HEAT LEDGER

Heuristically Enhanced Asynchronous Transactions

White Paper

V1.01

D.M. de Klerk | Svante Lehtinen | Heat Ledger Ltd

2016-08-02

Table of contents

1. Introduction

2. Unlimited scalability
   - 2.1. Removal of embedded database
   - 2.2. Use of off-heap memory
   - 2.3. Split balances from transactions and blocks
   - 2.4. Chain of blockchains

3. Slicing of the blockchain
   - 3.1. General description
   - 3.2. HEAT token distribution and the genesis block
   - 3.3. Details of slicing
   - 3.4. Added security from PoS consensus
   - 3.5. Validating the most recent blocks segment

4. Networking improvements

5. Real-time external replication
   - 5.1. The replication layer
   - 5.2. Extending the message protocol

6. Real-Time Asset-to-Asset Exchange
   - 6.1. Bringing Decentralized Asset Exchange to web scale
   - 6.2. Colored accounts and private assets
   - 6.3. Custom Asset Exchange [A2AE]
   - 6.4. Crowfunding and Fiat trading gateways

7. Distributed Services Architecture
   - 7.1. DSA
   - 7.2. Services are not smart contracts
   - 7.3. Writing Distributed Services
   - 7.4. Service example
   - 7.5. What type of services can we expect
   - 7.6. Service operator's privileges and liabilities
   - 7.7. Ability to chain services

8. Smart (offline) vouchers

9. E2EE off-chain messaging

10. Account structure and aliases
    - 10.1. HEAT account identifiers
    - 10.2. Aliases
    - 10.3. Empty balance pruning and temporary accounts
    - 10.4. Minimum forger balance

11. Account control and multi-sig
    11.1. Protecting your stake
    11.2. Limit on amount transferred per day
    11.3. Multi-sig accounts

12. HEAT Rewards mechanism
    12.1. Two-tier rewards
    12.2. How to reward block generators (PoS model)
    12.3. How to reward blockchain storage (PoP model)
    12.4. Each segment gets the same reward

13. PoP Challenges
    13.1. Fair beyond fair
    13.2. The challenger account
    13.4. Challenge Generation
    13.5. The actual challenge

14. Conclusion

15. White Paper Version history

Introduction

HEAT Ledger is a self-appointed “Gen 3.0” cryptocurrency platform focusing on resolving the two most pressing pitfalls of the existing 2.0 and 1.0 cryptocurrency server software solutions; Low transaction rate throughput and the burden of hosting large blockchain files on any single p2p node.

Like the HEAT software itself, this document is work in progress that will be amended and expanded as the project evolves. V1.0 is designed to deliver a reasonably complete, quasi-technical treatise on the basic elements of the initial HEAT system available at the time of genesis block in September 2016.

HEAT is in part inspired by open source NXT[1] as it uses the time proven Proof-of-Stake consensus algorithm but tuned up for 25-second blocks[2]. Feature wise, many parts of the HEAT service offering and middleware solutions are based on those originally developed for FIMK[3]. On other parts the HEAT core is fully rewritten in Java, making it optimally suited for custom financial business applications by providing microsecond latency (up to 15-30 million writes per second), vastly increased vertical scalability, and superior failover resiliency when compared to legacy datasystems.

HEAT general technical highlights:

- *Written in enterprise friendly Java*
- *Based on secure, MIT licensed NXT for cryptography and p2p consensus code*
- *Architected with re-use for private chains in mind*
- *Optimized for speed and low memory usage*
- *Massively scalable through the use of not one single blockchain, but a chain of blockchains (only the last one in the chain is required on each node)*
- *On application level contains all the latest 2.0 cryptocurrency features and more, including custom asset-to-asset trading, distributed apps and end-to-end encrypted instant messaging.*

HEAT as a cryptocurrency employs radically new approaches to the way cryptocurrencies are built, the most notable of these being **complete removal of the embedded database**. Another major core change consists of changes to the mechanism the network layer works.

For storing the consensus blockchain, HEAT does not use single blockchain file ever increasing in size. Instead HEAT makes use of serialized blockchain files of a limited size, accompanied by small balance files. When the latest blockchain file reaches threshold size (of a few GB, specified at genesis block) the protocol will automatically switch to a new blocks file cryptographically linked to the previous blocks & balance files. The details to achieve this are in principle rather similar to how blocks are chained to each other.

[1] https://nxt.org
[2] Average block target
[3] http://fimk.fi

Through these radical changes to both the storage layer and the networking layer we estimate it is possible to sustain at least **1000 transactions per second** load **24/7 all year round**.

A theoretical constant rate of 1000 tps would produce 6.3 TB worth of archivable blocks files per year or roughly 3 new block files per day, all on commodity (affordable) hardware.

Would however the need arise to process even higher numbers - eventually approaching the VisaNet max level of 56.000 tps[4]) thanks to the vertical scalability of HEAT all that's needed to support these kind of boost rates are just stronger servers with more processing cores and RAM onboard.

Unlimited Scalability

Many of us have heard of the issue of "scalability" which most popular blockchains face. The difficulty of handling an ever increasing number of transactions on the blockchain puts a growing pressure on the resources of each node in the network.

Of the current generation of blockchains (eg. Bitcoin[5], Ethereum[6], Dash[7], NXT[8]) most have an embedded database where they store both all blocks data and all balances.

Those embedded databases give the user the ability to ask the running coin daemon for any block or transaction from the past. While in lack of a better solution this sure helps users, it in fact does not help the core p2p operations at all. Instead it comes at a brobdingnagian cost, especially for blockchains that skip the usage of more performant LevelDB[9] but opt for the featureful and thus slower H2[10] SQL database.

LevelDB has its limitations also; you cannot store anything in LevelDB that requires more than one hard disk, more practically the maximum size is as low as 100 GB.

As stated, the core p2p process has no use for the past blocks and transactions. All access that is really required on constant basis is access to the most recent 720 blocks. This is needed for removing transactions in the rare case of a blockchain fork resolution.

Removal of embedded database

Not embedding LevelDB or H2 to store transactions and blocks, account balances and the order books for all asset pairs, HEAT uses custom build storage and access solution based on Memory Mapped Files and tailored specifically to address the needs of a cryptocurrency node.

[4] https://usa.visa.com/dam/VCOM/download/corporate/media/visa-fact-sheet-Jun2015.pdf
[5] https://github.com/bitcoin/bitcoin
[6] https://github.com/ethereum/go-ethereum
[7] https://github.com/dashpay/dash
[8] https://bitbucket.org/JeanLucPicard/nxt/src
[9] http://leveldb.org/
[10] http://www.h2database.com/html/main.html

> *In computing, mmap(2) is a POSIX-compliant Unix system call that maps files or devices into memory. It is a method of memory-mapped file I/O. It naturally implements demand paging, because file contents are not read from disk initially and do not use physical RAM at all. The actual reads from disk are performed in a "lazy" manner, after a specific location is accessed.* [11]

Memory mapped files offer us the possibility to access disk stored data in random sequence and at near RAM memory speeds - without the need to load this data in RAM. (The idea originally inspired by *Ramchains*[12] implementation by jl777).

