employ DCLs across many businesses, practical development and actual implementations have been surprisingly low. In the recently published journal article, "*Blockchain and Financial Market Innovation*", the Federal Reserve Bank of Chicago Economic Perspectives, Vol. 41, No. 7, 2017, the federal bank writes "In order to achieve their full potential, implementations of block chain technology will likely be accompanied by smart contracts. Smart contracts are legal contracts written in computer code that execute automatically once certain conditions, specified in the contract, are fulfilled. Smart contracts can be added to distributed ledgers to self-execute on the basis of information in the ledger . . . ". The Federal Reserve Bank of Chicago's position is representative of the consensus view that writing more data into a DCL framework is the future of growth and expansion.

The other known method for achieving non-homogeneity in the units of a DCL is called "colored coins". The website bitcoinwiki <https://en.bitcoin.it> defines colored coins as: " . . . a class of methods for representing and managing real world assets on top of the bitcoin block chain. While originally designed to be a currency, Bitcoin's scripting language allows to [sic] store small amounts of metadata on the block chain, which can be used to represent asset

manipulation instructions." <https://en.bitcoin.it/wiki/Colored_Coins>. Colored coins are a set of limited and preset encoding techniques for simple re-denominations of coins or for initiating self-executing transfers. The implementation of colored coins puts more overhead demands on already challenged DCL ledger processing by expanding the quantities of redundantly distributed data. Known colored coins are also limited in application, if for no other reason than their data demands become impractical when applied to universal solution sets.

The disclosed embodiment departs from consensus in that it is based on alternative storage processes and architecture. The disclosed embodiment is directed at separating the processes and storage of DCL computers, networks and systems, where only those items required for transaction record keeping are maintained in the fully distributed ledger, and all other data, functionality, and processing is stored in a system of decentralized or centralized storage and processing, linked to the distributed ledger through a combination including timestamps, cryptographic strings, cryptographic nonces, or identifying keys. The disclosed embodiment is directed at a material leap in functionality, utility and speed, and it can be applied to both existing DCL data architectures and newly purposed DCLs. Further, the storage architecture of the disclosed embodiment readily extends to highly efficient mixed real-world use, through a modular system of parallel stored and processed differentials; entirely new applications and enhancements can be added with no additional overhead in the transaction ledgers.

In addition to improving the functionality of the hardware and networking components, the disclosed embodiment is directed at transforming the properties and expanding the functionality entries in a DCL in many unconventional ways. To date DCLs have been limited to currencies of simple multi-node database applications. In order to make DCL technologies widely useful, their underlying entries and transaction records need storage and processing technologies to become representative of, convertible into, or based on real world items of markets and commerce, where those real world items of markets and commerce are represented by a mix of datasets of prices, volumes, dates, settlement particulars and other details. For example, an application of the disclosed embodiment relating to the storage, trade, and transport of steel can carry all of the necessary parameters without additional DCL overhead, where examples of the parameters include: type (carbon, alloy, stainless, tool), grade (SAE—Society of Automotive Engineers codes), application (structural, high tensile, pipe), delivery (date, port), and settlement price and currency (US dollars, Euro, Japanese Yen). Further, the same DCL system design and architecture can immediately accommodate any other application including health care, international trade, or financial instrument applications without recoding or providing for additional overhead.

The Achilles heel of the industry is achieving the combination of speed, functionality, and application versatility, while maintaining the benefits of secure, immutable, and distributed transaction records; the disclosed embodiment solves for the long felt needs of higher efficiency and greater functionality with a new data storage, data processing, and computer functionality. Applied to implementations, the system, method, and computer readable storage medium of the disclosed embodiment are capable of creating entirely new computerized ledger arrangements without additional processing headroom or distributed storage.

The disclosed embodiment employs a non-routine system in which a base DCL, or other computerized ledger, records transactions in sequence; the base computerized ledger transaction records may encrypted and distributed over an internet connected network or encrypted and stored on connected decentralized devices. Among the distinctions between the disclosed embodiment and known and conventional methods is that the disclosed embodiment operates at least one parallel, modular, and separate linked computer storage which accesses one or more exogenous published variables or at least one descriptor difference for the purpose of creating new functionality to a base DCL or other computerized ledger. At least one system created differential is generated from exogenous electronic published data, variables or descriptor, where the time frequency of differential generation and storage (while the electronic published data or variables are changing) is at least as frequent as DCL block creations (or similar transaction recording and sequencing formats) of the base computerized ledger. A system of parallel, modular and separate linked storage of generated differentials expands and redefines the use and application of a base computerized ledger without additional overhead or storage requirements for the base computerized ledger. Transactional records may be transmitted through a network with redundancies, but the differences, measurements, or descriptors are stored in parallel, modular and linked arrangements and not within the transaction records.

The above objects are met by the present invention. In addition the above objects and yet other objects and advantages of the present invention will become apparent from the hereinafter-set forth Brief Description of the Drawings, Detailed Description, and claims appended herewith. These features and other features are described and shown in the following drawings and detailed description.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** illustrates one example of data storage and data structure, where a distributed computerized ledger, a time sequence, and parallel storage of difference layers are indicated on the x, y, and z axes;

FIG. **2** illustrates one example of the population of data items in the three dimensional storage array;

FIG. **3** an example of a data storage array, where the transactional records which are transmitted between and among nodes are distinguished and separated from differential items processed and stored separately from the transactional ledger;

FIG. **4** illustrates an example of electronic linkage between and across a distributed computerized ledger storing transaction records and a related array of centrally or decentralized stored differentials;

FIG. **5** illustrates an example implementation of a distributed transaction ledger and a mixed type storage array of values and differentials;

FIG. **6** illustrates examples of three real world applications;

FIG. **7** illustrates an example of a time sequenced storage and linkage relationship between a base ledger and a related storage array of differentials;

FIG. **8** illustrates a second example of a time sequenced storage and linkage relationship between a base ledger and a related storage array of differentials;

FIG. **9** illustrates an example of a multilevel mixed differences storage array in an example of shipping, goods in transit, and international trade;

FIG. **10** illustrates an example of a multilevel mixed differences storage array in an example of an electronically tradable instrument.

FIG. **11** is a table illustration that is an example of exogenous differential array storage of a single item with an example record format;

FIG. **12** illustrates relative storage locations and functional alignments between distributed computerized ledgers and parallel storage of differences layers;

FIG. **13** illustrates a sample range of parallel storage of difference layer data types;

FIG. **14** is a table that illustrates descriptive and binary string storage;

FIG. **15** is a diagram a basic centralized computerized ledger;

FIG. **16** is a diagram a basic decentralized computerized ledger;

FIG. **17** is a diagram a basic distributed computerized ledger; and

FIG. **18** is a schematic diagram which depicts the internet and network connectivity of the system hardware including the differentials processor, differentials storage, and system non-transitory computer medium and internet linkage to nodes and networks.

DETAILED DESCRIPTION

In general, the present invention overcomes the disadvantages of current computer ledgers. In addition, the present invention works in an entirely new and different way than prior attempts to overcome computer ledger drawbacks. The disclosed embodiment is an unconventional system, method, and computer readable storage medium which creates and operates a high efficiency computerized ledger capable of universal application to a range of industrial, commercial and financial implementations. Included in the disclosed embodiment are computer storage, related processes, and methods which change and expand the use and applications of known computerized ledgers. The disclosed embodiment is directed at non-centralized electronic ledgers in which transaction records are distributed to multiple nodes within a network, and where multiple copies of transaction records (in summary or detail) are maintained on separate computers connected to a network. Such non-centralized networks include both decentralized networks and distributed networks, and also include networks in which participants engage in direct or indirect peer-to-peer transactions. The disclosed embodiment improves the functioning of the related computers, computer systems, the network communication equipment and related processes. Further, the disclosed embodiment expands the usefulness and versatility of existing ledgers, transactional storage systems and distributed ledger communication networks.

The disclosed embodiment reduces the storage requirements of known methods, improves network transaction speeds, reduces the amounts of transmitted data relating to transactions, and expands the utility and functionality of computerized ledgers. The detailed embodiments disclosed are examples, and the systems, methods, and computer readable storage medium can be embodied in many forms. The specific design, structure, and method details described herein are not meant to be limiting. In addition, terms and descriptors used herein are not intended to be limiting, but rather to illustrate the concepts.

Computerized ledgers are generally used to record the transaction records of interests or units, where the interests or units are purchased, sold, transferred, conveyed, split or where ownership stakes are otherwise altered. To date, large scale applications of computerized ledgers have been largely limited to electronic or so-called cryptocurrencies. A cryptocurrency operating on a network has extremely limited data demands; coins are identical and fungible, and the only requirement to track movements in the network is a basic sequenced listing record of transactions of the homogeneous units; all units or interests on the DCL network are identical and fungible and require no stored or process descriptors. The DCL has many benefits in the context of a single coin network, in that the ledger can be fully distributed and transparent. Coin based DCL networks track a single transaction type where validation and immutability of the post-validation record trail require basic cryptographic techniques to properly sequence the peer-to-peer transactions. However, as the industry has attempted expansions beyond homogeneous coins, the simple architecture of the basic ledger has introduced limitations. Widely promoted add-ons such as colored coins and smart contracts, are very limited in implementation, require implementation-specific customization, put much larger demands on data transmission and data storage, and introduce unique security risks.

A number of DCLs are currently in implemented as block chain operations over cryptocurrencies. Block chain (or block chain) ledgers are typically DCL formats where transactions are sequenced and bundled into discrete units or blocks, aggregated by time, where the blocks are sequenced through a combination of timestamps and internal generated cryptographic codes. The basic design objectives of the block chain DCLs are to ensure that transactions which get recorded in a block are validated through a consensus process, that blocks are properly sequenced as they are appended onto the chain, and that their sequencing and recording is immutable.

Known computerized ledgers are principally designed and built for electronic currencies, which did not previously exist, and only exist within the framework of the DCL block chain. Known systems and methods have attempted to extend the DCL to other types of items and applications, most of which require a high degree of detailed specification and data overhead. Systems developers are finding that extending the known DCL methods to applications requiring specification requires overly complex bespoke solutions for each application, and that the bespoke solutions create material burdens on networks.

The disclosed embodiment is a departure in systems, storage, method, and data architecture. The disclosed embodiment changes design and methods of data storage and the functionality of a DCL. The disclosed embodiment is partially based on the concepts: (i) electronic transactions within a DCL can be independent and separately processed from the data items required to specify a value, disposition, distribution, or resolution of a unit of the DCL, (ii) direct processing of a DCL and available network and system capacity must be directed at the highest levels of transaction and execution speed, rather than DCL internal specification, and (iii) many real world applications of DCL will relate to already specified real world objects, and the articulation of those items can generally be dynamically imputed to the DCL interests through linked and modular storage systems.

Included in the purposes of the system's separated parallel storage of differentials dynamically linked to a base DCL are a reduction in redundant data transmission and faster processing of transactions; a base DCL can be entirely repurposed through the disclosed embodiment with no increase in base DCL storage requirements or computational overhead. Also included in the purposes of the modular structure of separated parallel storage of differentials is increased utility and functionality of the systems running basic DCLs; the disclosed embodiment can repurpose a base DCL, by cre-

ating new electronically tradable items without any customization to the DCL, and the modularity of the disclosed embodiment allows rapid and dynamic changes without increased data transmission or data storage overhead.

Beginning with FIG. **1** diagram **10**, the diagram is a simplified illustration of system storage. A base electronic computerized ledger **11** is illustrated as the single layer where computerized ledger transaction records are created and modified including the actions of writing, appending, and reading, where the base electronic ledger resides on a distributed or decentralized ledger. All of the transaction records reside on the base electronic ledger **11**, and transaction records may be appended to the ledger grouped within blocks or appended individually. Transaction records will be identifiable by at least one of a system timestamp, a network timestamp, a unique system generated identifier, or a cryptographic identifier. When written or appended in groups or blocks, the related group or block and component transaction records may be identified collectively in some implementations (the disclosed embodiment reads and processes transaction record data in time sequenced groups), and may be identified independently in other implementations (the disclosed embodiment reads and processes transaction record data based on individual transactions).

Continuing with FIG. **1** diagram **10**, the parallel storage of differences is indicated by parallel storage of difference layer (PSDL) **12** and PSDL **13**. An implementation of the system includes at least one PSDL. Each PSDL will store at least one system written and system accessible time sequenced differential or descriptor, where differentials are created by the system from exogenous and electronically published data streams, and where at least one differences processing engine running on the system computes and stores time sequenced differences from values in the published data stream. Differentials recorded on a PSDL may also include descriptive differentials which can indicate difference types, grades, timeframes or other discriminatory identifiers; descriptive differentials may be utilized with or without data stream differentials. In certain implementations, a descriptive differential is an indirect reference to electronically published data streams; for example a descriptive differential which indicates a certain type of steel of a certain grade to a DCL unit imparts a delivery obligation or value which aligns with one or more electronically published data streams.

The differences residing on a PDSL are applied to the units (or interests) of a DCL upon a system occurrence of an action or process including a value polling, a distribution, a resolution or settlement, or other processes requiring the supplementary data in the PSDL. Cryptographic encoding, transaction validation, and consensus proofing process operations on the DCL may or may not access PSDLs. The system may apply each PDSL to the related units in sequence (i.e. PSDL**1**, then PSDL**2**) or simultaneously (i.e. PSDL**1** and PSDL**2** at the same time). Examples of the time sequenced exogenous and electronically published data include: (i) the prices of computer memory storage devices, (ii) prices of crude oil of differing grades, at different delivery points, denominated in different currencies, (iii) voter counts in statewide election by demographic, party affiliation, and geographic location.

Continuing with FIG. **1** diagram **10**, in DCL implementations of fully distributed or decentralized transaction record storage, the base electronic computerized ledger **11** may be transmitted across a network of computers, where multiple original copies reside on multiple network nodes. The PDSL data **12** and PSDL data **13** (and related processes) are not distributed to every node within the network, and in certain implementations of the system, all PSDL items and processes are maintained on a centralized devices or limited decentralized storage devices. The system's interaction with and between system's stored data layers, and the modifications to the base DCL ledger units (based on the PSDL data storage) may include any mathematical or logic computer operator or operation.

Moving to FIG. **2** diagram **20**, the diagram is an extension of diagram **10**, where diagram **20** illustrates an example of recording transaction records to the base DCL (which is distributed with multiple original copies in a distributed or decentralized network), and the simultaneous, advanced, or lagged writing of differences (on the PSDL) from one or more exogenous electronically published internet data streams or descriptors. The modularity of the linked and mixed storage are illustrated within the three dimensional axes of: (x) time as measured by network computer clocks, or other highly accurate timekeeping devices, (y) the layer or depth of items which relates to the content detail of each appended record in the PSDLs **12** and **13**, and (z) the changeable number and modularity of layers, where storage layers can be added, removed or otherwise modified, and where such modifications directly interoperate with the base DCL without the addition of network or system overhead, and without the security risks of distributed code. Illustrative detailed examples of transaction records are illustrated on the base DCL, where example appended blocks are denoted as "B", and the values "001" through "N" denote the sequencing of the appended blocks. Illustrative examples of PSDL stored differences are denoted on PSDL **12** and PSDL **13**, where "DIF1" and "DIS1" indicate differential items, and the values "001" through "N" illustrate a time alignment of PSDL items with base DCL transaction records.

Diagram **30** of FIG. **3** illustrates the system's separation of the base DCL transaction records **11**, from the parallel storage of difference layers **12**, **13**, and **31**, where a third storage layer in the PSDL is indicated at **31**. Diagram **30** is an example illustration of three separate PSDLs operating over a single base DCL. It can be noted that only the base DCL **11** is redundantly distributed (each may be an original and consensus proofed copy) over individual nodes of the network, while the PSDL data is stored in a centralized or decentralized network. FIG. **3** diagram **30** illustrates an example where multiple modular layers of stored (and operative) differences (the PSDLs) are time sequenced, and where time sequences are aligned with system writing and appending of transaction records in the base DCL individually or in groups (or blocks). Diagram **30** is an example of the modularity of the system and an illustration in the system's efficiency in storage operations. The system's PSDLs are modular, and implementations of the system can create entirely new computerized storage of entirely new functional electronic ledger items using already implemented or new DCLs. Further, because the system separates the storage requirements and computer processing overhead related to the PSDL data, there are no additional DCL storage requirements, and no additional computing overhead on the transaction record keeping network and systems; all while the disclosed embodiment transforms the utility and function of the base DCL.

Referring briefly to FIG. **3**, a base computerized ledger is indicated at **11** where appended blocks (of records) or individual records are added to the ledger in sequence (from left to right in the figure). A modular layer one (**12**), a modular layer two (**13**), and a modular layer three (**31**) are system generated stored differentials or descriptors which

are collectively applied to the base computerized ledger to create a useful, convertible and real world applications where entries within the computerized ledger **11** can have attributable and constructive discriminatory properties, and where differentials relate to any number of objects or transactions including, shipping records, commodity delivery, or a bespoke financial instrument. The base computerized ledger **11** may be redundantly transmitted and modified across a distributed network similar to the manner in which known electronic currencies operate, but the operative modifying modular layers are generated and stored once or in other limited decentralized distribution, and linked to the base computerized ledger **11** through a computer generated timestamp, a timestamp sequenced key, a unique character string, a cryptographic nonce, or similar unique identifier; records of the modular layers which differ in value or descriptor and time will have a unique alignment with records in the base DCL. Complex and multifactor arrangements may be stored, processed, and transmitted without the introduction of additional overhead in the transacting network and without the customization which is required with the implementation of techniques such as colored coins or smart contracts.

FIG. **4**, diagram **40** is an illustration of an example of system data linkages of the distributed or decentralized base DCL to the centralized or decentralized PSDL. Beginning with the base DCL **11**, basic details relating to a transaction or a group of transactions are indicated in the base DCL diagram **11**. Beginning with "Timestamp" and moving clockwise, "Timestamp" is generally a unique entry in the DCL, recording the system time when a block was found (in certain cryptographic mining processes) or recording the system time when one or more transactions are written on or appended to the base DCL dataset. "Nonce" is generally a one-time use number added to a group or block record in certain base DCL applications. The "Nonce" serves many purposes including cryptographic security, immutable record sequencing and it is part of the consensus validation in certain applications involving cryptographic mining. The "Tx_Root" is generally the connection to recorded transactions, and "Prev_Hash" is a hash from the immediately preceding block, which ensures that each block is immutably tied to previous block.

Continuing with FIG. **4** diagram **40**, example detail is added to PSDL **12** and PSDL **13**. Differing implementations may include different PSDL detail, and importantly, there is no requirement that the data stored in each PSDL be identical. "Time PSDL" is an example of the principle linkage, where the field stores a unique identifier which may be in the form of a string, or value such that the value is linked to a concurrent, delayed, or advanced identifying entry in the DCL. "Dif1_C(N)" is a difference measuring a subject data items over a defined period of time, where "Dif1_C(N)" can be descriptive or arithmetic. "Cur_Val" and "Prev_Val" are descriptive or arithmetic values stored on PSDL **12** and PSDL **13** and relate to exogenous items, where "Prev_Val" indicates the immediately preceding or early system time, and "Cur_Val" indicates a later time or current time. Both the DCL and PSDL have independent record sequences, and each record, block, or transaction within a DCL will have an attribution to a location in the stored PSDL, and the DCL-to-PSDL connector **41** links the distributed DCL to the centralized or decentralized PSDLs based on unique codes, time indicators, or similar items.

