Analysis of a Blockchain-Based Stock Exchange

Ryan Wolfe[1] advised by Robert Brunner[2]

***Abstract*— As blockchain technology has gained traction, stock exchanges and regulators have begun to see its potential to perform automated, distributed clearing. Particularly interesting is the fact that an unalterable public ledger provides the opportunity to enforce certain regulations automatically. For example, if we require participants to join the network by exchanging traditional currency for a cryptocurrency, it is trivial to prevent a party from attempting to buy an instrument without sufficient funds. In this paper, we describe a blockchain-based stock exchange, as well as the advantages and risks of such an approach. This exchange could be extended to support derivatives as well by adding standardized contracts to the protocol.**

I. INTRODUCTION

A. Purpose and Origin of Blockchain

Blockchain was invented as the infrastructure of the Bitcoin network. Bitcoin is a cryptocurrency, a store of value whose ownership is determined by the consensus of a network, rather than by physical currency ownership or the records of a central party such as a bank. A cryptocurrency requires consensus on the state of transactions and ownership status, fault-tolerance, transaction source authentication, and unalterability and publicness of records. Blockchain solves these problems, and as such has become useful in a wide variety of applications ranging from insurance claim processing to supply-chain monitoring.

B. Blockchain Mechanics and Properties

A blockchain is a growing list of public records known throughout a network. Each block contains transaction data, a block index, a timestamp for ordering purposes, and the hash of the previous block to prevent alteration of the chain. A new block is confirmed when a node completes a proof-of-work for that block. A proof-of-work is a problem which takes significantly more time to solve than to verify. For example, a problem that can be solved in polynomial time but whose solution can checked in constant time would work well as a proof-of-work in many cases. In most blockchains, the proof of work is finding a value for the block such that the hash of the block satisfies a certain property, such as beginning with a certain number of 0s. In order to allow nodes to change the blocks hash without waiting for or issuing new transactions, an additional field called a nonce is also included. While waiting for new transactions, each node changes the nonce repeatedly until it funds a value that causes the blocks hash to satisfy the proof of work condition, or until it receives a new transaction. Nodes reject any invalid transactions they receive, such as those not signed with the alleged issuers private key, or those attempting to trade using funds the issuer does not have. Nodes also reject any completed blocks they receive that contain invalid transactions, fail to meet the proof-of-work condition, or have an incorrect hash of the previous block. Conflicts in the blockchain are possible: even if a node completes a valid block, it is possible for another well-behaved node to complete a different valid block before the other block has been accepted by the network. This creates a fork which is typically resolved by having each client continue from the block it receives first, and switch to the longer branch as soon as it is aware that one has become longer. While it is possible for a dishonest node or group of nodes to alter the history of the blockchain by intentionally forking at a block they want to change and creating a longer chain than the correct chain, it is highly unlikely to happen in practice, as discussed in section 11 of [1]. Additionally, greater resistance to collusion by a minority can be obtained by having honest clients propagate all equal-length branches they are aware of and randomly choosing a branch to work on, as described in [2].

C. Blockchain-Based exchanges

The public, distributed, and typically unalterable nature of blockchains has drawn significant attention from securities exchanges and regulators. In December of 2017, the Australian Securities Exchange announced that it would adopt a blockchain-based clearing system, albeit with some centralized components for regulation purposes [3]. In February of 2018, the Canadian Securities Exchange followed suit in the hope that blockchain-based clearing would be faster and cheaper than the traditional clearing model [4]. In practice, a working blockchain-based exchange would likely require some centralization. However, we will begin by analyzing the implementation and properties of a blockchain-based exchange consisting of only honest nodes, then continue by examining the security and regulation measures prompted by the possibility of dishonest actors.

II. ASSUMPTIONS

We ask whether a blockchain is a practical model for building a stock exchange, and what compromises with

[1]R. Wolfe is a student in the Department of Computer Science at the University of Illinois at Urbana-Champaign

[2]R. Brunner is a professor in the School of Information Sciences and in the Department of Accountancy in the College of Business. He has affiliate appointments in the Astronomy, Computer Science, Electrical and Computer Engineering, Informatics, Physics, and Statistics Departments; at the Beckman Institute, in the Computational Science and Engineering program; and at the National Center for Supercomputing Applications. He is also the Data Science Expert in Residence at the Research Park at the University of Illinois.

