Decentralized, Off-Chain, per-Block Accounting, Monitoring, and Reconciliation of Blockchains

Thomas Jay Rush
http://quickblocks.io
July 24, 2017

If one wishes to acquire a list of transactions per account (or series of accounts) from a blockchain in the hope of producing a full accounting for those accounts, there is no independent, decentralized method to verify that the list of transactions retrieved is correct and all inclusive. Nor has it been possible, previously, to assure that the account balances, as reported by state retrieval functionality of the blockchain software, are in agreement with the balances implicitly represented by lists of transactions stored in the blocks. Nor has it previously been practical to acquire such a list of transactions in a timely manner. **Keywords: blockchain, Ethereum, data analytics, blockchain auditing, blockchain accounting**

Introduction

Blockchains store lists of programmatically-validated transactions in cryptographically-linked blocks. Most users presume that the list of transactions, which represent incoming and outgoing value transfers or access to a smart contract, accurately represents the account balances as maintained by the system.

Each block, which is appended to the blockchain depending on configuration, contains anywhere from zero to hundreds of transactions. Many transactions represent transfers of value to and from more than one account. These transactions are appended and maintained in the blockchain in time order.

While some blockchains, such as the Ethereum chain, store per-account account balances directly, generally speaking the account balances in a blockchain are implicitly built from its list of transactions. The cryptographic nature of the hash-based linking of each block to its parent, and each account's balance to the balances of those in parent blocks, may lead one to conclude that the two sources of information concerning an account's balance are always in agreement.

While we at QuickBlocks™ believe with very high certainty that blockchain balances and list of transactions are identical, we also recognize a tickling thought in back of our minds that insists on saying, "Yeah, but what if they're not?"

Below we detail our method for accomplishing per-block accounting and reconciliation for a given account (or series of accounts) for any blockchain. We do this off-chain, which means we rely as little as possible on the blockchain itself. This does not entirely eliminate the tickling thought, but it does scratch it a little bit.

Why the Problem is Difficult

In many programmable blockchains, including the Ethereum blockchain, there are two different modes of exchanging value or messages between accounts.

In the following, we will call the first mode a 'regular' or 'external' transaction. External

transactions, once processed by the system, appear directly in the list of transactions that constitute a block. All external transactions, once they are mined, even those that end in failure or transfer zero value, are stored in the blockchain.

The second mode of exchanging value or messages between accounts is called an 'internal transaction' or 'contract message call.' This type of transaction is initiated internally to the system as the result of an external invocation of a smart contract. Internal transactions or message calls do not appear as separate transactions listed directly in a block. Instead they are represented in the trace of the initiating external transaction.

From the perspective of the technologist developing the blockchain, these two types of transactions are different and necessarily must be recorded separately (and indeed are recorded separately in the blockchain).

From the perspective of an accountant, auditor, a regulator, or even a regular end user, who is simply looking for all debits and credits on his/her accounts, these two types of transactions are—and very much should be—indistinguishable.

It is this dichotomy between the differing views of internal vs. external transactions that our solution, QuickBlocks™, rectifies.

One of the outcomes of the present paper is to better clarify these two differing views of "transactions" on a blockchain. We will refer to the technologist's view of transactions (i.e. internal vs. external) as "blockchain transactions." We will refer to an accountant's view of a transaction as "accounting transactions."

The methods and techniques described below accomplish the task of translating blockchain-centric internal and external transactions into regular accounting or auditing transactions.

It is important to note that completing this "translation task" may introduce a source of errors on its own. At the same time, it is important to recognize that the blockchain itself may contain errors. A further method is therefore described that verifies the blockchain's accounting both implicitly through its list of transactions and explicit through its reported account balances.

This additional verification is accomplished while at the same time verifying that QuickBlocks™ own acquired list of transactions for an account or series of accounts is complete and accurate.

Background

Before we begin our discussion of our processes, we describe the data structures that support our work. We describe a stylized view of the data structures in current blockchain implementations. We then describe additional off-chain data and modifications we've made to these structures that enable our accumulation of transactions per account in an efficient manner.

Data Structures

Addressing no specific blockchain in particular, below we present a sample representation of the data stored in a programmable blockchain.

Note that we've removed many of the actual data as these removed fields relate to issues of no interest to our purposes. For example, we've removed many of the cryptographic hashing information as well as information relating to the mining mechanisms. Our system is able retrieve this information if needed, however, in pursuit of clarity, this unrelated information has been removed.

CBlock Data Structure

The primary data structure of the system, obviously, is the block. A representative description of which is presented in Appendix A-1.

Each of the data fields mentioned in the appendix is retrieved for each block directly

from a running blockchain node using the RPC interface.

CTransaction Data Structure

The transactions included in a block's *transactions* list are initiated by external accounts. In this sense, external transactions come from "outside" of the blockchain.

Contra-wise, internal transactions are initiated by smart contracts from "inside" the blockchain.

It is this distinction between transactions initiated by external accounts and smart-contract accounts that is of importance in distinguishing external from internal transactions. Only external transactions are included in a block's *transactions* list.

The transaction data structure is detailed in Appendix A-2.

CReceipt Data Structure

After processing a transaction, the blockchain node produces a receipt. The receipt carries information indicating how much gas was consumed, a list of any traces, and the final disposition (or error status) of the transaction. This information is needed to translate the transaction from a blockchain transaction to an accounting transaction.

The format of the receipt data is presented in Appendix A-3.

CLog Data Structure

The Ethereum virtual machine (EVM) provides the capability to write arbitrary data to a log. This log may deliver information to a distributed application via an interface such as the RPC, indicating the disposition of a particular transaction.

For example, a "Mint" log entry may be used to indicate the creation of an ERC20 token. This mechanism is intended to help the application developer understand what's happening in the smart contract, but as we shall see, the mechanism is limited, poorly used, and ultimately inadequate if one wishes to build a full list of transactions per account.

The transaction's receipt includes a, possibly empty, list of log entries. The format of a log entry is presented in Appendix A-4.