FIG. **5** diagram **50** is an example of the increased functionality of a base DCL format through the storage of the disclosed embodiment. Three PSDLs are illustrated as examples relating to using the disclosed embodiment in the context of commodities transport and trade. The base DCL **11** is shown, and the PSDLs **51**, **52**, and **53** illustrate example specifications for a PSDL linked to the base DCL. In each illustrated example (of three sets of PSDLs), the lower layer indicates a shipped commodity type (Type W, B, and D), the middle layer indicates the currency in the data (EUR=Euro, USD=U.S. Dollar, and SGD=Singapore Dollar), and the topmost layer indicates descriptive differences (−1, +1 and −1, from right to left at **51**, **52**, and **53** respectively). PSDL **51** is an example where the exogenous data streams are Type W (WTI Crude Oil), and Euro currency (EUR), and a descriptive difference of −1, where −1 may indicate an obligation to deliver WTI Crude Oil denominated in Euros. PSDL **52** is an example where the exogenous data streams are Type B (Brent Crude Oil), and U.S. Dollars currency (USD), and a descriptive difference of +1, where +1 may indicate an obligation (or operative entitlement) to take delivery of Brent Crude Oil denominated in USD. PSDL **53** is an example where the exogenous data streams are Type D (Dubai Crude Oil), and Singapore Dollars currency (SGD), and a descriptive difference of −1, where −1 may indicate an obligation (or operative entitlement) to make delivery of Dubai Crude Oil denominated in SGD. The lowest two layers of each PSDL illustrate stored data and operations are over a time sequenced exogenous data item; values are drawn from one or more internet data streams containing the requisite commodities and currency values. The topmost layer is a descriptive difference layer, where −1 may be used to indicate a delivery obligation, and where +1 may be used to indicate an obligation to take delivery; descriptive difference layers may be used to change the direction and responsiveness to one or more exogenous data items. The disclosed embodiment is not expected to alter the quantities, prices, or types of commodities which might be transacted and monitored in a computer system organized to manage and monitor commercial trade. However, the disclosed embodiment is expected to materially decrease the storage and transmission requirements of related distributed ledger networks and computer systems. Further, the centralized or decentralized storage of the PSDL data and operations is expected to material increase the network security of the computer systems and networks.

Moving to FIG. **6** diagram **60**, values of differences are illustrated for PSDL **51**, PSDL **52**, and PSDL **53**. The value of differences is generated by the system from one of more internet data streams, and where practical, the values of differences are generated, stored, and linked with a frequency which matches or exceeds the frequency used for appending transactions records to the base DCL **11** during periods in which the values of difference published in an internet data streams are changing. In each case, the value differences illustrated for TYPE W, TYPE B, TYPE D, EUR, USD, and SGD are differences in percentage changes from the immediately preceding period; in alternate implementations, absolute values or other measured changes may be generated, stored and applied. The value differences in the right-most layer of each PSDL are descriptive value differences which may indicate a relative directionality or degree of difference application. For clarity, one PSDL is generally applied to only one base DCL, however a single PSDL may have more stored layers than are indicated in the figures.

Continuing with FIG. **6** diagram **60**, one example of applying a PSDL to the units of the base DCL **11** is through the use of computer mathematic operators, where each layer's numerical difference storage layer is applied to

produce an aggregate impact. Beginning with PSDL **51**, the base DCL **11** units can be modified in one example by using a multiplication operator for each layer, where the units are modified by the product of: 1.1% (Type W), 0.1% (EUR), and −1 (L:−1) for a modification of −1.012011% (1.011× 1.001×−1). Applying the same operator to PSDL **52** and PSDL **53** results in 1.009%, and −1.00798% respectively. For clarity, the base DCL is stored and transacted independent of the PSDL storage. Also for clarity, PSDL storage may impart absolute values (where absolute values are subject to differences over time sequences), descriptive characteristics, or a mix of absolute values, relative values, and descriptors. The mathematical functions, operators and results are incidental to the disclosed embodiment's storage and processing and not central to the operation of the specialized system or storage design. The PSDL modifications may be electronically published over a network or internet such that holders or transactors can monitor aggregate modifying impacts, and the impact of PSDL modification may be coupled with the base DCL to direct reporting to holders, transactors, and other participants. Importantly, the system data, differences, and processes of the PSDLs is not propagated through or contained in the transaction ledger.

FIG. **7** diagram **70** is an example of alternative alignments of PSDL data storage **71**, and base DCL data storage **11**. Timeline **72** indicates a starting time of T(**0**) and three forward time intervals indicated as T(**1**), T(**2**), and T(**3**). PSDL data storage **71** is an aggregation of individual PSDL layers **12**, **13**, and **31**. In diagram **70**, the units of the base DCL **11** are impacted by forward-looking or advance values and differences of three PSDLs. Importantly, the PSDL data and processes are not stored in the base DCL nor is the PSDL data and processes redundantly distributed throughout nodes of the network as are known methods such as smart contracts and colored coins.

FIG. **8** diagram **80** is a second example of an alternative alignment of PSDL storage **71**, and base DCL storage **11**. Timeline **81** indicates a starting time of T(**0**) and two additional arrears time intervals indicated as T(**1**), T(**2**), and a current time of T(**3**). PSDL data storage **71** is an aggregation of individual PSDL layers **12**, **13**, and **31**. In diagram **80**, the units of the base DCL **11** are impacted by backward-looking or arrears values and differences of three PSDLs. Relating to diagrams **70** and **80**, because non-transaction record data is not distributed through all nodes of the network, but rather is separately and centrally stored in a modular framework, functionality and utility is materially increased because PSDL are limited in distribution and modular, transaction and speeds are improved because only transaction records need be fully distributed, and security is enhanced because transaction records can benefit from an immutable sequencing on the ledger, and operative differences PSDL data and processes can be stored on secured centralized or decentralized systems.

Referring briefly to FIG. **9**, a base computerized ledger (**11**) is operatively linked to parallel and modular differences of factors one through three (**91**, **92**, and **93** respectively, stored in parallel storage of differences layers **11**, **12**, and **31** respectively) over a multi-time period measured and stored differences, only the single record indicated at base computerized ledger **11** is transmitted across the multiple nodes of the electronic network; the parallel and modular differences of factors are centrally stored, not distributed with the transactions ledger, and accessed only when units (or interests) underlying the base computerized ledger (**11**) are subject to a resolution, disposition, valuation, settlement, distribution or another action requiring differentials and descriptors. FIG. **9** is an illustration of one particular implementation where highly reduced storage and high functionality is achieved. Based on 3 modular levels of parallel storage, each containing 6 stored attributes, 6^3 or 216 unique applications are available with one base computerized ledger data structure **11**, and data transmission requirements and storage redundancy is minimized. In contrast, units of known distributed electronic ledgers are homogeneous; one bitcoin or Ethereum coin is identical to any other and the addition of features using known methods where additional items are added to the blocks (or other sequential records) expand the block storage and transmission overhead exponential and cause an increase is the amount of redundant data which propagates throughout network nodes.

FIG. **9**, diagram **90** illustrates an example of the storage of illustrative PSDLs **12**, **13**, and **31**. Diagram **90** is an example of the PSDL storage and operation over base DCL **11**, where a PSDL may contain value differences or descriptive differences. The PSDLs in the example relate to a real world example of oil trade in which PSDL **12** indicates a type of crude oil (WTI, WTI-H, WTS, MARS, LOOP indicated at **91**), PSDL **13** indicates a port (or delivery) location (PORT_A, PORT_B, PORT_C indicated at **92**), and PSDL **31** indicates a delivery time or delivery cycle (1 W, 1 M, 2 M indicated at **93**). PSDL **12**, PSDL **13**, and PSDL **31** all operate over the units or interests of the base DCL **11**, however all storage relating to the PSDL data is not stored in the base DCL, but rather is stored centrally and linked to the resultant transaction records of the base DCL. One example of value differentials for oil type **91** is a storage of time-sequenced price values where the prices of one or more of the oil types is recorded in the PSDL for the storage and application of differences. Another example for oil type differentials **91** is the use of descriptive differentials relating to oil types **91**, where the system stores and applies strings or binary encoding rather than basic numerical values; a string example is "WTI-H", and the binary descriptive coding is 01000 to indicate the second oil type in a list of five.

FIG. **10** diagram **100** is an extended example of diagram **90**, which illustrates an implementation in the context of the creation a range of bespoke instruments over a base DCL **11** with 3 parallel storage of difference layers. In diagram **100**, the lowest storage layer **12** is a differences of currencies, where the currencies indicated in a selector **101** are U.S. Dollar (USD), Euro (EUR), Japanese Yen (JPY), Chinese Yuan (CNY), and British Pound Sterling (GBP); storage layer **12** can contain descriptive differences (e.g. which among the group of currencies is indicated), stored value absolute differences for relating to absolute values such as foreign exchange rates in a time sequence, or relative change or percentage differentials in a time sequences; descriptive differentials may be stored and indicated as indicators, indicator flags, binary values, or character strings. PSDL **13**, the second storage layer is used to indicate tenor of an actual or synthetic position desired for the aggregate PSDL (where the aggregate PSDL is the combination of storage levels **12**, **13**, and **31**). Relating to PSDL **13**, selector **102** can indicate "1 d", "1 w", "1 m", "1 Y" (1 day, 1 week, 1 month, and 1 year respectively). Relating to PSDL **31**, selector **103** references specific instruments which can be stored as description differences or time sequenced differences of absolute values or relative percentage changes. Selector **103** can indicate and store SPX, GT10, NDX, or HYLD (an equity market index, a bond index, a tech equity index, and a credit index respectively).

FIG. **11** diagram **110** is an example of a system storage record which may be stored within a PSDL. Table **111** indicates an example of a particular record where the fields are indicated as "identifier", "timestamp(0)", timestamp(t)" "val(0)", "val(t)", "dif(t)", and "cond". Identifier is an example of an encoding which is used to identify the subject of the stored differentials or descriptors. Timestamp(0) and timestamp(1) indicate the beginning and ending timestamps of a computer system or network time (and the related period) over which the stored difference is based. Val(0) and val(1) indicate the respective values of the subject item taken from an electronic published data stream, where val(0) relates to timestamp(0) in observation time, and where val(1) related to timestamp(1) in observation time. Dif(t) is an example of a numerical difference over the subject data stream object from timestamp(0) to timestamp(1). The field "cond." may be used to indicate successful retrieval from an internet data stream. The values and forms of values in table **111** are examples; implementations may present different record architecture, different stored data items, and different types of values.

FIG. **12** diagram **120** illustrates the separation of storage of the base DCL **122** and the PSDL layers **125** and **126**. Line **121** illustrates the dividing line of storage where items above the line are stored on the system in centralized or decentralized storage, and those items below the line are stored in a distributed ledger. Diagram **120** is an example of the modularity of the system, and two PSDLs are illustrated at **125** and **126** respectively. The single base DCL (labelled "BB" or base block records) is indicated at **122**. In each of the four columns of diagram **120**, the base DCL **122** may be impacted by each PSDL **126** and PSDL **125** vertical pair individually, where each time-sequenced entry in the aggregate PSDL is independent from the others, or the base DCL **122** may be impacted by the PSDL **126** and PSDL **125** aggregate PSDL in a cumulative or compounding manner, where the impact is cumulative as time moves from left to right as indicated by the time indicators t(**1**), t(**2**), t(**3**), and t(**4**).

FIG. **13** diagram **130** illustrates an example of differentials stored and processed by the system, where differentials are based on any mathematical or computer operator. Row **131** illustrates a time sequenced vector of values equal to 100, 101, 101.5, and 99. Rows **132**, **133**, and **134** are illustrations of differentials stored and processed by the system. In the example of row **132**, the system stores and processes differentials based on value change; value changes may be based on differences or absolute values. In the example of row **133**, the system stores and processes changing time sequenced values as the value of the time sequenced different and changing values. In the example if row **134**, the system stores and processes the values as percentage differences based on the immediately preceding value.

FIG. **14** diagram **140** illustrates an example of differentials where differentials are descriptive. As illustrated in the table of diagram **140**, the first column entitled "VAL" lists a set of unique string values "USD", "EUR", "JPY", "CNY", and "GBP". As illustrated in the example of the column entitled "DIF1", the differentials stored and processed by the system may be the actual string values which differ within the set of values (i.e. the differential descriptor). As illustrated in example of the column entitled "DIF2", the differentials stored and processed may be an indicator, flag, or binary string which indicates a value out of a set; where the set is comprised of 6 values, "100000" may be used to indicate the first value ("USD"), and "000001" may be used to indicate the last value "GBP". The remote and linked storage of descriptive items can be used to transform the nature of units within a base DCL or to strategically discriminate within the units for specialized remote processing or handling; units become representative of other items such as an industrial good, a commodity, or an airline ticket, and units within the block can be further discriminated by time or other transaction record parameters.

FIG. **15** diagram **150** is an example of a centralized network, as such term is used in the disclosed embodiment. Generally, all nodes in the network connect in a centralized manner to a central point or hub. Security in a centralized network is generally based on securing the media, device, and operating processes at the central point, and limiting write, modification and certain read access at the centralized point. Centralized networks enjoy the security advantage of having limited points or vulnerability, however, centralized networks can become highly compromised if the central point is breached.

FIG. **16** diagram **160** is an example of a decentralized network as such term is used in the disclosed embodiment. Generally, there are a number of central or connecting points at which individual nodes may connect. Similar to a centralized network, security in a decentralized network is generally based on securing the media, device, and operating processes at the common connection points, and limiting write, modification, and certain read access at the decentralized points.

FIG. **17** diagram **170** is an example of a distributed network as such term is used in the disclosed embodiment. Generally, each individual node is connected (or capable of being connected) to every other node in the network. Implementations of distributed computerized ledgers are generally configured in a manner similar to FIG. **17**. Security in a distributed network is often achieved through cryptographic techniques within a ledger in combination with a consensus system for validating transaction records written and appended to the ledger. Further, diagram **170** is an illustrating example of the configuration of one base DCL, where a ledger of transaction records is distributed among connected computer nodes. For clarity, intermediary nodes may be used as surrogates for discrete nodes, where such intermediary nodes are exchanges or other intermediary service providers who hold interests in a base DCL for limited periods of time or for rapid collective settlement.

Referring to FIG. **18** diagram **180**, it is to be understood that the disclosed embodiment includes storage of differentials and processing of differentials computer node **191**, where differentials and storage processing software is stored on system RAM **192**. The system software instructs the system continuously relating to all aspects of the differentials storage, including processing and storage of each modular PSDL, interconnectivity with at least one time-sequenced electronically published data stream or descriptive differential, and linkage to at least one base DCL. The differentials computer node **191** and related RAM may also be linked to a differentials computer node storage device **193**.

Continuing with FIG. **18**, an example of a base DCL is indicated at **181** where the separate computing nodes of a network are interconnected through the internet or a network, and where a ledger of transactions is distributed across the network. An example of the base DCL connection with the rest of the system is indicated at the distributed ledger I/O interface(s) **189**, where the distributed ledger I/O interface(s) **189** may operate in both a transmission and receiving mode. The distributed ledger I/O interface(s) **189** is further connected to a system processing unit **190**.

Continuing with FIG. **18**, an example of a raw differential I/O interface(s) **186** is connected through an internet connection **182** for the purposes of retrieving one or more time-sequenced electronically published data streams or a descriptive differentials, where a time-sequenced data stream may relate to prices, trade flows, trade variables, shipping details, economic variables, performance measures or other numerical or descriptive data. Examples of source nodes connected to the internet include: (i) a commercial trade, tracking, or shipping network run by a company, industry group, or governmental entity **183**, (ii) an electronic exchange **184** which publishes a price data stream of changing market prices, and (iii) an electronic news outlet **185** which publishes electronic data relating to changing news. Continuing to the differential processor **187**, the differential processor **187** assimilates at least one value or descriptive differential through a computer or mathematical operation, forming the data into a useable format where it can be stored on differential storage unit **188**, for simultaneous or subsequent application to the units or interests of a base DCL with network **181**. Continuing with the processing of differentials, the system processing unit **190** is connected to the differential processor and the differentials computer node **191** such that the system can effect differential data transmissions and responses to differential data queries through a processed differential data I/O interface **194** which is controlled by a differentials administration gateway terminal **195**. An example of processed differential transmission, dissemination, and query management is illustrated at internet connection **196** in which the system can both broadcast processed differential data and respond to queries. When a unit or interest of the base DCL requires valuation, settlement, exchange, or resolution, the system can be polled for a valuation or impact on a unit, record, or interest of the DCL as of a particular time, or over a particular period of time.

Continuing with diagram **180**, an example of an administrator interface **197** is controlled by an administrator console **198**, where the specifications of PSDLs, related exogenous data streams, and connectivity to a base DCL is established, controlled, and modified.

The disclosed embodiment is a system, method, and computer readable storage medium related to the systems, network connectivity, software, and data storage architecture in applications over computerized ledgers. The disclosed system, method and computer readable storage medium is directed at a range of computerized ledgers, including distributed ledgers, decentralized ledgers, and centralized ledgers, where computerized ledgers store and report encrypted, or otherwise secured, electronic transactions. The disclosed system, including its data storage features, computer readable storage medium, and methods are directed at universal solutions to improve the efficiency and utility of computers and networks operating computerized ledgers.

The above disclosed embodiments are not intended to limit the scope of the invention but are examples thereof. Although the invention herein has been described with reference to particular embodiments, it is to be understood that these embodiments are merely illustrative of the principles and applications of the present invention. The above disclosed embodiments were chosen and described to most clearly explain the principles of the invention and practical applications, and to enable others skilled in the art to understand the invention for various embodiments. It is therefore to be understood that numerous modifications may be made to the illustrative embodiments and that other arrangements may be devised without departing from the scope and spirit of the present invention as defined by the appended claims.

What is claimed:

1. A computer based method comprising:

creating at least one electronic parallel storage of a differences layer linked to a distributed computer ledger (DCL); the DCL contains an electronic transaction record by a time-sequenced value or a time-sequenced string;

accessing and storing a value through the at least one electronic parallel storage of the differences layer, the value from a group comprising of at least one time-sequenced electronically published data stream and at least one descriptive differential, wherein at least one differences processing engine running on a specialized computer system creates and stores parameters from a group comprised of a measurement differences and a descriptive differences;

storing the DCL containing an electronic transactions record on at least one of a distributed network of connected independent computers or a decentralized network of computers wherein the electronic transaction record is time sequenced, and a writing or an appending of the electronic transaction records is performed on the distributed network of connected independent computers or the decentralized network of computers;

storing the at least one electronic parallel storage of the differences layer on at least one of a centralized storage device controlled by the specialized computer system or a decentralized storage device controlled by the specialized computer system for increasing functionality and utility of the DCL, reducing data storage requirements, eliminating transmission of redundant data, and improving data security;

linking the electronic transaction record in the DCL to records of the at least one electronic parallel storage of the differences layer utilizing at least one time sequenced value, string, code, or key; and

imputing at least one measured differential with a descriptive identifier or at least one descriptive identifier to the electronic transaction record of the DCL through data storage and processing on the at least one electronic parallel storage of the differences layer.

2. The method of claim **1**, wherein records of the at least one electronic parallel storage of the differences layer are written and stored separately from the distributed electronic ledger containing electronic transaction records, where the records of the at least one electronic parallel storage of the differences layer are encoded for time-sequenced alignment with the electronic transaction records when values from a group comprised of the at least one time-sequenced electronically published data stream and the at least one descriptive differential change in value or specification.

3. The method of claim **1**, wherein values and descriptors from a group comprised of the at least one time-sequenced electronically published data stream and the at least one descriptive differential alter the functionality and transactional value of the electronic transaction records of the distributed electronic ledger.

4. The method of claim **1**, wherein values and descriptors from a group comprised of the at least one time-sequenced electronically published data stream and the at least one descriptive differential define the functionality and operative entitlement of the electronic transaction records of the distributed electronic ledger.

5. The method of claim 1, wherein values from a group consisting of at least one time-sequenced electronically published data stream and at least one descriptive differential are linked to the electronic transaction records within the distributed electronic ledger and the electronic transaction records are homogeneous on the distributed electronic ledger as identified by a timestamp or other unique record identifier.

6. The method of claim 1, wherein layers of the at least one electronic parallel storage of the differences layer linked are modular and changeable independent of the distributed electronic ledger containing electronic transaction records.

7. A system comprising:

a system having a memory device, the memory device further including a Random Access Memory (RAM);

a processor connected to the memory device, the processor is configured to:

create at least one electronic parallel storage of a differences layer linked to a distributed computer ledger (DCL), both the electronic parallel storage of the differences layer and the DCL containing a respective electronic transaction record, a time-sequenced value, or a time-sequenced string;

access a value from a group comprising of at least one time-sequenced electronically published data stream and at least one descriptive differential;

store the values from a group comprising of at least one time-sequenced electronically published data stream and at least one descriptive differential on the at least one electronic parallel storage of the differences layer;

align and link a stored value record of the at least one electronic parallel storage of the differences layer to the electronic transaction record of the DCL utilizing at least one time sequenced value, string, code, or key; and

impute at least one measured differential with a descriptive identifier or at least one descriptive identifier to the electronic transaction record of the DCL.

8. The system of claim 7, wherein the memory device includes a separation of storage of the differences layer.

9. The system of claim 8, wherein the separation of storage is between the electronic transaction record of the DCL and the differences layer.

10. The system of claim 9, wherein a plurality of differences layer is parallel stored to create a parallel storage of differences layer (PSDL).