Use of off-heap memory

HEAT is written in Java and thus uses the Java Garbage Collector to manage and free app memory not used any longer. While Java offers much safer code with regard to memory management than do apps written in C or C++, this safety comes at a big cost. Applications written in C or C++ don't have a Garbage Collector and thus don't have to pause execution of other code while the GC is doing its thing.

Inspired by the High-Frequency Trading[13] industry we were lead to investigate the software components used by these HFT companies. HFT requires massive amounts of data that need to be accessed at lightning speeds. The needs of HFT systems and Blockchains systems overlap.

Our search lead us to Chronicle Software[14], more specifically their LGPL licenced product Chronicle Map.[15]

Chronicle Map/Queue Properties

1. High throughput, low latency, cross process, persisted key value store.
2. Off heap data storage, with microsecond latency.
3. Designed to be used in Java, in a pauseless, garbage free manner.
4. Better utilizes your existing hardware by offering vertical scalability.
5. Concurrent writes on one machine using single copy in memory.
6. Resilient writes (if a process dies, the last write is not lost).
7. Entirely embedded, no need for any additional processes.
8. 30 million updates per second with 500 million key-values on a 16 core machine, with a 64 MB heap and no minor collections.
9. Data is not lost even if the JVM crashes on the next line.

An important part is the application startup process, where it might be needed to scan all transactions on the blockchain to rebuild or validate the various running balances. Loading a multi-giga-byte block file into a memory mapped file is not a problem since this takes just about a second.

[11] https://en.wikipedia.org/wiki/Mmap
[12] http://wiki.supernet.org/wiki/Ramchains
[13] https://en.wikipedia.org/wiki/High-frequency_trading
[14] http://chronicle.software/
[15] https://github.com/OpenHFT/Chronicle-Map

Subsequently scanning through all the transactions is made extremely fast by using off-heap memory during the scanning process exclusively, eliminating the need for the Garbage Collector to even run.

Regular startups of HEAT without the need to scan transactions are near instant since all that has to be loaded are the balance files which require much less space than the block files.

Split Balance Storage from Transactions & Blocks

In cryptocurrency / decentralized ledger architecture you always have a list of transactions since the Genesis block. Reading all those transactions allows you to build up the balances for each account.

The storage needed for balances is much less than what we need to store the actual transactions. As an example let's imagine that 100,000 individual accounts have made a total of 100,000,000 transactions.

	Total	**Size (in bytes)**	**Total size**
Accounts	100,000	20 (key=8,val=12)	2 MB
Transactions	100,000,000	200	20 GB

As can be seen the total amount of storage needed to store the balances for 100,000 accounts is just 2 MB. Translate this to an account for all the ~3 billion[16] users on the internet and we'll end up with a balance file of just 60 GB.

HEAT produces several distinct files all loaded as memory mapped files and all serving a homogenous data type. There are files for unconfirmed balances, confirmed balances, unconfirmed and confirmed asset balances but also for open buy and sell orders. Each of these files comes with an accompanying *recovery* file that holds the transactional data of the last grouped update.

The recovery files serve dual purpose: we use them to rollback a transaction (grouped set of updates) and we use them to recover on startup when the app process has crashed or was stopped for another reason (system crash/power outage).

A Chain of Blockchains

[to be implemented after the HEAT main net has been successfully released]

In HEAT instead of one single blockchain we have segmented parts of the blockchain. Segments are numbered and there is no limit to the number of segments.

Balances are distributed separately from the blocks and transactions in the blockchain segment. Balance files are much smaller than blockchain files.

The balance files accompany each segment file, you need to scan a segment using the balance file as a starting point to get at the final balance for that segment. Contained in the balance file is a cryptographic checksum of the expected post-scan balance which should match - if there is no match the segment is invalid.

[16] http://www.statista.com/statistics/273018/number-of-internet-users-worldwide/

Let’s take an example of a new HEAT user, Bob. Bob downloads HEAT client-server package, and with this comes the initial balance file. Bob then also downloads all balance files for all segments (these are around 2 MB per segment). Finally Bob downloads the most recently sealed segment.

Bob can verify that all balance files that he has downloaded all link back all the way to the original genesis segment.

Each balance file comes with a SHA256 hash of the previous balance file, plus a 265 bit CHECKSUM of all the balances in the previous balance file.

So when Bob downloads all balance files (1/~10,000 the size of the blocks) his HEAT software can verify all balance files all the way to the last one for which he has downloaded the segment.

When Bob’s HEAT server software now applies all transactions he gets the starting balance from where he can start downloading transactions from the network.

Block height counter is tracked separately and it keeps increasing normally when a new blockchain segment is started.

Slicing of the blockchain

General description

Note that these parts are still in development, up for discussion and we cannot rule out changes in the details of the implementation plan.

Why is HEAT launched without the blockchain slicing operational? There is no great urgency to get the first block split on the main net, and that fact does not arise only because the mass of stored data is small right after release. More importantly - unlike cryptocurrencies driven by database storage - HEAT does not get significantly slower over time for the growing pile of archival transactions. The blocks and transactions are stored in HEAT in a way that imposes very small computational and memory-wise cost to the resources.

HEAT is already capable of supporting the first block segment of a huge size, while for Bitcoin or NXT for instance a blockchain of several hundred gigabytes up to 1 or more terabyte would require extremely powerful servers. All HEAT needs is just the disk space, relatively fast disks (ssd) and a modest amount of RAM. Servers with 2 to 4 GB RAM work fine.

This is possible since we store blocks and transactions all one after the other on disk, scanning the blocks and transactions in order literally means reading each byte on disk one after the other. No faster method than this exists in non-experimental computing up to date.

All the other parts of the HEAT framework are ready for block segment splitting. Already all balance storages have been constructed in ways to support this.

The practical benefits of allowing distributed archival of small sliced block files, and also proper implementation of the HEAT Proof-of-Presence incentive mechanism demand eventual implementation of the sliced blockchain structure not long after the launch of HEAT main net.

HEAT token distribution and the genesis block

From July 11th through August 8th 2016 HEAT has had its initial public distribution, or “Initial Coin Offering” [ICO] period, during which founders reserve their stake of the HEAT tokens. Each founder gets her share of HEAT tokens based on the proportional amount of price laddered HEAT ICO credits they have pledged through paying a freely chosen amount of BTC, NXT, FIMK or ETH to Heat Ledger Ltd’s ICO accounts - or through one of the 3rd party escrow agents used during the ICO.