CTrace Data Structures

A final series of data structures, traces, are described next. Traces, because of space considerations, are not stored in the QuickBlocks™ binary cache. This allows us to adhere to our design principal of imposing a minimal impact on the target machine. The trace data carries two further data structures, *traceAction* and *traceResult*, detailed in Appendices A-5 through A-7.

Having now arrived at the bottom of the data structures, we remind the reader that we are deep inside the trace of an external transaction.

Smart contracts are able to call into other smart contracts ad-infinitum. These 'message calls' or 'internal transactions' potentially transfer value from one account to another (and in many case do).

Notice how the `CTraceAction` and `CTraceResult` data structures reflect the `CTransaction` and `CReceipt` data mentioned above. The trace data represents what we've called 'internal transactions' above. It is not a surprise that the data structures mirror each other.

Hopefully it is becoming obvious to the reader why a technologist might not recognize a transaction's trace as an 'accounting transaction' even though it does transfer value. The names and meanings of the trace data fields reflect the transaction and receipt data fields, and it is this aspect of the data that QuickBlocks™ takes advantage of.

This, then, is the dichotomy we mentioned earlier. Technologists see the trace data and the top-level transaction data as two different data structures. Accountants and auditors view them as identical in meaning because in

each case value is being transferred from one account to another. Accountants need debits and credits on an account. Technologists properly note that the trace data structures are not the same as a transaction or receipt.

We have now completed our description of the existing data structures, and have tried to explain the meaning of that data. In the remainder of this paper, we explain how QuickBlocks™ accomplishes its primary task: building a complete and accurate, per-account list of transactions (both internal and external) needed to furnish accounting or audit data on a particular account or set of accounts.

Our Method

The first part of the QuickBlocks™ system is the storage of information needed to quickly build per-address lists of transactions while at the same time ensuring that doing so does not overly burden the target system. This process is called the `blockScrape` application.

The blockScrape Application

For each block and for each transaction in each block and for all receipts and all log entries in each transaction, `blockScrape` retrieves and stores that data in a highly-optimized binary format for fast later retrieval. Note that QuickBlocks™ does not attach traces to the binary data at this point so as to not overburden the target machine. Under the monitoring section below, we explain how and when we are forced to access trace data.

`blockScrape` does not create an index of transactions per account. We are building a system that monitors only particular lists of addresses, therefore, we avoid the computationally large amount of work needed to produce a sorted index of all accounts.

This design choice, we believe, has allowed us to maintain our "fully decentralized" stance in relationship to the blockchain node.

By avoiding unnecessary indexing of accounts, the huge majority of which our end-users will never be interested in, and therefore will never request, we allow for faster operation at each block. Maintaining such a sorted index, even with sophisticated database software, in our judgment, would have a negative performance impact at every block.

Our avoidance of maintaining a sorted index of accounts imposes a negative impact on the first retrieval of an account's transaction list, but we believe this is a reasonable engineering trade-off.

A second consequence of the design decision not to store a sorted indexed list of accounts is that we may parallelize the block retrieval task in the future. Currently, the process is not parallelized. Maintaining a sorted list of accounts lessens the chance of parallelizing the process and is therefore avoided.

Furthermore, because a particular end-user on a given machine will ask only for only a miniscule percentage of accounts (i.e. only those accounts he or she is interested in), we effectively amortized the cost of building these lists of transactions over time.

Above we mentioned that we retrieve the data from the running node at every block. We do this only in the conceptual sense. In reality, we store only those blocks, transactions, receipts and logs requested directly for the particular accounts our end-users are interested in.

For example, if our end user is only interested in accounts that did not transact prior to block 3,000,000, then the earliest binary block in our cache would be block 3,000,000. This, again, lessens the impact of the QuickBlocks™ software on the target machine helping us meet our mission of running in a fully decentralized manner.

Data Storage

We store the block data (along with its associated transactional data) in individual files, one per block, 1,000 blocks per folder in subfolders of one hundred 1,000 block groups this under one hundred such groups. We do this for easy, quick access to a particular block without having to traverse a directory structure.

For example, the binary file for block 3,872,300 is stored in the folder …/scraper /00/38/72/003872300.bin where '…/' is a user specified top of the data folder allowing for storage of the QuickBlocks™ data on an external media (a highly-recommended practice).

We foresee experimentation to explore storing of more blocks per file (perhaps as many as 1,000 blocks per file) depending on the performance improvements relative to a single block per file as we currently do.

While our method of storing the data in binary files is significantly faster (up to 80 times faster) than repeatedly retrieving the same data using the RPC, it is still not fast enough. For this reason, we've implemented two different methods of improving the retrieval of the data. We describe these methods below. We intend to continue our experimentation and ultimately choose whichever method provides the fastest access to the data while imposing a minimal data storage requirement on the target machine.

Enhanced Bloom Filters

We mentioned in the description of the block data a field called `logsBloom` which stores both a transaction's receipt-level filter and a block-level bloom filter.

In an effort to improve the performance of the account filtering function which is necessary to properly build lists of transactions, and furthermore as a way to avoid having to scan through the entire data structure of each block in order to decide if a block contains transactions of interest, we've improved upon (i.e., enhanced) the node-generated block-level bloom filters by inserting the addresses noted below as "addresses of interest" into each filter. We call this improved method of the building bloom filters "Enhanced Bloom Filters."

The blockchain node stores the following addresses into a transaction's `logsBloom`:

Addresses stored in bloom filters
1) The initiating address of the log entry
2) Indexed log parameters (i.e. `topics`) that happen to be account addresses
3) Addresses involved in internal transactions[1]

In other words, only addresses involved in the generation of the log entries or internal calls are included. To this, we add the following addresses found elsewhere in a transaction:

Addresses in Enhanced Bloom Filters
The `to` field in the `CTransaction`
The `from` field in the `CTransaction`
`contractAddress` in the `CReceipt` if present
`To` addresses found in `CTraceAction` data
`From` addresses found in `CTraceAction` data

The node-generated bloom filters are inadequate for our purposes as they do not contain all address involved in a transaction. (Only those addresses that were involved in generating logs are included.)