11. The system of claim 7, wherein the difference layer is stored on a centralized storage or a decentralized storage apart from the electronic transaction record of the DCL.

12. The system of claim 11, wherein the electronic transaction record of the DCL is impacted by a parallel storage of differences layer.

13. The system of claim 12, wherein impact is done from each of the parallel storage of differences layer (PSDL) in an individual manner.

14. The system of claim 13, wherein the parallel storage of differences layer (PSDL) has a time-sequence entry, and each time-sequenced entry is independent in the PSDL.

15. The system of claim 12, wherein impact is done from the parallel storage of differences layer (PSDL) in a cumulative manner, or a compounding manner, wherein impact is cumulative based on a time indicator.

16. The system of claim 15, wherein the parallel storage of differences layer (PSDL) has a time-sequence entry, and each time-sequenced entry is independent or dependent in the PSDL.

17. The system of claim 7, wherein the difference layer is stored on a distributed network, a centralized network, or a decentralized network, and wherein the difference layer is stored apart from the electronic transaction record of the DCL.

18. The system of claim 17, wherein the electronic transaction record of the DCL is impacted by the differences layer.

19. A non-transitory computer readable storage medium, comprising storage, retrieval, modification, and linking system software which instructs at least one computer processor residing on a specialized computer system to implement a process to:

create at least one electronic parallel storage of a differences layer linked to a distributed computer ledger (DCL) containing an electronic transaction record arranged by a time-sequenced value or time-sequenced string, wherein the at least one electronic parallel storage of the differences layer accesses and stores values from a group consisting of at least one time-sequenced electronically published data stream and a list of descriptive differentials, and wherein at least one differences processing engine running on a specialized computer system creates and stores parameters from a group consisting of measurement differences and descriptive differences;

store the DCL containing the electronic transactions records on at least one of a distributed network of connected independent computers or a decentralized network of computers wherein the electronic transaction records are time sequenced, and the writing or appending of the electronic transaction records is performed on the distributed network of connected independent computers or the decentralized network of computers;

store the at least one electronic parallel storage of the differences layer on at least one of a centralized storage device controlled by the specialized computer system or a decentralized storage device for increasing functionality and utility of the DCL, reducing data storage requirements, eliminating transmission of redundant data, and improving data security;

link the transaction records in the DCL to the at least one electronic parallel storage of the differences layer utilizing at least one time sequenced value, string, code, or key; and

impute at least one measured differential with a descriptive identifier or at least one descriptive identifier to the electronic transaction records of the DCL, wherein a data storage and a processing of the imputing resides on a centralized device or a decentralized device controlled by the specialized computer system.

20. The non-transitory computer readable storage medium of claim 19, wherein the difference layer is stored apart from the electronic transaction record of the DCL, and the electronic transaction record of the DCL is impacted by the differences layer.

* * * * *

EXHIBIT B

Compound:
The Money Market Protocol

Version 1.0
February 2019

Authors
Robert Leshner, Geoffrey Hayes
https://compound.finance

Abstract
In this paper we introduce a decentralized protocol which establishes money markets with algorithmically set interest rates based on supply and demand, allowing users to frictionlessly exchange the time value of Ethereum assets.

Contents

1 Introduction

The market for cryptocurrencies and digital blockchain assets has developed into a vibrant ecosystem of investors, speculators, and traders, exchanging thousands [1] of blockchain assets. Unfortunately, the sophistication of financial markets hasn't followed: participants have little capability of trading the *time value* of assets.

Interest rates fill the gap between people with surplus assets they can't use, and people without assets (that have a productive or investment use); trading the time value of assets benefits both parties, and creates non-zero-sum wealth. For blockchain assets, two major flaws exist today:

- Borrowing mechanisms are extremely limited, which contributes to mispriced assets (e.g. "scamcoins" with unfathomable valuations, because there's no way to short them).
- Blockchain assets have negative yield, resulting from significant storage costs and risks (both on-exchange and off-exchange), without natural interest rates to offset those costs. This contributes to volatility, as holding is disincentivized.

Centralized exchanges (including Bitfinex, Poloniex...) allow customers to trade blockchain assets on margin, with "borrowing markets" built into the exchange. These are trust-based systems (you have to trust that the exchange won't get hacked, abscond with your assets, or incorrectly close out your position), are limited to certain customer groups, and limited to a small number of (the most mainstream) assets. Finally, balances and positions are virtual; you can't move a position on-chain, for example to use borrowed Ether or tokens in a smart contract or ICO, making these facilities inaccessible to dApps [2].

Peer to peer protocols facilitate collateralized and uncollateralized loans between market participants directly. Unfortunately, decentralization forces significant costs and frictions onto users; in every protocol reviewed, lenders are required to post, manage, and (in the event of collateralized loans) supervise loan offers and active loans, and loan fulfillment is often slow & asynchronous (loans have to be funded, which takes time) [3-6].

In this paper, we introduce a decentralized system for the frictionless borrowing of Ethereum tokens without the flaws of existing approaches, enabling proper money markets to function, and creating a safe positive-yield approach to storing assets.

2 The Compound Protocol

Compound is a protocol on the Ethereum blockchain that establishes money markets, which are pools of assets with algorithmically derived interest rates, based on the supply and demand for the asset. Suppliers (and borrowers) of an asset interact directly with the protocol, earning (and paying) a floating interest rate, without having to negotiate terms such as maturity, interest rate, or collateral with a peer or counterparty.

Each money market is unique to an Ethereum asset (such as Ether, an ERC-20 stablecoin such as Dai, or an ERC-20 utility token such as Augur), and contains a transparent and publicly-inspectable ledger, with a record of all transactions and historical interest rates.

2.1 Supplying Assets

Unlike an exchange or peer-to-peer platform, where a user's assets are matched and lent to another user, the Compound protocol aggregates the supply of each user; when a user supplies an asset, it becomes a fungible resource. This approach offers significantly more liquidity than direct lending; unless *every* asset in a market is borrowed (see below: the protocol incentivizes liquidity), users can withdraw their assets at any time, without waiting for a specific loan to mature.

Assets supplied to a market are represented by an ERC-20 token balance ("cToken"), which entitles the owner to an increasing quantity of the underlying asset. As the money market accrues interest, which is a function of borrowing demand, cTokens become convertible into an increasing amount of the underlying asset. In this way, earning interest is as simple as holding a ERC-20 cToken.

2.1.1 Primary Use Cases

Individuals with long-term investments in Ether and tokens ("HODLers") can use a Compound money market as a source of additional returns on their investment. For example, a user that owns Augur can supply their tokens to the Compound protocol, and earn interest (denominated in Augur) without having to manage their asset, fulfill loan requests or take speculative risks.

dApps, machines, and exchanges with token balances can use the Compound protocol as a source of monetization and incremental returns by "sweeping" balances; this has the potential to unlock entirely new business models for the Ethereum ecosystem.

2.2 Borrowing Assets

Compound allows users to frictionlessly borrow from the protocol, using cTokens as collateral, for use anywhere in the Ethereum ecosystem. Unlike peer-to-peer protocols, borrowing from Compound simply requires a user to specify a desired asset; there are no terms to negotiate, maturity dates, or funding periods; borrowing is instant and predictable. Similar to supplying an asset, each money market has a floating interest rate, set by market forces, which determines the borrowing cost for each asset.

2.2.1 Collateral Value

Assets held by the protocol (represented by ownership of a cToken) are used as collateral to borrow from the protocol. Each market has a collateral factor, ranging from 0 to 1, that represents the portion of the underlying asset value that can be borrowed. Illiquid, small-cap assets have low collateral factors; they do not make good collateral, while liquid, high-cap assets have high collateral

factors. The sum of the value of an accounts underlying token balances, multiplied by the collateral factors, equals a user's ***borrowing capacity***.

Users are able to borrow up to, but not exceeding, their borrowing capacity, and an account can take no action (e.g. borrow, transfer cToken collateral, or redeem cToken collateral) that would raise the total value of borrowed assets above their borrowing capacity; this protects the protocol from default risk.

2.2.2 Risk & Liquidation

If the value of an account's borrowing outstanding exceeds their borrowing capacity, a portion of the outstanding borrowing may be repaid in exchange for the user's cToken collateral, at the current market price minus a ***liquidation discount***; this incentives an ecosystem of arbitrageurs to quickly step in to reduce the borrower's exposure, and eliminate the protocol's risk.

The proportion eligible to be closed, a ***close factor***, is the portion of the borrowed asset that can be repaid, and ranges from 0 to 1, such as 25%. The liquidation process may continue to be called until the user's borrowing is less than their borrowing capacity.

Any Ethereum address that possesses the borrowed asset may invoke the liquidation function, exchanging their asset for the borrower's cToken collateral. As both users, both assets, and prices are all contained within the Compound protocol, liquidation is frictionless and does not rely on any outside systems or order-books.

2.2.3 Primary Use Cases

The ability to seamlessly hold new assets (without selling or rearranging a portfolio) gives new superpowers to dApp consumers, traders and developers:

- Without having to wait for an order to fill, or requiring off-chain behavior, dApps can borrow tokens to use in the Ethereum ecosystem, such as to purchase computing power on the Golem network
- Traders can finance new ICO investments by borrowing Ether, using their existing portfolio as collateral
- Traders looking to short a token can borrow it, send it to an exchange and sell the token, profiting from declines in overvalued tokens

2.3 Interest Rate Model

Rather than individual suppliers or borrowers having to negotiate over terms and rates, the Compound protocol utilizes an interest rate model that achieves an interest rate equilibrium, in each money market, based on supply and demand. Following economic theory, interest rates (the "price" of money) should increase as a function of demand; when demand is low, interest rates

should be low, and vise versa when demand is high. The utilization ratio $\boldsymbol{U}$ for each market $\boldsymbol{a}$ unifies supply and demand into a single variable:

$$U_a = Borrows_a \,/\, (Cash_a + Borrows_a)$$

The demand curve is codified through governance and is expressed as a function of utilization. As an example, borrowing interest rates may resemble the following:

$$Borrowing\ Interest\ Rate_a \;=\; 2.5\% + U_a * 20\%$$

The interest rate earned by suppliers is *implicit*, and is equal to the borrowing interest rate, multiplied by the utilization rate.

2.3.1 Liquidity Incentive Structure

The protocol does not guarantee liquidity; instead, it relies on the interest rate model to incentivize it. In periods of extreme demand for an asset, the liquidity of the protocol (the tokens available to withdraw or borrow) will decline; when this occur, interest rates rise, incentivizing supply, and disincentivizing borrowing.

3 Implementation & Architecture

At its core, a Compound money market is a ledger that allows Ethereum accounts to supply or borrow assets, while computing interest, a function of time. The protocol's smart contracts will be publicly accessible and completely free to use for machines, dApps and humans.

3.1 cToken Contracts

Each money market is structured as a smart contract that implements the ERC-20 token specification. User's balances are represented as cToken balances; users can `mint(uint amountUnderlying)` cTokens by supplying assets to the market, or `redeem(uint amount)` cTokens for the underlying asset. The price (exchange rate) between cTokens and the underlying asset increases over time, as interest is accrued by borrowers of the asset, and is equal to:

$$exchangeRate \;=\; \frac{underlyingBalance + totalBorrowBalance_a - reserves_a}{cTokenSupply_a}$$

As the market's total borrowing balance increases (as a function of borrower interest accruing), the exchange rate between cTokens and the underlying asset increases.

Function ABI	Description
`mint(uint256 amountUnderlying)`	Transfers an underlying asset into the market, updates msg.sender's cToken balance.

`redeem(uint256 amount)` `redeemUnderlying(uint256 amountUnderlying)`	Transfers an underlying asset out of the market, updates msg.sender's cToken balance.
`borrow(uint amount)`	Checks msg.sender collateral value, and if sufficient, transfers the underlying asset out of the market to msg.sender, and updates msg.sender's borrow balance.
`repayBorrow(uint amount)` `repayBorrowBehalf(address account, uint amount)`	Transfers the underlying asset into the market, updates the borrower's borrow balance.
`liquidate(address borrower, address collateralAsset, uint closeAmount)`	Transfers the underlying asset into the market, updates the borrower's borrow balance, then transfers cToken collateral from the borrower to msg.sender

Table 2. ABI and summary of primary cToken smart contract functions

3.2 Interest Rate Mechanics

Compound money markets are defined by an interest rate, applied to all borrowers uniformly, which adjust over time as the relationship between supply and demand changes.

The history of each interest rate, for each money market, is captured by an *Interest Rate Index*, which is calculated each time an interest rate changes, resulting from a user minting, redeeming, borrowing, repaying or liquidating the asset.

3.2.1 Market Dynamics

Each time a transaction occurs, the Interest Rate Index for the asset is updated to compound the interest since the prior index, using the interest for the period, denominated by r * t, calculated using a per-block interest rate:

$$Index_{a,n} = Index_{a,(n-1)} * (1 + r * t)$$

The market's total borrowing outstanding is updated to include interest accrued since the last index:

$$totalBorrowBalance_{a,n} = totalBorrowBalance_{a,(n-1)} * (1 + r * t)$$

And a portion of the accrued interest is retained (set aside) as reserves, determined by a ***reserveFactor***, ranging from 0 to 1:

$$reserves_a = reserves_{a,(n-1)} + totalBorrowBalance_{a,(n-1)} * (r * t * reserveFactor)$$

3.2.2 Borrower Dynamics

A borrower's balance, including accrued interest, is simply the ratio of the current index divided by the index when the user's balance was last checkpointed.

The balance for each borrower address in the cToken is stored as an *account checkpoint*. An account checkpoint is a Solidity tuple *<uint256 balance, uint256 interestIndex>*. This tuple describes the balance at the time interest was last applied to that account.

3.3 Borrowing

A user who wishes to borrow and who has sufficient balances stored in Compound may call `borrow(uint amount)` on the relevant cToken contract. This function call checks the user's account value, and given sufficient collateral, will update the user's borrow balance, transfer the tokens to the user's Ethereum address, and update the money market's floating interest rate.

Borrows accrue interest in the exact same fashion as balance interest was calculated in section 3.2; a borrower has the right to repay an outstanding loan at any time, by calling `repayBorrow(uint amount)` which repays the outstanding balance.

3.4 Liquidation

If a user's borrowing balance exceeds their total collateral value (borrowing capacity) due to the value of collateral falling, or borrowed assets increasing in value, the public function `liquidate(address target, address collateralAsset, address borrowAsset, uint closeAmount)` can be called, which exchanges the invoking user's asset for the borrower's collateral, at a slightly better than market price.

3.5 Price Feeds

A *Price Oracle* maintains the current exchange rate of each supported asset; the Compound protocol delegates the ability to set the value of assets to a committee which pools prices from the top 10 exchanges. These exchange rates are used to determine borrowing capacity and collateral requirements, and for all functions which require calculating the value equivalent of an account.

3.6 Comptroller

The Compound protocol does not support specific tokens by default; instead, markets must be whitelisted. This is accomplished with an admin function, `supportMarket(address market, address interest rate model)` that allows users to begin interacting with the asset. In order to borrow an asset, there must be a valid price from the Price Oracle; in order to use an asset as collateral, there must be a valid price and a collateralFactor.

Each function call is validated through a policy layer, referred to as the *Comptroller*; this contract validates collateral and liquidity, before allowing a user action to proceed.

3.7 Governance

Compound will begin with centralized control of the protocol (such as choosing the interest rate model per asset), and over time, will transition to complete community and stakeholder control. The following rights in the protocol are controlled by the admin:

- The ability to list a new cToken market
- The ability to update the interest rate model per market
- The ability to update the oracle address
- The ability to withdraw the reserve of a cToken
- The ability to choose a new admin, such as a DAO controlled by the community; because this DAO can itself choose a new admin, the administration has the ability to evolve over time, based on the decisions of the stakeholders

4 Summary

- Compound creates properly functioning money markets for Ethereum assets
- Each money market has interest rates that are determined by the supply and demand of the underlying asset; when demand to borrow an asset grows, or when supply is removed, interest rates increase, incentivizing additional liquidity
- Users can supply tokens to a money market to earn interest, without trusting a central party
- Users can borrow a token (to use, sell, or re-lend) by using their balances in the protocol as collateral

---

References

[1] Cryptocurrency Market Capitalizations. https://coinmarketcap.com/
[2] Bitfixex Margin Funding Guide. https://support.bitfinex.com/
[3] ETHLend White Paper. https://github.com/ETHLend
[4] Ripio White Paper. https://ripiocredit.network/
[5] Lendroid White Paper. https://lendroid.com/
[6] dYdX White Paper. https://whitepaper.dydx.exchange/
[7] Fred Ehrsam: The Decentralized Business Model. https://blog.coinbase.com/

EXHIBIT C

compound-finance / **open-oracle** Public

The Compound Open Price Feed

MIT License

174 stars 79 forks

Star Notifications

Code Issues 1 Pull requests 4 Actions Projects Wiki Secur

master Go to file

hayesgm and **coburncoburn** Add Compound Config ... on Mar 24 107

View code

Open Oracle

The Open Oracle is a standard and SDK allowing reporters to sign key-value pairs (e.g. a price feed) that interested users can post to the blockchain. The system has a built-in view system that allows clients to easily share data and build aggregates (e.g. the median price from several sources).

Contracts

First, you will need solc 0.6.6 installed. Additionally for testing, you will need TypeScript installed and will need to build the open-oracle-reporter project by running `cd sdk/javascript && yarn .`

To fetch dependencies run:

```
yarn install
```

To compile everything run:

```
yarn run compile
```

To deploy contracts locally, you can run:

README.md

Note: you will need to be running an Ethereum node locally in order for this to work. E.g., start ganache-cli in another shell.

You can add a view in `MyView.sol` and run (default is `network=development`):

```
yarn run deploy MyView arg1 arg2 ...
```

To run tests:

```
yarn run test
```

To track deployed contracts in a saddle console:

```
yarn run console
```

Reporter SDK

This repository contains a set of SDKs for reporters to easily sign "reporter" data in any supported languages. We currently support the following languages:

- JavaScript (in TypeScript)
- Elixir

Poster

The poster is a simple application that reads from a given feed (or set of feeds) and posts...

Contributing

Note: all code contributed to this repository must be licensed under each of 1. MIT, 2. BSD-3, and 3. GPLv3. By contributing code to this repository, you accept that your code is allowed to be released under any or all of these licenses or licenses in substantially similar form to these listed above.

Please submit an issue (or create a pull request) for any issues or contributions to the project. Make sure that all test cases pass, including the integration tests in the root of this project.

Releases

2 tags

Packages

No packages published

Contributors 9

Languages

JavaScript 47.1% Solidity 29.4% TypeScript 21.6% Shell 1.4% Dockerfile 0.5%

EXHIBIT D

API

Compound API

Introduction

The Compound API input and output formats are specified by Protocol Buffers, known colloquially as protobufs. Unlike typical protobufs endpoints, the Compound endpoints support JSON for input and output in addition to the protobufs binary format. To use JSON in both the input and the output, specify the headers `"Content-Type: application/json"` and `"Accept: application/json"` in the request.

The Compound API no longer supports the Ethereum testnets.

It is a possibility that in the future, API keys will be required to access the API.

AccountService

The Account API retrieves information for various accounts which have interacted with Compound. You can use this API to pull data about a specific account by address, or alternatively, pull data for a list of unhealthy accounts (that is, accounts which are approaching under-collateralization).

```
// Retreives list of accounts and related supply and borrow balances.
fetch("https://api.compound.finance/api/v2/account");

// Returns details for given account
fetch("https://api.compound.finance/api/v2/account?addresses[]=0x00..");
```

GET: `/account`

AccountRequest

The request to the account API can specify a number filters, such as which addresses to retrieve

The request to the account API can specify a number filters, such as which addresses to retrieve information about or general health requirements. The following shows an example set of reques parameters in JSON:

```
{
  "addresses": [] // returns all accounts if empty or not included
  "block_number": 0 // returns latest if given 0
  "max_health": { "value": "10.0" }
  "min_borrow_value_in_eth": { "value": "0.002" }
  "page_number": 1
  "page_size": 10
}
```

Type	Key	Description
bytes	`addresses`	List of account addresses to filter on, e.g.: ["0x...", ,"0x..."] (Optional)
Precise	`min_borrow_value_in_eth`	Filter for accounts which total outstanding borrows exceeding given amount. (Optional)
Precise	`max_health`	Filter for accounts where outstanding borrows divided by collateral value is less than the provided amount. If returned value is less than 1.0, for instance, the account is subject to liquidation. If provided, should be given as `{ "value": "...string formatted number..." }` (Optional)
uint32	`block_number`	If provided, API returns data for given block number from our historical data. Otherwise, API defaults to returning the latest information. (Optional)
uint32	`block_timestamp`	If provided, API returns data for given timestamp from our historical data. Otherwise, API defaults to returning the latest information. (Optional)
uint32	`page_size`	Number of accounts to include in the response, default is 10 e.g. page_size=10 (Optional)
uint32	`page_number`	Pagination number for accounts in the response, default is 1 e.g. page_number=1 (Optional)

AccountResponse

AccountResponse

The account API returns an overall picture of accounts matching the filters on Compound.