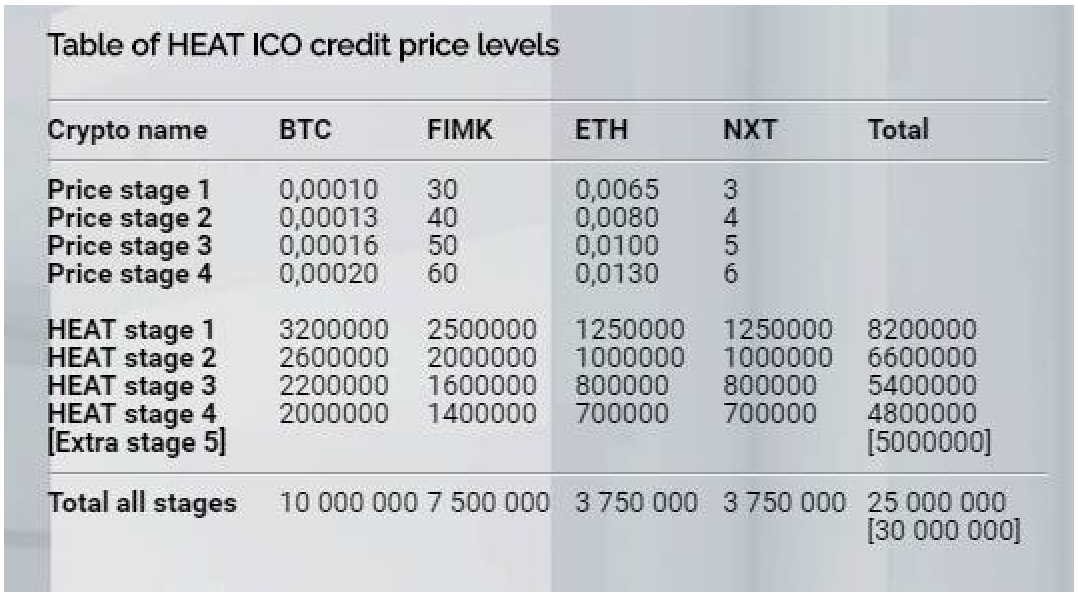

Table of HEAT ICO credit price levels

Crypto name	BTC	FIMK	ETH	NXT	Total
Price stage 1	0,00010	30	0,0065	3	
Price stage 2	0,00013	40	0,0080	4	
Price stage 3	0,00016	50	0,0100	5	
Price stage 4	0,00020	60	0,0130	6	
HEAT stage 1	3200000	2500000	1250000	1250000	8200000
HEAT stage 2	2600000	2000000	1000000	1000000	6600000
HEAT stage 3	2200000	1600000	800000	800000	5400000
HEAT stage 4	2000000	1400000	700000	700000	4800000
[Extra stage 5]					[5000000]
Total all stages	10 000 000	7 500 000	3 750 000	3 750 000	25 000 000 [30 000 000]

To distribute the 25 Million HEAT [or 30 million in case of full reservation] after the ICO period is over, Heat Ledger Ltd arranges the required facilities on the heatledger.com web site. Stakeholders need to go through an authentication process, consisting of creation of an individual HEAT founder account to which the HEAT tokens are transferred immediately after automated ownership verification of the original credit purchasing cryptocurrency account.

All these initial accounts are imported into a special piece of data file called the genesis block. In practice this part of the genesis block inclusion looks like a plain text file with entries on each new line where each entry contains a public key in HEX form and a balance displayed in the smallest unit of HEAT (HQT, Heat QuotienT).

In the HEAT server source code there is a hardcoded SHA256 hash with which we first validate the genesis plain text file before we generate the genesis block that starts the chain. The choice to place the genesis accounts in a separate file instead of hard coding them in the source code is to save application memory and to more easily switch to a different genesis block when applications of the HEAT software are used in corporate private blockchain products.

When the genesis block is in place, it will have an entry in the blocks segment file (sequentially stored data) and there will be an entry in the balance store for each genesis account. The balance of each account initially will be the amount assigned in the genesis plain text file.

Details of slicing

Now that the system is ready for use, stakeholders can start sending transactions and node operators can unlock their HEAT server and start forging blocks.

Once this process has run for a while we see some smaller data stores that hold balances, asset balances, open orders and public keys. And one much larger (several GB) data store that holds all transactions.

When the block slicing mechanism is deployed, all nodes on the network will know exactly at what block height a new blocks segment will be started. Whether we hardcode that height in the source code (easiest solution), publish that on the blockchain or use some other method where we base the moment of the split for instance on the actual size of all blocks on disk, remains to be decided.

At the time of a split, all live running nodes - active and connected at the moment of the split - will see a new file appear on their hard disk. If the previous segment was called heat-blockchain (for instance) the new file would be named heat-blockchain-2. Those online servers can now serve transactions from the past block segment to any peer that wants those as per the standard peer 2 peer operations.

Since balance stores belong to a specific blocks store, at the moment of the split all active nodes on the network will also generate new balance stores. There are several balance stores for all the different types of balances. The balance stores for the new block file will initially start as exact copies of their previous versions. This cloning as such happens extremely fast since the balance stores are based on memory mapped files and mostly use off-heap memory.

During the cloning process when we visit each balance in a predetermined order we create a checksum of all balances. The checksum is generated by building a SHA256 hash which is updated with each account id + balance combo. Also added to the SHA256 hash are all the final checksums that came out of each previous balance store before this one. So all previous checksums keep travelling forward to the future in each new balance store.

By this chaining of checksums we make it impossible to later present an altered balance file - simply because its checksum would never match the one for the genuine file we got during cloning.

In other words, the digest hash (checksum) from the previous step is added as a seed with all other previous checksums for each balance file before that, stored in the new balance file. This arrangement requires new users who come online the first time on the network later on - to download all the past balance files to be able to validate the last transaction segment. However the new user does **not** need to download the archived block segments, resulting in significant savings in the size of downloaded data from peers.

Added security from PoS consensus

To add a second layer of security to the sliced block files, we apply the strength of the PoS consensus protocol and the generally recognized inability for any attacker to obtain access to over half of all the total HEAT stake in existence.

After all nodes on the network have switched to a new segment and all have generated their new balance stores which include the checksum of the previous balance store - starting at block 1440 of the new segment and during 1440 blocks, each forger must include the carryover checksum in every block it forges. If any node on the network receiving a new block finds the carryover checksum does not match the actual correct checksum, the block is rejected.

The chance of forking the network is rather slim this way. The maximum rollback of 720 blocks has already been passed so the network cannot go back to a point before the segment split. The only nodes landing on a fork would be the ones who include a checksum that is considered false by the rest of the network.

A new user’s HEAT server software automatically verifies that all the balance files really belong together by calculating and comparing the balance checksums itself.

Let’s take an example: We are 20 block segments in the future, so this means the total chain size could have become perhaps larger than available space on the user’s hard disk. So downloading all segments is no longer even an option for him. What he can do is download the very latest completed blocks segment. Still a largish file of a few GB but well within the standards we are used to in the cryptocurrency scene.

The user’s server software also downloads all the past balance files, but since these are 1/10,000 or less the size of a blocks segment this has minimal performance impact. His HEAT server scans each balance file and before accepting them makes sure all the checksums match up all the way from the genesis to the final balance file.

Next comes the big moment; the user’s server needs to validate the final blocks segment it had downloaded from the network. It needs to validate all transactions and blocks in that segment, *but without access to all previous blocks and transactions*! **If** the protocol did require access to those past transactions, then by definition the whole mechanism would never work as a truly global scale crypto platform. Relying on having access to a virtually endless set of transactions does not scale under the technology available for humankind currently.