For this reason, we do not store the node-generated bloom filters in our binary cache, instead we store Enhanced Bloom Filters at the block level which include all transactions of interest.

Identically to blocks, Enhanced Bloom Filters are stored in folders of 1,000 blooms under two layers of one hundred per bloom folders. We therefore are able to remove the

[1] Currently, this is only true for the Parity Ethereum node. We hope in the future, it will be true for all nodes on all blockchains.

less-informative blooms stored in the binary block data which greatly reduces the amount of data stored in our cache.

Subsequently, we use the Enhanced Bloom Filters to determine if a block is of interest for a particular account or set of accounts. This method is fully explored in [3].

Per-Block Lists of Involved Accounts

An alternative method to the one just mentioned is to collect, at each block, a straight list of all accounts involved in that block. This method is similar to the Enhanced Bloom Filter method mentioned above, however, instead of storing bloom filters per block, we store the entire (sorted) list of addresses found in each block. Note that this does not mean that we store a sorted and fully-indexed list of all accounts for the entire blockchain.

This method is a compromise between a fully sorted index of addresses, which we believe places too high a burden on the target machine and also infringes on the ability to process blocks in parallel, and a method by which to acquire quickly-accessed lists of accounts involved in each block. By storing lists of addresses involved in each block, we retain the ability to parallelize the process.

Comparison of Storage Methods

The difference between the Enhanced Bloom Filter method mentioned above is the current method is that whereas under the first method the actual list of accounts per block is not determined until later, the current method does fully determine the list of accounts per block. In either case, we do not store the full block until the end-user explicitly asks for that data.

Under the Enhanced Bloom Filter method, the addresses involved in a transaction are compressed into a small bit-array that represents those addresses. A bloom filter, as described in [5], never reports a false negative, which means one need not, in that case, process the block to determine if there are transactions of interest. If the bloom filter reports "not present," then the address was definitely not involved in that block. If the bloom filter reports "yes," it means "maybe." In this case, the account requires further processing.

In the second method of listing each address per block, we've eliminated both false positives and false negatives at the cost of having to search sorted lists. As is described in the monitoring section below, building list of transactions involving a particular address is then simply a matter of traversing the per-block lists of accounts to find blocks of interest.

We delay until the account monitoring step to further detail where in the block an account is involved. For example, we do not store the *transactionIndex* an account was involved in at this stage. Furthermore, we do not store which traces (if any) were involved.

As we will see below, an implicit three-part numbering scheme, which uniquely identifies an "accounting transaction" begins to emerge. In order to identify accounting transactions, we need store only the *blockNumber*, *transactionIndex*, and *traceID* of a transaction. We refer to these tuples as a per-account caches below.

The above discussion relates to the `blockScrape` portion on the QuickBlocks™ system. `blockScrape` prepares the data for further processing. It does not fully answer the question of a per address index for particular accounts. Instead it simply prepares the way for the monitoring portion of the system to ascertain particular transaction lists.

In this way, we are able to keep the data storage requirements of the system at a minimum while at the same time keeping the per-block processing time at a minimum.

We now move on to a discussion of account monitoring applications.

The Account Monitoring Applications

Given the existence of the prepared data as described above, the next task is to accumulate full and accurate lists of transactions per requested address. Following this we are in a position to provide an accounting at the end of every block for one or more inter-related addresses.

In the following discussion, we are interested in two sets of two different data entities. The two data entities are the account balances as represented by the list of transactions on a particular account and the separate (but related) account balances stored by the node. The two sets are (1) these values as stored by the node, and (2) these values re-calculated by QuickBlocks™.

There are, furthermore, two competing motivations for doing this.

The first motivation for building two different sets of two different versions of the data has QuickBlocks™ searching for, and (hopefully) verifying the absence of, bugs or miscalculations in the blockchain accounting itself.

While we feel confident that the chance of error in the blockchain accounting is astronomically small, we do not feel quite so confident in the accounting of application-level code for the assets built on top of the blockchain such as ERC20 tokens.

One need only consider "The Great DAO Debacle of 2016™" to recognize the truth of this last statement.

In the summer of 2016, a smart contract called "The DAO" was hacked and had more than $50,000,000 US stolen.

While a smart contract monitoring system such as QuickBlocks™ could never have prevented such a hack, it could have identified the hack many hours prior to when it was first identified. Instead of a young man on a commuter train posting a Reddit post saying, "I think the DAO is being drained…" [and it has been being drained, once every fourteen seconds, for more than six hours]," A QuickBlocks™ monitoring system might have allowed him to say, "There was an attack on the DAO 14 seconds ago—someone should do something."

With foreknowledge of the existence of a monitoring system, developers can make design decisions purposefully intended to insert messages that may have helped to identify potential trouble.

As an example of this latter point, consider this pseudo-code from a simple ERC20 token:

```
contract Token {
   event Sending (address to, uint256 amt);
   event Sent (address to, uint256 amt);
   function Token () {}
   …
   function Transfer (address t, uint amt) {
     // to protect send and process changes
     …
     // event notes the send started
     Sending (to, amt);
     if (msg.send.transfer(to, amt)) {
      // event notes the send completed
      Sent (to, amt);
      return;
     }
     throw;
  }
}
```

At first glance, the above code seems redundant because if the `throw` after the `msg.send` fails, no `Sent` log will be written. Why would one need to signal both the beginning of a Transfer and the end of a Transfer? Wouldn't the existence of the `Sent` log indicate success and the lack of a `Sent` log indicate failure?

The answer to this question is revealed in the easy to understand stream of log entries generated by the above pattern. There are two cases to consider.

In the first case the `to` account is either a non-malicious smart contract or regular account. In this highly-likely case, the stream of transactions and events look like this:

```
transaction:
  transfer { from: 0xabcd, amount: 0x100 }
  event → Sending (0xabcd, 0x100);
  event → Sent (0xabcd, 0x100);

transaction:
  transfer { from: 0xabcd, amount: 0x250 }
  event → Sending (0xabcd, 0x250);
  event → Sent (0xabcd, 0x250);
```

For each `Transfer` transaction, there is a single `Sending` log followed by a single `Sent` log. In the case of an error, there would be no events.