Type	Key	Description	
Error	`error`	If set and non-zero, indicates an error returning data. `NO_ERROR = 0; INTERNAL_ERROR = 1; INVALID_PAGE_NUMBER = 2; INVALID_PAGE_SIZE = 3;`	
AccountRequest	`request`	The request parameters are echoed in the response.	
PaginationSummary	`pagination_summary`	For example	`{ "page_number": 1, "page_size": 100, "total_entries": 83, "total_pages": 1 }`
Account	`accounts`	The list of accounts (see Account below) matching the requested filter, with the associated account and cToken data.	

Account

This includes a list of cTokens contextualized to each account.

```
{
  "address": "0xbac065be2e8ca097e9ac924e94af000dd3a5663"
  "health": { "value": "1.072642756730503489907555994311947974318022395231132936826196057515919 01" }
  "tokens": [
```

```
    {
      "address": "0xf5dce57282a584d2746faf1593d3121fcac444dc"
      "borrow_balance_underlying": {"value": "131.4682716123015"}
      "lifetime_borrow_interest_accrued": {"value": "0.44430505829286"}
      "lifetime_supply_interest_accrued": {"value": "0.000002167182986489997 6"}
      "supply_balance_underlying": {"value": "0.0"}
    }
  ],
  "total_borrow_value_in_eth": {"value": "0.5100157047140227313856015174794473200000000000000000000000000000" }
  "total_collateral_value_in_eth": {"value": "0.5470646514802997866413544729358720112412173120000000000000000000
}
```


Type	Key	Description
bytes	`address`	The public Ethereum address of the account
Precise	`total_collateral_value_in_eth`	The value of all collateral supplied by the account. Calculated as cTokens held • exchange rate • collateral factor. Note: assets can be supplied and gain interest without being counted as collateral.
Precise	`total_borrow_value_in_eth`	The value of all outstanding borrows with accumulated interest.
Precise	`health`	`total_collateral_value_in_eth / total_borrow_value_in_eth`. If this value is less than 1.0, the account is subject to liquidation.
int32	`block_updated`	
AccountCToken	`tokens`	A list of tokens held by this account, see `AccountCToken` below for details.

AccountCToken

An account's supply, borrow, and interest information for a particular cToken.


```
{
  "address": "0xf5dce57282a584d2746faf1593d3121fcac444dc"
  "borrow_balance_underlying": {"value": "131.4682716123015"}
  "lifetime_borrow_interest_accrued": {"value": "0.44430505829286"}
  "lifetime_supply_interest_accrued": {"value": "0.000002167182986489997 6"}
  "supply_balance_underlying": {"value": "0.0"}
}
```

Type	Key	Description
bytes	`address`	The address of the cToken
string	`symbol`	The symbol of the cToken
Precise	`supply_balance_underlying`	The cToken balance converted to underlying `tokenscTokens held • exchange rate`
Precise	`borrow_balance_underlying`	The borrow balance (this is denominated in the underlying token, not in cTokens)
Precise	`lifetime_supply_interest_accrued`	The amount of supply interest accrued for the lifetime of this account-cToken pair.
Precise	`lifetime_borrow_interest_accrued`	The amount of borrow interest accrued for the lifetime of this account-cToken pair.
Precise	`safe_withdraw_amount_underlying`	The amount of supply that can be withdrawn such that the user's health remains at 1.25 or higher.

CTokenService

GET: `/ctoken`

CTokenRequest

The request to the cToken API can specify a number filters, such as which tokens to retrieve information about or moment in time. The following shows an example set of request parameters in JSON:

```
{
```

```
    "addresses": [] // returns all tokens if empty or not included
    "block_timestamp": 0 // returns latest information if given 0
}
```

Type	Key	Description
bytes	`addresses`	List of token addresses to filter on, e.g.: ["0x...", ,"0x..."] (Optional)
uint32	`block_number`	Only one of block_number or block timestamp should be provided. If provided, API returns data for given block number from our historical data. Otherwise, API defaults to returning the latest information. (Optional)
uint32	`block_timestamp`	Only one of block_number or block timestamp should be provided. If provided, API returns data for given block timestamp from our historical data. Otherwise, API defaults to returning the latest information. (Optional)
bool	`meta`	Pass true to get metadata for the token addresses specified. (Optional)

CTokenResponse

The cToken API returns an overall picture of cTokens matching the filter.

Type	Key	Description
Error	`error`	
CTokenRequest	`request`	The request parameters are echoed in the response.
CToken	`cToken`	The list of cToken (see `CToken` below) matching the requested filter.
CTokenMeta	`meta`	Metadata for all CTokens specified

CToken

This includes a list of cTokens contextualized to the full market.


```
{
 "cToken": [{
   "borrow_rate": {"value": "0.051453109785093843"},
   "cash": {"value": "514.078443"},
   "collateral_factor": {"value": "0.80000000000000000"},
   "exchange_rate": {"value": "0.020024242770802729"},
   "interest_rate_model_address": "0x1a43bfd39b15dcf444e17ab408c4b5be32deb7f5",
   "name": "Compound USD Coin",
   "number_of_borrowers": 3,
   "number_of_suppliers": 34,
   "reserves": {"value": "0"},
   "reserve_factor": {"value": "0.10000000000000000"},
   "supply_rate": {"value": "0.013237112532748109"},
   "symbol": "cUSDC",
   "token_address": "0x5b281a6dda0b271e91ae35de655ad301c976edb1",
   "total_borrows": {"value": "178.064546"},
   "total_supply": {"value": "34565.25157651"},
   "underlying_address": "0x4dbcdf9b62e891a7cec5a2568c3f4faf9e8abe2b",
   "underlying_name": "USD Coin",
   "underlying_price": {"value": "0.004136828705595353000000000"},
   "underlying_symbol":"USDC"
  }],
 "error": null,
 "request": {
   "addresses": ["0x5b281a6dda0b271e91ae35de655ad301c976edb1"],
   "block_number": 4515576,
   "block_timestamp": 0
  }
}
```


Type	Key	Description
bytes	`token_address`	The public Ethereum address of the cToken
Precise	`total_supply`	The number of cTokens in existence
Precise	`total_borrows`	The amount of underlying tokens borrowed from the cToken
Precise	`reserves`	The amount of underylying tokens held by reserves
Precise	`cash`	The current liquidity of the cToken
Precise	`exchange_rate`	The cToken / underlying exchange rate. This rate increases over time as supply interest accrues.
Precise	`supply_rate`	The floating supply interest rate
Precise	`borrow_rate`	The floating borrow interest rate

Type	Key	Description
Precise	`collateral_factor`	The amount of the value of the underlying token that will count as collateral. eg. cEth with collataral factor 0.75 means 1 eth of supply allows 0.75 eth of borrowing.
uint32	`number_of_suppliers`	The number of accounts holding this cToken
uint32	`number_of_borrowers`	The number of accounts with oustanding borrows
Precise	`underlying_price`	The price of the underlying token in eth
bytes	`underlying_address`	The address of the underlying token
string	`symbol`	The symbol of the ctoken
string	`name`	The name of the ctoken
string	`underlying_symbol`	The symbol of the underlying token
string	`underlying_name`	The name of the underlying token
bytes	`interest_rate_model_address`	The address of the interest rate model
Precise	`reserve_factor`	The amount of borrow interest that is converted into reserves
Precise	`comp_supply_apy`	The floating comp apy for supplying this token
Precise	`comp_borrow_apy`	The floating comp apy for borrowing this token
Precise	`borrow_cap`	The maximum size of total borrows for this market, beyond which no new borrows will be given

CTokenMeta

Type	Key	Description
uint32	`unique_suppliers`	Number of non-duplicate suppliers between all specified markets
uint32	`unique_borrowers`	Number of non-duplicate borrowers between all specified markets

MarketHistoryService

The market history service retrieves historical information about a market. You can use this API to find out the values of interest rates at a certain point in time. Its especially useful for making charts and graphs of the time-series values.

```
// Returns 10 buckets of market data
fetch("https://api.compound.finance/api/v2/market_history/graph?asset=0xf5dce57282a584d2746faf1593d3121fcac444d
```

GET: `/graph`

MarketHistoryGraphRequest

The market history graph API returns information about a market between two timestamps. The requestor can choose the asset and number of buckets to return within the range. For example:

```
{
 "asset": "0xf5dce57282a584d2746faf1593d3121fcac444dc",
 "min_block_timestamp": 1556747900,
 "max_block_timestamp": 1559339900,
 "num_buckets": 10
}
```

Type	Key	Description

Type	Key	Description
bytes	asset	The requested asset
uint32	min_block_timestamp	Unix epoch time in seconds
uint32	max_block_timestamp	Unix epoch time in seconds
uint32	num_buckets	How many buckets to group data points in

MarketHistoryGraphResponse

The market history graph API response contains the rates for both suppliers and borrowers, as well as the sequence of total supply and borrows for the given market.

Type	Key	Description
Error	error	If set and non-zero, indicates an error returning data. NO_ERROR = 0; INTERNAL_ERROR = 1; INVALID_REQUEST = 2
bytes	asset	The asset in question
Rate	supply_rates	The historical interest rates for suppliers
Rate	borrow_rates	The historical interest rates for borrowers
MarketTotal	total_supply_history	The historical total supply amounts for the market
MarketTotal	total_borrows_history	The historical total borrow amounts for the market
Rate	exchange_rates	The historical exchange rate
Price	prices_usd	The historical usd price of asset

MarketTotal

Type	Key	Description
uint32	`block_number`	The block number of the data point
uint32	`block_timestamp`	The timestamp of the block of the data point
Precise	`total`	The total value of the asset in asset-WEI terms.

Price

Type	Key	Description
uint32	`block_number`	The block number of the data point
uint32	`block_timestamp`	The timestamp of the block of the data point
Precise	`price`	The price of the underlying token in usd at that block number

Rate

Type	Key	Description
uint32	`block_number`	The block number of the data point
uint32	`block_timestamp`	The timestamp of the block of the data point
double	`rate`	The rate as a value between 0 and 1

GovernanceService

The Governance Service includes three endpoints to retrieve information about COMP accounts, governance proposals, and proposal vote receipts. You can use the APIs below to pull data about the Compound governance system:

```
// Retreives a list of governance proposals
fetch("https://api.compound.finance/api/v2/governance/proposals");

// Retreives a list of governance proposal vote receipts
fetch("https://api.compound.finance/api/v2/governance/proposal_vote_receipts");

// Retreives a list of COMP accounts
fetch("https://api.compound.finance/api/v2/governance/accounts");
```

GET: `/governance/proposals`

ProposalRequest

The request to the Proposal API can specify a number of filters, such as which ids to retrieve information about or state of proposals.

Type	Key	Description
uint32	`proposal_ids`	List of ids to filter on, e.g.: `?proposal_ids[]=23,25` (Optional)
string	`state`	The state of the proposal to filter on, (e.g.: "pending", "active", "canceled", "defeated", "succeeded", "queued", "expired", "executed") (Optional)
bool	`with_detail`	Set as true to include proposer and action data, default is false (Optional)
uint32	`page_size`	Number of proposals to include in the response, default is 10 e.g. `page_size=10` (Optional)
uint32	`page_number`	Pagination number for proposals in the response, default is 1 e.g. `page_number=1` (Optional)

ProposalResponse

The Proposal API returns a list of proposals that match the given filters on the request in

The Proposal API returns a list of proposals that match the given filters on the request in descending order by `proposal_id`.

Type	Key	Description
Error	`error`	If set and non-zero, indicates an error returning data. `NO_ERROR = 0; INTERNAL_ERROR = 1; INVALID_PAGE_NUMBER = 2; INVALID_PAGE_SIZE = 3; INVALID_PROPOSAL_ID = 4; INVALID_PROPOSAL_STATE = 5;`
ProposalRequest	`request`	The request parameters are echoed in the response.
PaginationSummary	`pagination_summary`	For example: `{ "page_number": 1, "page_size": 100, "total_entries": 83, "total_pages": 1 }`
Proposal	`proposals`	The list of proposals matching the requested filter

DisplayCompAccount

Type	Key	Description
bytes	`address`	The address of the given COMP account
string	`display_name`	A human readable name that describes who owns the account
string	`image_url`	A url to retrieve an account image
string	`account_url`	A url for the organization/user of the COMP account

Proposal

Type	Key	Description

Type	Key	Description
uint32	`id`	Unique ID for looking up a proposal
string	`title`	The title that describes the proposal
string	`description`	A description of the actions the proposal will take if successful
DisplayCompAccount	`proposer`	Either `null` or an object with data about the creator of the proposal (See DisplayCompAccount). Only populated when request submitted `with_detail=true`
ProposalAction	`actions`	Either `null` or an array of actions (See ProposalAction) that will be queued and executed if proposal succeeds. Only populated when request submitted `with_detail=true`
ProposalState	`states`	An array of states (See ProposalState) that represent the state transitions that the proposal has undergone
string	`for_votes`	The number of votes in support of the proposal
string	`against_votes`	The number of votes in opposition to this proposal

ProposalAction

Type	Key	Description
string	`title`	The title that describes the action
bytes	`target`	The address to send the calldata to

Type	Key	Description
string	value	The value of ETH to send with the transaction
string	signature	The function signature of the function to call at the target address
string	data	The encoded argument data for the action

ProposalState

Type	Key	Description
string	state	The state objects type, (e.g.: pending, active, canceled, defeated, succeeded, queued, expired, executed)
uint32	start_time	The start timestamp of state
uint32	end_time	Either `null` or the definitive end timestamp or an estimated end timestamp of the state
string	trx_hash	Either `null` or the transaction hash that represents the state transition

GET: `/governance/proposal_vote_receipts`

ProposalVoteReceiptRequest

The request to the Proposal Vote Receipt API can specify a number of filters, such as which id to retrieve information about or which account.

Type	Key	Description
uint32	proposal_id	A proposal id to filter on, e.g. `?proposal_id=23 (Optional)

Type		Description
bytes	`account`	Filter for proposals receipts for the given account (Optional)
bool	`support`	Filter for proposals receipts by for votes with `support=true` or against votes with `support=false`. If support not specified, response will return paginated votes for both for and against votes (Optional)
bool	`with_proposal_data`	Will populate a proposal object on the vote receipt when request submitted with `with_proposal_data=true`, default is false (Optional)
uint32	`page_size`	Number of proposal vote receipts to include in the response, default is 10 e.g. `page_size=10 (Optional)
uint32	`page_number`	Pagination number for proposal vote receipts in the response, default is 1 e.g. `page_number=1 (Optional)

ProposalVoteReceiptResponse

The Proposal Vote Receipt API returns a list of proposal vote receipts that match the given filters on the request

Type	Key	Description
Error	`error`	If set and non-zero, indicates an error returning data. `NO_ERROR = 0; INTERNAL_ERROR = 1; INVALID_PAGE_NUMBER = 2; INVALID_PAGE_SIZE = 3; INVALID_PROPOSAL_ID = 4; INVALID_ACCOUNT = 6;`
ProposalVoteReceiptRequest	`request`	The request parameters are echoed in the response.
		For example: `{ "page_number": 1,`

Type	Key	Description
PaginationSummary	`pagination_summary`	`"page_size": 100, "total_entries": 83, "total_pages": 1 }`
ProposalVoteReceipt	`proposal_vote_receipts`	The list of proposal vote receipts matching the requested filter

DisplayCompAccount

Type	Key	Description
bytes	`address`	The address of the given COMP account
string	`display_name`	A human readable name that describes who owns the account
string	`image_url`	A url to retrieve an account image
string	`account_url`	A url for the organization/user of the COMP account

Proposal

Type	Key	Description
uint32	`id`	Unique id for looking up a proposal
string	`title`	The title that describes the proposal
string	`description`	A description of the actions the proposal will take if successful
DisplayCompAccount	`proposer`	Either `null` or an object with data about the creator of the proposal (See DisplayCompAccount). Only populated when request submitted `with_detail=true`

Type	Key	Description
ProposalAction	`actions`	Either `null` or an array of actions (See ProposalAction) that will be queued and executed if proposal succeeds. Only populated when request submitted `with_detail=true`
ProposalState	`states`	An array of states (See ProposalState) that represent the state transitions that the proposal has undergone
string	`for_votes`	The number of votes in support of the proposal
string	`against_votes`	The number of votes in opposition to this proposal

ProposalVoteReceipt

Type	Key	Description
uint32	`proposal_id`	The proposal id the vote receipt corresponds to
Proposal	`proposal`	Either `null` or the object with proposal data (See Proposal). Only populated when request submitted `with_proposal_data=true`
DisplayCompAccount	`voter`	The object with voter data (See DisplayCompAccount)
bool	`support`	Whether or not the voter supports the proposal

Type		Description
string	votes	The number of votes cast by the voter

GET: `/governance/accounts`

GovernanceAccountRequest

The request to the Governance Account API can specify a number of filters, such as which accounts to retrieve information about.

Type	Key	Description
bytes	`addresses`	A list of accounts to filter on, e.g.: `?addresses=0x...` (Optional)
string	`order_by`	Filter for accounts by. E.g. "votes", "balance", or "proposals_created" (Optional)
bool	`with_history`	Will populate a list of transaction history for the accounts when request is submitted `with_history=true`
uint32	`page_size`	Number of accounts to include in the response, default is 10 e.g. `page_size=10` (Optional)
uint32	`page_number`	Pagination number for accounts in the response, default is 1 e.g. `page_number=1` (Optional)

GovernanceAccountResponse

The Governance Account API returns a list of accounts that match the given filters on the request

Type	Key	Description
Error	`error`	If set and non-zero, indicates an error returning data. `NO_ERROR = 0;` `INTERNAL_ERROR = 1;` `INVALID_PAGE_NUMBER = 2;`

Type	Key	Description

		`INVALID_PAGE_SIZE = 3;`
		`INVALID_ACCOUNT = 6;`
		`INVALID_FILTER_BY = 7;`

GovernanceAccountRequest	`request`	The request parameters are echoed in the response.
PaginationSummary	`pagination_summary`	For example: `{ "page_number": 1, "page_size": 100, "total_entries": 83, "total_pages": 1 }`
CompAccount	`accounts`	The list of governance accounts matching the requested filter

CompAccount

Type	Key	Description
bytes	`address`	The address of the given COMP account
string	`display_name`	A human readable name that describes who owns the account
string	`image_url`	A url to retrieve an account image
string	`account_url`	A url for the organization/user of the COMP account
string	`balance`	The balance of COMP for the given account
string	`votes`	The total votes delegated to the account
string	`vote_weight`	The percentage of voting weight of the 10,000,000 total COMP

Type	Key	Description
uint32	`proposals_created`	The number of proposals created in the Compound Governance System
DisplayCompAccount	`delegate`	The account this COMP account is delegating to (See DisplayCompAccount)
uint32	`rank`	Either `null` or the rank order of top 100 COMP accounts for votes
CompAccountTransaction	`transactions`	Either `null` or a list of historical transactions for the account (See CompAccountTransaction)
uint32	`proposals_voted`	The number of proposals voted on in the Compound Governance System
uint32	`total_delegates`	The number of addresses delegating to this account
CrowdProposal	`crowd_proposal`	Either `null` or a description of the crowd proposal the comp_moment represents (See CrowdProposal)

CompAccountTransaction

Type	Key	Description
string	title	A human readable title representing the transaction
uint32	timestamp	The timestamp the transaction occurred
string	trx_hash	The transaction hash of the transaction
string	delta	The change in value on the Comp Account

CrowdProposal

Type	Key	Description
bytes	`proposal_address`	The address of the given COMP Crowd Proposal
string	`title`	The title that describes the proposal
string	`description`	A description of the actions the proposal will take if successful
DisplayCompAccount	`author`	An object with data about the author of the proposal (See DisplayCompAccount).
ProposalAction	`actions`	An array of actions (See ProposalAction) that will be queued and executed if proposal succeeds.
uint32	`create_time`	The timestamp the crowd proposal was created
uint32	`propose_time`	The timestamp the crowd proposal was proposed
uint32	`terminate_time`	The timestamp the crowd proposal was terminated
string	`state`	The current state of the crowd proposal (e.g.: gathering_votes, proposed, terminated)

DisplayCompAccount

Type	Key	Description
bytes	`address`	The address of the given COMP account

Type	Key	Description
bytes	address	The address of the given COMP account
string	display_name	A human readable name that describes who owns the account
string	image_url	A url to retrieve an account image
string	account_url	A url for the organization/user of the COMP account

GET: `/governance/accounts/search`

GovernanceAccountSearchRequest

The request to the Governance Account API can specify a number of filters, such as which accounts to retrieve information about.

