Faster, Richer, Fully Customizable Data from Programmable Blockchains

Thomas Jay Rush
http://quickblocks.io
January 28, 2017
Revised: July 24, 2017

A software system, QuickBlocks™, is described that provides user-focused, speed-optimized, customizable per-smart-contract data from any blockchain, including public, consortia, and private chains. Through a collection of software libraries, applications, and automatically-generated source code, the system improves the quality and accessibility of blockchain data to programmers and end users. Given this improved accessibility, many previously unanticipated functionalities, such as fast delivery of smart-contract specific JSON data from RPC, detailed gas-usage analysis, live debugging and stress testing from previously recorded blockchain interactions, smart-contract control panels, and user-local, data-rich, fully-decentralized desktop and mobile applications become possible. **Keywords: blockchain, Ethereum, data analytics, finTech**

Introduction

Distributed. Consensus-driven. Immutable. Ledger of transactions.

These four phrases, when combined into a technology called the blockchain, hold great promise to alter the future of computing and perhaps the world.

In the context of a blockchain, the phrase, "ledger of transactions" means an accounting ledger or business information system, each transaction in the ledger consisting of a spender, a recipient, a timestamp, and a value. The word "distributed" in this context means that each participant maintains his/her own duplicate copy of the ledger[1]. The phrase "consensus-driven" refers to the fact that a majority of participants, prior to writing to the ledger, must agree on what will be written[2]. "Immutable" refers to the fact that once the ledger entry is consented to, it is nearly impossible for a single participant (or group of participants) to alter the ledger.

Consider how the world will change, when humans possess a shared, unalterable history of our own actions.

At the heart of this amazing system is data, and one of the great promises of the blockchain—if it can be realized—is that each participant has access to their individual data. It is our belief, however, that this data is not as easily accessible as it should be, especially for the less technical among us. Nor is the data presented in as rich a format or with as deep a content as it might be. Nor is it available, in its current implementations, at speed.

Our project aims to remedy each of these shortcomings.

[1] That is, the participant's software (called a node or client) maintains the ledger.

[2] Ignoring for a moment possible forks, which is discussed below.

The primary component of a blockchain is the node or client. A blockchain client is a piece of networking software that runs identically and simultaneously on many computers at the same time. Nodes continually broadcast transactions to other nodes on the network and listen for transactions from other nodes. Competing with each other to be the first to identify a suitably difficult-to-find stochastically-generated solution to a cryptographic puzzle, the winning node constructs a block (using a recent collection of transactions) and, once consensus is reached, is rewarded with a newly created 'coin' or 'coins' of then-current value.

Additionally, the winner of the block receives the accumulated transaction costs of the approved transactions. (These costs are called 'gas' below.)

It is this potential return on investment of a node's computing resources (block reward + gas) that incentivizes participants to both continue to participate and participate honestly. A dishonest action is assumed to lessen the value of any previously accumulated rewards, and therefore dishonesty becomes increasingly less likely as the value of the 'coin' increases.

In addition to providing "accounting services" in the form of block creation, each node provides an interface to its own copy of the blockchain data. This interface is provided either through RPC (remote procedure calls) or IPC (inter-process communication), each of which allow other software components to retrieve data from the ledger.

It is our opinion that these interfaces, in their current manifestation, are inadequate.

The RPC and IPC interfaces[3] expose the blockchain's data at a level that, we believe, is too close to the internal workings of the blockchain.

[3] Throughout the remainder of this document, we refer only to the RPC interface. The reader should consider that in all cases, IPC, could be used as well.

This makes it difficult for users of the system to effectively process the received data. The RPC interface furthermore delivers this inadequate data in a piece-meal fashion. The meaning of particular portions of the data is dependent on the contents of other portions, requiring multiple calls through the interface to fully determine the validity and meaning of each transaction.

The node's communication interfaces provide functionality for retrieving blocks, transactions, receipts, traces, account balances, and other highly-specific data such as mining information, block and transaction hashes, and, importantly, the ability to create, sign, and initiate transactions. These latter functionalities are not of interest to our system. Our system is concerned only with retrieving blocks, transactions, receipts, traces, and logs.

It is our belief that most users of the blockchain data, ranging from systems architects, software developers, all the way down to end users with individual accounts, are interested not in blockchain specific formats, but in data customized and optimized for their particular use.

In other words, regular human beings are interested in their own account data, not in blocks, not in hashes, and certainly not in mining data. This interest extends to both participants in, and purveyors of, smart contracts.

This document describes QuickBlocks™, our system for providing customized, per-account access to a richer and more useful set of blockchain data at speed.

A Quick Note on Blockchains

While the development of our code has been heavily focused on the public Ethereum blockchain [1][2], we believe, given the similarity of all blockchains, that providing improved speed and data quality for other blockchains will be a relatively simple extension. While what we discuss below is

Ethereum-centric, the reader should be aware that our code is intended to work with any blockchain, both public and private.

Architectural Overview

QuickBlocks™ consists of three primary types of components: libraries, tools, and applications.

Libraries

The first type of component are software libraries which organize reusable code by functionality. QuickBlocks™ contains six pre-compiled libraries (`utillib`, `abilib`, `etherlib`, `acctlib`, `tokenlib`, and `walletlib`) and an arbitrary number of automatically-generated customizable, per-smart-contact libraries discussed below.

The most basic functionality is contained in a library called `utillib`. This library consists of software code for carrying out common functions such as string and time manipulation; concurrency-protected data access; container classes such as lists, arrays and maps; and other utility functionality.

The second library component is called `abilib`. This library allows for reading, writing, and manipulation of application binary interface (ABI) files. ABI files, produced by the Solidity compiler, contain the information necessary to describe interfaces to Ethereum smart contracts. It is this library that allows QuickBlocks™ to automatically generate customized per-smart-contract libraries.

The third library component of QuickBlocks™ is called `etherlib`. This library mirrors the blocks, transactions, receipts, traces, and accounts found in the blockchain data. It is in the `etherlib` that we interact directly with the blockchain via RPC. We do this in order to collect raw blockchain data, which is then enhanced so as to provide more useful data to higher-level components such as the `tokenlib`; the customized, per-smart-contract libraries; and the various applications.

It is the job of `etherlib` to provide faster access to the data, and many speed optimizations, in addition to a collection of easy-to-use interfaces for traversing the blocks, transactions, and accounts, are contained in this library.

A small library called `acctlib` provides the ability to store per account data in a fast-insert, fast-search 16-way tree data structure keyed by account address. This library is currently in experimental status.

The final two pre-compiled libraries are called `tokenlib` and `walletlib`, which implement support for the Ethereum ERC 20 token standard and popular multi-sig wallets. These libraries are described further below.

Built upon these six libraries are automatically-generated, customizable, per-smart-contract libraries. The automatic generation of C++ code is accomplished with two co-operating applications called `grabABI` and `makeClass`, each described further below.

Customized per-smart-contract libraries are collections of C++ code, along with an automatically generated make-based build system, that allows QuickBlocks™ to "promote" generic blockchain transactions to smart-contract specific C++ classes aware of and capable of exporting their own data.

Additionally, these per-smart-contract specific C++ classes allow the programmer to make a smart contract's functional transactions and events available to other software components such as automated testing, analytics, or debugging applications.

An example of this functionality might be the delivery of the token standard *transferFrom(from, to, amount)* function, not as a generic RPC transaction, wherein the parameters of the function call are "trapped" inside a difficult to understand, and even more difficult to parse, input data field (i.e. 0xa9059cbb000000000000000000000000de 56d176c07a3b3776dfba86cbf…), but as fully parsed JSON data exposing the values directly of the *from*, *to*, and *amount* parameters.

`tokenlib`, which abstracts the Ethereum token standard, extracts the data and stores it in the `CTransfer` C++ class. This class, by virtue of being derived from the underlying classes in the `etherlib` and `utillib` libraries, possesses the ability to both serialize itself for later retrieval and generate rich JSON data fully produced on the fly.

Tools

Mostly as a method by which to test the various libraries, we've created a series of tools designed to give direct access to each of the various data structures found in the Ethereum blockchain.

For example, we've written tools to retrieve data at the block level, the transaction level, the transaction receipt level, the transaction log level, and even as deeply down into the guts of the blockchain to retrieve the transaction trace data.

At each level, the end user may include all levels below that level as well. For example, the user may request to include all traces in the transaction request.

We propose to release the libraries and the various tools to open source. Tools are described more fully below.

Applications

In addition to library code and tools, the QuickBlocks™ system delivers a collection of applications. Each of the applications is described in its own section below. Applications pull together various functionality provided by the libraries to accomplish particular tasks.

For example, one application, called `blockScrape`, distinguishes between blocks that contain one or more transactions and blocks that do not. Another application, called `miniBlocks`, separates user-focused portions of the data (such as *from*, *to*, *amount*) from blockchain-focused parts of the data (such as block and transaction hashes, mining rewards, etc.). By partitioning the data into user-focused and non-user-focused portions and eliminating blocks that contain no transactions (and therefore are not of interest to some applications), the system is able to greatly increase the speed with which it can analyze, deliver, and interact with the data.

In the following sections, we first discuss two of the primary, building-block applications of our system called `makeClass` and `grabABI`. After this, we discuss each individual library component. Thereafter, we discuss the various tools and application components. Following this we discuss a few potential future use cases that we've envisioned. We conclude our paper by speculating on a few unexpected implications of our work and a brief discussion of future issues of concern.

makeClass and grabABI

Before discussing the various libraries and applications, we discuss two particular applications that are central to operation of the system. Without these applications, creation of much of the QuickBlocks™ code would not have been possible. Neither would the creation of per-smart-contract parsing libraries from ABI—one of the unique ideas of the system—be possible.