Validating the most recent blocks segment

HEAT has already shown it is by far the fastest public decentralized ledger available - a feat made possible only by its unique design and the use of custom designed storage and balance components.

But speed is not enough if we cannot safely ignore the many tera- / petabytes of transactions that would no doubt make up such a transactions network of global span. Let's walk through the process of our friend Bob validating the block and balance files for his single node.

1. Bob has all *balance* files (lets say 20 generations of them)
2. Bob confirms each balance file all the way back from genesis up-to-now all connect to each other (confirmed through checksums - each follow up balance store references its past store)
3. Bob however doesn't know if all balances are the real balances, this remains to be seen later
4. Bob downloads the most recent completed *block+transactions* segment
5. Bob however still cannot be sure that this is the 'real' data set, this still remains to be seen.
6. Bob downloaded block segment number 19, the network currently is at segment number 20
7. Bob now takes the balance store of segment number 18 and starts there.. The number 18 balance store should consist of exactly all the final balances on the HEAT network at the time the number 18 segment was created and should also be the starting state of the number 19 blocks file.
8. Bob now assumes the number 18 balance file is the real balance (which could still be false - but he'll find out later).
9. Bob can already confirm if the number 17 balance store has the same checksum that was written 1440 times by all forgers at the start of the number 18 segment. This way Bob could at least know that the full POS forging weight at that point in time agreed upon the checksum for that balance store.
10. Bob now starts scanning segment 19, which is filled with transactions. Each transaction is validated and applied, each time updating the final state of the number 18 balance store. Each new transaction will update the balance store.
11. When Bob is done applying all the transactions in the number 19 blocks store, if all is correct, his final balances should match exactly to the same balances that can be found in the number 19 final balance store.
12. Of course, it's still possible now that Bob was duped into downloading fake balances files and he could have downloaded fake blocks segments, which all match up.
13. But to create such fake matching balances and matching transactions, that is no simple task. It is highly unlikely an attacker could do that, but theoretically possible. To succeed with a fake blockchain attack he needs to not only convince the majority of the network of this fact, but also make sure the rollback required is less than 720 blocks, since that's the maximum allowed number of blocks in a blockchain reorg. Brute forcing and discovering and distributing such fake datasets will not only cost a lot of time, it also exposes you the moment you start the attack. For all practical purposes this level of security is sufficient.
14. Now Bob is pretty sure his balances and transactions are valid and he starts to join the peer to peer block distribution network.
15. On joining the network Bob will find himself at the start of segment 20, the network will start feeding him blocks and transactions. Since at the start of the number 20 segment there are 1440 blocks that all carry the checksum for the number 19 balance store, all signed and validated by the full POS weight of the blockchain, Bob can now be sure that all his balance files and blocks segments are valid.

Networking Improvements

For its Peer-2-Peer and API connections, NXT and its clones rely on Jetty[17] which is an embeddable tried and tested servlet container and web server solution, incepted in 1995.

Newer networking libraries and paradigms have however emerged. Where Jetty is a blocking network framework - which means it schedules one thread per connection - more advanced solutions like Netty[18] used by HEAT exist.

> *A part of the mentioned scalability of Netty is a direct consequence of its asynchronous design: It does not require a thread per request and is therefore able to handle more concurrent connections with less available memory compared to a thread-per-request approach. With less threads running on your server, the operating system will be less busy doing context switches and other thread related overhead. This can lead to a performance increase. In the case of Netty this seems to be true as it has found its way into businesses such as Twitter and Facebook which handle impressive amounts of concurrent requests. (source [19])*

Existing earlier research[20] has also shown the inefficiency of most of the cryptocurrency peer-to-peer protocols. A major case of such inefficiencies is the double (or multiple) transport of blocks and transaction data to peers that already have those blocks.

In HEAT we use a smarter protocol where peers don't transfer blocks data to other peers unless those peers explicitly indicate they want that data.

Another difference is that we strive to connect much more peers to each other at the same time. Again, as a reference NXT currently has a default of 20 connections that are kept alive to public peers. With HEAT and non-blocking asynchronous websocket connections over Netty we believe we can support between 1000 and 5000 active connections to other peers. The optimal standard number of active connections is something still under investigation at the time of this writing.

To further optimize network performance, HEAT will exclusively use binary messages between peers. This will significantly clip the amount network bandwidth consumed as compared to the current situation where messages are encoded in JSON. Memory consumption can be further lowered by passing the binary messages received over the network directly to the off-heap storage layer, decreasing the need for garbage collection.

[17] http://www.eclipse.org/jetty/
[18] http://netty.io/
[19] http://ayedo.github.io/netty/2013/06/19/what-is-netty.html
[20] On Scaling Decentralized Blockchains | http://fc16.ifca.ai/bitcoin/papers/CDE+16.pdf

Real-Time External Replication

By discarding the database from the HEAT core we have removed one of the major bottlenecks of the daily p2p and consensus operations. By no longer having to worry about later discoverability of transactions and blocks, we have been able to:

1. **Scale infinitely** - since there is no more need for each peer to on demand produce data from any transactions from the past
2. **Improve performance** - We no longer need to re-index the whole database table upon each insert, because processing new transactions now only requires storing them at the end of the list of existing transactions.

Through these implementations we however lose the possibility to perform powerful SQL queries of the blocks and transaction data using only a regular peer. To accommodate for this loss of functionality and even go way beyond, we have created the replication layer.

The Replication Layer

The replication layer is an event based optional and configurable event sink mechanism where you run HEAT together with a MySQL[21] [or another brand of] database server. This server can be either on the same machine or somewhere else on the local network.

MySQL support comes out of the box, but in turn it builds on a generic implementation that allows to write an event sink to any type of database server. All you need to add is a vendor specific Java class that provides all the database specific SQL queries for that database and HEAT will happily replicate to that database type.

Standard parts we real-time replicate include:

1. Confirmed and unconfirmed transactions. While the HEAT core handles and stores confirmed and unconfirmed transactions separately, through the replication abstraction we are able to produce a coherent collection of transactions consisting of all unconfirmed transactions at the start of the collection and all confirmed transactions behind that.
   This greatly eases creating a client view of all real-time transactions.
2. All balances for all accounts
3. All blocks metadata. We don't store the block contents or signatures, saving a lot of storage.
4. All public keys for all accounts
5. All messages either sent as regular message transaction or included as a message attachment to any type of transaction. Optionally you can configure HEAT with your private key and all messages are replicated encrypted. This gives someone running replication the possibility to store all his private data encrypted on the blockchain - yet have it replicated and unencrypted in real-time to his/her MySQL server for fast indexing, custom queries etc.
6. All assets, trades and orders. Orders are matched in real-time based on unconfirmed transactions. In case a new block indicates we have not matched our orders correctly according to our view of the unconfirmed transaction ordering, this is corrected instantly upon seeing the new block over the event sink.

[21] https://www.mysql.com/

Extending The Message Protocol

For commercial projects before the concrete plan for HEAT materialized, we had to handle a number of domain objects that had to be stored on the blockchain, yet also be available in real-time in our MySQL application server.