Type	Key	Description
string	term	A string to search accounts by. The search term can be part of an address or display name, e.g. "0x1234" or "The Purple Diamonds"
uint32	page_size	Number of accounts to include in the response, default is 10 e.g. `page_size=10` (Optional)
uint32	page_number	Pagination number for accounts in the response, default is 1 e.g. `page_number=1` (Optional)

GovernanceAccountSearchResponse

The Governance Account API returns a list of accounts that match the given filters on the request

Type	Key	Description

Type	Key	Description
Error	`error`	If set and non-zero, indicates an error returning data. `NO_ERROR = 0; INTERNAL_ERROR = 1;`
GovernanceAccountSearchRequest	`request`	The request parameters are echoed in the response.
PaginationSummary	`pagination_summary`	For example: `{ "page_number": 1, "page_size": 100, "total_entries": 83, "total_pages": 1 }`
CompAccount	`accounts`	The list of governance accounts matching the requested search term

CompAccount

Type	Key	Description
bytes	`address`	The address of the given COMP account
string	`display_name`	A human readable name that describes who owns the account
string	`image_url`	A url to retrieve an account image
string	`account_url`	A url for the organization/user of the COMP account

Type	Key	Description
string	`balance`	The balance of COMP for the given account
string	`votes`	The total votes delegated to the account
string	`vote_weight`	The percentage of voting weight of the 10,000,000 total COMP
uint32	`proposals_created`	The number of proposals created in the Compound Governance System
DisplayCompAccount	`delegate`	The account this COMP account is delegating to (See DisplayCompAccount)
uint32	`rank`	Either `null` or the rank order of top 100 COMP accounts for votes
CompAccountTransaction	`transactions`	Either `null` or a list of historical transactions for the account (See CompAccountTransaction)
uint32	`proposals_voted`	The number of proposals voted on in the Compound Governance System
uint32	`total_delegates`	The number of addresses delegating to this account
CrowdProposal	`crowd_proposal`	Either `null` or a description of the crowd proposal the comp_moment represents (See CrowdProposal)

CompAccountTransaction

Type	Key	Description

Type		Description
string	`title`	A human readable title representing the transaction
uint32	`timestamp`	The timestamp the transaction occurred
string	`trx_hash`	The transaction hash of the transaction
string	`delta`	The change in value on the Comp Account

CrowdProposal

Type	Key	Description
bytes	`proposal_address`	The address of the given COMP Crowd Proposal
string	`title`	The title that describes the proposal
string	`description`	A description of the actions the proposal will take if successful
DisplayCompAccount	`author`	An object with data about the author of the proposal (See DisplayCompAccount).
ProposalAction	`actions`	An array of actions (See ProposalAction) that will be queued and executed if proposal succeeds.
uint32	`create_time`	The timestamp the crowd proposal was created
uint32	`propose_time`	The timestamp the crowd proposal was proposed
uint32	`terminate_time`	The timestamp the crowd proposal was terminated

Type	Key	Description
		terminated
string	`state`	The current state of the crowd proposal (e.g.: gathering_votes, proposed, terminated)

DisplayCompAccount

Type	Key	Description
bytes	`address`	The address of the given COMP account
string	`display_name`	A human readable name that describes who owns the account
string	`image_url`	A url to retrieve an account image
string	`account_url`	A url for the organization/user of the COMP account

GET: `/governance/history`

GovernanceHistoryRequest

The governance history API returns historical information about the Compound governance system.

GovernanceHistoryResponse

The governance history API response contains the values for votes delegate, total delegators, total delegatees, and proposals created.

Type	Key	Description
Error	`error`	If set and non-zero, indicates an error returning data.

Type	Key	Description
		NO_ERROR = 0; INTERNAL_ERROR = 1; INVALID_REQUEST = 2;

string	votes_delegated	The number of votes delegated
uint32	token_holders	The number of addresses with a COMP balance greater than 0
uint32	voting_addresses	The number of addresses that have votes greater than 0
uint32	proposals_created	The number of proposals created
string	total_comp_allocated	The number of COMP allocated to all markets, including COMP not yet transferred to users

POST: `/governance/profile`

GovernanceProfileRequest

Type	Key	Description
string	action	Action to take on the profile, either "upsert" or "delete"
bytes	address	Address of the record to perform action on.
string	display_name	Name to display for the profile
string	image_url	Profile image for the profile
string	account_url	A link of the profile owner's choice
string	key	

GovernanceProfileResponse

Type	Key	Description
Error	`error`	
GovernanceProfileRequest	`request`	
DisplayCompAccount	`profile`	

DisplayCompAccount

Type	Key	Description
bytes	`address`	The address of the given COMP account
string	`display_name`	A human readable name that describes who owns the account
string	`image_url`	A url to retrieve an account image
string	`account_url`	A url for the organization/user of the COMP account

GET: `/governance/comp`

GovernanceCompDistributionRequest

The governance COMP distribution API returns COMP distribution information for markets in the Compound protocol.

Type	Key	Description
bytes	`addresses`	List of token addresses to filter on, e.g.: ["0x...", ,"0x..."]

GovernanceCompDistributionResponse

The governance COMP distribution API response contains the values for COMP allocated, COMP borrow index, COMP distributed, COMP speed, and COMP supply index for each market.

Type	Key	Description
GovernanceCompDistributionRequest	`request`	The request parameters are echoed in the response.
string	`comp_rate`	The number of COMP allocated to all markets per block
string	`daily_comp`	The number of COMP allocated to all markets per day assuming a given block time
string	`total_comp_allocated`	The number of COMP allocated to all markets, including COMP not yet transferred to users
string	`total_comp_distributed`	The number of total COMP actually transferred to users
MarketCompDistribution	`markets`	A list of all cToken markets receiving COMP

MarketCompDistribution

Type	Key	Description
bytes	`address`	The address of the cToken market

Type	Name	Description
string	`name`	The name of the cToken market
string	`symbol`	The symbol of the cToken market
bytes	`underlying_address`	The address of undelrying token of the cToken market
string	`underlying_name`	The name of the underlying token of the cToken market
string	`underlying_symbol`	The symbol of the underlying token of the cToken market
string	`supplier_daily_comp`	The projected daily comp distribution to suppliers of the market given the current distribution rate for the market
string	`borrower_daily_comp`	The projected daily comp distribution to borrowers of the market given the current distribution rate for the market
string	`comp_allocated`	The number of COMP allocated to the market, including COMP not yet transferred to borrowers/suppliers of the market
string	`comp_borrow_index`	The index used to calculate how much COMP an individual borrower should receive
string	`comp_distributed`	The number of COMP already transferred to the borrowers/suppliers of the market
string	`comp_speed`	The number of COMP allocated to the market each block
string	`comp_supply_index`	The index used to calculate how much COMP an individual supplier should receive

GET: `/governance/comp/account`

GovernanceAccountCompDistributionRequest

The governance COMP account distribution API returns COMP distribution information across al markets for a given account

Type	Key	Description
bytes	`address`	The account address to use for the request

GovernanceAccountCompDistributionResponse

The governance COMP account distribution API response contains the values for COMP allocated, COMP borrow index, COMP distributed, daily COMP, and COMP supply index for each market.

Type	Key	Description
GovernanceCompDistributionRequest	`request`	The request parameters are echoed in the response.
AccountCompDistribution	`markets`	A list of all cToken markets the account has been allocated COMP

AccountCompDistribution

Type	Key	Description
bytes	`address`	The address of the cToken market
string	`name`	The name of the cToken market
string	`symbol`	The symbol of the cToken market
bytes	`underlying_address`	The address of undelrying token of the cToken market

Type	Key	Description
string	`underlying_name`	The name of the underlying token of the cToken market
string	`underlying_symbol`	The symbol of the underlying token of the cToken market
string	`daily_comp`	The projected daily comp distribution to the account
string	`comp_allocated`	The number of COMP allocated to the account, including COMP not yet transferred to the account
string	`comp_borrow_index`	The index used to calculate how much COMP the account should receive based on it's borrows from the market
string	`comp_distributed`	The number of COMP already transferred to the the account
string	`comp_supply_index`	The index used to calculate how much COMP the account should receive based on it's supply to market

GovernanceCompDistributionRequest

The governance COMP distribution API returns COMP distribution information for markets in the Compound protocol.

Type	Key	Description
bytes	`addresses`	List of token addresses to filter on, e.g.: ["0x...", ,"0x..."] (Optional)

Shared Data Types

Custom data types that are shared between services.

Pagination Summary

Used for paginating results.

Type	Key	Description
uint32	`page_number`	The current page number
uint32	`page_size`	The number of entries to show per page.
uint32	`total_entries`	The number of items matching the request across all pages.
uint32	`total_pages`	The number of pages need to show total_entries at the given page_size.

Precise

For non-negative numbers only.

Type	Key	Description
string	`value`	The full UNSIGNED number in string form. max value is 2^2 231584178474632390847141970017375815706539969331

Docs

Community

Discord

Forums

Grants

EXHIBIT E

OPEN PRICE FEED

Open Price Feed

Introduction

The Open Price Feed accounts price data for the Compound protocol. The protocol's Comptroller contract uses it as a source of truth for prices. Prices are updated by Chainlink Price Feeds. The codebase is hosted on GitHub, and maintained by the community.

The Compound Protocol uses a View contract ("Price Feed") which verifies that reported prices fall within an acceptable bound of the time-weighted average price of the token/ETH pair on Uniswap v2, a sanity check referred to as the Anchor price.

The Chainlink price feeds submit prices for each cToken through an individual ValidatorProxy contract. Each ValidatorProxy is the only valid reporter for the underlying asset price. The contracts can be found on-chain as follows:

Contract name	address
AAVE ValidatorProxy	0x0238247E71AD0aB272203Af13bAEa72e99EE7c3c
BAT ValidatorProxy	0xeBa6F33730B9751a8BA0b18d9C256093E82f6bC2
COMP ValidatorProxy	0xE270B8E9d7a7d2A7eE35a45E43d17D56b3e272b1
DAI ValidatorProxy	0xb2419f587f497CDd64437f1B367E2e80889631ea
ETH ValidatorProxy	0x264BDDFD9D93D48d759FBDB0670bE1C6fDd50236
LINK ValidatorProxy	0xBcFd9b1a97cCD0a3942f0408350cdc281cDCa1B1
MKR ValidatorProxy	0xbA895504a8E286691E7dacFb47ae8A3A737e2Ce1
REP ValidatorProxy	0x9065531647938379541 6B615B61282C72D8382C1
SUSHI ValidatorProxy	0x875acA7030B75b5D8cB59c913910a7405337dFf7
UNI ValidatorProxy	0x70f4D236FD678c9DB41a52d28f90E299676d9D90

Contract name	address
WBTC ValidatorProxy	0x4846efc15CC725456597044e6267ad0b3B51353E
YFI ValidatorProxy	0xBa4319741782151D2B1df4799d757892EFda4165
ZRX ValidatorProxy	0x5c5db112c98dbe5977A4c37AD33F8a4c9ebd5575
UniswapAnchoredView	0x6d2299c48a8dd07a872fdd0f8233924872ad1071

Architecture

The Open Price Feed consists of two main contracts.

- `ValidatorProxy` is a contract that calls `validate` on the `UniswapAnchoredView`. This queries Uniswap v2 to check if a new price is within the Uniswap v2 TWAP anchor. If valid, the `UniswapAnchoredView` is updated with the asset's price. If invalid, the price data is not stored.
- `UniswapAnchoredView` only stores prices that are within an acceptable bound of the Uniswap time-weighted average price and are signed by a reporter. Also contains logic that upscales the posted prices into the format that Compound's Comptroller expects.

This architecture allows multiple views to use the same underlying price data, but to verify the prices in their own way.

Stablecoins like USDC, USDT, and TUSD are fixed at $1. SAI is fixed at 0.005285 ETH.

As a precaution, the Compound community multisig has the ability to engage a failover that will switch a market's primary oracle from the Chainlink Price Feeds to Uniswap v2. The multisig is able to change to a failover for single markets. The Uniswap V2 TWAP price is used as the failover. The community can enable or disable the failover using `activateFailover` or `deactivateFailover`.

Price

Get the most recent price for a token in USD with 6 decimals of precision.

- `symbol` : Symbol as a string

UniswapAnchoredView

```
function price(string memory symbol) external view returns (uint)
```

Solidity

```
UniswapAnchoredView view = UniswapAnchoredView(0xABCD...);
uint price = view.price("ETH");
```

Web3 1.0

```
const view = UniswapAnchoredView.at("0xABCD...");
//eg: returns 200e6
const price = await view.methods.price("ETH").call();
```

Underlying Price

Get the most recent price for a token in USD with 18 decimals of precision.

- `cToken`: The address of the cToken contract of the underlying asset.

UniswapAnchoredView

```
function getUnderlyingPrice(address cToken) external view returns (uint)
```

Solidity

```
UniswapAnchoredView view = UniswapAnchoredView(0xABCD...);
uint price = view.getUnderlyingPrice(0x1230...);
```

Web3 1.0

```
const view = UniswapAnchoredView.at("0xABCD...");
//eg: returns 400e6
const price = await view.methods.getUnderlyingPrice("0x1230...").call();
```

Config

Each token the Open Price Feed supports needs corresponding configuration metadata. The configuration for each token is set in the constructor and is immutable.

The fields of the config are:

- `cToken`: The address of the underlying token's corresponding cToken. This field is null for tokens that are not supported as cTokens.
- `underlying`: Address of the token whose price is being reported.
- `symbolHash`: The `keccak256` of the byte-encoded string of the token's symbol.
- `baseUnit`: The number of decimals of precision that the underlying token has. Eg: USDC has 6 decimals.
- `PriceSource`: An enum describing the whether or not to special case the prices for this token. `FIXED_ETH` is used to set the SAI price to a fixed amount of ETH, and `FIXED_USD` is used to peg stablecoin prices to $1. `REPORTER` is used for all other assets to indicate the reported prices and Uniswap anchoring should be used.
- `fixedPrice`: The fixed dollar amount to use if `PriceSource` is `FIXED_USD` or the number of ETH in the case of `FIXED_ETH` (namely for SAI).
- `uniswapMarket`: The token's market on Uniswap, used for price anchoring. Only filled if `PriceSource` is `REPORTER`.
- `isUniswapReversed`: A boolean indicating the order of the market's reserves.
- `reporter`: The address that submits prices for a particular cToken. This is always a `ValidatorProxy` contract that is always called by a price feed reference contract for each relevant price update posted by Chainlink oracle nodes.
- `reporterMultiplier`: An unsigned integer that is used to transform the price reported by the Chainlink price feeds to the correct base unit that the `UniswapAnchoredView` expects. This is required because the price feeds report prices with different decimal placement than the `UniswapAnchoredView`.

UniswapAnchoredView

```
function getTokenConfigBySymbol(string memory symbol) public view returns (TokenConfig memory)
function getTokenConfigBySymbolHash(bytes32 symbolHash) public view returns (TokenConfig memory)
function getTokenConfigByCToken(address cToken) public view returns (TokenConfig memory)
```

Web3 1.0

```
const view = UniswapAnchoredView.at("0xABCD...");
const config = await view.methods.getTokenConfigBySymbol("ETH").call();
```

Solidity

```
UniswapAnchoredView view = UniswapAnchoredView(0xABCD...);
uint price = view.getTokenConfigBySymbol("ETH");
```

Anchor Period

Get the anchor period, the minimum amount of time in seconds over which to take the time-weighted average price from Uniswap.

UniswapAnchoredView

```
function anchorPeriod() returns (uint)
```

Web3 1.0

```
const view = UniswapAnchoredView.at("0xABCD...");
const anchorPeriod = await view.methods.anchorPeriod().call();
```

Solidity

```
UniswapAnchoredView view = UniswapAnchoredView(0xABCD...);
uint anchorPeriod = view.anchorPeriod();
```

Anchor Bounds

Get the highest and lowest ratio of the reported price to the anchor price that will still trigger the price to be updated. Given in 18 decimals of precision (eg: 90% => 90e16).

UniswapAnchoredView

```
function upperBoundAnchorRatio() returns (uint)
function lowerBoundAnchorRatio() returns (uint)
```

Web3 1.0

```
const view = UniswapAnchoredView.at("0xABCD...");
const upperBoundRatio = await view.methods.upperBoundAnchorRatio().call();

```

Solidity

```
UniswapAnchoredView view = UniswapAnchoredView(0xABCD...);
uint upperBoundRatio = view.upperBoundAnchorRatio();
```

Protocol

Markets

Prices

Developers

Docs

Governance

Overview

COMP

Leaderboard

Community

Discord

Forums

Grants

EXHIBIT F

Compound FAQ

Robert Leshner Follow
Dec 5, 2018 · 4 min read

Compound is an algorithmic, autonomous interest rate protocol— allowing users & applications to frictionlessly earn interest or borrow Ethereum assets.

If you have questions or to learn more, join the Compound Discord.

Getting Started

How do I use Compound?

How to Earn Interest and Borrow Ethereum Assets
Earn interest on ETH, USDC, DAI, REP, WBTC, BAT, and ZRX
medium.com

Where else can I access the Compound protocol?

There are a number of community-built interfaces that you can use to access the Compound protocol & markets, including Zerion, InstaDapp, etc. You can find the full list on Compound Labs's website.

How does the Compound protocol work?

Compound is the first "liquidity pool" — instead of lending assets directly to another user, you supply liquidity to a market, and users borrow from that market.

In each market, interest rates are determined algorithmically (based on supply and demand), and interest accrues every Ethereum block.

There are no pre-defined durations or terms (such as “90 days”) — you can use the Compound protocol for as short as one block, or as long as you’d like; you’re free to withdraw or repay at any time.

I’ve heard of cTokens, what are those?

When you supply assets to the Compound protocol, your balance is represented as a cToken, which can be transferred, traded, or programmed by developers to create new experiences.

Think a cToken like a receipt — it’s used to show who owns a balance inside Compound. *Please be careful — if you transfer a cToken, your balance inside Compound will decrease*.

Interest Rates

How are interest rates set?

Interest rates are a function of the liquidity available in each market, and fluctuate in real-time based on supply and demand. When liquidity is plentiful, interest rates are low. As liquidity becomes scarce, interest rates increase, incentivizing new supply and the repayment of borrowing.

You aren’t locked into an interest rate — expect it to change. On each market page, you can view the interest rate model, and graphs of interest rates (for suppliers, and borrowers) over the past two months.

Why is the supply rate lower than the borrow rate?

In each market there is excess liquidity (assets supplied > assets borrowed), which allows you to quickly withdraw or borrow funds from the protocol.

The interest paid by borrowers is earned by the suppliers of the asset. Because there are more suppliers, the interest rate they earn is proportionately lower; this measured by an asset’s *Utilization Rate*.

Second, a portion of the interest paid by borrowers is set aside as *Reserves*, which acts as insurance and is controlled by COMP token-holders.

On each market page, you can view the accumulated Reserves and Reserve Factor (the portion of interest set aside).

How is interest calculated?

The interest rates you see in the Interface are quoted as *annual* interest rates. Interest accrues each Ethereum block; every ~15 seconds, your balance will increase by `(1/2102400)` of the quoted interest rate. Really!

Security

Is the Compound protocol safe? Has it been audited?

The security of the Compound protocol is our highest priority; our development team, alongside third-party auditors and consultants, has invested considerable effort to create a protocol that we believe is safe and dependable. All contract code and balances are publicly verifiable, and security researchers are eligible for a bug bounty for reporting undiscovered vulnerabilities.

Our security page contains details on each security audit, and the formal verification of the protocol.

How does the protocol's price feed work?