centralization might be necessary for its use. In our initial analysis, we will consider a reliable network consisting entirely of honest clients. Messages may arrive out of order, but will typically arrive in a short amount of time. The network will not be partitioned and all nodes will follow the protocol we have specified. We simulated a realistic exchange scenario by creating multiple client objects with different identifiers and having the clients execute the orders contained in past orderbook data we obtained from NYSE. Our simulated network runs on a single machine. Each client is assigned an unused port to listen on and receive messages, and sends messages to other clients by sending messages to their ports. In our simulation, different ports are analogous to different IP addresses in a real network. Because this network runs on a single machine, we achieved consensus by simply having each client broadcast messages to all other clients. However, in a real network, more sophisticated consensus algorithms would be required. In the course of running our simulation, we discovered that Pythons pickle module performs unreliably for sufficiently large objects like our blocks (the testing machine ran Python 3.5.2 on Ubuntu 16.04). We mitigated the effects of this by subsampling from our data source, only using orders for certain symbols, in order to keep the data structures in our blocks relatively small. In the cases where pickle still fails to load objects from messages, we drop the messages. A better solution for a real system would be to break up blocks into multiple, smaller objects, send them in multiple messages, and have recipients reconstruct the block once all the messages for that block are received.

III. DATA

We used historical orderbook data from NYSE as our testing data. The NYSE provides historical orderbook data as time-sorted records in the following format: Date, symbol, update time, status, buy/sell side, reference price, price point, shares, num orders. For each day, a start-of-day orderbook state file is provided, and a change file is provided. Professor Vic Anand converted the raw zip files provided by NYSE into sas7bdat files. The format of the sas7bdat files can be observed in fig.1. We wrote a python script to consume from the sas7bdat files and write to a database. The script connects to a postgresql database and creates an orderbook table with the fields [Date, Symbol, Update Time, Status, Side, Reference Price, Price, Shares]. The script inserts information from beginning of day files into the table, while constructing a dictionary to keep track of the current state of orders in the market. It then processes the change files. For each change, the script updates its market state dictionary, then writes a record to the database, where time is the time of the update to the orders for that symbol, and shares represents the total number of shares ordered at the given price for the symbol after the update and before the next update. To retrieve the orderbook for a particular symbol at a particular time, query by symbol, and for each price point, take the record with the latest update date/time that is before or at the specified time. While this database is useful for performing analysis which only requires certain snapshots of the orderbook and not necessarily the entire evolution of the orderbook, we decided it was not necessary for testing our blockchain implementation of a stock exchange. For our tests, we consumed directly from the sas7bdat files for a days worth of trades. Because NYSE did not publish the breakdowns of owners of order changes in this data set, we assumed that each change to the orderbook was caused by a single market participant. We chose which client would execute each order in a round-robin fashion. Because we arbitrarily chose clients to execute orders, the final state of the market does not reflect that given by the data set, because the quantities of stocks and funds owned made some orders invalid. However, these orders still represent a realistic trading scenario, and our use of them shows that a blockchain-based exchange can be viable.

IV. METHODS

A. ARCHITECTURE

For ease of development and testing, we wrote our exchange simulation to run on a single machine. A network is simulated by having each client object listen to a unique port on the local machine, and broadcast its messages to the ports of all the clients it is aware of. Multiprocessing is used to run the clients in parallel. Diagrams of this process, as well as that of trade matching and block verification, can be found in the next section.

B. CODE

The code for running this exchange can be found here. In its current state it cannot be run without the NYSE data, which we cannot publish. However, it can be run if you obtain the NYSE data or modify it to run with other data you provide. We have provided the most relevant parts of the code in the next section, with docstrings and additional comments.

Fig. 1. *Format of NYSE Data*

	Standard	Standard	Standard	Standard	Standard	Standard	Standard	Standard	Standard
1	Date	Symbol	Updatetime	Status	BS	Ref_Price	Price_Point	Shares	NumOrders
2	20120806.0	A	2012-08-06 07:30:01	1	0.0	39.11	39.1	-44700.0	0.0
3	20120806.0	A	2012-08-06 07:30:01	1	0.0	39.11	39.09	-15100.0	0.0
4	20120806.0	A	2012-08-06 07:30:01	1	0.0	39.11	39.08	-6600.0	0.0
5	20120806.0	A	2012-08-06 07:30:01	1	0.0	39.11	39.07	-12600.0	0.0
6	20120806.0	A	2012-08-06 07:30:01	1	0.0	39.11	39.06	-12000.0	0.0
7	20120806.0	A	2012-08-06 07:30:01	1	0.0	39.11	39.05	-11200.0	0.0
8	20120806.0	A	2012-08-06 07:30:01	1	0.0	39.11	39.04	-200.0	0.0