In the second scenario, where the receiving smart contract is malicious (which was the case with The DAO), the stream of transactions and logs would look much different:

```
transaction:
  transfer { from: 0xabcd, amount: 0x100 }
  event → Sending (0xabcd, 0x100);
  event → Sending (0xabcd, 0x100);
  event → Sending (0xabcd, 0x100);
  event → Sending (0xabcd, 0x100);
  event → Sending (0xabcd, 0x100);
  event → Sending (0xabcd, 0x100);
  // until the malicious smart contract
  // purposefully quits so as to
  // avoid running out of gas
  event → Sent (0xabcd, 0x250);
  event → Sent (0xabcd, 0x250);
  event → Sent (0xabcd, 0x250);
  event → Sent (0xabcd, 0x250);
  event → Sent (0xabcd, 0x250);
  event → Sent (0xabcd, 0x250);
```

It should be obvious that a simple, watchful monitoring program, looking for a pattern of more than a single `Sending` event before its corresponding `Sent` event is a clear indication of a recursive attack.

Another example of a smart contract programmer anticipating the use of a monitoring system might be at the creation of ERC20 tokens during an ICO. In many of the current implementations we've seen, the following logs are generated by the ERC20 smart contract:

```
event Mint (address account,
            uint amount);

event Transfer (address toAccount,
                uint amount);
```

If instead, the programmer provided the following data in his/her events,

```
event Mint (address account,
            uint amount,
            uint accountBalance);

event Transfer (address toAccount,
                uint amount,
                uint toAccountBalance,
                uint fromAccountBalance);
```

the task of accounting for the ICO would be made easier. Here, the smart contract is reporting data to the monitor information the monitor otherwise need to calculate itself.

The monitoring software may simply accept the balances as correct, but it could just as well duplicate the accounting of the smart contract thereby providing a second layer of confidence in the smart contract code.

Moving on from a discussion about the reasons behind using a smart contract monitoring system, we now present a description of how QuickBlocks™ monitors work. The reader is reminded that, in almost every case, QuickBlocks™ makes design decisions towards maintaining a low-impact, decentralized stance on the end user's machine. For this reason, some things may work in unexpected ways.

The reader is also reminded, again in order to maintain a decentralized stance, a QuickBlocks™ monitor does not solve the monitoring problem for every account on the blockchain, but only for particular accounts or collections of accounts. This design choice has imposed methods that may not be obvious if the monitoring system admits centralization on large-scale computers.

Components of a Decentralized Account Monitoring System

A new monitor, which is a collection of software modules as described in this section, is built automatically for each blockchain address or series of blockchain addresses requested by an end user.

An interesting attribute of a smart contract is that (a) once it is deployed the source code

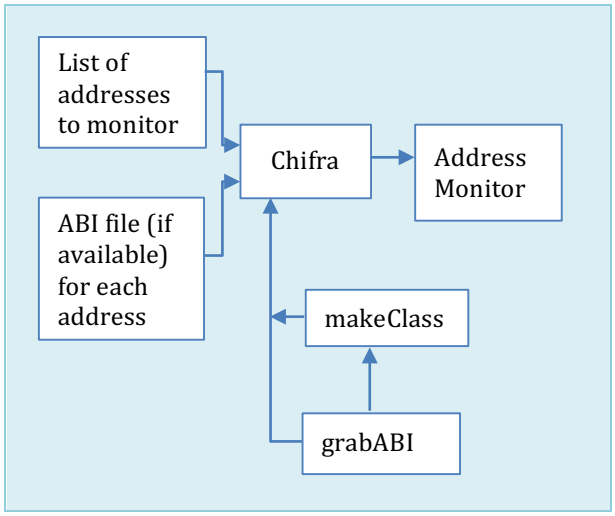
for that smart contract never changes, and (b) the full interface for the smart contract is specified in a single file called an Application Binary Interface (or ABI) file. This ABI file, which represents an immutable piece of code, also never changes.

These two facts inform the nature of the monitors in two profound ways. First, since deployed smart contracts are immutable, the monitor code may be immutable as well. This eliminates the need for maintenance and minimizes long-term operational costs of a smart contract monitor. Once deployed, a smart contract behaves identically forever, and so may its associated monitor.

The second implication of the above facts is that the source code of the monitor may be generated automatically from the ABI file. This means that a QuickBlocks™ monitor may be built automatically and will operate without modification forever.

These facts allow us to automatically create monitor systems requiring no further modification. Having said that, our system generates C++ code. Therefore, customization per monitor is possible if desired.

A high-level overview of the creation of a QuickBlocks™ monitor is presented next.



grabABI / makeClass

The two components, `grabABI` and `makeClass`, are described in [4]. Their collective task is to produce, for each monitored collection of smart contracts, a C++ class library capable of parsing the data produced by that system. This process is described in the referenced paper.

Generally, given one or more smart contract addresses, `grabABI` pulls the ABI file(s) for those address from any available source (for example the .eth domain name system or http://Etherscan.io—a centralized solution). The user may also provide the ABI file directly if full decentralization is desired.

The `grabABI` process does two things: (1) it makes canonical and then encodes each of the smart contract's function and event signatures. It does this using a process described in the Ethereum Yellow Paper [1]. After encoding the function and event signatures, `grabABI` generates a class definition file for each function or event. `makeClass` uses these files to generate off-chain C++ classes that are subsequently packaged into a parser library and linked to the monitor.

Note that all of the above is created automatically by the `chifra` application given the contract's address alone. Further note that, even though the C++ code is generated automatically, it is C++ code and therefore may be customized as the end user wishes.

As is well known, smart contracts cannot interact with the real world. We envision future smart contracts written with a QuickBlocks™ monitor in mind and able to control real world devices.