Compound uses the Open Price Feed, in which Reporters (like Coinbase) sign price data using a known public key, that Posters (any Ethereum address) can submit on-chain, to create a transparent, decentralized, resilient, and tamper-proof price feed.

How can I view my balance, without trusting the interface?

Sometimes, a balance appears as 0 (typically due to an issue with MetaMask or Infura). Relax — this is common.

To view your balance on the Ethereum blockchain, visit the Etherscan contract for the cToken, and scroll to `13. balanceOf`. Enter your address, click Query, and your cToken balance (with 8 decimals) will be shown.

To calculate your balance in the underlying asset, multiply your cToken balance by `4.exchangeRateStored`, and divide by 1e18.

Help! I can't access Compound!

The Compound protocol lives on the Ethereum blockchain, and is "always-on". In the event that MetaMask or the Compound Interface are malfunctioning, you can always access the Compound protocol manually.

Governance

Who controls the Compound protocol?

Compound is managed by a decentralized community of COMP token-holders and their delegates, who propose and vote on upgrades to the protocol.

How does Governance work?

Any address with 100 COMP can propose governance actions, which are executable code (like changes to the parameters of a market). When a proposal has gathered 100,000 COMP in support, voting begins, and lasts for 3 days. If a majority, and at least 400,000 votes are cast for the proposal, it is queued in a Timelock contract, and can be implemented after a 2 day waiting period.

Governance has complete control over the protocol, and all COMP tokens held by the protocol — the community manages the protocol as it sees fit.

How do I get involved in Compound Governance?

The community has created a Compound Forum to discuss governance proposals, and share ideas.

FAQ Compound Defi

EXHIBIT G

compound-finance / **open-oracle** Public

jflatow Address audit feedback, fix up poster, and other improvements ✓ History

3 contributors   

64 lines (40 sloc) | 2.01 KB

Open Oracle

The Open Oracle is a standard and SDK allowing reporters to sign key-value pairs (e.g. a price feed) that interested users can post to the blockchain. The system has a built-in view system that allows clients to easily share data and build aggregates (e.g. the median price from several sources).

Contracts

First, you will need solc 0.6.6 installed. Additionally for testing, you will need TypeScript installed and will need to build the open-oracle-reporter project by running `cd sdk/javascript && yarn .`

To fetch dependencies run:

```
yarn install
```

To compile everything run:

```
yarn run compile
```

To deploy contracts locally, you can run:

```
yarn run deploy --network development OpenOraclePriceData
```

Note: you will need to be running an Ethereum node locally in order for this to work. E.g., start ganache-cli in another shell.

You can add a view in `MyView.sol` and run (default is `network=development`):

```
yarn run deploy MyView arg1 arg2 ...
```

To run tests:

```
yarn run test
```

To track deployed contracts in a saddle console:

```
yarn run console
```

Reporter SDK

This repository contains a set of SDKs for reporters to easily sign "reporter" data in any supported languages. We currently support the following languages:

- JavaScript (in TypeScript)
- Elixir

Poster

The poster is a simple application that reads from a given feed (or set of feeds) and posts...

Contributing

Note: all code contributed to this repository must be licensed under each of 1. MIT, 2. BSD-3, and 3. GPLv3. By contributing code to this repository, you accept that your code is allowed to be released under any or all of these licenses or licenses in substantially similar form to these listed above.

Please submit an issue (or create a pull request) for any issues or contributions to the project. Make sure that all test cases pass, including the integration tests in the root of this project.

EXHIBIT H

CTOKENS

cTokens

Introduction

Each asset supported by the Compound Protocol is integrated through a cToken contract, which is an EIP-20 compliant representation of balances supplied to the protocol. By minting cTokens, users (1) earn interest through the cToken's exchange rate, which increases in value relative to the underlying asset, and (2) gain the ability to use cTokens as collateral.

cTokens are the primary means of interacting with the Compound Protocol; when a user mints, redeems, borrows, repays a borrow, liquidates a borrow, or transfers cTokens, she will do so using the cToken contract.

There are currently two types of cTokens: CErc20 and CEther. Though both types expose the EIP-20 interface, CErc20 wraps an underlying ERC-20 asset, while CEther simply wraps Ether itself. As such, the core functions which involve transferring an asset into the protocol have slightly different interfaces depending on the type, each of which is shown below.

How do cTokens earn interest?

Each market has its own Supply interest rate (APR). Interest isn't distributed; instead, simply by holding cTokens, you'll earn interest.

cTokens accumulates interest through their exchange rate — over time, each cToken becomes convertible into an increasing amount of it's underlying asset, even while the number of cTokens in your wallet stays the same.

Do I need to calculate the cToken exchange rate?

When a market is launched, the cToken exchange rate (how much ETH one cETH is worth) begins at 0.020000 — and increases at a rate equal to the compounding market interest rate. For example, after one year, the exchange rate might equal 0.021591.

Each user has the same cToken exchange rate; there's nothing unique to your wallet that you have to worry about.

Can you walk me through an example?

Let's say you supply 1,000 DAI to the Compound protocol, when the exchange rate is 0.020070; you would receive 49,825.61 cDAI (1,000/0.020070).

A few months later, you decide it's time to withdraw your DAI from the protocol; the exchange rate is now 0.021591:

- Your 49,825.61 cDAI is now equal to 1,075.78 DAI (49,825.61 * 0.021591)
- You could withdraw 1,075.78 DAI, which would redeem all 49,825.61 cDAI
- Or, you could withdraw a portion, such as your original 1,000 DAI, which would redeem 46,315.59 cDAI (keeping 3,510.01 cDAI in your wallet)

How do I view my cTokens?

Each cToken is visible on Etherscan, and you should be able to view them in the list of tokens associated with your address

cToken balances have been integrated into Coinbase Wallet and MetaMask; other wallets may add cToken support

Can I transfer cTokens?

Yes, but exercise caution! By transferring cTokens, you're transferring your balance of the underlying asset inside the Compound protocol. If you send a cToken to your friend, your balance (viewable in the Compound Interface) will decline, and your friend will see their balance increase.

A cToken transfer will fail if the account has entered that cToken market and the transfer would have put the account into a state of negative liquidity.

Mint

The mint function transfers an asset into the protocol, which begins accumulating interest based on the current Supply Rate for the asset. The user receives a quantity of cTokens equal to the underlying tokens supplied, divided by the current Exchange Rate.

CErc20

```
function mint(uint mintAmount) returns (uint)
```

- `msg.sender`: The account which shall supply the asset, and own the minted cTokens.
- `mintAmount`: The amount of the asset to be supplied, in units of the underlying asset.
- `RETURN`: 0 on success, otherwise an Error code

Before supplying an asset, users must first approve the cToken to access their token balance.

CEther

```
function mint() payable
```

- `msg.value` `payable`: The amount of ether to be supplied, in wei.
- `msg.sender`: The account which shall supply the ether, and own the minted cTokens.
- `RETURN`: No return, reverts on error.

Solidity

```
Erc20 underlying = Erc20(0xToken...);     // get a handle for the underlying asset contract
CErc20 cToken = CErc20(0x3FDA...);        // get a handle for the corresponding cToken contract
underlying.approve(address(cToken), 100); // approve the transfer
assert(cToken.mint(100) == 0);            // mint the cTokens and assert there is no error
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);
await cToken.methods.mint().send({from: myAccount, value: 50});
```

Redeem

The redeem function converts a specified quantity of cTokens into the underlying asset, and returns them to the user. The amount of underlying tokens received is equal to the quantity of cTokens redeemed, multiplied by the current Exchange Rate. The amount redeemed must be less than the user's Account Liquidity and the market's available liquidity.

CErc20 / CEther

```
function redeem(uint redeemTokens) returns (uint)
```

- `msg.sender`: The account to which redeemed funds shall be transferred.
- `redeemTokens`: The number of cTokens to be redeemed.
- `RETURN`: 0 on success, otherwise an Error code

Solidity

```
CEther cToken = CEther(0x3FDB...);
require(cToken.redeem(7) == 0, "something went wrong");
```

Web3 1.0

```
const cToken = CErc20.at(0x3FDA...);
cToken.methods.redeem(1).send({from: ...});
```

Redeem Underlying

The redeem underlying function converts cTokens into a specified quantity of the underlying asset, and returns them to the user. The amount of cTokens redeemed is equal to the quantity of underlying tokens received, divided by the current Exchange Rate. The amount redeemed must be less than the user's Account Liquidity and the market's available liquidity.

CErc20 / CEther

```
function redeemUnderlying(uint redeemAmount) returns (uint)
```

- `msg.sender`: The account to which redeemed funds shall be transferred.
- `redeemAmount`: The amount of underlying to be redeemed.
- `RETURN`: 0 on success, otherwise an Error code

Solidity

```
CEther cToken = CEther(0x3FDB...);
require(cToken.redeemUnderlying(50) == 0, "something went wrong");
```

Web3 1.0

```
const cToken = CErc20.at(0x3FDA...);
cToken.methods.redeemUnderlying(10).send({from: ...});
```

Borrow

The borrow function transfers an asset from the protocol to the user, and creates a borrow balance which begins accumulating interest based on the Borrow Rate for the asset. The amount borrowed must be less than the user's Account Liquidity and the market's available liquidity.

To borrow Ether, the borrower must be 'payable' (solidity).

CErc20 / CEther

```
function borrow(uint borrowAmount) returns (uint)
```

- `msg.sender`: The account to which borrowed funds shall be transferred.
- `borrowAmount` : The amount of the underlying asset to be borrowed.
- `RETURN`: 0 on success, otherwise an Error code

Solidity

```
CErc20 cToken = CErc20(0x3FDA...);
require(cToken.borrow(100) == 0, "got collateral?");
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);
await cToken.methods.borrow(50).send({from: 0xMyAccount});
```

Repay Borrow

The repay function transfers an asset into the protocol, reducing the user's borrow balance.

CErc20

```
function repayBorrow(uint repayAmount) returns (uint)
```

- `msg.sender`: The account which borrowed the asset, and shall repay the borrow.
- `repayAmount`: The amount of the underlying borrowed asset to be repaid. A value of -1 (i.e. 2^{256} - 1) can be used to repay the full amount.
- `RETURN`: 0 on success, otherwise an Error code

Before repaying an asset, users must first approve the cToken to access their token balance.

CEther

```
function repayBorrow() payable
```

- `msg.value` `payable`: The amount of ether to be repaid, in wei.
- `msg.sender`: The account which borrowed the asset, and shall repay the borrow.
- `RETURN`: No return, reverts on error.

Solidity

```
CEther cToken = CEther(0x3FDB...);
require(cToken.repayBorrow.value(100)() == 0, "transfer approved?");
```

Web3 1.0

```
const cToken = CErc20.at(0x3FDA...);
cToken.methods.repayBorrow(10000).send({from: ...});
```

Repay Borrow Behalf

The repay function transfers an asset into the protocol, reducing the target user's borrow balance.

CErc20

```
function repayBorrowBehalf(address borrower, uint repayAmount) returns (uint)
```

- `msg.sender`: The account which shall repay the borrow.
- `borrower`: The account which borrowed the asset to be repaid.
- `repayAmount`: The amount of the underlying borrowed asset to be repaid. A value of -1 (i.e. 2^{256} - 1) can be used to repay the full amount.
- `RETURN`: 0 on success, otherwise an Error code

Before repaying an asset, users must first approve the cToken to access their token balance.

CEther

```
function repayBorrowBehalf(address borrower) payable
```

- `msg.value` `payable`: The amount of ether to be repaid, in wei.
- `msg.sender`: The account which shall repay the borrow.
- `borrower`: The account which borrowed the asset to be repaid.
- `RETURN`: No return, reverts on error.

Solidity

```
CEther cToken = CEther(0x3FDB...);
require(cToken.repayBorrowBehalf.value(100)(0xBorrower) == 0, "transfer approved?");
```

Web3 1.0

```
const cToken = CErc20.at(0x3FDA...);
await cToken.methods.repayBorrowBehalf(0xBorrower, 10000).send({from: 0xPayer});
```

Transfer

Transfer is an ERC-20 method that allows accounts to send tokens to other Ethereum addresses. A cToken transfer will fail if the account has entered that cToken market and the transfer would have put the account into a state of negative liquidity.

CErc20 / CEther

```
function transfer(address recipient, uint256 amount) returns (bool)
```

- `recipient`: The transfer recipient address.
- `amount`: The amount of cTokens to transfer.
- `RETURN`: Returns a boolean value indicating whether or not the operation succeeded.

Solidity

```
CEther cToken = CEther(0x3FDB...);
cToken.transfer(0xABCD..., 100000000000);
```

Web3 1.0

```
const cToken = CErc20.at(0x3FDA...);
await cToken.methods.transfer(0xABCD..., 100000000000).send({from: 0xSender});
```

Liquidate Borrow

A user who has negative account liquidity is subject to liquidation by other users of the protocol to return his/her account liquidity back to positive (i.e. above the collateral requirement). When a liquidation occurs, a liquidator may repay some or all of an outstanding borrow on behalf of a borrower and in return receive a discounted amount of collateral held by the borrower; this discount is defined as the liquidation incentive.

A liquidator may close up to a certain fixed percentage (i.e. close factor) of any individual outstanding borrow of the underwater account. Unlike in v1, liquidators must interact with each cToken contract in which they wish to repay a borrow and seize another asset as collateral. When collateral is seized, the liquidator is transferred cTokens, which they may redeem the same as if they had supplied the asset themselves. Users must approve each cToken contract before calling liquidate (i.e. on the borrowed asset which they are repaying), as they are transferring funds into the contract.

CErc20

```
function liquidateBorrow(address borrower, uint amount, address collateral) returns (uint)
```

- `msg.sender`: The account which shall liquidate the borrower by repaying their debt and seizing their collateral.
- `borrower`: The account with negative account liquidity that shall be liquidated.
- `repayAmount`: The amount of the borrowed asset to be repaid and converted into collateral, specified in units of the underlying borrowed asset.
- `cTokenCollateral`: The address of the cToken currently held as collateral by a borrower, that the liquidator shall seize.
- `RETURN`: 0 on success, otherwise an Error code

Before supplying an asset, users must first approve the cToken to access their token balance.

CEther

```
function liquidateBorrow(address borrower, address cTokenCollateral) payable
```

- `msg.value` `payable`: The amount of ether to be repaid and converted into collateral, in wei.
- `msg.sender`: The account which shall liquidate the borrower by repaying their debt and seizing their collateral.
- `borrower`: The account with negative account liquidity that shall be liquidated.
- `cTokenCollateral`: The address of the cToken currently held as collateral by a borrower, that the liquidator shall seize.
- `RETURN`: No return, reverts on error.

Solidity

```
CEther cToken = CEther(0x3FDB...);

CErc20 cTokenCollateral = CErc20(0x3FDA...);
require(cToken.liquidateBorrow.value(100)(0xBorrower, cTokenCollateral) == 0, "borrower underwater??");
```

Web3 1.0

```
const cToken = CErc20.at(0x3FDA...);
const cTokenCollateral = CEther.at(0x3FDB...);
await cToken.methods.liquidateBorrow(0xBorrower, 33, cTokenCollateral).send({from: 0xLiquidator});
```

Key Events

Event	Description
`Mint(address minter, uint mintAmount, uint mintTokens)`	Emitted upon a successful Mint.
`Redeem(address redeemer, uint redeemAmount, uint redeemTokens)`	Emitted upon a successful Redeem.
`Borrow(address borrower, uint borrowAmount, uint accountBorrows, uint totalBorrows)`	Emitted upon a successful Borrow.
`RepayBorrow(address payer, address borrower, uint repayAmount, uint accountBorrows, uint totalBorrows)`	Emitted upon a successful Repay Borrow.

Event	Description
`LiquidateBorrow(address liquidator, address borrower, uint repayAmount, address cTokenCollateral, uint seizeTokens)`	Emitted upon a successful Liquidate Borrow.

Error Codes

Code	Name	Description
0	`NO_ERROR`	Not a failure.
1	`UNAUTHORIZED`	The sender is not authorized to perform this action.
2	`BAD_INPUT`	An invalid argument was supplied by the caller.
3	`COMPTROLLER_REJECTION`	The action would violate the comptroller policy.
4	`COMPTROLLER_CALCULATION_ERROR`	An internal calculation has failed in the comptroller.
5	`INTEREST_RATE_MODEL_ERROR`	The interest rate model returned an invalid value.
6	`INVALID_ACCOUNT_PAIR`	The specified combination of accounts is invalid.
7	`INVALID_CLOSE_AMOUNT_REQUESTED`	The amount to liquidate is invalid.
8	`INVALID_COLLATERAL_FACTOR`	The collateral factor is invalid.
9	`MATH_ERROR`	A math calculation error occurred.
10	`MARKET_NOT_FRESH`	Interest has not been properly accrued.
11	`MARKET_NOT_LISTED`	The market is not currently listed by its comptroller.

Code	Name	Description
11	MARKET_NOT_LISTED	The market is not currently listed by its comptroller.
12	TOKEN_INSUFFICIENT_ALLOWANCE	ERC-20 contract must *allow* Money Market contract to call `transferFrom`. The current allowance is either 0 or less than the requested supply, repayBorrow or liquidate amount.
13	TOKEN_INSUFFICIENT_BALANCE	Caller does not have sufficient balance in the ERC-20 contract to complete the desired action.
14	TOKEN_INSUFFICIENT_CASH	The market does not have a sufficient cash balance to complete the transaction. You may attempt this transaction again later.
15	TOKEN_TRANSFER_IN_FAILED	Failure in ERC-20 when transfering token into the market.
16	TOKEN_TRANSFER_OUT_FAILED	Failure in ERC-20 when transfering token out of the market.