For this we have created an addition to the replication layer where you can create binary messages [either encrypted or not] recognized by the replication code and handed off to their correct handlers. These handlers have to be written by the user; handlers are Java classes that have an unique ID and that know how to interpret binary message input - and apply that data to the MySQL database.

The replication extensions in most cases need additional table definitions, for this out of the box there is a schema versioning solution where all you need to enter are your table definitions or updates. HEAT replication will take care of applying these updates when needed.

It turned out to be very simple to integrate binary message support into the HEAT HTML5 client framework. Things get even easier through use of **TypeScript** as the client language. TypeScript allows us to create interfaces that perfectly wrap and handle the binary data encodings.

For now HEAT has only used custom built message handlers. But this is not how we intend our users to utilize them. A big part of the handler work consists of manually entering code that interprets the binary data and translates that to variables (numbers, strings etc). This is tedious work and perfectly suited for automation. All that’s needed is a set of table definitions and some formalized message data structures.

We have not concluded exactly how to best expose the extended messaging protocol to allow anyone to run his own custom protocol on top of HEAT. Ideas range from Java helper classes to JSON to a custom DSL[22] based on JRuby[23].

Real-Time Asset-to-Asset Exchange

Bringing Decentralized Asset Exchange to web scale

Through the help of real-time replication presented in the previous section, we have been able to create a relatively efficient live view of the state of all orders in the HEAT Asset-to-Asset Exchange [A2AE].

Traditionally, asset exchange applications based on cryptographic ledgers are not suited for high speed trading. In fact they are usually anything other but real-time, due to the serious speed limitations of applying transaction data to blocks with a delay of up to several minutes before becoming available for further transactions.

[22] https://en.wikipedia.org/wiki/Domain-specific_language
[23] http://jruby.org/

In order to expose the HEAT A2AE on web scale to a massive audience, we have implemented a secondary web application that serves the APIs that the client uses to show the full trading UI. All of this is external to the functioning of the HEAT core and thus has no impact on the magnitude of orders and users the protocol can handle.

For a client prototype we have created a **Play!** Framework Application in **Scala** that connects to the replicated MySQL database to expose this real-time exchange data. Tests have shown we can serve huge numbers of users and process enormous numbers of orders all from a relatively light server.

NOTE: Real time asset exchange data is not new. In FIMK and **NXT+**[24] (See example of Supernet Asset over Virtual Exchange[25]) we have already created a mechanism called Virtual Exchange Layer which does a similar thing. The difference to HEAT is that whereas the Virtual Exchange Layer is an API that calculates the most recent order state on each request, with HEAT the final order state is always available at any time straight from MySQL (or whatever database you use for replication).

As a part of the process to implement a new generation scalable blockchain and bring most of the features of legacy trading platforms to blockchain, HEAT introduces mandatory *expiration of orders*. Expiration is achieved in a two step process; First the orders are recorded with an expiration timestamp between 1 second and 2,592,000 seconds (30 days) from order time. While the HEAT client UI and order matching middle layer ignore any expired orders at any time, such orders are completely purged in a blockchain rescan every 24 hours.

Colored accounts and Private assets

The unique colored account implementation that FIMK has provided since early 2016, enabling custom assets[26] pricing and trading, is ported to HEAT as is. The idea of a colored account is that you can create “colored coins” within the HEAT ledger by simply tagging an account instead of tagging a subset of tokens. As the creator of a color you can *transmutate* any HEAT you own into that color and change any colored HEAT back into HEAT again.

Once having created new colored tokens you can use all the features of HEAT. The features available work only as long as you are using them within a subset of accounts assigned to the same account color.

While all standard assets in the HEAT asset exchange are priced in HEAT, creating an asset through a colored account imposes the created asset’s price base for the particular account color (for instance EUR) instead of HEAT. This step alone gives birth to custom assets priced in another asset. However, to isolate the colored asset from the general HEAT token flow, all accounts wishing to trade that asset need to be again assigned the correct color by the asset issuer.

HEAT includes a yet another asset feature making it possible to mark an asset *private*, for both colored and standard assets. Private assets can only be traded by accounts selected by the asset issuer. In addition private assets can use custom order and trade fees charged from the users eventually broadcasting their buy / sell / cancel order transactions.

[24] https://github.com/fimkrypto/nxt-plus
[25] https://www.mofowallet.com/launch.html#/assets/nxt/12071612744977229797/trade
[26] https://lompsa.com/#/activity/fim/assets/latest

Custom Asset Exchange [A2AE]

As the term implies, Asset-to-Asset Exchange enables trading of custom tokens, priced in any other custom token. The pricing currency issue has generally been a problem child for the few crypto ledgers that have the asset exchange functionality available at all. In addition to FIMK, at least Bitshares[27] and more recently the Waves[28] technology have made it possible to price assets in any other token.

HEAT's initial version achieves the custom assets functionality through the colored accounts mechanism. We have implementation plan standing by for more direct native Asset-to-Asset trading, which will be deployed once the HEAT main net has been launched and is running in a stable condition for a while. At that time, the HEAT system will support multiple different technical routes to issue, price and trade custom assets, making it suitable for a wide variety of real world business applications.

Crowdfunding and Fiat trading gateways

The flexibility and robust processing capabilities of the HEAT A2AE are specifically fit for any kind of crowdfunding and share issuance applications. Evolved from traditional crowdfunding, projects crowdfunded through HEAT's system may issue tokens for investors and enable immediate or later phase trade of these tokens on the HEAT decentralized [or company's privately commissioned] crypto ledger. Large volume trading with instantly matched trades no longer produce any hindrance thanks to HEAT's HFT processing speed.

Colored semi-public tokens - or private proprietary blockchains - are the obvious choice to represent national fiat money [EUR, USD] against which real world assets are priced in. While it is technically possible to arrange fiat assets for any of the competing platforms mentioned previously, the obstacles to achieve real fiat trading on blockchain are more acutely related to regulatory and legal obstacles. Thus the point of entry is often too high - or rather obscure - for licensed money transmitter business to engage in mission critical fiat operations through experimental p2p public ledgers.

Heat Ledger Ltd's business model is to provide custom trading blockchains for corporate use, as well as low point of entry to the public HEAT blockchain for medium weight agile FSPs[29]. Through this incentive the company strives towards joint venture projects with licensed money transmitters, to provide a showcase of real world fiat applications used through the HEAT software. The non-technical details of this subject are outside the scope of this document and are discussed elsewhere when permitted by the business arrangements.

[27] https://bitshares.org/
[28] https://wavesplatform.com
[29] Financial Service Provider

Distributed Services Architecture (DSA)

NOTE: The name distributed services architecture is already in use http://iot-dsa.org/

HEAT answers the call for blockchain based distributed applications in its own ingenious way. We have named the technique "Distributed Services Architecture" [DSA]. With DSA HEAT users are able to write software applications in either Java or JavaScript and have these apps serve clients interactively using the blockchain as (encrypted) distributed state full memory space. DSA is based on proprietary stateful interactive communication protocol, indeed being very similar to the well known HTTP protocol powering the internet today.