Fig. 2. *Order Dispatch*

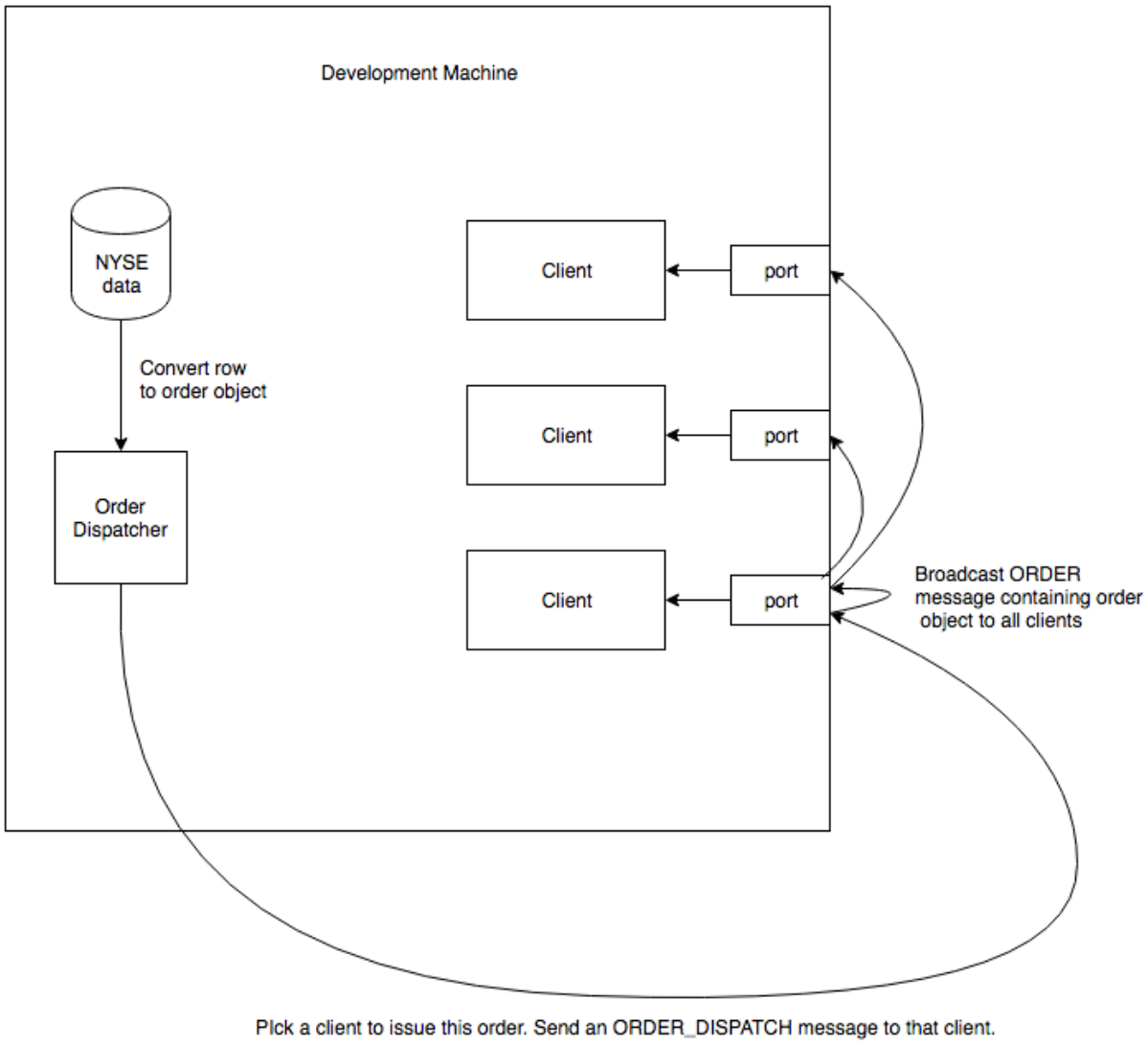


Fig. 3. *Trade Matching*