Failure Info

Code	Name
0	ACCEPT_ADMIN_PENDING_ADMIN_CHECK
1	ACCRUE_INTEREST_ACCUMULATED_INTEREST_CALCULATION_FAILED
2	ACCRUE_INTEREST_BORROW_RATE_CALCULATION_FAILED
3	ACCRUE_INTEREST_NEW_BORROW_INDEX_CALCULATION_FAILED
4	ACCRUE_INTEREST_NEW_TOTAL_BORROWS_CALCULATION_FAILED
5	ACCRUE_INTEREST_NEW_TOTAL_RESERVES_CALCULATION_FAILED
6	ACCRUE_INTEREST_SIMPLE_INTEREST_FACTOR_CALCULATION_FAILED
7	BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED
8	BORROW_ACCRUE_INTEREST_FAILED

Code	Name
8	BORROW_ACCRUE_INTEREST_FAILED
9	BORROW_CASH_NOT_AVAILABLE
10	BORROW_FRESHNESS_CHECK
11	BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED
12	BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED
13	BORROW_MARKET_NOT_LISTED
14	BORROW_COMPTROLLER_REJECTION
15	LIQUIDATE_ACCRUE_BORROW_INTEREST_FAILED
16	LIQUIDATE_ACCRUE_COLLATERAL_INTEREST_FAILED
17	LIQUIDATE_COLLATERAL_FRESHNESS_CHECK
18	LIQUIDATE_COMPTROLLER_REJECTION
19	LIQUIDATE_COMPTROLLER_CALCULATE_AMOUNT_SEIZE_FAILED
20	LIQUIDATE_CLOSE_AMOUNT_IS_UINT_MAX
21	LIQUIDATE_CLOSE_AMOUNT_IS_ZERO
22	LIQUIDATE_FRESHNESS_CHECK
23	LIQUIDATE_LIQUIDATOR_IS_BORROWER
24	LIQUIDATE_REPAY_BORROW_FRESH_FAILED
25	LIQUIDATE_SEIZE_BALANCE_INCREMENT_FAILED
26	LIQUIDATE_SEIZE_BALANCE_DECREMENT_FAILED
27	LIQUIDATE_SEIZE_COMPTROLLER_REJECTION

Code	
28	LIQUIDATE_SEIZE_LIQUIDATOR_IS_BORROWER
29	LIQUIDATE_SEIZE_TOO_MUCH
30	MINT_ACCRUE_INTEREST_FAILED
31	MINT_COMPTROLLER_REJECTION
32	MINT_EXCHANGE_CALCULATION_FAILED
33	MINT_EXCHANGE_RATE_READ_FAILED
34	MINT_FRESHNESS_CHECK
35	MINT_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED
36	MINT_NEW_TOTAL_SUPPLY_CALCULATION_FAILED
37	MINT_TRANSFER_IN_FAILED
38	MINT_TRANSFER_IN_NOT_POSSIBLE
39	REDEEM_ACCRUE_INTEREST_FAILED
40	REDEEM_COMPTROLLER_REJECTION
41	REDEEM_EXCHANGE_TOKENS_CALCULATION_FAILED
42	REDEEM_EXCHANGE_AMOUNT_CALCULATION_FAILED
43	REDEEM_EXCHANGE_RATE_READ_FAILED
44	REDEEM_FRESHNESS_CHECK
45	REDEEM_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED
46	REDEEM_NEW_TOTAL_SUPPLY_CALCULATION_FAILED

Code	
47	REDEEM_TRANSFER_OUT_NOT_POSSIBLE
48	REDUCE_RESERVES_ACCRUE_INTEREST_FAILED
49	REDUCE_RESERVES_ADMIN_CHECK
50	REDUCE_RESERVES_CASH_NOT_AVAILABLE
51	REDUCE_RESERVES_FRESH_CHECK
52	REDUCE_RESERVES_VALIDATION
53	REPAY_BEHALF_ACCRUE_INTEREST_FAILED
54	REPAY_BORROW_ACCRUE_INTEREST_FAILED
55	REPAY_BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED
56	REPAY_BORROW_COMPTROLLER_REJECTION
57	REPAY_BORROW_FRESHNESS_CHECK
58	REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED
59	REPAY_BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED
60	REPAY_BORROW_TRANSFER_IN_NOT_POSSIBLE
61	SET_COLLATERAL_FACTOR_OWNER_CHECK
62	SET_COLLATERAL_FACTOR_VALIDATION
63	SET_COMPTROLLER_OWNER_CHECK
64	SET_INTEREST_RATE_MODEL_ACCRUE_INTEREST_FAILED
65	SET_INTEREST_RATE_MODEL_FRESH_CHECK

```
66  SET_INTEREST_RATE_MODEL_OWNER_CHECK
67  SET_MAX_ASSETS_OWNER_CHECK
68  SET_ORACLE_MARKET_NOT_LISTED
69  SET_PENDING_ADMIN_OWNER_CHECK
70  SET_RESERVE_FACTOR_ACCRUE_INTEREST_FAILED
71  SET_RESERVE_FACTOR_ADMIN_CHECK
72  SET_RESERVE_FACTOR_FRESH_CHECK
73  SET_RESERVE_FACTOR_BOUNDS_CHECK
74  TRANSFER_COMPTROLLER_REJECTION
75  TRANSFER_NOT_ALLOWED
76  TRANSFER_NOT_ENOUGH
77  TRANSFER_TOO_MUCH
```

Exchange Rate

Each cToken is convertible into an ever increasing quantity of the underlying asset, as interest accrues in the market. The exchange rate between a cToken and the underlying asset is equal to:

```
exchangeRate = (getCash() + totalBorrows() - totalReserves()) / totalSupply()
```

CErc20 / CEther

```
function exchangeRateCurrent() returns (uint)
```

- `RETURN`: The current exchange rate as an unsigned integer, scaled by 1 * 10^(18 - 8 + Underlying Token Decimals).

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint exchangeRateMantissa = cToken.exchangeRateCurrent();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);
const exchangeRate = (await cToken.methods.exchangeRateCurrent().call()) / 1e18;
```

Tip: note the use of `call` vs. `send` to invoke the function from off-chain without incurring gas costs.

Get Cash

Cash is the amount of underlying balance owned by this cToken contract. One may query the total amount of cash currently available to this market.

CErc20 / CEther

```
function getCash() returns (uint)
```

- `RETURN`: The quantity of underlying asset owned by the contract.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint cash = cToken.getCash();
```

Web3 1.0

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);
const cash = (await cToken.methods.getCash().call());
```

Total Borrow

Total Borrows is the amount of underlying currently loaned out by the market, and the amount upon which interest is accumulated to suppliers of the market.

CErc20 / CEther

```
function totalBorrowsCurrent() returns (uint)
```

- `RETURN`: The total amount of borrowed underlying, with interest.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint borrows = cToken.totalBorrowsCurrent();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);
const borrows = (await cToken.methods.totalBorrowsCurrent().call());
```

Borrow Balance

A user who borrows assets from the protocol is subject to accumulated interest based on the current borrow rate. Interest is accumulated every block and integrations may use this function to obtain the current value of a user's borrow balance with interest.

CErc20 / CEther

```
function borrowBalanceCurrent(address account) returns (uint)
```

- `account`: The account which borrowed the assets.
- `RETURN`: The user's current borrow balance (with interest) in units of the underlying asset.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint borrows = cToken.borrowBalanceCurrent(msg.caller);
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);
const borrows = await cToken.methods.borrowBalanceCurrent(account).call();
```

Borrow Rate

At any point in time one may query the contract to get the current borrow rate per block.

CErc20 / CEther

```
function borrowRatePerBlock() returns (uint)
```

- `RETURN`: The current borrow rate as an unsigned integer, scaled by 1e18.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint borrowRateMantissa = cToken.borrowRatePerBlock();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);
const borrowRate = (await cToken.methods.borrowRatePerBlock().call()) / 1e18;
```

Total Supply

Total Supply is the number of tokens currently in circulation in this cToken market. It is part of the EIP-20 interface of the cToken contract.

CErc20 / CEther

```
function totalSupply() returns (uint)
```

- `RETURN`: The total number of tokens in circulation for the market.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint tokens = cToken.totalSupply();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);
const tokens = (await cToken.methods.totalSupply().call());
```

Underlying Balance

The user's underlying balance, representing their assets in the protocol, is equal to the user's cToken balance multiplied by the Exchange Rate.

CErc20 / CEther

```
function balanceOfUnderlying(address account) returns (uint)
```

- `account` : The account to get the underlying balance of.
- `RETURN` : The amount of underlying currently owned by the account.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint tokens = cToken.balanceOfUnderlying(msg.caller);
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);
const tokens = await cToken.methods.balanceOfUnderlying(account).call();
```

Supply Rate

At any point in time one may query the contract to get the current supply rate per block. The supply rate is derived from the borrow rate, reserve factor and the amount of total borrows.

CErc20 / CEther

```
function supplyRatePerBlock() returns (uint)
```

- `RETURN` : The current supply rate as an unsigned integer, scaled by 1e18.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint supplyRateMantissa = cToken.supplyRatePerBlock();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);
const supplyRate = (await cToken.methods.supplyRatePerBlock().call()) / 1e18;
```

Total Reserves

Reserves are an accounting entry in each cToken contract that represents a portion of historical interest set aside as cash which can be withdrawn or transferred through the protocol's governance. A small portion of borrower interest accrues into the protocol, determined by the reserve factor.

CErc20 / CEther

```
function totalReserves() returns (uint)
```

- `RETURN`: The total amount of reserves held in the market.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint reserves = cToken.totalReserves();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);
const reserves = (await cToken.methods.totalReserves().call());
```

Reserve Factor

The reserve factor defines the portion of borrower interest that is converted into reserves.

CErc20 / CEther

```
function reserveFactorMantissa() returns (uint)
```

- `RETURN`: The current reserve factor as an unsigned integer, scaled by 1e18.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint reserveFactorMantissa = cToken.reserveFactorMantissa();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);
const reserveFactor = (await cToken.methods.reserveFactorMantissa().call()) / 1e18;
```

Forums

Grants

EXHIBIT I

COMPTROLLER

Comptroller

Introduction

The Comptroller is the risk management layer of the Compound protocol; it determines how much collateral a user is required to maintain, and whether (and by how much) a user can be liquidated. Each time a user interacts with a cToken, the Comptroller is asked to approve or deny the transaction.

The Comptroller maps user balances to prices (via the Price Oracle) to risk weights (called Collateral Factors) to make its determinations. Users explicitly list which assets they would like included in their risk scoring, by calling Enter Markets and Exit Market.

Architecture

The Comptroller is implemented as an upgradeable proxy. The Unitroller proxies all logic to the Comptroller implementation, but storage values are set on the Unitroller. To call Comptroller functions, use the Comptroller ABI on the Unitroller address.

Enter Markets

Enter into a list of markets - it is not an error to enter the same market more than once. In order to supply collateral or borrow in a market, it must be entered first.

Comptroller

```
function enterMarkets(address[] calldata cTokens) returns (uint[] memory)
```

- `msg.sender`: The account which shall enter the given markets.
- `cTokens`: The addresses of the cToken markets to enter.
- `RETURN`: For each market, returns an error code indicating whether or not it was entered. Each is 0 on success, otherwise an Error code.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);
CToken[] memory cTokens = new CToken[](2);
cTokens[0] = CErc20(0x3FDA...);
cTokens[1] = CEther(0x3FDB...);
uint[] memory errors = troll.enterMarkets(cTokens);
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);
const cTokens = [CErc20.at(0x3FDA...), CEther.at(0x3FDB...)];
const errors = await troll.methods.enterMarkets(cTokens).send({from: ...});
```

Exit Market

Exit a market - it is not an error to exit a market which is not currently entered. Exited markets will not count towards account liquidity calculations.

Comptroller

```
function exitMarket(address cToken) returns (uint)
```

- `msg.sender`: The account which shall exit the given market.
- `cTokens`: The addresses of the cToken market to exit.
- `RETURN`: 0 on success, otherwise an Error code.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);
uint error = troll.exitMarket(CToken(0x3FDA...));
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);
const errors = await troll.methods.exitMarket(CEther.at(0x3FDB...)).send({from: ...});
```

Get Assets In

Get the list of markets an account is currently entered into. In order to supply collateral or borrow in a market, it must be entered first. Entered markets count towards account liquidity calculations.

Comptroller

```
function getAssetsIn(address account) view returns (address[] memory)
```

- `account`: The account whose list of entered markets shall be queried.
- `RETURN`: The address of each market which is currently entered into.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);
address[] memory markets = troll.getAssetsIn(0xMyAccount);
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);
const markets = await troll.methods.getAssetsIn(cTokens).call();
```

Collateral Factor

A cToken's collateral factor can range from 0-90%, and represents the proportionate increase in liquidity (borrow limit) that an account receives by minting the cToken.

Generally, large or liquid assets have high collateral factors, while small or illiquid assets have low collateral factors. If an asset has a 0% collateral factor, it can't be used as collateral (or seized in

liquidation), though it can still be borrowed.

> Collateral factors can be increased (or decreased) through Compound Governance, as market conditions change.

Comptroller

```
function markets(address cTokenAddress) view returns (bool, uint, bool)
```

- `cTokenAddress`: The address of the cToken to check if listed and get the collateral factor for.
- `RETURN`: Tuple of values (isListed, collateralFactorMantissa, isComped); isListed represents whether the comptroller recognizes this cToken; collateralFactorMantissa, scaled by 1e18, is multiplied by a supply balance to determine how much value can be borrowed. The isComped boolean indicates whether or not suppliers and borrowers are distributed COMP tokens.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);
(bool isListed, uint collateralFactorMantissa, bool isComped) = troll.markets(0x3FDA...);
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);
const result = await troll.methods.markets(0x3FDA...).call();
const {0: isListed, 1: collateralFactorMantissa, 2: isComped} = result;
```

Get Account Liquidity

Account Liquidity represents the USD value borrowable by a user, before it reaches liquidation. Users with a shortfall (negative liquidity) are subject to liquidation, and can't withdraw or borrow assets until Account Liquidity is positive again.

For each market the user has entered into, their supplied balance is multiplied by the market's collateral factor, and summed; borrow balances are then subtracted, to equal Account Liquidity. Borrowing an asset reduces Account Liquidity for each USD borrowed; withdrawing an asset reduces Account Liquidity by the asset's collateral factor times each USD withdrawn.

Because the Compound Protocol exclusively uses unsigned integers, Account Liquidity returns either a surplus or shortfall.

Comptroller

```
function getAccountLiquidity(address account) view returns (uint, uint, uint)
```

- `account`: The account whose liquidity shall be calculated.
- `RETURN`: Tuple of values (error, liquidity, shortfall). The error shall be 0 on success, otherwise an error code. A non-zero liquidity value indicates the account has available account liquidity. A non-zero shortfall value indicates the account is currently below his/her collateral requirement and is subject to liquidation. At most one of liquidity or shortfall shall be non-zero.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);
(uint error, uint liquidity, uint shortfall) = troll.getAccountLiquidity(msg.caller);
require(error == 0, "join the Discord");
require(shortfall == 0, "account underwater");
require(liquidity > 0, "account has excess collateral");
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);
const result = await troll.methods.getAccountLiquidity(0xBorrower).call();
const {0: error, 1: liquidity, 2: shortfall} = result;
```

Close Factor

The percent, ranging from 0% to 100%, of a liquidatable account's borrow that can be repaid in a single liquidate transaction. If a user has multiple borrowed assets, the closeFactor applies to any single borrowed asset, not the aggregated value of a user's outstanding borrowing.

Comptroller

```
function closeFactorMantissa() view returns (uint)
```

- `RETURN`: The closeFactor, scaled by 1e18, is multiplied by an outstanding borrow balance to determine how much could be closed.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);
uint closeFactor = troll.closeFactorMantissa();
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);
const closeFactor = await troll.methods.closeFactorMantissa().call();
```

Liquidation Incentive

The additional collateral given to liquidators as an incentive to perform liquidation of underwater accounts. For example, if the liquidation incentive is 1.1, liquidators receive an extra 10% of the borrowers collateral for every unit they close.

Comptroller

```
function liquidationIncentiveMantissa() view returns (uint)
```

- `RETURN`: The liquidationIncentive, scaled by 1e18, is multiplied by the closed borrow amount from the liquidator to determine how much collateral can be seized.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);
uint closeFactor = troll.liquidationIncentiveMantissa();
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);
const closeFactor = await troll.methods.liquidationIncentiveMantissa().call();
```

Key Events

Event	Description
`MarketEntered(CToken cToken, address account)`	Emitted upon a successful Enter Market.
`MarketExited(CToken cToken, address account)`	Emitted upon a successful Exit Market.

Error Codes

Code	Name	Description
0	`NO_ERROR`	Not a failure.
1	`UNAUTHORIZED`	The sender is not authorized to perform this action.
2	`COMPTROLLER_MISMATCH`	Liquidation cannot be performed in markets with different comptrollers.
3	`INSUFFICIENT_SHORTFALL`	The account does not have sufficient shortfall to perform this action.
4	`INSUFFICIENT_LIQUIDITY`	The account does not have sufficient liquidity to perform this action.
5	`INVALID_CLOSE_FACTOR`	The close factor is not valid.
6	`INVALID_COLLATERAL_FACTOR`	The collateral factor is not valid.

Code	Name	Description
7	INVALID_LIQUIDATION_INCENTIVE	The liquidation incentive is invalid.
8	MARKET_NOT_ENTERED	The market has not been entered by the account.
9	MARKET_NOT_LISTED	The market is not currently listed by the comptroller.
10	MARKET_ALREADY_LISTED	An admin tried to list the same market more than once.
11	MATH_ERROR	A math calculation error occurred.
12	NONZERO_BORROW_BALANCE	The action cannot be performed since the account carries a borrow balance.
13	PRICE_ERROR	The comptroller could not obtain a required price of an asset.
14	REJECTION	The comptroller rejects the action requested by the market.
15	SNAPSHOT_ERROR	The comptroller could not get the account borrows and exchange rate from the market.
16	TOO_MANY_ASSETS	Attempted to enter more markets than are currently supported
17	TOO_MUCH_REPAY	Attempted to repay more than is allowed by the protocol.

Failure Info

Code	Name
0	ACCEPT_ADMIN_PENDING_ADMIN_CHECK
1	ACCEPT_PENDING_IMPLEMENTATION_ADDRESS_CHECK
2	EXIT_MARKET_BALANCE_OWED
3	EXIT_MARKET_REJECTION
4	SET_CLOSE_FACTOR_OWNER_CHECK

Code	
5	SET_CLOSE_FACTOR_VALIDATION
6	SET_COLLATERAL_FACTOR_OWNER_CHECK
7	SET_COLLATERAL_FACTOR_NO_EXISTS
8	SET_COLLATERAL_FACTOR_VALIDATION
9	SET_COLLATERAL_FACTOR_WITHOUT_PRICE
10	SET_IMPLEMENTATION_OWNER_CHECK
11	SET_LIQUIDATION_INCENTIVE_OWNER_CHECK
12	SET_LIQUIDATION_INCENTIVE_VALIDATION
13	SET_MAX_ASSETS_OWNER_CHECK
14	SET_PENDING_ADMIN_OWNER_CHECK
15	SET_PENDING_IMPLEMENTATION_OWNER_CHECK
16	SET_PRICE_ORACLE_OWNER_CHECK
17	SUPPORT_MARKET_EXISTS
18	SUPPORT_MARKET_OWNER_CHECK

COMP Distribution Speeds

COMP Speed

The "COMP speed" unique to each market is an unsigned integer that specifies the amount of COMP that is distributed, per block, to suppliers and borrowers in each market. This number can be changed for individual markets by calling the `_setCompSpeed` method through a successful Compound Governance proposal.

The following is the formula for calculating the rate that COMP is distributed to each supported market.

```
utility = cTokenTotalBorrows * assetPrice

utilityFraction = utility / sumOfAllCOMPedMarketUtilities

marketCompSpeed = compRate * utilityFraction
```

COMP Distributed Per Block (All Markets)

The Comptroller contract's `compRate` is an unsigned integer that indicates the rate at which the protocol distributes COMP to markets' suppliers or borrowers, every Ethereum block. The value is the amount of COMP (in wei), per block, allocated for the markets. Note that not every market has COMP distributed to its participants (see Market Metadata).

The compRate indicates how much COMP goes to the suppliers or borrowers, so doubling this number shows how much COMP goes to all suppliers and borrowers combined. The code examples implement reading the amount of COMP distributed, per Ethereum block, to all markets.

Comptroller

```
uint public compRate;
```

Solidity

```
Comptroller troll = Comptroller(0xABCD...);

// COMP issued per block to suppliers OR borrowers * (1 * 10 ^ 18)
uint compRate = troll.compRate();

// Approximate COMP issued per day to suppliers OR borrowers * (1 * 10 ^ 18)
uint compRatePerDay = compRate * 4 * 60 * 24;

// Approximate COMP issued per day to suppliers AND borrowers * (1 * 10 ^ 18)
uint compRatePerDayTotal = compRatePerDay * 2;
```

Web3 1.2.6

```
const comptroller = new web3.eth.Contract(comptrollerAbi, comptrollerAddress);

let compRate = await comptroller.methods.compRate().call();
compRate = compRate / 1e18;

```

```
// COMP issued to suppliers OR borrowers
const compRatePerDay = compRate * 4 * 60 * 24;

// COMP issued to suppliers AND borrowers
const compRatePerDayTotal = compRatePerDay * 2;
```

COMP Distributed Per Block (Single Market)

The Comptroller contract has a mapping called `compSpeeds`. It maps cToken addresses to an integer of each market's COMP distribution per Ethereum block. The integer indicates the rate at which the protocol distributes COMP to markets' suppliers or borrowers. The value is the amount of COMP (in wei), per block, allocated for the market. Note that not every market has COMP distributed to its participants (see Market Metadata).

The speed indicates how much COMP goes to the suppliers or the borrowers, so doubling this number shows how much COMP goes to market suppliers and borrowers combined. The code examples implement reading the amount of COMP distributed, per Ethereum block, to a single market.

Comptroller

```
mapping(address => uint) public compSpeeds;
```

Solidity

```
Comptroller troll = Comptroller(0x123...);
address cToken = 0xabc...;

// COMP issued per block to suppliers OR borrowers * (1 * 10 ^ 18)
uint compSpeed = troll.compSpeeds(cToken);

// Approximate COMP issued per day to suppliers OR borrowers * (1 * 10 ^ 18)
uint compSpeedPerDay = compSpeed * 4 * 60 * 24;

// Approximate COMP issued per day to suppliers AND borrowers * (1 * 10 ^ 18)
uint compSpeedPerDayTotal = compSpeedPerDay * 2;
```

Web3 1.2.6

```
const cTokenAddress = '0xabc...';

const comptroller = new web3.eth.Contract(comptrollerAbi, comptrollerAddress);

let compSpeed = await comptroller.methods.compSpeeds(cTokenAddress).call();
compSpeed = compSpeed / 1e18;

// COMP issued to suppliers OR borrowers
const compSpeedPerDay = compSpeed * 4 * 60 * 24;
```

```
// COMP issued to suppliers AND borrowers
const compSpeedPerDayTotal = compSpeedPerDay * 2;
```

Claim COMP

Every Compound user accrues COMP for each block they are supplying to or borrowing from the protocol. Users may call the Comptroller's `claimComp` method at any time to transfer COMP accrued to their address.