`makeClass` is a code generation tool of our own devising. Given two template files (one for the C++ header file and one for the C++ implementation file[4]) and a configuration file described in Appendix A, `makeClass` is able to generate fully functional software code that supports dynamic self-creation; serialization; file import from common formats including JSON; and importantly the ability to export arbitrary data formats including JSON, tab-delimited text, and comma-separated values. Furthermore, we've

[4] Nothing precludes makeClass from generating code for other programming languages. One would simply have to create template files for that language.

recently implemented direct injection of the output of our system into arbitrary databases.

All three of the classes in the abilib (CAbi, CFunction, CParameter) were created with makeClass, as were CBlock, CTransaction, CReceipt, CLogEntry, CTrace, CStructLog, CPriceQuote, CAccount, and CRPCResult in etherlib.

The entirety of the tokenlib library was created automatically using makeClass and grabABI. For this library, unlike the classes mentioned above, all of which are derived from CBaseNode, six of the nine classes in tokenlib (CDefaultFunc, CTransfer, CTransferFrom, CApprove, CCreate-TokenProxy, CUnknown) are derived from the etherlib class CTransaction, while three of the classes are derived from the etherlib class CLogEntry (CApproval-Event, CTransferEvent, CCreated-TokenEvent, CMintEvent).

This ability to derive classes from base classes is what allows us to "promote" the blockchain data into full fledged self-aware classes in their own right. Classes that can export fully customized data given generic transactions and events on the blockchain. This capability is at the core of our idea.

All classes in each per-smart-contract library are created automatically by makeClass. In order to generate these classes, we use another of our applications called grabABI and its --generate function.

grabABI reads the ABI file for a particular smart contract either from its local cache (because it's been encountered before) or from a web API such as that provided by http://etherscan.io.

Given an ABI file, grabABI, which normally simply prints the ABI to the screen, generates—not the code for the per-smart-contract libraries—but the configuration files for the makeClass application which subsequently generates the code.

This additional level of abstraction (i.e. code generation directly from a smart contract's ABI) is further to the very heart of our idea.

The code generated by makeClass, includes hooks and extensibility provisions we call //EXISTING_CODE, which may be used to surround existing programmer-customized code inside each automatically generated file. This allows the programmer to both automatically generate the majority of the code for a particular class and at the same time customize the code to whatever extent is deemed necessary for his/her application.

Next, we describe in greater detail each of the three major groups of work: libraries, tools, and applications.

Libraries

The following section describes, in detail, each of the six library components and introduces the idea of per-smart-contract customized libraries. Per-smart-contract libraries are further explored in the application section of the paper under the grabABI sub-section, particularly as it relates to the --generate option.

Utillib Library

We call the first of the six libraries utillib. The utillib library provides support routines that one will find in most utility libraries, such as support for basic data types including unsigned and signed integers, floats and doubles; big number support; time and date support; support for manipulating, searching, and processing strings; various collection types such as arrays, lists, and maps; runtime typing; serialization; and a proprietary technology known as display strings (as described in Appendix B).

Additionally, support for concurrency protected file access; screen, file, and string export and import functionality; various operating-system specific file and folder manipulations; configuration file support; command line parsing and options support, and finally performance timing support is provided.

Many of these functions are now available in the standard C++ libraries, and therefore much of the code in `utillib` will, over time, be removed. Primary among this soon-to-be removed code is our proprietary `SFString` and collection classes, which, while capable, are sub-optimal.

If a user of the QuickBlocks™ system wishes to replace the `utillib`, he or she may, but certain features would be required. These are features not provided in the standard C++ libraries that are needed to support applications and higher-level libraries built on top of `utillib`. This includes class hierarchy support (auto-creation and identification), serialization support, and display strings.

Class hierarchy support makes available a number of features including the ability to create instances of a class given a string input parameter. This feature makes possible the promotion of a blockchain's transactions to the various token and per-smart-contract classes.

Serialization allows for fast, binary storage and retrieval of the class data. This, along with significant pre-computations, is the primary way we speed up data access.

Display strings allow our software to export the state of a particular instance of a class in any desired format given an appropriately constructed formatting string. These formatting strings may represent tab-delimited text, comma-separated values, JSON, or any other desired format including the ability to inject the data directly into an arbitrary database. Display string technology is described in Appendix B.

Abilib Library

The `abilib` library provides capabilities for reading, writing, and manipulating application binary interface files which are one of the outputs of the compilation and deployment stage of an Ethereum smart contract. These files list each smart-contract interface function and all events that may be emitted from a smart contract. Both public and private, constant and non-constant function calls are included in the ABI, however because only non-constant, public functions place transactions on the blockchain, we concern ourselves only with that subset of functions. In future versions, we intend to support all functions from the ABI. All events found in the ABI are of interest.

Three classes comprise the `abilib` library: `CAbi`, `CFunction`, and `CParameter`.

The primary class in the library is `CAbi` which carries the list of functions and events found in the smart contract's interface. We choose to record this list of functions and events twice, once sorted by name, and a second list sorted by the item's encoding.

In this way, we allow ourselves much faster access to the data than would be possible otherwise. In some parts of the processing, we need to find an encoding given a name, in others we need to find a name given an encoding; therefore, we store the list twice, sorted respectively by name or encoding.

A second C++ class, stored as the above mentioned two lists, is the `CFunction` class. `CFunction` stores both the functions and the events of a smart contract. `CFunctions` store lists of `CParameters` which are tuples of field types, field names and other attributes.

In this way, the `CAbi` class is able to represent a full description of the interface of any smart contract.

Each of these three classes in the `abilib` library was generated automatically from configuration files using the `makeClass` application as detailed in Appendix A.

By nature of having been derived from the `CBaseNode` class, each of these classes implement the ability to serialize and deserialize themselves, display themselves using display strings, and import and export themselves to various file formats.

In Appendix C, we've included the configuration files for the `abilib` classes, as well as the definition files for all of the automatically generated code in the `etherlib` and `tokenlib` libraries. Because most of the code is automatically generated in these libraries, we show only the configuration files in the appendix.

`abilib` is the primary component of two significant portions of our system: (1) the ABI data for a smart contract is used when parsing the input data field of a blockchain transaction, thereby associating the appropriate function call to the transaction awaiting further use. It does this by providing the parameter lists to the function as found in the ABI in an easy to use format. Additionally, event encoding and parameter data is parsed using the similar functionality.

It is this ability to parse a transaction's input data and event logs that allows us to promote generic transactions to per-smart-contract classes as mentioned above. This realizes the second major function of the `abilib` library.

Etherlib Library

`etherlib` is at the heart of the entire system and consists of four primary functionalities: (1) communication with the blockchain node, (2) mirroring and caching the received blockchain data, (3) parsing of both transactional input data and receipt logs addresses and topics (it is this function that underpins the `tokenlib` and per-smart-contract libraries), (4) a series of convenience functions for traversing all or part of the blocks and the transactions and accounts encountered while traversing the blocks. A final functionality is the acquisition and application of the currency's price data.

Communication with the node

Using either the RPC via the `curl` libraries or inter-process communication, the `etherlib` communicates with the node using functions similarly named to those found in the RPC. For example, functions `getBlock`, `getTrans`, `getReceipt`, traceTransaction, `lastestBlock`, and getBalance mirror names from the node's RPC.

Many of the interfaces provided by the RPC for retrieving data are mimicked by `etherlib`. We do not provide interfaces for communicating back to the node to do things such as sign or initiate transactions. We feel that this type of interaction is best handled by secured wallet software or some other means where there has been a much heavier focus on security. While having done everything we can to make our code secure, our system is dealing only with immutable data extraction from the blockchain.

The reader is invited to read about using the node's RPC in documentation to be found outside of this white paper [3][4].

Mirroring of blockchain data

The mirroring (or caching) of the blockchain data is provided by six classes: `CBlock`, `CTransaction`, `CReceipt`, `CLogEntry`, `CTrace`, and `CStructLog`. Each of these classes is documented in Appendix D.

As has been discussed earlier, the various fields in the `CBlock` class include a list of transactions. The `CTransaction` class, the most used class in the system, contains the *from*, *to*, *value*, and *timestamp* values expected of any transaction in addition to a transaction *receipt*. The `CReceipt` class contains, importantly, *gasUsed* and the *logs* and *traces* necessary to determine the outcome of a transaction.

Parsing of transaction input and event topics

Given an ABI for a smart contract, we first ensure that the function and event signatures are canonical. Given the canonical signatures we then proceed to encode these signatures so they may be used later to decode the input data fields of each transaction and the indexed topics and data of each receipt log.

This involves a two-step process of first converting the Ascii representation of the canonical signature to hexadecimal and then using RPC's sha3 function to encrypt the signature. For smart contract functions found in the ABI, only the first four bytes (eight hexadecimal characters of the resulting sha3) are used as the encoding. For event in the ABI, the entire 32 bytes (64 hexadecimal characters) are used.

Armed with the encoded input and event signatures, the system uses the ABI to decode the input data field. It does this by reading the type of each parameter to a function or event and extracting that data type from the input field (in the case of a transaction) or in the indexed topics (in the case of the receipt logs).

Armed with this extracted data, the function parser returns a delimited string of the function or event name plus the various parameters.

The `tokenlib`, `walletlib`, and customized per-smart-contract libraries further use this parsed function and event data to promote particular transactions or receipt logs to a customized, recognizable, C++ class fully exposing the function or event's meaning.

Convenience functions for traversing blocks, transactions, and accounts.

There are numerous functions in the `etherlib` that begin with forEvery…, such as forEveryBlockFromClient, or forEveryMini-Block. There are too many to enumerate here. Please see Appendix E for a complete list.

These functions expect to have been previously instructed how the end user wishes to access the requested items by use of an initialization function called `etherlib_init()`.

The initialization function may be called with values such as "`infura`," "`parity`," "`geth`," "`binary`," or "`fastest`." The system determines how best to deliver the data requested in the most efficient way.