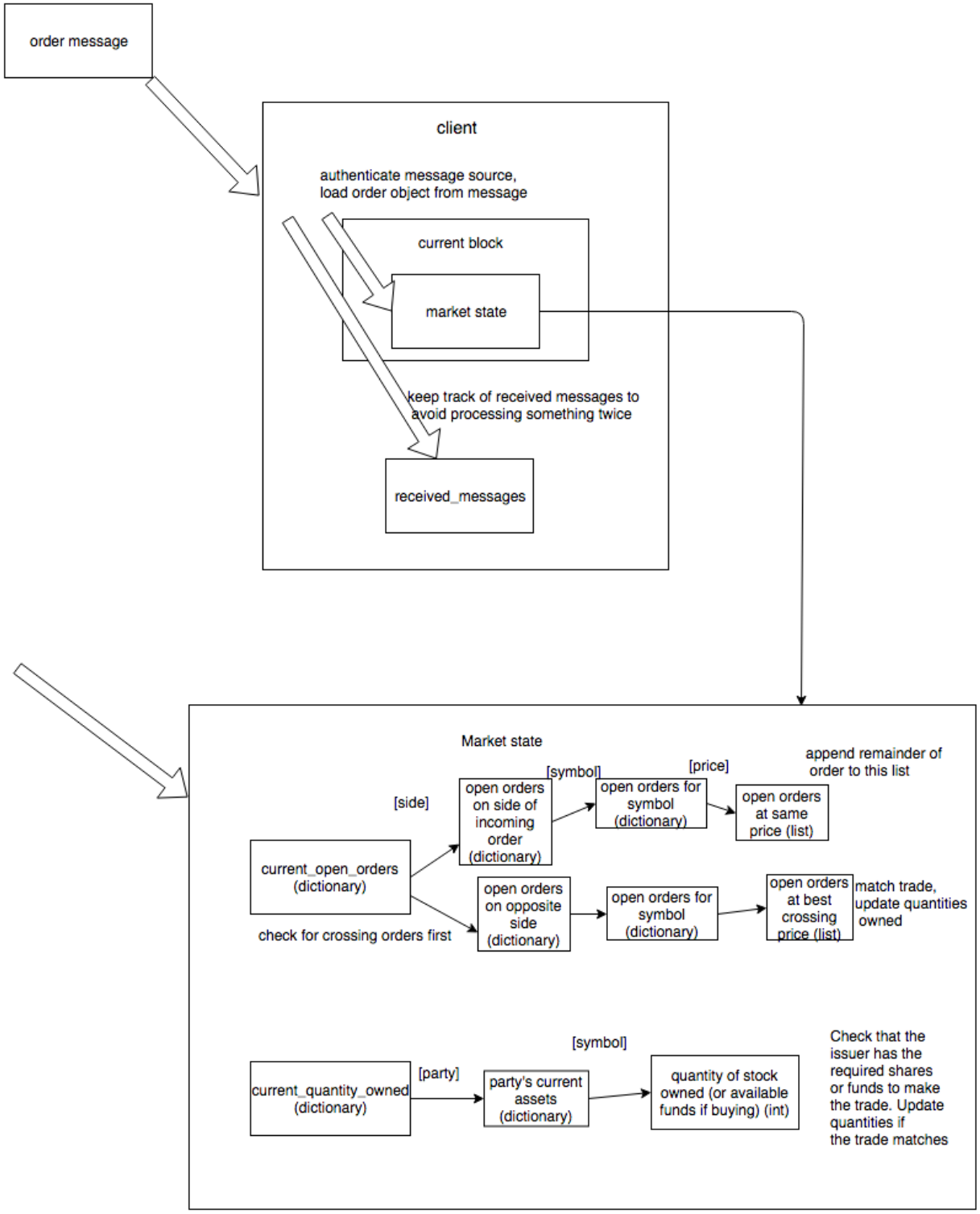
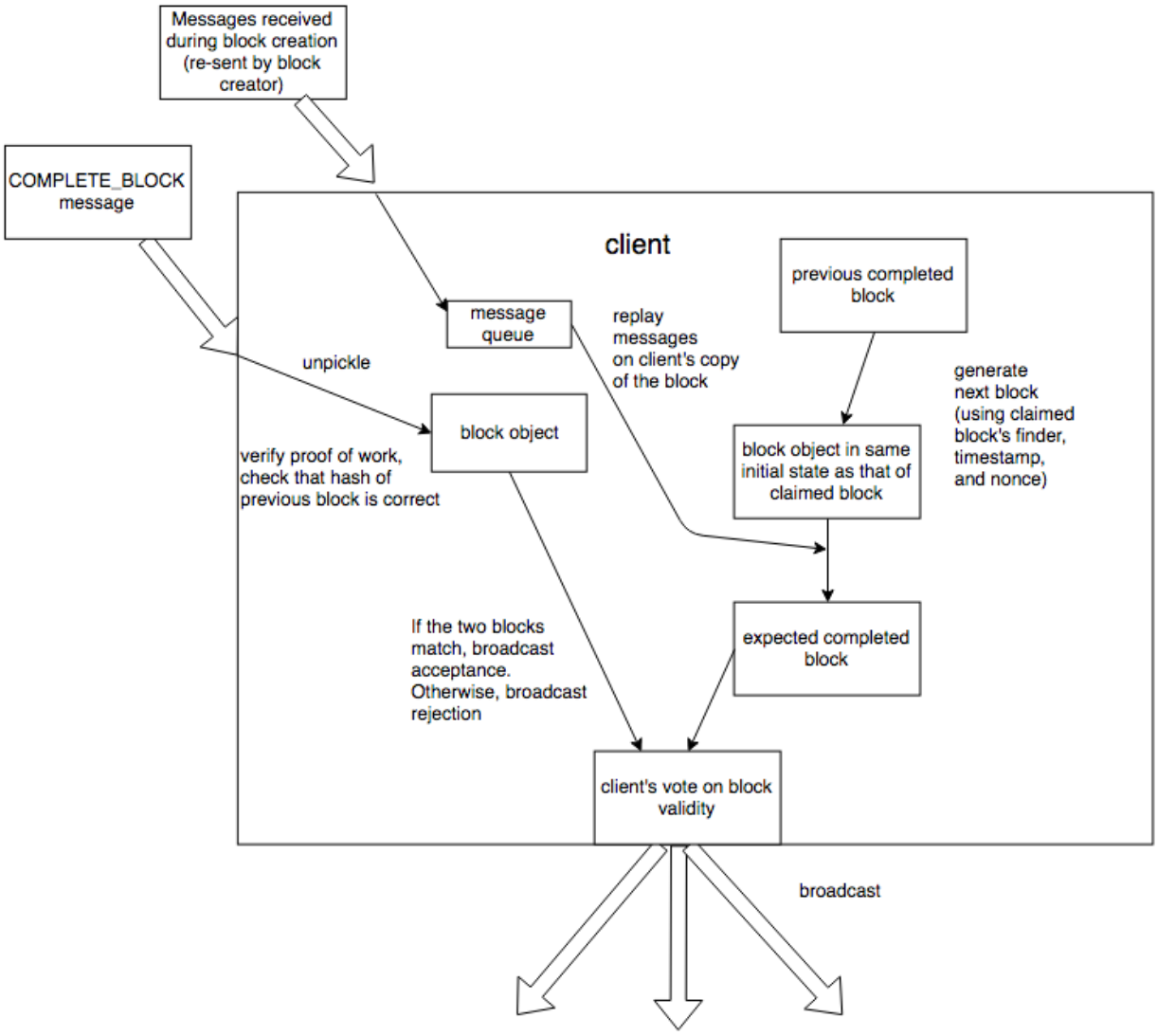