Comptroller

```
// Claim all the COMP accrued by holder in all markets
function claimComp(address holder) public

// Claim all the COMP accrued by holder in specific markets
function claimComp(address holder, CToken[] memory cTokens) public

// Claim all the COMP accrued by specific holders in specific markets for their supplies and/or borrows
function claimComp(address[] memory holders, CToken[] memory cTokens, bool borrowers, bool suppliers) public
```

Solidity

```
Comptroller troll = Comptroller(0xABCD...);
troll.claimComp(0x1234...);
```

Web3 1.2.6

```
const comptroller = new web3.eth.Contract(comptrollerAbi, comptrollerAddress);
await comptroller.methods.claimComp("0x1234...").send({ from: sender });
```

Market Metadata

The Comptroller contract has an array called `getAllMarkets` that contains the addresses of each cToken contract. Each address in the `getAllMarkets` array can be used to fetch a metadata struct in the Comptroller's markets constant. See the Comptroller Storage contract for the Market struct definition.

Comptroller

```
CToken[] public getAllMarkets;
```

Solidity

```
Comptroller troll = Comptroller(0xABCD...);
CToken cTokens[] = troll.getAllMarkets();
```

Web3 1.2.6

```
const comptroller = new web3.eth.Contract(comptrollerAbi, comptrollerAddress);
const cTokens = await comptroller.methods.getAllMarkets().call();
const cToken = cTokens[0]; // address of a cToken
```

Protocol

Markets

Prices

Developers

Docs

Governance

Overview

COMP

Leaderboard

Community

Discord

Forums

Grants

EXHIBIT J

Supplying Assets to the Compound Protocol

Quick Start Guide

Adam Bavosa Follow

Feb 12, 2020 · 14 min read

The Compound Protocol is a series of interest rate markets running on the Ethereum blockchain. When users and applications supply an asset to the Compound Protocol, they begin earning a variable interest rate instantly. Interest accrues every Ethereum block (currently ~13 seconds), and users can withdraw their principal plus interest anytime.

Under the hood, users are contributing their assets to a large pool of liquidity (a "market") that is available for other users to borrow, and they share in the interest that borrowers pay back to the pool.

When users supply assets, they receive cTokens from Compound in exchange. cTokens are ERC20 tokens that can be redeemed for their underlying assets at any time. As interest accrues to the assets supplied, cTokens are redeemable at an exchange rate (relative to the underlying asset) that constantly increases over time, based on the rate of interest earned by the underlying asset.

Non-technical users can interact with the Compound Protocol using an interface like Argent, Coinbase Wallet, or app.compound.finance; developers can create their own applications that interact with Compound's smart contracts.

In this guide, we're going to walk through **supplying assets via Web3.js JSON RPC and via proxy smart contracts that live on the blockchain**. These are two methods in which developers can write software to utilize the Compound Protocol.

There are examples in **JavaScript** and also **Solidity**.

Table of Contents for This Guide

If you are new to Ethereum, we suggest that you start by Setting up your Development Environment for Ethereum.

> *All of the **code** referenced in this guide can be found in this **GitHub Repository**: Quick Start: Supplying Assets to the Compound Protocol.*

To copy the repository to your computer, run this on the command line after you've installed git:

```
git clone git@github.com:compound-developers/compound-supply-examples.git
```

Compound Markets

The Compound Protocol enables developers to build innovative products on DeFi. So far, we've seen crypto wallets equipped with savings APRs, a no-loss lottery system, an interest-earning system for donation income, and more.

The smart contracts that power the protocol are deployed to the Ethereum blockchain. This means that at the time of this guide's writing, the only types of assets that Compound can support are Ether and ERC-20 tokens.

The currently supported assets are listed here https://compound.finance/markets. Based on the different implementation of Ether (ETH) and ERC-20 tokens, we have to utilize two similar processes:

- The ETH supply method
- The ERC20 token supply method

Like mentioned earlier, when someone supplies an asset to the protocol, they are given **cTokens** in exchange. The method for getting **cETH** is different from the method for getting **cDAI**, **cUNI**, or any other cToken for an ERC-20 asset. We'll run through code examples and explanations for the two different asset supply methods.

When supplying Ether to the Compound protocol, an application can send ETH directly to the payable **mint** function in the cEther contract. Following that mint, cEther is minted for the wallet or contract that invoked the mint function. Remember that if you are calling this function from another smart contract, that contract needs a **payable** function in order to receive ETH when you redeem the cTokens later.

The operation is slightly different for cERC20 tokens. In order to mint cERC20 tokens, the invoking wallet or contract needs to first call the **approve** function on the **underlying token's contract**. All ERC20 token contracts have an **approve** function.

The approval needs to indicate that the corresponding cToken contract is permitted to take *up to the specified amount* from the sender address. Subsequently, when the **mint** function is invoked, the cToken contract retrieves the indicated amount of underlying tokens from the sender address, based on the prior approve call.

Example code for each method (JS and Solidity) is available, open source, in the GitHub Repository linked above.

Connecting to the Ethereum Network

You will need to use the contract address for the particular network that you're developing on; start by identifying the contract address for each network in the Docs. In this guide, we'll create a fork of Mainnet, which will run on our localhost; copy the Mainnet addresses.

If you want to use a public test net (like Ropsten, Göerli, Kovan, or Rinkeby), make an Infura account at https://infura.io/ to get your API key. If you are using your own localhost test net, or the production mainnet, we will also use Infura.

If you are not hosting your own Ethereum node to access the blockchain, make an Infura account before continuing.

For more on connecting to a public Ethereum network, see the instructions in Setting up your Development Environment for Ethereum.

Supplying to the Compound Protocol on a Localhost Network

To run an Ethereum local test net on your machine, we will fork the Main network (a.k.a Homestead or Mainnet). This means that you can interact with the production smart contracts in a test environment. **No real ETH will be used and no modifications to the production blockchain will occur**. If you haven't already, install Node.js. Click here to install the LTS of Node.js and NPM.

Let's install all of the dependencies required by the project.

```
cd compound-supply-examples/
npm install
npm install -g npx

## or for yarn fans:
## yarn install
## yarn add global npx
```

This will install all of the dependencies listed in the **package.json** file, as well as a CLI tool called **npx**.

Run this command in **a second command line window** before you start running the code referenced later in this guide. The command spins up a test Ethereum blockchain on your localhost. It also seeds your localhost account with ERC20 tokens referenced at

the top of the script file. **Be sure to add your Infura project ID and Ethereum mnemonic as environment variables beforehand.**

```
## Set environment variables for the script to use

export MAINNET_PROVIDER_URL="https://mainnet.infura.io/v3/<YOUR
INFURA PROJECT ID HERE>"

export DEV_ETH_MNEMONIC="clutch captain shoe salt awake harvest setup
primary inmate ugly among become"

## Runs the Hardhat node locally
## Also seeds your first mnemonic account with test Ether and ERC20s

node ./scripts/run-localhost-fork.js
```

The script that we are running uses Hardhat to run an Ethereum node locally which has a *fork* of the history of Ethereum Mainnet. We have test Ether and test ERC-20 tokens that are seeded in the first account of the development mnemonic. Wait for the script logs to appear before continuing.

```
Running a hardhat localhost fork of mainnet at http://localhost:8545

Impersonating address on localhost...
0x5d3a536E4D6DbD6114cc1Ead35777bAB948E3643
Impersonating address on localhost...
0x35a18000230da775cac24873d00ff85bccded550
Impersonating address on localhost...
0x39AA39c021dfbaE8faC545936693aC917d5E7563
Local test account successfully seeded with DAI
Local test account successfully seeded with UNI
Local test account successfully seeded with USDC
DAI amount in first localhost account wallet: 100
UNI amount in first localhost account wallet: 10
USDC amount in first localhost account wallet: 100

Ready to test locally! To exit, hold Ctrl+C.
```

To change the types of ERC-20 assets provided to the account, uncomment the lines in the **amounts** object near the top of the **run-localhost-fork.js** script.

Supplying to Compound on a Public Network

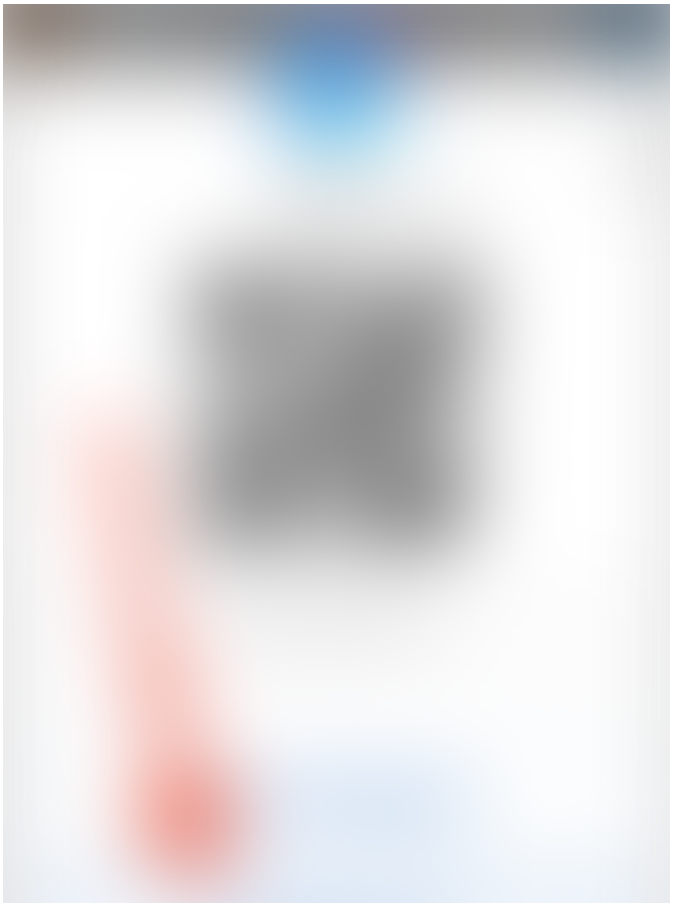
If you are supplying to the protocol on the Mainnet, Ropsten, Göerli, Kovan, or Rinkeby, you should have already located and copied the **Compound contract address** for that network (see how above). You'll need it for later.

You also should have collected some ETH for that network by purchasing/mining (Main), or a test net's faucet (all the others). This is not necessary when using a localhost fork.

For example, here is Ropsten's faucet https://faucet.ropsten.be/. You can send yourself 1 ETH every 24 hours from a single IP address. This is test ETH that is only applicable to the Ropsten test network.

Next, copy and safely store your wallet's private key. **Don't do this if you are only testing on your localhost.** The private key is used to sign transactions that are sent on the Ethereum network. The purpose of this is to certify that the transaction was created and submitted by a unique wallet.

If you are using MetaMask for your Ethereum wallet, open the menu, click the 3 dots on the right, Account Details, Export Private Key, and input your MetaMask password. This will reveal your private key. **Keep it safe!** Copy this value and save it for later.



It is a **best practice** to store a test key like this as an environment variable on your local machine. When a key is stored as an environment variable, it can be referenced in code files by a variable name, instead of explicitly with a string. This promotes code cleanliness, and **reduces the risk of exposing your secret**.

Again, if you are only testing smart contracts on your localhost Hardhat node today, **don't get your MetaMask private key**. We'll rely on a private key that comes from your environment variable mnemonic (see instructions above).

How to Supply ETH to Compound via Web3.js

Supplying Ether (ETH) to the Compound Protocol is as easy as calling the "mint" function in the Compound cEther smart contract. The "mint" function transfers ETH to the Compound contract address, and mints cETH tokens. The cETH tokens are transferred to the wallet of the supplier.

Remember that the amount of ETH that can be exchanged for cETH increases every Ethereum block, which is about every 13 seconds. **There is no minimum or maximum**

amount of time that suppliers need to keep their asset in the protocol. See the varying exchange rate for each cToken by clicking on one at https://compound.finance/markets.

For more information on cToken concepts see the cToken documentation.

In order to call the mint function, you need to first:

- Have ETH in your Ethereum wallet.
- Find your Ethereum wallet's private key.
- Connect to the network via Infura API key (see above section **Connecting to the Ethereum Network**)

There are several programming languages that have Ethereum Web3 libraries, but the most popular at the time of this guide's writing is **JavaScript**.

We'll be using Node.js JavaScript to call the mint function. The following code snippets are from this Node.js file in the supplying assets guide GitHub Repository. **Web browser JavaScript** is nearly identical to these code examples.

Let's import Web3.js, and initialize the Web3 object. It's pointing to our localhost's Hardhat node, which has 10000 test ETH in each of the test wallets. We get the same test wallet addresses every time we run a Hardhat node with the mnemonic environment variable (from the **Connecting to the Ethereum Network** section).

If you are using a public network (Ropsten, Kovan, etc.), make sure your wallet has ETH, and that you have your wallet private key stored as an environment variable. Also, have your Infura API key ready if you are deploying to a public test net.

Replace the HTTP provider URL in the Web3 declaration line with the appropriate network's provider if you are not using the Hardhat test environment.

Next, we'll add our wallet's private key as a variable. It's a best practice to access this as an environment variable.

If you are writing **web browser JavaScript instead of Node.js**, you can add the user's private key to the Web3 object by using the **ethereum.enable()** command. Here is the alternative code snippet.

Next we'll make some variables for the contract address and the contract ABI. The contract addresses are posted on this page: https://compound.finance/docs#networks. Remember to use the mainnet address if you are testing with Hardhat locally. The ABI is the same regardless of the Ethereum network that we are using.

The next section of code is where the magic happens. The first call in the main function supplies our ETH to the protocol by calling the **mint** function, which mints cETH. The

cETH is transferred to our wallet address.

The three subsequent function calls are not necessary, but they are here for illustration. The first method calls a getter function in the Compound contract that shows how much **underlying ETH** our cToken balance entitles us to. The second function shows our wallet's **cToken balance**. The third function gets the current **exchange rate** of cETH to ETH.

Our code sends 1 ETH to the contract, and gives our wallet cETH. The ratio of cETH to ETH should be in the ballpark of 50 to 1. Remember that the exchange rate of underlying to cToken **increases** over time.

Lastly, after the supply operation is complete, we'll redeem our cTokens. This is what a user or application will do when they want to withdraw their crypto asset from the Compound protocol.

The first method, **redeem**, redeems based on the cToken amount passed to the function call.

The second method, **redeemUnderlying**, which is commented out, redeems ETH based on the amount passed to the function call.

Finally, we execute the main function and declare an error handler.

Here is the command for running the script from the root directory of the project:

```
node ./examples-js/web3-js/supply-eth.js
```

Script example output:

How to Supply a Supported ERC20 Token to Compound via Solidity

The following will run through an example of adding an ERC20 token to the Compound protocol using Solidity smart contracts. The full Solidity file can be found in the project GitHub repository.

Here's an overview of supplying a token to the Compound Protocol with Solidity:

Prerequisites

- Get some ETH into your own Ethereum wallet by purchasing/mining (or faucets on test nets). This will be used for gas costs. If you're using Hardhat on localhost, you're ready.

- Get some ERC20 token, in this case Dai. If you are working in the production environment, purchase some Dai for your Ethereum wallet. If you are working with a Hardhat test blockchain locally, your test wallet will receive some Dai when you run the **run-localhost-fork.js** script.
- Get the address of the ERC20 contract.
- Get the address of the Compound cToken contract. See **Dai** on this page: https://compound.finance/docs#networks.

Order of Operations

- You **transfer** Dai from your wallet to your custom contract. This is not done in Solidity, but instead with Web3.js and JSON RPC.
- You call your custom contract's function for supplying to the Compound Protocol.
- Your custom contract's function calls the approve function from the original ERC20 token contract. This allows an amount of the token to be withdrawn by cToken from your custom contract's token balance.
- Your custom contract's function calls the **mint** function in the Compound cToken contract.
- Finally, we call your custom contract's function for redeeming, to get the ERC20 token back.

Let's get started. First we'll walk through the code in our Solidity file, **MyContracts.sol**.

We added contract interfaces. The first is for our ERC20 token contract, and the second is for Compound's corresponding cToken contract.

We'll be able to call the production versions of the 3rd party contracts using these definitions. We need to initialize them with the production address of the deployed contracts, which we pass to each of the functions in **MyContract**.

The first function in **MyContract** allows the caller to supply an ERC20 token to the Compound Protocol. We will need to pass the underlying contract address, the cToken contract address, and the number of tokens we want to supply.

The function first creates references to the production instances of Dai and cDAI contracts using our interface definitions.

Then the function logs the **exchange rate** and the **supply rate**. These calls are not necessary for supplying. They are there for illustration. You can see the amounts in the "events" output later in JavaScript.

Next, our function approves the transfer of ERC20 token from our contract's address to Compound's cToken contract using the **approve** method.

Finally, our contract calls the cToken contract **mint** function. This supplies some Dai to the protocol, and gives our custom contract a balance of cDAI.

After we have supplied some Dai, we can redeem it at any time. The following function shows how we can accomplish that in Solidity.

The **redeemCErc20Tokens** function allows the caller to redeem based on the amount of underlying or the amount of cTokens. This is indicated by calling the function with a boolean for redeem type; True for cToken, and false for underlying amount.

If there is an error with redeeming, the error code is logged using **MyLog**. Error codes for cToken contracts are described in the documentation.

Now that we have our code written, let's run it!

Compiling

Hardhat has compilation as a simple command in the development environment. The compiler settings are configured in the **hardhat.config.js** file in the root directory of the project. Run the following command to compile the Solidity smart contracts in the **contracts** folder.

```
npx hardhat compile
```

Deploying

Once you have **deployed** your contract, the script will log the new **MyContract** address.

```
npx hardhat run ./scripts/deploy.js --network localhost

Deploying contracts with the account:
0xa0df350d2637096571F7A701CBc1C5fdE30dF76A
Account balance: 100000000000000000000
MyContract address: 0x0Bb909b7c3817F8fB7188e8fbaA2763028956E30
```

Copy this and save it for later. We'll need it to call the smart contract's function to supply Dai to Compound.

Executing

The Web3.js code that will invoke our custom smart contract can be found in the **examples-solidity/** folder. Let's run through the Web3.js **supply-erc20.js** script.

First, the script makes a Web3 object and points it to the blockchain network that we want to use to supply to Compound.

Next, we make a reference to our Ethereum wallet private key. This should be a wallet that has some ETH (for gas) and also Dai (to supply to Compound). Our script's main function first transfers Dai from our wallet to **MyContract**.

Next, we make some references to **MyContract**, the **Dai contract**, and also the **Compound cDAI contract**.

Remember, the cToken contract addresses and ABIs can be found here: https://compound.finance/docs#networks, and **MyContract's** address was **logged** when we deployed the contract.

Finally, we call our main function, which first transfers Dai from our wallet to **MyContract**.

Next we call the **supplyErc20ToCompound** function in **MyContract**, which sends 10 Dai to Compound in exchange for cDAI.

The next 2 function calls are not necessary for supplying. They illustrate how to get the **balance of underlying** ERC20 asset in the protocol and the **amount of cTokens** that

MyContract now holds.

Lastly we call the **redeemCErc20Tokens** function in MyContract to redeem the cDAI for Dai. The example utilizes the **redeem** method by passing a cToken amount. Under that, there is a redeem **underlying amount** example, which is commented out.

Now we're ready to run!

If you are running this on a public network, you'll need to acquire Dai for that network.

To execute the script, navigate to the project root directory and run:

```
node ./examples-solidity/web3-js/supply-erc20.js
```

If successful, the output of the script will show something like this:

Remember that this code will work with any of the ERC20 tokens that Compound supports. You will need to swap in the corresponding ERC20 token contract address and ABI into the JavaScript.

Thanks for reading! You are now able to supply assets to the Compound protocol using Solidity or JavaScript. We walked through **supplying** Ether, and also the supported ERC20 token assets. We can also **redeem** cTokens for those underlying assets later on. There are several code examples available in the Supply Examples GitHub repository.

The next key concept of developing a DApp with the Compound protocol is **borrowing assets**. Be sure to check out the next developer quick start guide: Borrowing Assets from the Compound Protocol.

Be sure to subscribe to the Compound Newsletter. Feel free to comment on this post, or get in touch in the #development room on our very active Discord server.

Ethereum Ethereum Development Solidity Tutorial Web3 Compound