For example, because QuickBlocks™ does not store input data in the fixed-width miniTransaction arrays (because it is of variable length), we fall back to the binary representation if that data is requested.

Furthermore, if the system is directed to deliver blocks containing no transactions, it reverts to requesting that data from the RPC. In this way, the `etherlib` is able to always deliver the requested data as quickly as possible. Thereby comes our system's name— QuickBlocks™.

Currency pricing information is available, as a convenience, through the `CPriceQuote` class.

Acctlib Library

The `acctlib` provides functionality to store per-account data, such as a list of transactions of interest for an account, in a quickly accessible 16-way tree data structure [5] keyed by account address. This data structure lends itself very well to storing per-account data particularly given the nature of the indexing key which is a 20-byte account address.

While this is not how QuickBlocks™ actually implements storing per account data, conceptually, as each block is produced, we extract the list of transactions from that block. The transactions are traversed searching for accounts involved in that transaction. For each such account, we append the *block_number.transaction_id* to the end of that account's growing list of references. These lists of "interesting" blocks and transactions are stored in the tree keyed by account address.

We say "conceptually" above because QuickBlocks™ does not actually do this processing at each block. Instead, and in response to our desire to minimize the impact of our work on the target machine, we store these lists only when requested by an end user. This minimize the on-disc size of the growing account-bases list of transactions.

As requests are made to certain tools and applications the 16-way tree data structure provides very fast access, $O(\log n)$, to the list per account.

This library is currently experimental.

Tokenlib and Walletlib Libraries

Created automatically entirely by the `makeClass` and `grabABI` applications, the `tokenlib` and `walletlib` libraries implement functionality to provide support for common Ethereum smart contract functionality.

We could have allowed each smart-contract that supports the ERC20 token standard to implement these functions on their own, but because of the prevalence of these functions, and because for many use cases the ABI file may not be available, we chose to implement this common functionality in libraries. This allows us to support these common types of smart contracts without requiring an ABI file from the end user (which is some cases may not be available).

By virtue of being created automatically by `makeClass`, each of the classes in these libraries may be customized using the `//EXISTING_CODE` functionality. However, at present, these class are not customized.

As a further consequence of being derived from the base classes of the `etherlib` library, each of these classes may be easily "promoted" to more capable and informative classes. The two built-in functions, *promote_to_transaction(CTransaction *t)*, and *promote_to_event(CLogEntry *e)* are provided to promote any given `etherlib` class.

In each of these two functions, the parsed data is retrieved and then split to reveal the function or event name and its parameters. Given this information the appropriate class is created dynamically and the particular fields of the class are assigned from the parsed parameters.

As difficult as this is to explain, this is one of the fundamental capabilities provided by QuickBlocks™. By being able to promote generic `CTransaction` instances to richer, more informative classes particular to a smart contract, QuickBlocks™ is able to regurgitate or re-deliver richer data than it was given.

Instead of simply delivering the *from*, *to*, *amount,* and *timestamp* from a transaction along with an unparsed input data field (as is available, at best, from the RPC), QuickBlocks™ delivers fully articulated, parsed JSON, tab-delimited, comma-separated or any other format data to a smart contract's front end or desktop application.

Furthermore, by virtue of being derived ultimately from the `CBaseNode`, each of these "promoted" classes inherit the ability to create itself dynamically and serialize itself. It is therefore possible to build a list of all transactions for a particular account (or series of accounts) or smart contract and store this data in a serialized, binary format for very fast, later retrieval.

The `tokenlib` and `walletlib` are provided so as to support parsing of a small collection of functions and events related to the Ethereum token standard and multi-sig wallets even in the absence of a smart contract's ABI definition.

In other words, any smart contract that implements the ERC 20 token interface may be successfully promoted even without its ABI. In the presence of an ABI, an even more useful and valuable promotion of transactions and events may be accomplished. This feature is described in the next session.

Per-Smart Contract Libraries

The final type of library included in the QuickBlocks™ system is not included until the end user of the system creates it. This type of library is called here a customized per-smart-contract library.

These libraries require the presence of an ABI file defining a smart contract's interface. Given the ABI definition, and using the `makeClass` application, and further anticipating the creation of a `makeClass` configuration file using the `grabABI` application, the system may automatically create a collection of C++ classes along with a make-based build system for generating a static library customized to represent the smart contract.

We've completed an example using "The DAO's" ABI and present the results at http://dao.quickblocks.io.

Notwithstanding what we mention above about the need for the ABI function, the special case of the ERC token interface and common multi-sig wallets is implemented in the `tokenlib` and `walletlib` libraries; therefore, the few functions and events related to these types of transactions are not generated in the automated process.

This then is the most central idea expressed in this white paper: the QuickBlocks™ system is able to generate a full accounting for any Ethereum smart contract for which an ABI is available by virtue of being able to fully parse all transactions and events associated with that contract. Furthermore, the functionality needed to do so is fully automated.

With the QuickBlocks™ system installed, and an in-sync local blockchain node, the end user simply needs to run the command

```
grabABI <address> -g
```

The resulting C++ code, which is placed in a folder called ./parselib will contain the software code and build system that will allow a programmer to retrieve raw RPC data from the node and produce fully parsed data that may be exported to any file format. The resulting data can be used by the programmer to better understand, test, and stress test his/her smart contract, or provide data to a front-end website or desktop application.

Tools and Applications

The following sections describe each individual tool and application that we've written against the above libraries.

Prior to this we briefly discuss a few topics that are applicable to all tools and applications, such as keeping the scraped blockchain cache fresh, accounting for forks in the blockchain, serialization and append only files, and the exportation and display of the data.

Freshening the Local QuickBlocks™ Cache

Data accumulates on the blockchain continually. Transactions are cast against the network many times each second, and the number of transactions is steadily increasing. Therefore, a process is needed to repeatedly freshen the local cache that implements the system.

At each invocation of the `blockScrape` process (detailed below) we read a previously stored value indicating the last visited block. Initially the last visited block is zero. The freshening process requests, through the RPC interface, blocks up to and including the latest block on the chain (a value that easily available through the RPC).

On each invocation, `blockScrape` requests recent blocks along with their transactions. Requests for each transaction's receipts are then initiated. The receipt along with its associated transaction is processed for error identification, and the block, if it contains at least one transaction, is then serialized to the binary cache.

Once installed and fully running, the `blockScrape` process is automated to ensure data freshness using a functionality such as the Linux cron command or similar. In this way, the QuickBlocks™ binary cache remains as fresh as the blockchain data itself.

Handling Forks in the Chain

Blockchains are guaranteed to be consistent across all nodes—eventually. There are times, however, when a certain subset of nodes believes a certain block is the latest valid block, while a different subset believes a different block to be valid. This is an unavoidable consequence of the way blockchains operate.

In order to account for this aspect of the blockchain, QuickBlocks™ assumes that recent blocks, which may have been previously processed and stored, are open for reevaluation. We do this by assigning a boolean 'finalized' flag to each block, initially set to false.

Our system does determine if a particular block is valid (that is the job of the blockchain itself). We, instead, assume that any previously processed block within the last 25 must be reevaluated. Once reevaluated, if the block is more than 25 from the head of the chain, the finalized flag is set to true, and the block is not re-evaluated again.

So, while above, we did indicate that the `blockScrape` process evaluates blocks from the most recently visited block to latest block, in reality it processes to the latest block on the chain minus 25, replacing any values it finds in any previously stored blocks.

In this way, one may be certain that any block older than 25 blocks from latest block is 'finalized,' while blocks within 25 of the latest block—while most likely valid—are subject to reevaluation.

While solutions to freshening and forking are detailed here in the context of the `blockScrape` application, these issues apply to many of the applications described below. Where appropriate, each application handles these issues similarly.

Serialization and Append Only Files

Blockchain data is immutable. This means that, once finalized, a block will never be modified. Furthermore, it is a truism that appending binary data to a file on most operating systems is significantly faster than opening that same file for random access and editing, especially when the file is large.

We take advantage of this fact throughout our work.

For example, the `fullBlock` index is append only, however, because of potential forks, we cannot simply append non-empty block indices to this list. We therefore only append blocks older than 25 blocks from the top of the chain to this data file. We maintain a separate file for non-finalized blocks where appropriate. This includes the `fullBlock` index, the `miniBlocks` index, and various other files. This allows us to open the larger, finalized block indices in append-only mode and operate on separate, smaller files via random read/write access for non-finalized data.

In a server environment, or a desktop application environment, these concerns are lessened as one may keep the data in memory and write to file when necessary.

Export Formatting and Display Strings

Using particularly formatted strings, each class derived from the `CBaseNode` may display its own contents into whatever format is requested.

This is accomplished using a technology of our own devising called display strings. A display string contains squirrely-bracket surrounded field names stored inside a square-bracketed sections of the string. The square-bracketed surrounded template will only display if the field value of the identified field is non-empty.

This allows us to display tabular data which may contain empty rows which we desire not to display. Furthermore, this allows us to display minimally small JSON data as fields with default or empty values are not displayed. Display string technology is more fully described in Appendix B.

In the remainder of this section we discuss individual applications and tools built upon various combinations of the above libraries.

The blockScrape Application

The first application component of QuickBlocks™ interacts with the blockchain node directly using RPC or IPC to request the blockchain data. Delivered blocks arrive from RPC in a format called JSON. The first task, therefore, of the `blockScrape` app is to parse the JSON data into an internal, proprietary, in-memory format. As each block is parsed, its list of transactions is also retrieved and parsed.

At this point, our system determines if the block is either (a) empty of transactions, or (b) contains one or more transactions. In the later case, the block is marked for later writing to the binary cache.

The JSON data is comprised of Ascii strings, however, both internally and on permanent storage, it is stored as bytes. This conversion reduces the data's footprint by nearly ½.