Fig. 4. *Block Validation*



```
class OpenOrder:
    """
    Order waiting for hits

    Participant: identifying string for trading party
    Symbol: identifying string for stock
    side: True for selling, False for buying
    price: price desired
    initial_quantity: number of shares desired
    remaining_quantity: number of shares that haven't been fulfilled for this order
 ↪  yet
    Timestamp: Time order is created
    """

    def __init__(self, participant, symbol, side, price, quantity, timestamp):
        self.participant = participant
        self.symbol = symbol
        self.side = side
        self.price = price
        self.initial_quantity = quantity
        self.remaining_quantity = quantity
        self.timestamp = timestamp


class ExecutedTrade:
    """
    A trade occurs when a counterparty hits an open order.
    May not complete the entire order.
    Verification process for transactions should include trade matching to
    automatically convert the appropriate quantity of crossing orders to a trade.
    Should probably wait until end of block, match crossing trades first by price
 ↪  and then by time

    Order: OpenOrder object the trade is hitting
    Counterparty: public identifier string of party hitting the order
    quantity: number of shares counterparty is buying/selling
    timestamp: time at which trade is matched
    """

    def __init__(self, order, counterparty, quantity, timestamp):
        self.order = order
        self.counterparty = counterparty
        self.quantity = quantity
        self.timestamp = timestamp
```

```
class MarketState:
    """
    initial_open_orders: dictionary: side -> symbol -> price -> list of open
 ↪  orders. represents state of orderbook at object creation
    initial_quantities_owned: dictionary: party -> symbol -> quantity. represents
 ↪  state of market ownership at object creation
    transaction_list: list of new orders and trades detected since object creation.
 ↪  Note that in practice, clients only issue orders instead of hitting specific
 ↪  trades. Allows us to replay market evolution from beginning
```

```
    new_transactions: deque of new orders and executed trades. Sorted by time
↪   before processing. used to keep track of executed trades created by crossing
↪   orders, destroyed after transaction matching
    current_open_orders: dictionary: side -> symbol -> price -> list of open
↪   orders. represents current state of orderbook after match_transactions is
↪   called
    current_quantities_owned: dictionary: party -> symbol -> quantity. represents
↪   market ownership after match_transactions is called
    current_matched_trades: list of executed trades created by matching orders.
↪   populated during match_transactions.
    """
    def __init__(self, initial_open_orders, initial_quantities_owned):
        self.initial_open_orders = initial_open_orders
        self.initial_quantities_owned = initial_quantities_owned
        self.transaction_list = []
        self.new_transactions = deque()
        self.current_open_orders = deepcopy(
            initial_open_orders)
        self.current_quantities_owned = deepcopy(
            initial_quantities_owned)
        self.matched_trades = []


    def add_transaction(self, transaction):
        self.new_transactions.append(transaction)
        self.transaction_list.append(transaction)


    def match_order(self, order):
        """
        assumes all lists of orders are sorted by time, so it should only be called
↪   during match_transactions
        """
        if order.side == SELL:
            quantity_owned = nested_dict_get(self.current_quantities_owned,
                                             (order.participant, order.symbol))
            if quantity_owned is None:
                print("Failed to place sell order for {} shares of {} because you
                 ↪  own no shares".format(
                    order.quantity, order.symbol))
            elif quantity_owned < order.initial_quantity:
                print(
                    "Failed to place sell order for {0} shares of {1} because you
                     ↪  only own {0}. Placing sell order for {0} shares".format(
                        order.initial_quantity, order.symbol))
                order.initial_quantity = quantity_owned
                order.remaining_quantity = order.initial_quantity
            # match against open buy orders

            relevant_orders = nested_dict_get(self.current_open_orders, (BUY,
             ↪  order.symbol)) # dict of price to list of orders
            if relevant_orders is None:
                if nested_dict_get(self.current_open_orders, (SELL, order.symbol,
                 ↪  order.price)) is None:
                    nested_dict_insert(self.current_open_orders, (SELL,
                     ↪  order.symbol, order.price), [])
```

```
                self.current_open_orders[SELL]
                 ↪ [order.symbol][order.price].append(order)
                return

            price, order_list = max(relevant_orders.items()) #sorted by first
             ↪ element of tuple, price
            relevant_order = min(order_list, key=lambda rel_order:
             ↪ rel_order.timestamp)

            if relevant_order.price < order.price:
                #place new order
                if nested_dict_get(self.current_open_orders, (SELL, order.symbol,
                 ↪ order.price)) is None:
                    nested_dict_insert(self.current_open_orders, (SELL,
                     ↪ order.symbol, order.price), [])

                self.current_open_orders[SELL]
                 ↪ [order.symbol][order.price].append(order)
                return

            # match with existing open orders