With DSA in the HEAT core will allow the creation of a wide range of digital currency and p2p database services. The services can be offered in a decentralized and anonymous way and service providers can prove they have acted fair.

Anyone is free to offer any kind of mostly automated services by just offering them on their HEAT software node. Any of these services would normally require fairly technical setup which at least requires running and securing an always online server and writing software that handles the automation part.

The distributed services architecture solves the following difficulties normally faced with when developing and offering cryptocurrency or other digital services.

1. No need for web server - you run HEAT node and install your automated services on HEAT
2. No need to write the service's software: That is if you use one of the pre-built services and offer it as your own
3. Service operator can stay completely anonymous
    a. Instead of an HTTP web server, the DSA protocol allows you to run your service even by being online briefly only, for instance only a few minutes each day. The protocol would still allow users to interact with your service as if you were always online.
    b. HEAT can be run over TOR to even further mask the service provider's identity
4. No way to cheat for service operator
    a. The service implementation code can be made publicly available.Since all service inputs and outputs are on the blockchain they can all be verified by anyone with access to the blockchain.
    b. Imagine a service where you can exchange cryptocurrencies. When you interact with a DSA service and after first contact it gives you a price/quote for which it will exchange your cryptocurrencies. We now have a public record of that service provider handing you that quote. This helps the customer to inspect all the previous uses of the service (and how the service obliged to them). On the other hand the service that gave you the quote also uses it during the processing of the service requests. The stored record from the point of view of the service is basically an execution context that never expires - giving the service script access to all archived inputs and outputs as if they were simple variables to the service script.

5. Ability to chain services and create service networks: Distributed services are meant for reusability. The idea here is that eventually thousands of individual services are run on the HEAT blockchain, then new services can be constructed by re-using existing services and making them available to end users.

Services Are Not Smart Contracts

DS are by no means Smart Contracts, the main reason being that according to our definition of smart contract, a smart contract runs independently and runs on all peers in the network.

Distributed services are different in several ways.

	Distributed Service	**Smart Contract**
Is invoked by blockchain "events"	YES	YES
Receives its input over the blockchain	YES	YES
Returns its output over the blockchain	YES	YES
Runs at massive scale, individual services don't run on all nodes in the network.	YES	NO
Author can stay anonymous	YES	YES
Comes with built in, powerful payment solution	YES	YES
Can access data from outside the blockchain (exchange rates, betting scores etc.)	YES	NO
Can invoke external services or perform outside of blockchain actions (like sending email or wiring EUR to a receiver)	YES	NO
Can be chained: You can build new services/contracts by combining existing services/contracts	YES	YES
Can interact with other blockchains	YES	NO
Running contract/service is free	YES	NO

Writing Distributed Services

Writing distributed services is made easy. While all the input and outputs to and from all services are encoded as dense binary messages, this does not mean creating services (especially from JavaScript or even better in TypeScript) requires any knowledge of this.

All the input and output marshalling is handled by the HEAT DS libs. What remains for the service creator is only implementing the service class itself.

A Service Example

Without going into too much detail let's describe how we write a very simple service that automates sending an email for us. Why do we need such a service you might ask?

It could be ideal as a building block for other services to include the option to send emails, for instance a service that provides Two-Factor Authentication (just an example).

The following service is written in TypeScript, a language created by Microsoft and that compiles to Javascript. The HEAT client framework is completely written in TypeScript.

```
class SendEmailInput {
  sender = ""
  recipient = ""
  subject = ""
  body = ""
}

class SendEmailOutput {
  success = true
}

class SendEmailService extends DistributedService {
  public api = {
    "sendEmail": {
      "input": SendEmailInput,
      "output": SendEmailOutput,
      "fee": (input: SendEmailInput) => {
              return input.body.length * 0.001;
            }
    }
  };

 public sendEmail(transaction: Transaction,
                 input: SendEmailInput,
                 callback: (output: SendEmailOutput) => void) {
   emailAPI.sendEmail(input.recipient, input.sender,
                   input.subject, input.body).then(
     (success: boolean) => {
              callback({ success: success });
            }
   )
  };
}
```

As you can see there isn’t a lot of code. Yet with this piece of code alone, compiled and installed in your HEAT services directory you could be serving anyone in the world who wants to send an automated email.

You would each time be receiving a fee payment in HEAT and your service will not run until the price you declared was paid in full to your account.

Besides sending email, there are numerous other - some seen before in other environments and some never thought about - small software services you or other users can create on the HEAT DSA. Those service implementers don't have to write the actual service code. Instead they could use your service (such as the email snippet above), or another service your service is networking with, and pass on the (minimal) cost for using your service to the customer.

What Type of Services Can we Expect

The sky is the limit with regard to the types services that can be expected to run on the HEAT network. However below is described an example set of services to broadly convey an idea of what is possible.

We list a set of service categories - although the list is limited in size note that these categories can be applied to almost any cryptocurrency and even every FIAT currency out there today.

1. **Oracle service**
   An oracle writes a (foreign) fact to the blockchain. Through an oracle it becomes possible to create a transaction on the HEAT blockchain that does not execute until an event outside the HEAT blockchain has happened. A Good example of using an oracle: Bob sends Alice HEAT tokens but the transaction will not execute before the oracle service writes to blockchain the transaction details of Alice about her BTC payment to Bob.
2. **Escrow service**
   Requesting a new escrow, both parties send money to the escrow. Escrow will not release funds before both parties agree. Could be combined with oracle service where escrow releases funds based on an oracle saying the requirements are met.
3. **Time-based data decryption (secret reveal) service**
   Based on a certain date [block height], transaction on the HEAT blockchain or some other real world event coming from an oracle this service decrypts and publishes / sends to pre-defined recipient some secret you gave it in advance.
4. **Co-signer service**
   For use with multi-sig transactions on BTC/ETH/HEAT/etc. The user of this service will use the service provider public key as a required co-signer of a multi signature transaction. When the requirement is met the service provider will add his signature to the multi-sig transaction which will then execute.
5. **Gateway trade bot service**
   With a gateway trade bot service you can buy/sell cryptocurrency on an exchange.
   An example of such a service would be users sending ETH to the service ETH address. When received the service will immediately sell the ETH for BTC and transfer the real BTC for the seller.
6. **Webshop simplification**
   If you sell digital products on the internet (music or video files for instance) you could create a service where you either return a link to download the digital product, or you could return a cryptographic key for the buyer to unlock the digital product.

Creating a new service implementation for one of the categories would consist of copying an existing implementation and only changing the details, which in some cases could be just a single line of code.

Service operator’s privileges and liabilities

1. Anonymity and the option for strong cryptographic identification - Service operators can remain anonymous. Because all contact with the service operator runs over the blockchain, additionally protected by TOR IP anonymizing relay layer if desired, there is no direct way to pinpoint the physical location of the service unless the service operator wants so.

On the other hand, since the service is identified by its HEAT account id this offers a strong non-breakable proof of identity when desired. Also hallmarking [public registration of your node IP address] a HEAT node can be used to provide higher level of cryptographic proof about the service operator’s identity and even location, where desired by the service operator himself.