Prior to writing blocks to the binary cache, we further request of the node each transaction's receipt. The transaction receipt provides information needed to determine if the transaction successfully completed or ended in error. This high-cost determination is performed once, on our initial encounter with the block, eliminating repeated calculation of a value that will never change (due to the immutability of the data). Determining whether or not a transaction was in-error is described in detail below.

In addition to each transaction's receipt, we accumulate each transaction's events logs. Logs are generated into a receipt during the execution of a smart contract at each point in the smart contract where the programmer has inserted such events.

The presence or absence of an event in a transaction's receipt is a partial indication of the transaction's success or failure, but not a dispositive one. Due to the flexibility of the event system, and the possibility of poorly designed event placement, logs, while useful, are not an iron-clad method by which to document the operational history of a smart contract. Transactions may either (a) succeed without generating accurate events, or (b) fail while generating misleading or confusing events[5].

It is generally accepted that an understanding of a smart contract's behavior may be gotten from an analysis of its logs, and, in fact, most existing smart contract front-ends assume this to be true. We believe this is error–prone and suggest that full, fast access to the transaction and event data from a smart contract is a better alternative than relying on the events logs alone.

[5] While transaction that fail will not generate events, transaction that call into other contracts will write events even if the called-into transaction fails. If poorly programmed, this may result in one contract recording an event indicating success, while the called-into invocation failed.

Our method of determining the success or failure of an individual smart contract invocation is described next.

Each transaction carries a data item called gas, or as we call it *gasAllowed*[6]. This value indicates the maximum allowable cost (as set by the initiator of the transaction) allowed for a particular transaction. Each transaction's receipt contains an additional field called *gasUsed*. *gasUsed* indicates the total cost, recorded by the node, of the invocation.

If *gasAllowed* is smaller than *gasUsed*, the transaction unequivocally succeeded and is marked as such.

Unfortunately, one may not rely on the inverse. While *gasUsed* will never be greater than *gasAllowed*, it may be equal; however, this is not a clear indication of a failed transaction. A successful transaction may, by happenstance, use exactly as much gas as was allowed.

In order to distinguish between successful transactions that used exactly as much gas as was allowed and an in-error transaction, that is, a transaction that failed either due to (a) trying to use more gas than was provided, or (b) as a result of the source code throwing an exception, one must further request of the RPC a trace of the transaction.

While not confirmed experimentally, the RPC trace request is likely the most resource intensive RPC call because the node software must re-create the trace on the fly, and therefore the slowest operationally.

In our analysis of transactions, we've discovered that during the first three million blocks, consisting of 15,362,847 transactions, 18.6% (2,859,376) needed to be traced because *gasAllowed* was equal to *gasUsed*.

By far the predominant reason for this need for tracing is direct transfers of value to non-contract address, each of which costs 21,000 gas.

Oddly, some Ethereum wallet software (such as Mist at the time of this writing) send

[6] We hereafter call this field gasAllowed. The node's RPC documentation [3][4] calls it gas. We prefer the more accurate name.

exactly 21,000 gas for these straight transfers of value. Of the 2,859,376 transactions that needed to be traced in order to determine error status, 2,680,783 (93.75%) provided exactly 21,000 gas.

This appears to be an oversight by wallet providers. This unnecessary need to trace transactions to determine in-error status could be easily removed if the wallet software sent a slightly higher value of gas, say 21,001 gas. The overage would be refunded, but the transaction would not need to be traced to determine if it was in error.

Another reason for the high prevalence of the need to trace transactions is the liberal use of the 'throw' keyword in many smart contracts; however, this is obviously less worrisome than the 21,000 gas issue.

Returning to the discussion, we store the result of the in-error calculation on first encounter[7], and thereafter realize significant performance improvements by retrieving this cached in-error information from our proprietary binary cache.

After accumulating the full data for each transaction cohesively into a block so as to avoid the repeated RPC requests necessary to build it on the fly, we store the data to permanent binary storage in a format optimized for speed of access.

This involved an experimentally determined optimal number of blocks per file so as to balance speed of access with size of storage on disc. The number of blocks stored per file is parameterized allowing for different characteristics on different hardware/software installations.

During our initial evaluation of the blockchain data we discovered, to our surprise, that nearly than 35% of all blocks (1,356,704 of 3,250,000) contain zero transactions. Because we are interested only in transactions and accounts, and in order to save significantly in the size of the data written to permanent storage, we store only blocks with one or more transactions to disc.

In order to avoid having to query the permanent storage to determine information that we already know (whether or not a particular block contains transactions), we store an index of blocks that contain transactions.

With this information, we may iterate over all non-empty blocks without checking for the existence of known empty blocks on disc.

Additionally, through a simple calculation, we can iterate over all empty blocks if we wish. Because we do not store empty blocks on disc, here we need to revert to the RPC interface to retrieve the empty block data. Because there are no transactions, and therefore no need to determine if any transactions are in-error, the "slow" RPC interface is not overly burdensome. We do not need to request receipts nor trace transactions for empty blocks (because there are no transactions).

The `blockScrape` application allows us to balance between speed, access to the data, and a minimization of permanent storage needed for the binary cache.

The `blockScrape` process runs continually, and it is here that we leave it to its operation in order to discuss the next process in the suite of applications that fully describe QuickBlocks™.

The miniBlocks Application

As stated earlier, our initial implementations are interested primarily in transactions. For this reason we store only blocks that contain transactions.

We find, however, that simply storing the blocks, which contain a significant amount of non-transactional data such as hashes, bloom filters, mining information and the like, does not provide the performance improvements we seek.

We use the `miniBlocks` application to strip unwanted data from each block. This necessitates a duplication of some of the data,

[7] To be precise after the block is finalized when it recedes more than 25 blocks from the top of the chain.

as we retain the original block data and store only a subset of the data.

In addition to removing unwanted data from the blocks, we also collapse transactions, receipts, and traces, into a single item called a miniTransaction, arranging the data in a way that significantly increases the speed with which we can deliver it to our end user's applications. We do this by storing blocks and transactions in two separate fixed-width arrays both on disc and in memory. In this way, we can simply read entire tables of data into memory with a single binary read operation.

Reading large chunks of data into memory is profoundly faster than serializing it into memory. Furthermore, it is orders of magnitude faster than parsing JSON string data.

A `miniBlock` contains four fields: the *timestamp* of the block, the *blockNumber* of the block, the *index* of the first transaction in the array of miniTransactions, and the *number* of miniTransactions included in this block starting at the *index*.

For each miniTransaction, we store the following fields: the *index* of the transaction within the block, a boolean flag indicating *success* or failure of the transaction, the *gasPrice* of the transaction, the *gasUsed* by the transaction, the *gasAllowed* of the transaction, the *from* and *to* addresses of the transaction, and *value* of the transaction.

The remaining transaction related data (receipt logs and input data), being of variable length are not stored in the `miniBlock` array as this would make it impossible to read the entire array in a single operation. These items may be later loaded and parsed from the binary blocks, but only in the case where an end user has requested that data explicitly.

This lesser amount of data, in itself, greatly speeds up access to the data. However, we realize a profound increase (two order of magnitude) in the speed of access to the data by storing the `miniBlock` data in arrays as opposed to blocks which store arbitrarily long lists of transactions. We are able to use this circumscribed data when possible, and revert to our `blockScrape`-created binary data when necessary. In some cases, depending on the user's needs, we may have to further fall back to the RPC, for example if the user wants to visit empty blocks or analyze mining activity.

The accountTree Application

The `accountTree` application allows us to accumulate—for the first time, we believe—an off-chain list of all Ethereum accounts and various data per account.

During the creation of this list of accounts, we are able to summarize various data related to the account, including its ether balance if any, a boolean indicating if the account contains contract code, the first and most recent blocks in which the account has participated (including a marker indicating the existence of incoming internal transactions sent by smart contracts), the account's address (obviously), and any other data we deem useful.

We store the data using our serialization functionality which provides lightning fast access to a binary list of accounts along with this associated, user-specified data.

This type of data will, we hope, allow us to increase the speed with which we can gather transactional data per address. Much of the discussion until now has concerned the block chain data as a whole, but we think the primary use for QuickBlocks™ is to build accounting and analysis data for individual accounts and smart contracts. With a per-account list of the first and latest block access, building such transaction lists will be made easier. Quick access to each account is accomplished with a 16-way k-ary tree data structure.

The first and most recent block number references must account for the possibility that a smart contract having sent a message or value to an account "internally." The existence of these incoming internal transactions is the reason why it is not adequate to simply use a relational link to pick up all transactions with either a *to* or *from* address for the account.

The `accountTree` application is currently under development. Parts of this code are under an open source license.

The gasHole Tool

Notwithstanding its name, nor the fact that my son recently stole the name for use in `his hackathon` project, the `gasHole` tool is useful for analyzing a smart contract's gas usage. This tool looks at each transaction on the blockchain and attempts to summarize and analyze aspects of gas usage. Simple summaries, averages and statistics are created.

In addition, if the end user requests such, the tool can analyze the gas usage for a particular account or smart contract. For example, a smart contract purveyor might be interested to know how much each function invocation is costing his/her users.

As a further example, the programmer may find a particular smart-contract function uses a higher per-invocation gas than a similar function. Upon analysis, the programmer may learn that this is due to repeated 'throws' and poorly understood or poorly documented user input data. Or, perhaps, the function is written in such a way that it uses more gas than it should. (In other words, it's a `gasHole`.)

We've been experimenting with an analysis of The DAO's gas usage, but do not have useful results to report at this time.

Tokenomics™

The ERC20 token standard defines an interface that makes is easier for tools such as blockchain wallets and token trading markets to interact with the token. By observing this pre-specified interface, ERC20 tokens become fungible between token holders. QuickBlocks™, being tool for blockchains, also supports the ERC20 token through its `tokenlib` library.