            if relevant_order.quantity >= order.quantity: #convert this open order
             ↪ to a trade.
                matched_trade = ExecutedTrade(relevant_order, order.participant,
                 ↪ order.quantity, order.timestamp)
                self.new_transactions.appendleft(matched_trade)
                return

            else:  # re-prepend the remainder of the new order, then prepend the
             ↪ matched trade
                self.new_transactions.appendleft(order)

                matched_trade = ExecutedTrade(relevant_order, order.participant,
                 ↪ order.quantity, order.timestamp)
                self.new_transactions.appendleft(matched_trade)
                return

        # NOTE: WE HAVE OMITTED THE BUY-SIDE LOGIC HERE BECAUSE IT IS ONLY
         ↪ NEGLIGIBLY DIFFERENT FROM THE SELL-SIDE LOGIC


    def process_trade(self, trade):
        """
        process an ExecutedTrade object
        """
        #make sure referenced order is still open and large enough
        referenced_order = trade.order

        if trade.order.remaining_quantity < trade.quantity:
            #log error
            print("Trade size larger than order size. cancelling trade.")

        trade.order.remaining_quantity -= trade.quantity

        if trade.order.remaining_quantity == 0: # close the order by removing from
         ↪ list of open orders.
```

```
            relevant_orders = self.current_open_orders[trade.order.side]
             ↪ [trade.order.symbol][trade.order.price]
            self.current_open_orders[trade.order.side]
             ↪ [trade.order.symbol][trade.order.price] = [relevant_order for
             ↪ relevant_order in relevant_orders if
             ↪ relevant_order.remaining_quantity >0 ]

        self.matched_trades.append(trade)

    def match_transactions(self):
        """
        match crossing orders and update current state of market
        """
        self.new_transactions = deque(sorted(self.new_transactions, key=lambda
         ↪ transaction: transaction.timestamp))

        for symbol_dict in self.initial_open_orders.values():
            for price_dict in symbol_dict.values():
                for price, order_list in price_dict.items():
                    price_dict[price] = sorted(order_list, key=lambda order:
                     ↪ order.timestamp)

        while len(self.new_transactions) > 0:
            transaction = self.new_transactions.popleft()
            if type(transaction) is OpenOrder:
                self.match_order(transaction)
            elif type(transaction) is ExecutedTrade:
                self.process_trade(transaction)
            else:
                print("invalid transaction type found")
```

```
class Block:
    """
    index: int. this block's place in the blockchain
    timestamp: datetime. time at which block was found to be valid
    market_state: object representing evolution of orderbook from block
 ↪  initialization to block validation
    previous_hash: hash of previous block.
    nonce: used to change hash of block even when new orders haven't been received
    finder: identifier of client who found valid block
    """

    def __init__(self, index, initial_market_state, previous_hash):
        self.index = index
        self.timestamp = None
        self.market_state = initial_market_state
        self.previous_hash = previous_hash
        self.nonce = 0
        self.finder = None

    def hash_block(self):
        return hash(pickle.dumps(self))

    def add_transaction(self, transaction):
        # transaction can be a new order or a hit on an order
        self.market_state.add_transaction(transaction)
```

```
    def generate_next_block(self): # ONLY CALL AFTER SETTING TIMESTAMP AND FINDER
        next_market_state = MarketState(self.market_state.current_open_orders,
         ↪ self.market_state.current_quantities_owned)
        next_block = Block(self.index+1, next_market_state, self.hash_block())
        return next_block
```

```
class Client:
    """
    Object representing blockchain clients. In order to run the network on a single
 ↪ machine, each client listens to a port, and broadcasts to the ports of other
 ↪ clients it is aware of.
    Clients are run in parallel with multiprocessing.
    identifier: string identifying the client to the public
    port: the port this client will listen on
    port_set: ports of other clients
    private_key: private key for this client. used to sign messages
    public_key: public key for this client. Distributed to other clients, used to
 ↪ authenticate message origin
    public_key_pem: dump of public key, sent in messages since public_key objects
 ↪ can't be pickled
    current_block: block which this client is currently trying to complete
    identifier to public key: dict of identifier->public key for other clients.
 ↪ used to authenticate message origin
    received_messages_set: set of messages received during current block
    received_messages_queue: list of messages received during current block (in
 ↪ order they were received)
    """

    def __init__(self, identifier, port, port_set):
        self.identifier = identifier
        self.port = port
        self.port_set = port_set
        self.private_key = rsa.generate_private_key(
             public_exponent=65537,
             key_size=2048,
             backend=default_backend()
            )
        self.public_key = self.private_key.public_key()
        self.public_key_pem = self.public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo
        )
        self.current_block = None
        self.identifier_to_public_key = {self.identifier: self.public_key}
        self.received_messages_set = set()
        self.received_messages_queue = []