An off-chain smart contract monitor, mirroring the behavior of the on-chain smart contract, can, if it encounters a particular function, initiate an interaction with an IoT device. An example of this type of interaction might be a smart contract controlling a bike lock, for example.

Chifra

The `chifra` process controls the above described process of creating a monitor's parser library as well as building the remainder of the monitor. `Chifra` generates commands that kick-off the `grabABI` process which generates class definition files that `makeClass` uses to build the source code. `Chifra` then copies a templated (empty) monitor that provides all the functionality of a monitor as described in the next section.

Smart Contract Monitors

A smart contract monitor is an off-chain piece of software that "watches" an on-chain smart contract. It does this by listening to the blocks being added to the head of the chain, identifying the transactions of interest (related to a series of addresses comprising the system), parsing the related transactions, possibly displaying or exporting the transactions, potentially looking for specific anomalous behavior, and then storing the parsed data into a local cache for significantly faster later retrieval.

(We find it odd, as you may, that the off-chain monitor "watches" an on-chain smart contract by "listening" to the blockchain.)

The first process (watching the end of the chain) uses the `blockScrape` application mentioned above. The monitor records the most-recently visited block number, asks the node for the newest block, and proceeds to pull previously un-processed blocks.

This process runs periodically as per the settings of the monitor. A common setting is once every 15 seconds which ensures a relatively fresh cache while ensuring low-impact on the target machine.

The monitor writes its findings to a transactional cache specific to that smart contract system.

A monitor's transactional cache is very simple. It consists of nothing more than a binary array of blockNumber / transactionIndex / traceID representing the encountered transactions of interest.

The second mode of operation for a monitor is that of displaying the transactions from the cache. This is accomplished simply by traversing the monitor's binary transactional cache, retrieving the full block from the `blockScrape` binary cache, and then rendering the transaction as described in [4, Appendix B].

What we've described is the accumulation and re-rendering of a series of transactions related to a smart contract system (which may include one or more accounts), in the remainder of this paper, we describe use cases for these per-account, quickly-accessible lists of transactions.

Use Cases

Per-Block Instantaneous Reconciliation

We mentioned earlier that one of the goals of the QuickBlocks™ system was to provide a methodology to verify the accounting of the blockchain. We mentioned that some blockchains maintain two versions of an account's balance: (1) the balances stored in the state, and (2) the balances as represented implicitly by the list of transactions.

Because QuickBlocks™ is able to build an accurate list of transactions, and thereby duplicate the blockchain's accounting, we can verify that the list of transactions does indeed reflect to the account balances at the end of each block.

This has the following implication: If we find that the QuickBlocks™ list of transactions (which is assumed to be full and accurate), does not agree with the blockchain balances, we've likely identified an error in the QuickBlocks™ code. If, on the other hand, and upon further investigation, we discover that the QuickBlocks™ accounting is correct, we've identified (however unlikely) an error in the blockchain accounting.

QuickBlocks™ and the blockchain must agree.

We extend this idea further to digital assets, such as ERC20 tokens. Here, the accounting provided by the blockchain node is not inherent to the node itself. Instead, the developer of the token contract has written the accounting system.

Anyone familiar with the field is aware that there have been numerous bugs and/or mistakes either in the security of a digital assets or in the accounting for a digital asset.

If the smart contract implementing the asset contains an error, and it is being monitored, there are opportunities to identify errors before they cause more damage than is necessary. If the contract is not being monitored, no-one will notice, and a potential error will go unnoticed as it did with The DAO.

While we at QuickBlocks™ are still investigating the effect of The DAO Hack had on The DAO's token balances, it is our belief that the relationship between the tokens and the ether they represented went out of kilter on the first block after the hack.

At the end of the first block after the hack, the ether in the smart contract was lower than it should have been, while the token count was correct. A per-block reconciliation of the ratio of tokens to ether may have made this obvious.

If one can identify such an imbalance between the token balance in a smart contract and the ether balance it represents, one may have an opportunity to forestall a larger loss than would otherwise occur.

Per Block Interactive Transaction Debugger

Given the ability of QuickBlocks™ to instantaneously reconcile each block with the blockchain, we've been able to build an interactive per-block debugging tool.

This software, which is created automatically by `chifra`, traverses through the account's list of transactions, doing a full reconciliation at each block. It comes to a halt and enters debugging mode if there is a discrepancy between the expected balance (according to QuickBlocks™) and the account balances as retrieved from the node.

As of this writing, we are able to fully account for the ether. We have already embarked on a similar functionality for accounting for digital assets such as tokens.

The various functionality provided by the debugger is presented in Appendix A-8.

Decentralized Blockchain Explorer

A distinguishing characteristic of our work is that it remains entirely decentralized. This is simpatico with the philosophy behind the blockchain. Unlike many blockchain explorers, QuickBlocks™ resides locally, running on the same computer as the blockchain node itself.

In addition to running locally, QuickBlocks™ parses and delivers the data in a format that is familiar to the smart contract developer. Instead of the hexadecimal, difficult-to-understand, "ugly" data from the RPC, QuickBlocks™ delivers easily digested data already translated into the language of the smart contract.

Due to many factors including optimizations to the stored data, and because QuickBlocks™ runs locally to the machine and does not require a transport of data across the Internet, our data delivery is furthermore significantly faster than other methods of data retrieval we've seen.

Database and Test Case Generation

We anticipate smart contract developers using our software to populate traditional web 2.0 databases and thereby driving much richer and more meaningful web-based user interfaces for their dApps.

At the same time, the transactional history of the smart contract can be used to generate live-recording test cases that may be used on a testing environment to study aspects of a smart contract system such as stress testing and gas accounting.

Furthermore, because the monitor system is, after all, C++ source code and represents

the exact functional and event interface of the smart contract system, it is conceivable that a testing engineer could write arbitrarily complex test data streams to purposefully attack or stress specific parts of the smart contract system.