We've developed the concept of `Tokenomics™` to explore this capability.

We've produced a study [6] using one of the above-mentioned per-smart-contract libraries for "The DAO."

In that study, we duplicated the "extraBalance accounting" accomplished in late summer 2016. Reports online have suggested that the extraBalance accounting mentioned here may have taken more than twenty hours to complete.

QuickBlocks™ is able to generate identical results (in fact, improving on the accuracy) in less than four minutes.

This is type of speed-up, more than two orders of magnitude, is what we have been in search of.

The ethslurp Application

The ethslurp application was the first application we built on this system. Because it was written prior to the `blockScrape` application, it retrieves its data from the http://etherscan.io web APIs, which makes it an odd duckling in our system.

All of the other applications mentioned in this paper read directly from the blockchain, or in some cases, from our binary cache depending on the required speed and level of detail desired.

ethslurp does have one feature worth noting, however, that does not appear in any of our other applications. This is per-account binary caches, reflecting a central design criteria of all of our code.

The ethslurp application asks for and caches account data only when an end user requests it. With each request, the data is freshened and perhaps re-cached.

Additionally, a --file command line is provided which allows multiple command lines to be stored in a text file for repeated invocations much like the `sed –f` command line option from Linux. This capability allows for a single load of an account's transaction data and repeated application of a series of commands.

At all points in all of our applications and libraries we use this type of *lazy computation* so as to allow for the fastest possible access we can give to the data.

The verifyBlock Application

Because we are concerned with the accuracy of the data provided by our system, we provide the `verifyBlock` application.

In this simple app, we take a block number or a range of block numbers from the user's command line. We then retrieve the RPC data for that block, all of the block's transactions, and each of the transaction's receipts and logs. We store this in a single concatenated string.

We then build the same type of string from our internal cache with the expectation that we get the same result. Any deviation is quickly inspected. There should be no deviations[8].

We provide the source code for this application as open source. In this way, our community can convince themselves that the QuickBlocks™ cache delivers identical data to that available through the RPC.

On our local testing machines, `verifyBlock` runs continually, repeatedly asking for and verifying randomly-selected blocks. We have yet to have seen a reported difference between the blocks produced by our cache and the blocks returned by the RPC.

Other Applications and Tools

There are a number of other applications and tools written against the various libraries. Among these are `lastestBlock` which simply retrieves the latest block on the chain and delivers it to the screen or for use in a shell script; `getBlock` which takes a single block number or a range of blocks and delivers a JSON representation of the block to the screen or file; `getTrans`, `getReceipt`, `getLogs`, `getBloom`, `getTrace` which each return the corresponding data structure (as JSON) given a transaction hash (or blockNumber.transID). All of the 'get' tools accept a --depth parameter which allows the user to retrieve the entire data stored below a particular level.

Various other tools include `whenBlock` which, given either a date or blockNumber returns the other; `ethName` which takes either a full or partial account address or a word or phrase and returns the other to the command line, and `getBalance` and `getTokenBalance` which retrieve various balances given an address and a blockNumber stored on the Ethereum chain.

Each of these other applications or tools may be used in scripting scenarios to accomplish various useful different tasks.

Use Cases

Here, we provide a list of partially-implemented or imagined applications or processes that can be built upon our system.

Infura-like Enhanced RPC (Infura++)

Infura is a web-based system running Ethereum nodes and providing RPC interfaces to its clients. It supports an increasingly large number of end users in a manner similar to the way a traditional website might deliver a web API.

We propose a system, that runs in tandem with Infura, that we call `Infura++` or `Enhanced RPC`. This imaged system would allow an end user to specify the address of a smart contract, whereby QuickBlocks™ would retrieve the contract's ABI file, automatically generate a collection of C++ files and a make-based build system, and create a dynamically linked library for that contract. This per-smart-contract interface could provide customized web-delivered, smart-contract-specific JSON data to the user via a Web API.

Even if the user's smart contract address does not have an available ABI, the `Enhanced RPC` application mentioned here can provide JSON data for any ERC20 token functions or events without customization of the installation.

This system might encourage end users to develop their front-end websites using the `Infura++` JSON, thereby tying them into a

[8] Except perhaps when the block is within 25 blocks of the head, but this is taken care of elsewhere.

longer-term relationship the provider, including on-going remunerated patronage.

QuickBooks® Integration

Given that QuickBlocks™ provides very access to particular account transactions as describe above, it is conceivable that this full transaction data could be exported into existing account systems such as QuickBooks® from Intuit.

QuickBooks® has existing capability to handle this data. We have begun initial discussion with various members of our community on how this might be accomplished. Progress is slow.

Other existing accounting software can easily be accommodated as well.

Blockchain on a Stick

We've modified the Ethereum node running on our systems to store its data on a detachable SSD Drive for ease of movement from one machine to another. Furthermore, we've stored our various caches and indexes on the same drive.

Prior to a personal meeting, we synchronize all available data on this SSD drive and take it with us to our meeting. Being attached via a USB cable, the drive detaches from one computer and attaches to another looking very much like a popsicle, with a long wire projecting from one end a bulbous drive at the other. We call this popsicle-like removable drive Blockchain-on-a-Stick™.

One possibility product idea is to produce, with the low latency of a day or two, a shippable Blockchain-on-a-Stick™ for end users.

This physical object could include an installation of the blockchain node software and the QuickBlocks™ system fully installed and able to run directly from the SSD Drive.

This would greatly lessen the time needed to synchronize the blockchain, while at the same time providing a fully-functioning node. Furthermore, Blockchain-on-a-Stock™ could include the QuickBlocks™ indexes and caches and the various software libraries, tools, and applications.

It is conceivable that Blockchain-on-a-Stick™ could be shipped via one-day shipping. A simple plug-and-play would allow an end-user to be up and running quickly without a long process of downloading what is certain to become increasingly large amount of data.

Continued production of such a system would involve simply automating the updating of the data.

Debugging Support

For the first time (to our knowledge) QuickBlocks™ allows a smart contact's author to record live transactions from a running smart contract. This recording can be directly fed back into a testing version of the same smart contract (or more importantly, with some translation of the data, into a proposed upgrade to the smart contract).

Because it is possible for QuickBlocks™ to convert (through parsing) each transaction back into the language of the smart-contract, this enriched data may be used to test various scenarios and interaction patterns.

Furthermore, QuickBlocks™ provides C++ classes (via its per-smart contract parsing libraries) that may be used to mimic transactional data. These on-demand transactions may be used to generate high-volume testing data and tools for analyzing possible implementations of a smart contract prior to deployment, or to test gas usage, attack vectors or other aspects of the contract.

Another use for debugging support would be to optimize the Ethereum node data as a whole. There have been many days in recent months where the blockchain has been over-burdened. For example, recent ICOs brought the system to a slow grind for many hours. If one used QuickBlocks™ to record the period of over-use, one could repeatedly replay those transactions against a test version of the node (with the block-timing reduced) as a method by which to identify, and perhaps remove, bottlenecks.

Had we only some of this capability before the deployment of the DAO…

Blockchain-Wide Reporting and Analysis

There is clearly a need for blockchain-wide reporting and analysis. This ability, which we believe is available to us at speed for the first time, has allowed us to identify two significant things about the Ethereum blockchain that had not previously been widely known.

We found, by analyzing blockchain-wide data that nearly than 35% of all blocks contain zero transactions. The implications of this are not clear, but a serious analysis of this fact may provide insight into the activities of the miners.

More importantly we found that 97% of all transactions where *gasAllowed* is equal to *gasUsed*, and therefore which needs to be traced to determine if the transaction was in error, provide exactly 21,000 *gasAllowed* (indicating simple value transfers).

If those same transactions were sent with 22,000 *gasAllowed* instead, the time needed to identify in-error transactions by blockchain scrapers would be significantly reduced. This applies to all front-end and desktop applications as well. In fact, it applies to any application that consumes RPC data.

If Ethereum wallets provided 22,000 *gasAllowed* for simple value transfers, and only 21,000 is consumed, the extra 1,000 gas would be refunded to the sender. Importantly, this would decrease the number of transactions that need to be traced because *gasAllowed* would not be equal to *gasUsed* as it is today.

Without QuickBlocks™, and its fast delivery of this sort of blockchain-wide data, this issue would have never been noticed. There are certainly many other insights to be grocked from the blockchain data.

Another analysis, perhaps as an extension of the `accountTree` application, might be to notate each smart contract on the blockchain. With such a list, one could retrieve ABI files from various online sources. With this information, one could build a fully automated version the 4byte.directory website extending it by adding not only all available functions found in all available ABI files but also, importantly, all event signature tokens as well (which are currently not available).

Another possible use for blockchain-wide analysis might be to analyze the smart contract code being stored on the chain. Our guess is, without having done the work, that much of the code on the blockchain is duplicated. Being immutable, one possible optimization to the node that might lessen storage space, would be to store identical code in a single location addressed by a hash of the byte code. In other words, instead of storing the code per address of creation, store it in a content-addressable location as do systems such as IPFS. This would allow for de-duplication of the byte code, thereby lowering storage requirements on the node.

Implications and Future Concerns

To complete our presentation, we present a few implications of our work and conclude our paper.

Client-Side Decentralized Applications

As we more fully developed our system, we soon realized that, given the nature of the blockchain node software, which in its fundamental form is fully decentralized on each end user machine, is that a new day may be dawning for application developers. Or, better said, an old day is re-dawning.

Prior to the advent of the wide-spread networking and the Internet, application development concerned itself only with the end user's desktop. In a very real way, this means that the world's computing environment was fully decentralized. With the advent of the Internet and the world-wide-web, all that changed. The center of the computation moved to the web server.

With the recent re-appearance of decentralized databases such as the blockchain and related technologies such as IPFS, Swarm, and Whisper, it is now possible to return to client-side, desktop-focused environment that runs on individual computer using local sources of data.