2. Fraud proof technically, enhanced with decentralized reputation system - The service operator can of course cheat, for instance if an oracle service writes to the blockchain that a certain BTC transaction has happened while in reality it did not, the operator could that way keep funds that do not belong to him.

But anyone who wants can simply verify that what the oracle has written in such a case is actually false. This could even be automated from the HEAT client where with a single click your client downloads the service implementation code and runs it, using the same inputs since they are on the blockchain, thus revealing a fake result.

What also becomes an important factor when multiple service providers offer the same service, is how much trust or reputation the provider has. Simple feedback can be made available as comments attached to the service operator, or if required a more detailed reputation system may be created which will use the blockchain as its backbone.

Ability to chain services

Since service inputs and outputs are both on the blockchain it is possible to chain one service after the other. Because you call a service through a payment transaction that transaction could be made to execute only when certain requirements are met. This chaining feature enables the emergence of automated software service networks of massive scale, that could become sophisticated enough to provide services based on AI implementations or neural expert systems.

Smart (offline) Vouchers

Smart Vouchers is a unique HEAT feature that allows anyone with a HEAT account to generate possibly millions of transactions and hand those out to anyone in the world, all **for the price of a single transaction fee**.

Smart Vouchers are conditional transactions (payments, messages, asset transfers etc.) that can be generated offline, completely free of charge.

Unlike standard transactions Smart Vouchers are not stored on the blockchain. Instead they are generated offline and handed out one by one to their recipients in the form of either a very small file or a piece of text of varying length.

Either way, the voucher is a fully signed transaction which can be sent to the blockchain either by anyone (in case the voucher is not fixed to one account) or only by the account the voucher was created for.

The *smart* part of HEAT vouchers is their conditionality. While you can generate a million vouchers, if you make each voucher only valid if a certain alias has a certain value (or falls in a certain range of values) you now have a way to securely hand out vouchers in mass quantities and externally control their validity.

Use cases for smart vouchers range from decentralized lotteries to methods of more easily introducing new users to the HEAT network services through for instance rebate coupons. Smart vouchers allow business operators to hand out free HEAT assets but at zero cost if the user does not use his voucher.

E2EE off-chain messaging

HEAT contains native messenger that can send not only blockchain archived message transactions, but also off-chain messages free of charge between two endpoints. This kind of messaging is end-to-end encrypted, as the only way to decrypt the transmitted data is by using either the sender's or the receiver's private key. Off-chain messaging also does not leave an archive of the encrypted message (or metadata) to the blockchain, giving an impression of more secure way of communication than blockchain based messaging.

To use off-chain messaging, both parties of the message channel need to be online at the time of message sending. Gossip[30] protocol is used to transmit the messages. The gossip protocol is basically the peer-to-peer protocol that assures all transactions are distributed throughout the network among all peers. HEAT reuses that protocol to allow flow of selected messages over the same protocol, free of charge yet still limited and overseen, prioritized lower than transactions and based on the space available in the p2p transmit channel. HEAT's binary data format makes it highly efficient to transmit data between peers, so it is expected that there in nearly all cases is sufficient allowance to transmit off-chain messages throughout the network so that they reach the recipient.

Account structure and aliases

HEAT account identifiers

For new HEAT accounts users by default create human readable identifiers through the standard HEAT client. They have a choice of a public identifier or a private indentifier. Identifiers are usually email accounts - in the form of either inhouse user@heatledger.com, or external user@herownemail.com.

The email account format is chosen to facilitate user migration from legacy email software that often is insecure to use. External email identifiers are verified through 2FA (verifier email sent by Heat Ledger Ltd's server). The HEAT client's messaging solution embeds notification messages through standard email when desirable.

[30] https://en.wikipedia.org/wiki/Gossip_protocol

Public identifiers are stored on the blockchain and are visible to all. Private identifiers are different - instead of their plain text identifier, HEAT stores a 32 byte hash of the identifier. This way participants need to know the identifier before they can use it, yet we can assure delivery of transactions to the correct account through checking the identifier hash from the HEAT blockchain.

Advanced users are able to create standard HEAT accounts in a completely decentralized manner, by just publishing the 128-bit public key to the blockchain and without storing any extra identifiers to the blockchain.

Aliases

Aliases are free form text identifiers with some free form data associated to them. Aliases are always temporary, meaning they will be available for a while and will automatically be deleted after a certain number of blocks.

The reason it is not feasible to keep aliases around forever is that such a system would not scale. There is a virtual limitless demand for aliases of all sorts (to power vouchers for one thing). However, before an alias expires the alias assigner has a window of a certain number of blocks in which he can extend the alias registration by paying a standard alias tx fee.

Alias keys consist of 32 bytes, which is enough to fit a HEAT public key or basically any other unique key. Alias values consist of 64 bytes which is more than enough to fit even the highest of numbers. These sizes are much smaller than the original NXT aliases for instance, which had 100 byte keys and 1000 byte values. However in return for stripping the more extensive data store will be able to scale to massive worldwide usage of smart vouchers as a way to affordably support all sorts of off-line blockchain uses.

Such deployments can assist businesses to securely create perfectly scalable voucher solutions to power anything from lotteries, crowdsales, and ticket sales to distributed on-blockchain voting solutions which include strong (off-chain) identities of voters and no loss of HEAT transaction costs in case users fail to redeem their vouchers.

Empty balance pruning and temporary accounts

In HEAT balances and accounts are identified internally by their 8 byte [java long primitive] numeric id. Everywhere where account details are stored they always are identified by their 8 byte key. This is highly efficient means to process and store data.

On top layer however, HEAT accounts are identified by their 32 byte or 128 bit public keys, and in turn secured by their 256 bit private key.

To be able to internally link 8 byte account ids to 32 byte account public keys, a record of all public keys must be kept and track what account id represents which public key respectively. This is very costly in terms of storage, since each new account (unlike each transaction) has to permanently be stored and distributed over the network.

To mitigate this resource cost HEAT will actively start removing all public keys and balances for each account having had a balance lower than X (eg. 0.00000001 HEAT) for more than N blocks. If users don't want their public keys removed they have to make sure their balances never decrease below X.

The HEAT Client and API are designed to not accidentally overdraft the minimum balance. This function effectively allows complete deletion of accounts from the blockchain, and the use of temporary accounts in the form of vouchers for instance.

The limit for minimum balance is subject to increase in the future with the expansion in the number of HEAT accounts and consequently the size of the HEAT balance files that are mandatory for all new nodes.

Minimum forger balance

In HEAT you need at least 1 HEAT to be able to forge blocks. In practice an amount that small has negligible chance of forging a block, however it is reasonable to allow testing of forging setup for new users with low HEAT token balances.

HEAT account control and multi-sig

Protecting your stake

An extremely important aspect of holding possibly large sums of HEAT in your account, is to never lose the private key to that account. If you lose your key you either lose access to your funds, or someone else can get access to your funds. Unfortunately, over the several years that we have worked with and created new blockchain solutions we have seen it happen too often that large stakeholders lose considerable sums or all even of their holdings.