Generation of Transactional Data for Use in Accounting and Auditing Systems

The smart contract monitoring system can generate data to any format. One envisioned use case is to generate data in specific formats for use by non-technical users such as accountants and audit personnel. For example, generation of the transactional history of a smart contract system could easily be fed into an accounting package such as QuickBooks® through its already existing import format specification.

Tokenomics

We define the field of Tokenomics to be the accounting for and analysis of ERC20 (or related) smart contract tokens or other digital assets. This might involve accounting for ICO crowd sales including analysis of lost opportunity costs due to oversubscription.

In addition, for the first time, we are able to build what would be effectively off-chain capitalization tables for the entity whose ownership is represented by the token. This has not been possible previously.

"Bridging Code" for Real World Interactions

A smart contract runs entirely in the context of its virtual machine. In this sense, there is no "outside world." Neither is there a way to get data into a smart contract (other than an external transaction).

A smart contract monitor, such as those provided by QuickBlocks™, is able to operate as a bridge to the real world. While watching the smart contract, and in response to certain events or transactions, a monitor may initiate an off-chain process. QuickBlocks™ monitors allow smart contracts to interact with the world.

Say, for example, the smart contract accepts ether in return for opening a bicycle lock. One could implement a fully autonomous bike sharing system with such a monitor. The smart contract controls the money, while the smart contract monitor unlocks and re-locks the bicycle.

Spot Pricing of Transactions

With a simple modification of the data, QuickBooks™ may price transactions in whatever currency is of interest to the end user. This modification involves storing a single floating-point number representing the then-current price of the underlying crypto-currency at that moment. This information may be collected using any available method, perhaps by averaging multiple different pricing data at each block. This would be the basis for tax reporting software.

It would not be necessary to retrieve and store the then-current prices of the various digital assets built on top of a particular blockchain (unless those assets are inter-operable). At the time of purchase, the then-current value of the crypto-currency is representative of the cost-basis price of the asset. At the time of the sale, the then-current value of the cryptocurrency is representative of the value of the token. It is a fact that only the value of the underlying crypto-currency of the asset matters when calculating net gain or loss.

There is an opportunity to provide a service to end users wherein all debits and credits into a particular token can be calculated, and for the first time, an automated best pricing method for the transfer of assets can be calculate to minimize tax implications.

Method for Storing Minimal Data per User and yet Providing Maximum Availability via IPFS

One of our potential use cases is data delivery, not only to smart contract

developers but also to any other user who may be interested in the data.

Because we wish to remain fully decentralized, the traditional method of sharing data (i.e. through a centralized web-based API service) is inadequate.

We therefore propose to store fully parsed and improved data on a distributed file system such as IPFS. Distributed file systems allow for a minimum impact on a single machine (because only the blocks of interest need to be stored on the hard drive), while at the same time, the entire binary cache is "conceptually" available to all participants.

It is presumed that the transfer of an already improved binary block file will be lower than fully re-processing the block on each individual machine.

This minimal imposition of storage costs while at the same time infinite availability of the data may allow us to provide a fully decentralized data delivery mechanism.

Other Ideas and Closing Remarks

In the `blockScrape` application, we insure the accuracy of the data we've stored in our own proprietary binary format by retrieving it into memory, rendering it to a minimal length string and compare the same to the data retrieved and processed from the block. Any difference indicates an error. In this way, we insure accurate data handling.

In the `blockScrape` application, we store only blocks with one or more transactions in the binary cache. At the same time, we store an array of 64-bit integers indicating the block numbers of those blocks that contain at least one transaction. Because we are writing transactional accounting software, there is no need to store blocks that contain no transactions. This lessens, by about 35%, the amount of data stored and the time needed to traverse the entire blockchain.

For the monitoring software, we initially store no binary blocks in the cache. We later only store a block if the block contains transactions of interest to the monitor. This significantly lowers, by orders of magnitude, the amount of binary data we need to store on the target machine's hard drive.

By further storing the data in a distributed file system such as IPFS, we are able to lessen the amount of processing needed to create each binary block as, if already present somewhere in the distributed system, the block can be retrieved directly.

The `chifra` program allows us to accept a list of blockchain addresses, and from there retrieve ABI files, from which we generate the entire parsing library and other parts of the monitor software from a template.

For the `ethPrice` application, we acquire price data from multiple sources and store US dollar prices in separate binary files (for later reference) along with an average price attached at each block.

For the `grabABI` program, we reach out to multiple data sources looking for ABI files from which to build a monitor program. If an ABI file is found, we cache it locally and generate a list of canonical function interfaces and their associated encodings. This generates a list of 4-byte function signatures and 32-byte event signatures.

When a monitor program needs to 'decode' a transaction's *input* data field, we use the function signature to (a) determine which function the *input* represents, and (b) parse out the remaining data from the *input* and assign it to each of the class's fields.

In the monitor applications, while presenting records from a monitored set of addresses, the application may enter into a debugging mode that allows the user to view the transaction in whole by presenting the detailed transaction, the detailed trace, the detailed logs, the bloom filter as it tests itself against each monitored address.

At each block, the debugger's accounting system reconciles to the address's beginning balance, the total value of inputs and outputs per block and the ending balance. At the end of each block, the debugger asks the node for its balances.

If there is a discrepancy, the debugger enters into single step mode. Here the end-user may save a record of the mismatch for later investigation or simply correct the error interactively and move on to the next block. The debugger also is able to display each monitored address with human-readable names instead of hexadecimal addresses.

Monitor applications operate on a configuration file which includes (a) toggle flags for turning on and off various options of the program and (b) display specifications for displaying records from the cache. Each time the monitor runs, it first freshens its local cache using `blockScrape` and then adds to the list of transactions per account.

Since many smart contracts are token based, QuickBlocks™ pre-generates two fixed libraries called `tokenlib` and `walletlib` which handle the industry standard ERC20 smart contact functions and events, in addition to the most widely used multi-sig wallet functions and events. If the `grabABI` function cannot find an account's ABI file, the system automatically falls back to these standard libraries to process the transaction if possible.