Because of the nature of that data, it seems reasonable to contemplate a return to an era where an application behaves as if it were entirely stand-alone.

Unlike the pre-web epoch, where connectivity was basically non-existent, desktop application may now natively use data that is distributed across the planet. It is as if the world now has the entire Internet at its computing hands while at the same time running on a perfectly-secure, stand-alone, disconnected computer.

This is made more obvious when one may access, analyze, and respond to the blockchain data at speed, which is what QuickBlocks™ provides.

In one imagined conception of the future of computing, and given a scaled, high-speed blockchain client, we envision the appearance of smart contracts that store no data at all. Data storage may be accomplished via IPFS or Swarm. Messages sent through function calls and events to smart contracts could communicate the locations of these data, which can be retrieved by QuickBlocks™. This without having to store anything directly on the blockchain.

Give the ability to quickly access this data off-chain in an easily digestible way, a very capable client-side-only desktop application could respond to these messages as if the data was created locally.

This capability might one day provide a way to take full advantage of widely connected networked data, without the dangers of current inherently flawed Internet security mechanisms.

Proof of Stake / Sharding

With the pending advent of Proof of Stake, Casper, and Sharding, each of which holds promise for a significant increase in the speed and volume of transactions on the network, a challenge should be anticipated to our system.

The question is “Will QuickBlocks™ be able to keep up when there are thousands of transactions per second?”

In response, we will say that there will be at least one extant example of a system that can keep up with the blockchain: that is the blockchain itself. We ask, “Why, if at least one piece of software can keep up with the blockchain, can’t we.”

With this question in mind, we now end our presentation. ⊡

REFERENCES

[1] Wood, Gavin, “Ethereum: A Secured Decentralized Generalized Transaction 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] GitHub. n.d. Geth JSON RPC. Accessed 07 24, 2017. https://github.com/ethereum/wiki /wiki/JSON-RPC.

[4] GitHub. n.d. Parity JSON RPC. Accessed 07 24, 2017. https://github.com/paritytech/ parity/wiki/JSONRPC.

[5] Wikipedia. n.d. K-ary Tree. Accessed 07 24, 2017. https://en.wikipedia.org/wiki /K-ary_tree.

[6] QuickBlocks. n.d. Four Periods of "The DAO". Accessed 07 24, 2017. https://dao.quickblocks.io.

Appendix A: The makeClass Application

In this appendix, we present the configuration file and the two template files used by the makeClass application to generate C++ code that comprises much of the code in QuickBlocks.

During its operation, makeClass first reads settings from the configuration file and then calculates a few derived values such as SHORT3 from CLASSNAME by removing the leading character, setting the value to lower case, and splitting off the three left-most characters. As makeClass reads each template file, it simply replaces the token values in the code with the values represented in the configuration file or subsequently derived values.

This configuration file describes the CTransaction class from the etherlib library:

```
class:         CTransaction
fields:        hash blockHash|int blockNumber|string creates|int confirmations|
               addr contractAddress|string cumulativeGasUsed|addr from|int gas|
               string gasPrice|string gasUsed|hash hash|string input|bool isError|
               bool isInternalTx|int nonce|hash r|string raw|hash s|int timeStamp|
               addr to|int transactionIndex|hash v|string value|CReceipt receipt
includes:      ethtypes.h|abilib.h|receipt.h|trace.h
cIncs:         #include "etherlib.h"
destination:   ~/quickBlocks/src/libs/etherlib/
```

This is the template for the header file for makeClass generated code. Notice the // EXISTING_CODE comments. It is here that a programmer would put custom code for the class in the generated files.

```
#pragma once
/*-------------------------------------------------------------------------
 * This source code is confidential proprietary information which is
 * Copyright (c) 1999, 2017 by Great Hill Corporation.
 * All Rights Reserved
 *------------------------------------------------------------------------*/
/*
 * This file was generated with makeClass. Edit only those parts of the code inside
 * of 'EXISTING_CODE' tags.
 */
[H_INCLUDES]
//--------------------------------------------------------------------------
class [{CLASS_NAME}];
typedef SFArrayBase<[{CLASS_NAME}]>         [{CLASS_NAME}]Array;
typedef SFList<[{CLASS_NAME}]*>             [{CLASS_NAME}]List;
typedef CNotifyClass<const [{CLASS_NAME}]*> [{CLASS_NAME}]Notify;
typedef SFUniqueList<[{CLASS_NAME}]*>       [{CLASS_NAME}]ListU;

//---------------------------------------------------------------------------
extern int sort[{PROPER}]        (SFString& f1, SFString& f2, void *rr1, void *rr2);
extern int sort[{PROPER}]ByName  (const void *rr1, const void *rr2);
extern int sort[{PROPER}]ByID    (const void *rr1, const void *rr2);
extern int isDuplicate[{PROPER}] (const void *rr1, const void *rr2);

// EXISTING_CODE
// EXISTING_CODE

//--------------------------------------------------------------------------
class [{CLASS_NAME}] : public [{BASE_CLASS}]
{
public:
[FIELD_DEC]
public:
                  [{CLASS_NAME}]  (void);
                  [{CLASS_NAME}]  (const [{CLASS_NAME}]& [{SHORT}]);
                 ~[{CLASS_NAME}]  (void);
  [{CLASS_NAME}]& operator=   (const [{CLASS_NAME}]& [{SHORT}]);
  DECLARE_NODE ([{CLASS_NAME}]);

  // EXISTING_CODE
  // EXISTING_CODE

protected:
  void   Clear       (void);
  void   Init        (void);
  void   Copy        (const [{CLASS_NAME}]& [{SHORT}]);
  SFBool readBackLevel(SFArchive& archive);

  // EXISTING_CODE
  // EXISTING_CODE
};

//---------------------------------------------------------------------------
inline [{CLASS_NAME}]::[{CLASS_NAME}](void)
{
  Init();
  // EXISTING_CODE
  // EXISTING_CODE
}

//---------------------------------------------------------------------------
inline [{CLASS_NAME}]::[{CLASS_NAME}](const [{CLASS_NAME}]& [{SHORT}])
{
  // EXISTING_CODE
  // EXISTING_CODE
  Copy([{SHORT}]);
}

// EXISTING_CODE
// EXISTING_CODE

//---------------------------------------------------------------------------
inline [{CLASS_NAME}]::~[{CLASS_NAME}](void)
{
  Clear();
  // EXISTING_CODE
  // EXISTING_CODE
}

//---------------------------------------------------------------------------
inline void [{CLASS_NAME}]::Clear(void)
{
  // EXISTING_CODE
  // EXISTING_CODE
}

//---------------------------------------------------------------------------
inline void [{CLASS_NAME}]::Init(void)
{
  [{BASE_CLASS}]::Init();

[FIELD_SET]
  // EXISTING_CODE
  // EXISTING_CODE
}

//---------------------------------------------------------------------------
inline void [{CLASS_NAME}]::Copy(const [{CLASS_NAME}]& [{SHORT}])
{
  Clear();

  [{BASE_CLASS}]::Copy([{SHORT}]);
[FIELD_COPY]
  // EXISTING_CODE
  // EXISTING_CODE
  finishParse();
}

//---------------------------------------------------------------------------
```

```
inline [{CLASS_NAME}]& [{CLASS_NAME}]::operator=(const [{CLASS_NAME}]& [{SHORT}])
{
 Copy([{SHORT}]);
 // EXISTING_CODE
 // EXISTING_CODE
 return *this;
}

//---------------------------------------------------------------------------
inline SFString [{CLASS_NAME}]::getValueByName(const SFString& fieldName) const
{
 // EXISTING_CODE
 // EXISTING_CODE
 return Format("[{"+toUpper(fieldName)+"}]");
}

//---------------------------------------------------------------------------
extern SFString next[{PROPER}]Chunk(const SFString& fieldIn, SFBool& force, const void *data);

//---------------------------------------------------------------------------
IMPLEMENT_ARCHIVE_ARRAY([{CLASS_NAME}]Array);
IMPLEMENT_ARCHIVE_LIST([{CLASS_NAME}]List);

//---------------------------------------------------------------------------
extern SFString next[{PROPER}]Chunk_custom(const SFString& fieldIn, SFBool& force, const void
*data);

//---------------------------------------------------------------------------
// EXISTING_CODE
// EXISTING_CODE
```