    def receive_announcement(self, announcement_message):
        # currently not in use, but allows clients to handle addition of new nodes
         ↪ to the network even after initial network setup
        message_type, identifier, public_key_pem, port =
         ↪ announcement_message.split(",")
        public_key = serialization.load_pem_private_key(public_key_pem,
         ↪ password=None, backend=default_backend())
        self.port_set.add(port)
        if identifier not in self.identifier_to_public_key:
            self.identifier_to_public_key[identifier] = public_key
```

```
    def send_order(self, order):
        print("{} sending order".format(self.identifier))
        self.broadcast(pickle.dumps(("ORDER", order)))

    def receive_order(self, order_message):
        print("{} received order".format(self.identifier))
        message_type, order = pickle.loads(order_message)
        self.current_block.add_transaction(order)

    def receive_order_from_dispatcher(self, order_message):
        print("{} received order from dispatcher".format(self.identifier))
        message_type, order, sender = pickle.loads(order_message)
        if sender == self.identifier:
            self.send_order(order)

    def send_completed_block(self, block):
        print("{} is sending completed block {}".format(self.identifier,
         ↪ block.index))
        self.broadcast(pickle.dumps(("COMPLETE_BLOCK", block)))
        '''for message in self.received_messages_set:
            self.broadcast(pickle.dumps(("BLOCK_MESSAGE", block.index,
↪ message)))'''
        self.received_messages_set = set()
        self.received_messages_queue = []

    def receive_completed_block(self, block_message):
        message_type, block  = pickle.loads(block_message)
        print("{} received block candidate {}".format(self.identifier,
         ↪ block.index))
        if block.index >= self.current_block.index:
            self.received_messages_set = set()
            self.received_messages_queue = []
            self.current_block = block.generate_next_block()
            # to verify, replay authenticated messages on block's initial market
             ↪ state and verify that the resulting market state matches that in
             ↪ the given block.
            # also check hash of block and previous block

    def broadcast(self, message): #message must be encoded
        message = message + MESSAGE_DELIMITER
        self.received_messages_queue.append(message)
        self.received_messages_set.add(message)
        for port in self.port_set:
            if port != self.port:
                s = socket.socket()
                host = socket.gethostname()
                s.connect((host, port))
                s.sendall(message)
                s.close()

    def listen(self):
        message_type_to_handler = {"ANNOUNCE": self.receive_announcement,
         ↪ "ORDER":self.receive_order,
         ↪ "ORDER_DISPATCH":self.receive_order_from_dispatcher,
         ↪ "COMPLETE_BLOCK":self.receive_completed_block}
        s = socket.socket()
```

```
        host = socket.gethostname()
        s.bind((host, self.port))
        s.listen(5)
        while True:
            # looping over the socket buffer and using a message delimiter allows
             ↪ us to read arbitrary-length messages, although some messages can
             ↪ still be malformed because of pickling errors
            connection, address = s.accept()
            data = connection.recv(2048)
            message_bytes = data
            while data:
                message_bytes += data
                data = connection.recv(2048)
            message_list = [message + '.'.encode() for message in
             ↪ message_bytes.split(MESSAGE_DELIMITER)[:-1]]

            for message in message_list:
                try:
                    # synchronization protocol: broadcast message. keep list of
                     ↪ messages seen before. rebroadcast all received messages
                     ↪ that haven't been seen before.
                    if message not in self.received_messages_set: # ignore messages
                     ↪ you've seen before
                        decoded_message = pickle.loads(message)
                        message_type = decoded_message[0]
                        self.received_messages_set.add(message)
                        self.received_messages_queue.append(message)
                        self.broadcast(message)
                        handler = message_type_to_handler[message_type]
                        handler(message)
                except(pickle.UnpicklingError): # sometimes pickle fails to dump or
                 ↪ load large objects properly. We will consider this case a
                 ↪ dropped message.
                    print("pickle failure. dropping message")
                except(EOFError):
                    print("pickle failure, dropping message")
                except(UnicodeDecodeError):
                    print("pickle failure, dropping message")
                except(KeyError):
                    print("pickle failure, dropping message")

            self.attempt_block_completion()

    def attempt_block_completion(self):
        self.current_block.finder = self.identifier
        self.current_block.nonce = randint(0, 100000)
        self.current_block.market_state.match_transactions()
        self.current_block.timestamp = datetime.datetime.now()
        dump = pickle.dumps(self.current_block)
        first_hash_char = hash(dump)[0]
        if first_hash_char<10:
            self.send_completed_block(self.current_block)
            # move to next block.
            self.current_block = self.current_block.generate_next_block()
```