Conclusion

We believe there is a missing component in the current blockchain ecosystem. This is not surprising as the same person who wrote the first C++ implementation of the blockchain also wrote the first implementation of the RPC interface and furthermore wrote the first implementation of the web3.0 JavaScript interface.

This genius developer (calling Dr. Wood anything short of a genius seems unwise) foisted the task of accumulating per-account lists of transactions and parsing those transactions on the wrong part of the system—dApp developers.

This missing piece of the ecosystem is exactly the piece that QuickBlocks™ provides—that of a data translation layer.

Very few people are interested in blocks of intermingled transactions for arbitrary lists of accounts. Most people, in our opinion, including system architects, developers, testers, bookkeepers, accountants, auditors, regulators, and even individuals are interested in particular transactions cast against their own list of interesting accounts. This is what QuickBlocks™ provides…fully parsed…with minimal impact on the machine…fully decentralized…and fast ⊡

REFERENCES

[1] Wood, Gavin, "Ethereum: A Secured Ledger; EIP-150 Revision (8bb760b - 2017-01-19)" accessed 2017-01-28 at https://ethereum. github.io/yellowpaper /paper.pdf.

[2] Buterin, Vitalik, "A Next-Generation Smart Contract and Decentralized Application Platform" accessed 2017-01-28 at https://github.com/ethereum/wiki/wiki /White-Paper

[3] Rush, Thomas Jay, "Adaptive Enhanced Bloom Filters for Identifying Transactions of Interest"; July 2017.

[4] Rush, Thomas Jay, "Faster, Richer, Fully Customizable Data from Programmable Blockchains"; January 2017.

[5] Wikipedia. n.d. Bloom filter. Accessed 07 01, 2017. https://en.wikipedia.org/wiki/ Bloom_filter.

Appendix A.1 – The CBlock Data Structure

CBlock		
	Field Name	**Description**
	parentHash *32-byte hash*	A reference to the previous block in the blockchain. This value is identical to the previous block's hash. By virtue of including this hash in the current block's data prior to generating the current block's own hash, the blockchain insures the immutability of the data.
	hash *32-byte hash*	A cryptographic hash of the data contained in the current block, excluding, of course, the hash itself, but including the hash of the parent.
	blockNumber *unsigned*	The sequentially-ordered number of the given block starting from the 'zero' or 'genesis' block, and including no gaps even if the block is empty.
	timestamp *unsigned*	The epoch-based timestamp of the block. All transactions in a given block are assigned the same timestamp as the block itself.
	logsBloom *bloom*	An accumulation of the logsBloom data, if any, from the transaction's receipts. Note, in QuickBlocks™, this blockchain data has been removed and replaced with an Adaptive, Enhanced Bloom filter as described in [4].
	transactions *array of CTransaction*	An arbitrarily long, possibly empty, list of externally initiated and consented to transactions including those transactions that may have ended in error.

Appendix A.2 – The CTransaction Data Structure

CTransaction		
	Field Name	**Description**
	hash *32-byte hash*	A cryptographic hash of the data contained in this transaction, excluding, of course, the hash itself.
	blockHash *32-byte hash*	A reference to the hash of the block in which this transaction is included.
	blockNumber *unsigned*	A reference to the blockNumber of the block in which this transaction is included.
	timestamp *unsigned*	A reference to the timestamp of the block.
	transactionIndex *unsigned*	A zero-based sequential index of this transaction's location in the block's *transactions* array.
	from *20-byte address*	The *external* account that initiated this transaction. Note that this account, in current implementations, pays for the gas consumed by the transaction. The account is represented by a 20-byte public address.
	to *20-byte address*	The destination or receiving address of the transaction. This account may be either another external account or a smart contract account.
	value *big unsigned*	The value, denominated in the blockchain's base denomination, sent with this transaction. This value may be zero, and it is exclusive of gas costs.

CTransaction		
	Field Name	**Description**
	gasAllowed *big unsigned*	This field indicates a user-specified maximum amount of gas to be consumed by the transaction. This value is set prior to the initiation of the transaction. If the amount of gas consumed by the transaction during its processing is strictly larger than the amount specified here, the transaction fails. Furthermore, if a transaction fails for any other reason, for example, an incorrectly specified input value, all remaining gas is consumed. The calculated amount gasCost $$gasCost = gas * gasPrice$$ is needed to fully account for a transaction. gasCost is exclusive value above. gasCost is extracted from the sending (i.e., the from) account regardless of the error status of the transaction.
	gasPrice *big unsigned*	When a transaction is completed the amount of gas consumed during the execution of the transaction is multiplied by gasPrice and the resulting amount (gasCost) is extracted from the sending (i.e., the from) account and paid to the block's winning miner.
	input *bytes*	This arbitrarily long field may contain any data including no data at all. The owner of the sending account may, for example, use this field to send hexadecimal encoded messages to the owner of the receiving account. More likely, the input data field is used to carry information necessary to specify the called function of a smart contract and its associated parameters as specified in the function's ABI interface. The first four bytes of this field, if the recipient address is a smart contract and the transaction is a function call, contain the signature of the intended called function. The remaining data in this field contains that function's parameter data. In the special case when the to field is 0x0, the input data field carries the byte code of a compiled smart contract. The system, in this case, attempts to deploy the smart contract to the blockchain. On success, the location of the new smart contract, is returned in the transaction receipt's contractAddress field.

CTransaction		
	Field Name	**Description**
	isError *bool*	This calculated field is retrieved from the transaction's receipt as described below.
	isInternalTx *bool*	This calculated field is retrieved from the transaction's trace as described below.
	receipt *CReceipt*	Each transaction is assigned a receipt which must be retrieved separately. The receipt data structure is described below.