This is the template for the C++ implementation file for makeClass generated code:

```
/*-------------------------------------------------------------------------
 * This source code is confidential proprietary information which is
 * Copyright (c) 1999, 2017 by Great Hill Corporation.
 * All Rights Reserved
 *------------------------------------------------------------------------*/
/*
 * This file was generated with makeClass. Edit only those parts of
 * the code inside of 'EXISTING_CODE' tags.
 */
#include "[{LONG}].h"
[OTHER_INCS]
//---------------------------------------------------------------------------
IMPLEMENT_NODE([{CLASS_NAME}], [{BASE_CLASS}], curVersion);

//---------------------------------------------------------------------------
void [{CLASS_NAME}]::Format(CExportContext& ctx, const SFString& fmtIn, void *data) const
{
 if (!isShowing())
   return;

 if (fmtIn.IsEmpty())
 {
   ctx << toJson();
   return;
 }

 SFString fmt = fmtIn;
 if (handleCustomFormat(ctx, fmt, data))
   return;

 [{CLASS_NAME}]Notify dn(this);
 while (!fmt.IsEmpty())
   ctx << getNextChunk(fmt, next[{PROPER}]Chunk, &dn);
}

//---------------------------------------------------------------------------
```

```
SFString next[{PROPER}]Chunk(const SFString& fieldIn, SFBool& force, const void *data)
{
 [{CLASS_NAME}]Notify *[{SHORT}] = ([{CLASS_NAME}]Notify*)data;
 const [{CLASS_NAME}] *[{SHORT3}] = [{SHORT}]->getDataPtr();

 // Now give customized code a chance to override
 SFString ret = next[{PROPER}]Chunk_custom(fieldIn, force, data);
 if (!ret.IsEmpty())
  return ret;

 switch (tolower(fieldIn[0]))
 {
[FIELD_CASE]  }

 // Finally, give the parent class a chance
 [{PARENT_CHNK}]
 if (!ret.IsEmpty())
  return ret;

 return "<span class=warning>Field not found: [{" + fieldIn + "}]</span>\n";
}

//---------------------------------------------------------------------------
SFBool [{CLASS_NAME}]::setValueByName(const SFString& fieldName, const SFString& fieldValue)
{
 // EXISTING_CODE
 // EXISTING_CODE

[{PARENT_SET}]
 switch (tolower(fieldName[0]))
 {
[FIELD_SETCASE]  }
 return FALSE;
}

//---------------------------------------------------------------------------
void [{CLASS_NAME}]::finishParse()
{
 // EXISTING_CODE
 // EXISTING_CODE
}

//---------------------------------------------------------------------------
void [{CLASS_NAME}]::Serialize(SFArchive& archive)
{
[{PARENT_SER}]
 if (archive.isReading())
 {
[ARCHIVE_READ]   finishParse();
 } else
 {
[ARCHIVE_WRITE]
 }
}

//---------------------------------------------------------------------------
void [{CLASS_NAME}]::registerClass(void)
{
 static bool been_here=false;
 if (been_here) return;
 been_here=true;

[{PARENT_REG}]
 SFInt32 fieldNum=1000;
 ADD_FIELD([{CLASS_NAME}], "schema",  T_NUMBER|TS_LABEL, ++fieldNum);
 ADD_FIELD([{CLASS_NAME}], "deleted", T_BOOL|TS_LABEL,  ++fieldNum);
[REGISTER_FIELDS]

 // Hide our internal fields, user can turn them on if they like
 HIDE_FIELD([{CLASS_NAME}], "schema");
 HIDE_FIELD([{CLASS_NAME}], "deleted");
```

```
  // EXISTING_CODE
  // EXISTING_CODE
}

//---------------------------------------------------------------------------
int sort[{PROPER}](const SFString& f1, const SFString& f2, const void *rr1, const void *rr2)
{
  [{CLASS_NAME}] *g1 = ([{CLASS_NAME}]*)rr1;
  [{CLASS_NAME}] *g2 = ([{CLASS_NAME}]*)rr2;

  SFString v1 = g1->getValueByName(f1);
  SFString v2 = g2->getValueByName(f1);
  SFInt32 s = v1.Compare(v2);
  if (s || f2.IsEmpty())
    return (int)s;

  v1 = g1->getValueByName(f2);
  v2 = g2->getValueByName(f2);
  return (int)v1.Compare(v2);
}
int    sort[{PROPER}]ByName(const    void    *rr1,    const    void    *rr2)    {    return
sort[{PROPER}]("[{NAME_SORT1}]", "[{NAME_SORT2}]", rr1, rr2); }
int    sort[{PROPER}]ByID       (const    void    *rr1,    const    void    *rr2)    {    return
sort[{PROPER}]("[{ID_SORT1}]", "[{ID_SORT2}]", rr1, rr2); }

//---------------------------------------------------------------------------
SFString next[{PROPER}]Chunk_custom(const SFString& fieldIn, SFBool& force, const void *data)
{
  [{CLASS_NAME}]Notify *[{SHORT}] = ([{CLASS_NAME}]Notify*)data;
  const [{CLASS_NAME}] *[{SHORT3}] = [{SHORT}]->getDataPtr();
  switch (tolower(fieldIn[0]))
  {
    // EXISTING_CODE
    // EXISTING_CODE
    default:
      break;
  }

#pragma unused([{SHORT}])
#pragma unused([{SHORT3}])

  return EMPTY;
}

//---------------------------------------------------------------------------
SFBool  [{CLASS_NAME}]::handleCustomFormat(CExportContext&  ctx,  const  SFString&  fmtIn,  void
*data) const
{
  // EXISTING_CODE
  // EXISTING_CODE
  return FALSE;
}

//---------------------------------------------------------------------------
SFBool [{CLASS_NAME}]::readBackLevel(SFArchive& archive)
{
  SFBool done=FALSE;
  // EXISTING_CODE
  // EXISTING_CODE
  return done;
}

//---------------------------------------------------------------------------
// EXISTING_CODE
// EXISTING_CODE
```

Appendix B: Display Strings

What is a display string?

A display string is a regular 'c' character string made up of plain text and field tokens. Into these field tokens a class instance's field values are inserted on-the-fly as the string is displayed. Display strings are used in many places in QuickBlocks: during the rendering of JSON, tab-delimited, or comma-separated data to file or the screen, while generating C++ implementation and header files in makeClass, and even when retreiving arbirtrary name-specifiied values from a class.

Customizing display strings

The full power of display strings becomes apparent when one starts customizing them. Each class derived from CBaseClass must implement a pure virtual function called Format which takes as parameter a display string. An empty display string implies that JSON output is desired, but the display string may be specified in any format. For example "[The {p:FROM} field holds value][{FROM}]," would display "The from field holds value account_12" if the from field held a value of 'account 12,' but would display nothing if the 'from' were empty.

Using display strings

Display strings consist of three parts: clear text, optional text and field tokens.

Field tokens must exist, with no intervening spaces, inside of squiggle brackets {} and must correspond to one of the recognized field names for the class. There can be no spaces or other characters between the squiggles and the token name. For example {SUBJECT} is a valid field token, {text SUBJECT} is not.

Each {fieldToken} must further exist inside of square brackets []. Optional text may appear on either side of the {fieldToken} inside of the square brackets. Optional text is displayed only if the value of the field is non-empty. Optional text, as the name implies, need not exist. The square brackets themselves must exist.

Clear text may exist in a display string outside of the square brackets eitehr before or after the opening or closing bracket respectively. Clear text need not exist, but if it does, it is copied directly to the display unaltered.

A simple example

As an example, the display string:

Clear [Optional {CATEGORY} Optional] Clear

would render as 'Clear Optional Finance Optional Clear' if the category field evaluates to the value 'Finance'. If the category field is empty the string would render as 'Clear Clear' (text inside the square bracket is not rendered if the field value is empty).

Prompt tokens

Notwithstanding the paragraph above you may insert the text 'p:' immediately after the opening squiggle and just prior to the field name to turn the token into a prompt token.

A non-prompt (or regular) token is resolved to the value of the field. A prompt token is resolved to the name of the field. For example the token {FROM} evaluates to the value of the from field whereas the token {p:FROM} evaluates to the prompt for the field. (The names of a class's fields may be customized in a configuration file. Perhaps the programmer wishes to rename the 'From' to 'Spender' for his/her particular application.)

An extended example

A display string such as this

```
<table>
[<tr><td>{p:FROM}: </td><td>][<tr><td>{FROM}</td></tr>]
[<tr><td>{p:AMOUNT}: </td><td>][<tr><td>{AMOUNT} ether</td></tr>]
</table>
```

would render to an HTML browser as this

```
From:    Account 12
Value:   100 ether
```

if the 'from' field evaluates to a value of Account 12, the amount field has been renamed to "Value," and the 'amount' field holds a value of 100 ether. The same class would display

```
From:      Account 12
```

if the amount field were empty.

Appendix C: The makeClass Configuration Files

In this appendix, we present the configuration file of all automatically created code, including all classes in the tokenlib library, and nine of the twelve classes in the etherlib library. Each configuration file is presented without further comment.

For the abilib library

```
==================== abi.txt ==========================
class:          CAbi
fields:         CFunctionArray abiByName|CFunctionArray abiByEncoding
includes:       function.h|parameter.h
destination:    ~/quickBlocks/src/libs/abilib/

==================== function.txt ==========================
class:          CFunction
fields:         string name|string type|bool indexed|bool anonymous|bool constant|
                  bool payable|string encoding|CParameterArray inputs|CParameterArray outputs
includes:       utillib.h|parameter.h
destination:    ~/quickBlocks/src/libs/abilib/

==================== parameter.txt ==========================
class:          CParameter
fields:         string name|string type|bool isPointer|bool isArray
includes:       utillib.h
destination:    ~/quickBlocks/src/libs/abilib/
```

For the etherlib library

```
==================== account.txt ==========================
class:          CAccount
fields:         addr addr|string header|string displayString|bool pageSize|int lastPage|
                  int lastBlock|int nVisible|CTransactionArray transactions
includes:       ethtypes.h|abilib.h|transaction.h
destination:    ~/quickBlocks/src/libs/etherlib/

==================== block.txt ==========================
class:          CBlock
fields:         addr author|string difficulty|string extraData|string gasLimit|string gasUsed|
                  hash hash|string logsBloom|addr miner|hash mixHash|string nonce|
                  string number|hash parentHash|string receiptRoot|string receiptsRoot|
                  SFStringArray sealFields|string sha3Uncles|string size|string stateRoot|
                  string timestamp|string totalDifficulty|CTransactionArray transactions|
                  string transactionsRoot|SFStringArray uncles
includes:       ethtypes.h|abilib.h|transaction.h
cIncs:          #include "etherlib.h"
destination:    ~/quickBlocks/src/libs/etherlib/
```

```
==================== blockchain.txt ==========================
class:            CBlockChain
fields:           CBlockArray blocks
includes:         block.h|etherlib.h
cIncs:            #include "account.h"
destination:      ~/quickBlocks/src/libs/etherlib/

==================== pricequote.txt ==========================
class:            CPriceQuote
fields:           int timeStamp|float open|float high|float low|float close|float quoteVolume|
                    float volume|float weightedAvg
includes:         ethtypes.h|abilib.h
destination:      ~/quickBlocks/src/libs/etherlib/

==================== receipt.txt ==========================
class:            CReceipt
fields:           hash blockHash|int blockNumber|addr contractAddress|int cumulativeGasUsed|
                    addr from|int gasUsed|CLogEntryArray logs|string logsBloom|string root|
                    addr to|hash transactionHash|int transactionIndex
includes:         ethtypes.h|abilib.h|logentry.h
cIncs:            #include "etherlib.h"
destination:      ~/quickBlocks/src/libs/etherlib/

==================== trace.txt ==========================
class:            CTrace
fields:           int gas|string returnValue|CStructLogArray structLogs|CStructLog last
includes:         ethtypes.h|abilib.h|structlog.h
cIncs:            #include "etherlib.h"
destination:      ~/quickBlocks/src/libs/etherlib/

==================== transaction.txt ==========================
class:            CTransaction
fields:           hash blockHash|int blockNumber|string creates|int confirmations|
                    addr contractAddress|string cumulativeGasUsed|addr from|int gas|
                    string gasPrice|string gasUsed|hash hash|string input|bool isError|
                    bool isInternalTx|int nonce|hash r|string raw|hash s|int timeStamp|addr to|
                    int transactionIndex|hash v|string value|CReceipt receipt|CTrace trace
includes:         ethtypes.h|abilib.h|receipt.h|trace.h
cIncs:            #include "etherlib.h"
destination:      ~/quickBlocks/src/libs/etherlib/
```