V. CONCLUSION

We successfully provided a working prototype for a blockchain-based exchange. Admittedly, we made some aggressive assumptions in order to develop a working system quickly; a real exchange would allow clients to trade on far more symbols than we did, and messaging would likely be much less reliable. However, a network composed of trading firms would have far more processing power at its disposal than the single testing machine we used to develop our system, meaning that the additional complexity of verifying blocks would likely remain manageable. Additionally, even in a network with frequent transactions, blocks can be kept at a reasonable size by choosing a suitably easy-to-solve proof-of-work algorithm. We thus conclude that building a working blockchain-based exchange is practical. However, some additional concerns remain, as we shall discuss in the next section.

VI. FUTURE WORK

One problem with a blockchain-based exchange is that traders may not want to be forced to take on exposure to a cryptocurrency merely to do business. One solution to this could be backing the cryptocurrency used in our implementation with a traditional currency. This could be accomplished by refusing to issue cryptocurrency for block creation and creating a bank account for the use of the blockchain network. Cryptocurrency funds would be issued to participants who made verified deposits in the account, and participants would be required to relinquish funds and have the transaction verified in order to withdraw from the account. However, this solution would make the network dependent on a centralized party. While this is somewhat contrary to the purpose of using a blockchain, trade clearing would still be accomplished quickly in a distributed system. There may be another solution to this that will reveal itself with more research, but it is possible that a tradeoff between cryptocurrency exposure and network integrity is unavoidable. Another problem is that in our implementation, we assumed that nodes have no particular inclination towards lying, especially about transactions they are not involved in. This assumption is typically true for cryptocurrencies, but far less so for a stock exchange. Every order has an effect on the market and the profit that other participants stand to make. For example, this conflict can present itself in reaching consensus on the timestamp of an order. Multiple participants will frequently try to hit the same order, and hits on the order will be processed in chronological order. Thus, each participant is incentivized to make its order timestamp appear as early as possible, while making those of other participants appear as late as possible. Clearly we cannot simply trust an order issuer about their timestamp, so we must reach consensus on when the order was submitted, using its arrival times at other nodes. However, the other nodes might adopt a protocol that attempts to reach consensus maliciously, by voting for the latest timestamp they have seen for that order. However, this problem might be mitigated by adding nodes owned by regulators to the network, and adapting the protocol such that the earliest time at which a message is seen by any regulator node is used as the timestamp for that order. A major goal of future work on this prototype would be to implement derivatives trading through standardized smart contracts. Smart contracts allow the implementation of more complex transactions where a portion of the contract may be executed at a later date. This would allow us to implement derivatives clearing. While a blockchain-based exchange for stocks would be useful, the gains in clearing speed and regulation enforcement would be even greater in derivatives exchanges. For example, the network could automatically reject attempts at naked shorting, automatically update futures margin accounts, and it could employ a model based on market participants' past behavior and the nearness of a contract's execution date to prevent participants from making trades that would put them at a high risk of defaulting on a contract. The complexity and risk involved in derivative clearing make a blockchain-based exchange an attractive solution.

REFERENCES

[1] S. Nakamoto, *Bitcoin: A Peer-to-Peer Electronic Cash System*. Bitcoin.org, 2008

[2] I. Eyal and E. Gun Sirer, *Majority is not Enough: Bitcoin Mining is Vulnerable*. cornell.edu, 2013.

[3] David Meyer, *The Australian Securities Exchange Just Made Blockchain History*. Forbes, 2017.

[4] *CSE Unveils Canadas First Platform for Clearing and Settling Securities through Blockchain Technology*. Canadian Securities Exchange, 2018.