Appendix A-3 – The CReceipt Data Structure

CReceipt		
	Field Name	**Description**
	contractAddress *address*	If an externally generated transaction is sent to address 0x0, and the input data field of the transaction must contain valid byte code, and in this case a new smart contract is created. The address of that newly created contract is returned in this field of the transaction's receipt.
	gasUsed *big unsigned*	The total amount of gas consumed during the processing of the transaction is returned here. Note, in the case of an in-error calculation and subsequent "throw" from a smart contract, the gasUsed field and the gasAllowed field will be equal. This is because a "throw" consumes all remaining gas. The two values will be equal in the case when the transaction's processing would have consumed more gas than was allowed. Many software implementations interested in building per-address lists of transactions mistakenly think that a simple test of gasLimit vs. gasUsed, if equal, indicates an error. This is not the case. In order to fully understand the discussion below, one must realize that for many cases where gasLimit exactly equals gasUsed, the transaction did not end in error. This is the case, for example, in a simple value transfer from one external account to another. Many blockchain wallets send exactly 21,000 gas in such a case which is the identical to the cost of such a transfer. This makes searching for in-error transactions more difficult.

CReceipt		
	Field Name	**Description**
	logsBloom *bloom filter*	A transaction's receipt holds an arbitrarily long list of log entries (i.e. generated events). Log entries are returned in the receipt's logs field. Various fields in the log entries represent either account addresses involved in the transaction or event signatures. Through a rather simple calculation, three particular bits of a 4096 bit long binary array of bits are twiddled (set to 'true') indicating the possible presence (or, if the bit is set to 'false,' the absence) of that address in the logs. Because this filter may report false positives, it is not unequivocally true that a particular address was involved in the transaction. The logsBloom may be used to eliminate an account from consideration, however.
	logs *CLogEntry array*	The logs list includes an arbitrarily long list of event `CLogEntry` data structures, described below.

Appendix A.4 – The CLog Data Structure

CLogEntry		
	Field Name	**Description**
	address *20-byte address*	Only smart contracts may emit log entries. This address value represents the address of the smart contract that emitted the event represented by this log entry.
	logIndex *unsigned*	The index of the given log entry in the list of logs numbered sequentially starting from 'zero'.
	topics *big unsigned array*	Topics are between zero and four 32-byte entries representing 'indexed' fields in the event signature as specified in the source code. Indexing on these fields may improve search performance while filtering on these items as they are inserted into the transaction's logsBloom.
	data *bytes*	An arbitrarily long collection of data (packed to the nearest 32-byte boundary) representing the non-indexed fields in an event's signature.

Appendix A.5 – The CTrace Data Structure

CTrace		
	Field Name	**Description**
	blockHash *32-byte hash*	A reference to the blockHash of the block containing this trace.
	blockNumber *unsigned*	A reference to the blockNumber of the block containing this trace.
	subtraces *unsigned*	Smart contracts are able to call into other smart contracts which may, in turn, call into still other smart contracts ad-infinitum to the block-level gas limit imposed by the node. The node represents this potentially 'tree-like' structure as a flat list of indexes into traces. This field represents the number of times this level of the trace called into other smart contracts. If, during the operation of a particular external transaction, four internal message calls into four different smart contracts were made, this value would be '4'. Any code that is reading the traces (i.e. QuickBlocks™) must rebuild the tree-like structure if needed. Luckily, for our purposes we are only interested in the list of accounts involved in the transaction, therefore we can safely ignore this parameter.
	traceAddress *list of addresses*	If the blockchain node stored the traces of smart contract calls in a proper tree-like structure this field would be unnecessary. It represents, as a flat string, the zero-based position of the current trace in the call tree. As an example, without further explanation, the third call of the fifth call of the first call of a three-level deep trace tree would be represented as [0] [4] [2] (Trace indexes are zero based.) Because QuickBlocks™ is only interested in the addresses involved in a transaction, we can safely ignore this field.

CTrace		
	Field Name	**Description**
	transactionHash *32-byte hash*	A reference to the transactionHash of the transaction containing this trace.
	transactionPosition *unsigned*	A reference to the transactionIndex containing this trace.
	type *string*	One of "call," "create," or "suicide." In the case of "create" and "suicide" the callType field of the trace action is empty.
	action *CTraceAction*	A data structure containing, in effect, the to, from and input parameters, as per the type, into an internal transaction. Described below.
	result *CTraceResult*	A data structure containing, in effect, the output of the internal transactions as per the type above. Described below.

Appendix A.6 – The CTraceAction Data Structure

CTraceAction		
	Field Name	**Description**
	callType *string*	If the value of type in the containing CTrace is other than "create" or "suicide," this value indicates the type of call. This field takes on values from "call," "callcode," "delegatecall," or [other]?
	from *20-byte address*	The address of the calling smart contract, nearly identical in meaning to the from parameter in the transaction object.
	to *20-byte address*	The address of the smart contract being called, nearly identical in meaning to the to field in the external transaction object.
	input *bytes*	The input data for the current call with identical meaning to the input data described in the external transaction.
	value *big unsigned*	The value of the current call with identical meaning to the value data described in the external transaction.
	gas *big unsigned*	The maximum gas allowed for the current call with identical meaning to the gas data described in the external transaction.

Appendix A.7 – The CTraceResult Data Structure

CTraceResult		
	Field Name	**Description**
	gasUsed *big unsinged*	The amount of gas used during the execution of this internal contract call.
	output *bytes*	The return value of the internal call, if any.

Appendix A.8 – Command Structure of Interactive Blockchain Transaction Debugger

Commands available from smart contract transactional debugger		
	Command	**Meaning**
	(c)orrect	Correct the current imbalance and continue to the next imbalance (if any)
	(a)utocorrect	Toggle autocorrect (allows pressing enter to correct)
	(p)lay	Play a block.txid.traceid or '+' for next, '-' for prev
	(t)race	Retrace the previous transaction (enter 't' plus tab to scroll through recent transactions from the buffer)
	(b)uffer	Show the transaction buffer (including transaction hashes)
	(e)thscan	Open a block, account, or transaction in http://ethscan.io
	(s)ource	View a smart contract's source code (if found)
	(l)ist	Show the list of accounts being debugged
	confi(g)	Edit the monitor's config file
	(v)erbose	Toggle auto trace
	clea(r)	Clear the screen
	(q)uit	Quit the current monitor program
	(h)elp	Display this screen