For the tokenlib library

```
==================== approvalevent.txt ==========================
class:            CApprovalEvent
baseClass:        CLogEntry
fields:           address _owner|address _spender|uint256 _amount|
includes:         logentry.h
cIncs:            #include "etherlib.h"
destination:      ~/quickBlocks/src/libs/tokenlib/
```

```
==================== approve.txt ==========================
class:          CApprove
baseClass:      CTransaction
fields:         address _spender|uint256 _amount|
includes:       transaction.h
cIncs:          #include "etherlib.h"
destination:    ~/quickBlocks/src/libs/tokenlib/

==================== createdtokenevent.txt ==========================
class:          CCreatedTokenEvent
baseClass:      CLogEntry
fields:         address to|uint256 amount|
includes:       logentry.h
cIncs:          #include "etherlib.h"
destination:    ~/quickBlocks/src/libs/tokenlib/

==================== createtokenproxy.txt ==========================
class:          CCreateTokenProxy
baseClass:      CTransaction
fields:         address _tokenHolder|
includes:       transaction.h
cIncs:          #include "etherlib.h"
destination:    ~/quickBlocks/src/libs/tokenlib/

==================== transfer.txt ==========================
class:          CTransfer
baseClass:      CTransaction
fields:         address _to|uint256 _value|
includes:       transaction.h
cIncs:          #include "etherlib.h"
destination:    ~/quickBlocks/src/libs/tokenlib/

==================== transferevent.txt ==========================
class:          CTransferEvent
baseClass:      CLogEntry
fields:         address _from|address _to|uint256 _amount|
includes:       logentry.h
cIncs:          #include "etherlib.h"
destination:    ~/quickBlocks/src/libs/tokenlib/

==================== transferfrom.txt ==========================
class:          CTransferFrom
baseClass:      CTransaction
fields:         address _from|address _to|uint256 _value|
includes:       transaction.h
cIncs:          #include "etherlib.h"
destination:    ~/quickBlocks/src/libs/tokenlib/
```

```
==================== deffunction.txt ==========================
class:          CDefFunction
baseClass:      CTransaction
fields:         string _str|
includes:       transaction.h
cIncs:          #include "etherlib.h"
destination:    ~/quickBlocks/src/libs/tokenlib/

==================== unknown.txt ==========================
class:          CUnknown
baseClass:      CTransaction
fields:         string _str|
includes:       transaction.h
cIncs:          #include "etherlib.h"
destination:    ~/quickBlocks/src/libs/tokenlib/
```

Appendix D: Classes in the etherlib libraries

Here we present the various classes in the etherlib library:

```
class CAccount : public CBaseNode
{
public:
    SFAddress   addr;
    SFString    header;
    SFString    displayString;
    SFBool      pageSize;
    SFInt32     lastPage;
    SFInt32     lastBlock;
    SFInt32     nVisible;
    CTransactionArray transactions;
    ...
};
```

```
class CBlock : public CBaseNode
{
public:
    SFAddress author;
    SFString difficulty;
    SFString extraData;
    SFString gasLimit;
    SFString gasUsed;
    SFHash hash;
    SFString logsBloom;
    SFAddress miner;
    SFHash mixHash;
    SFString nonce;
    SFString number;
    SFHash parentHash;
    SFString receiptRoot;
    SFString receiptsRoot;
    SFStringArray sealFields;
    SFString sha3Uncles;
    SFString size;
    SFString stateRoot;
    SFString timestamp;
    SFString totalDifficulty;
    CTransactionArray transactions;
    SFString transactionsRoot;
    SFStringArray uncles;
    ...
};
```

```
class CBlockChain : public CBaseNode
{
public:
    CBlockArray blocks;
    ...
};
```

```
class CLogEntry : public CBaseNode
{
public:
    SFAddress address;
    SFHash blockHash;
    SFInt32 blockNumber;
    SFString data;
    SFInt32 logIndex;
    SFStringArray topics;
    SFHash transactionHash;
    SFInt32 transactionIndex;
    ...
};
```

```
class CMiniBlock
{
public:
    SFUint32 number;
    SFUint32 timestamp;
    SFUint32 firstTrans;
    SFUint32 nTrans;
    SFUint32 gasLimit;
    ...
};
```

```
class CMiniTrans
{
public:
    SFUint32  index;
    bool      isError;
    SFUint32  gasPrice;
    SFUint32  gasUsed;
    SFUint32  gas;
    char      from [41];
    char      to   [41];
    char      value[41];
    ...
};
```

```
class CPriceQuote : public CBaseNode
{
public:
    SFInt32 timeStamp;
    float open;
    float high;
    float low;
    float close;
    float quoteVolume;
    float volume;
    float weightedAvg;
    ...
};
```

```
class CReceipt : public CBaseNode
{
public:
    SFHash blockHash;
    SFInt32 blockNumber;
    SFAddress contractAddress;
    SFInt32 cumulativeGasUsed;
    SFAddress from;
    SFInt32 gasUsed;
    CLogEntryArray logs;
    SFString logsBloom;
    SFString root;
    SFAddress to;
    SFHash transactionHash;
    SFInt32 transactionIndex;
    ...
};
```

```
class IPCSocket
{
public:
    FILE *m_fp;
    SFString m_path;
    int m_socket;
    ...
};
```

```
class RPCSession
{
public:
    IPCSocket m_ipcSocket;
    size_t m_rpcSequence;
    ...
};
```

```
class CRPCResult : public CBaseNode
{
public:
    SFString jsonrpc;
    SFString result;
    SFString id;
    ...
};
```

```
class CStructLog : public CBaseNode
{
public:
    SFInt32 depth;
    SFBool error;
    SFInt32 gas;
    SFInt32 gasCost;
    SFStringArray memory;
    SFString op;
    SFInt32 pc;
    SFStringArray stack;
    SFStringArray storage;
    ...
};
```

```
class CTrace : public CBaseNode
{
public:
    SFInt32 gas;
    SFString returnValue;
    CStructLogArray structLogs;
    CStructLog last;
    ...
};
```

```
class CTransaction : public CBaseNode
{
public:
    SFHash blockHash;
    SFInt32 blockNumber;
    SFString creates;
    SFInt32 confirmations;
    SFAddress contractAddress;
    SFString cumulativeGasUsed;
    SFAddress from;
    SFInt32 gas;
    SFString gasPrice;
    SFString gasUsed;
    SFHash hash;
    SFString input;
    SFBool isError;
    SFBool isInternalTx;
    SFInt32 nonce;
    SFHash r;
    SFString raw;
    SFHash s;
    SFInt32 timeStamp;
    SFAddress to;
    SFInt32 transactionIndex;
    SFHash v;
    SFString value;
    CReceipt receipt;
    CTrace trace;
    ...
};
```

```
class CWebAPI
{
private:
    SFString key;
    SFString provider;
    SFString url;
    ...
};
```

Appendix E: The forEvery Functions

The etherlib library contains a number of ‘forEvery’ functions which aid in the tranversal of the locally stored binary and array data. Here is a full list of those functions.

This first few forEvery functions traverse full blocks with one or more transactions that reside on disc:

```
bool   forEveryBlockOnDisc            (BLOCKVISITFUNC func,         void *data, SFUint32 start, SFUint32 count);
bool   forEveryEmptyBlockOnDisc       (BLOCKVISITFUNC func,         void *data, SFUint32 start, SFUint32 count);
bool   forEveryNonEmptyBlockOnDisc    (BLOCKVISITFUNC func,         void *data, SFUint32 start, SFUint32 count);
```

These two functions operate on a circumscribed version of the blockchain data which resides in memory during the traversal:

```
bool   forEveryMiniBlockInMemory      (MINIBLOCKVISITFUNC func, void *data, SFUint32 start, SFUint32 count);
bool   forEveryFullBlockInMemory      (BLOCKVISITFUNC func,         void *data, SFUint32 start, SFUint32 count);
```

These few functions ignore blocks and simply traverse the transactions directly. Note, only miniTransaction version of the data are availabe here:

```
bool   forEveryTransaction            (MINITRANSVISITFUNC func,  void *data, SFUint32 start, SFUint32 count);
bool   forEveryTransactionTo          (MINITRANSVISITFUNC func,  void *data, SFUint32 start, SFUint32 count);
bool   forEveryTransactionFrom        (MINITRANSVISITFUNC func,  void *data, SFUint32 start, SFUint32 count);
```