A very efficient method to prevent losses from large holdings is through account control. HEAT will incorporate the following two forms of account control either in the initial main net release, or in an update shortly after.

Limit on amount transferred per day

Large holders like exchanges should definitely use this feature. To place an account under account control happens through sending a special transaction. With this transaction you include an emergency public key and an amount that can be transferred without using the emergency key each N blocks.

If an unauthorized party obtains access to that account, the most he/she can take is the maximum allowed amount. If you detect the breach in time and use your emergency key you can transfer all funds to a new secure account.

Since these account control instructions go in a storage section that cannot be discarded later on, the transaction fee for such an account is higher than sending a normal payment transaction.

Multi-sig accounts

HEAT accounts can be assigned multi-signature status. This means that the owner of the account can send a special transaction which includes definition of other accounts that from then on have to also sign each transaction coming from that original account, for the tx to be accepted in the blockchain.

Since these multi-sig settings also go in a storage section that cannot be easily pruned later on, the transaction fee for this initial "mark-account-multi-sig" transaction is higher than for normal payment transactions.

HEAT rewards mechanism

Two-tier rewards

HEAT, like most public decentralized ledgers, wants to incentivise active network participants and uses a hybrid award system to reward contributing users;

1. Reward for generating blocks (Proof-of-Stake, POS). This involves running a full node, unlocked and with the user's stake applied to generate blocks. Users who run a block generating node generally need to have at least a moderate amount of HEAT on their account.
2. Reward for storing the blockchain (Proof-of-Presence, POP). Only the most recent blocks file is distributed amongst all p2p nodes, previous blocks files are not needed for normal p2p operations. HEAT incentives users to store and make available previously archived block files.

How to reward block generators (PoS model)

Block generation (“forging”) is a mechanism of creating new HEAT tokens and distributing transactions fees as a reward to node runners, based on the amount of HEAT tokens available on account unlocked (signed in) by a node. An account that generates a block is awarded yearly declining guaranteed reward, plus all transaction fees, until after 4 years when the guaranteed amount becomes 0 and the block reward consists of just the transaction fees for the block. We expect there to be sufficient amount of tx fees by then to incentivize suitable number of active PoS nodes.

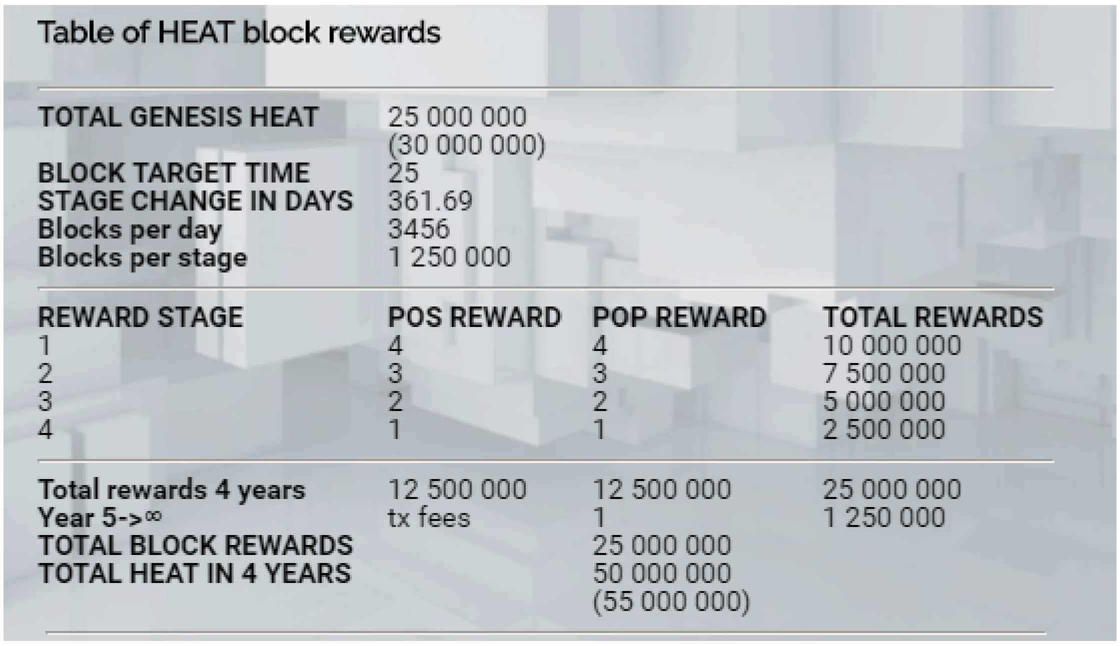

Table of HEAT block rewards

TOTAL GENESIS HEAT	25 000 000 (30 000 000)		
BLOCK TARGET TIME	25		
STAGE CHANGE IN DAYS	361.69		
Blocks per day	3456		
Blocks per stage	1 250 000		
REWARD STAGE	**POS REWARD**	**POP REWARD**	**TOTAL REWARDS**
1	4	4	10 000 000
2	3	3	7 500 000
3	2	2	5 000 000
4	1	1	2 500 000
Total rewards 4 years	12 500 000	12 500 000	25 000 000
Year 5->∞	tx fees	1	1 250 000
TOTAL BLOCK REWARDS		25 000 000	
TOTAL HEAT IN 4 YEARS		50 000 000 (55 000 000)	

How to reward blockchain storage (PoP model)

In HEAT we want nodes to host past blocks segments and past balance stores. It is however not required for a node to have stored these block segments and balance stores for it to be able to forge blocks. So the reward systems function independently of each other.

Before nodes are eligible for PoP rewards they must register a hallmark. Hallmark assigns a static IP to your account and stores this on the blockchain. Other peers on the network who connect to you over that same IP address now know what account runs that node and what his PoS weight (forging amount) is. The open source p2p protocol[31] readily favors hallmarked nodes over non-hallmarked when it comes to its choice of where to download blocks and transactions from.

Once you have registered your node's IP address through a hallmark, your node advertises exactly what past blocks segments you have archived and are making available on the network.

This registration of available segment serves 3 purposes:

1. It establishes an index for nodes on the network that wish to download that segment. Through your registration they now know your node is able to serve blocks from whichever segment it registered.
2. It serves as an indicator for nodes that are looking for the most profitable segments to host, since the most rare segments get the highest rewards.
3. The challenge protocol that demands proof of individual archival nodes uses this registry to randomly select nodes and segments for which it generates a custom challenge based on the hallmarked account.

[31] https://bitbucket.org/JeanLucPicard/nxt/downloads

After you have hallmarked your node you can register or unregister your node as having certain blocks segments. Having registered a segment (which are identified by their numeric index - the genesis segment being segment 0) automatically means you will participate in the PoP reward contest for the next block.

It is important that your node is online all the time you wish to participate in the PoP rewards, since the protocol penalizes you by skipping a turn in case you failed to respond to a challenge or in case you responded with a wrong answer to the challenge.

Each segment gets the same reward

An important goal of the PoP protocol rewards is to make an even distribution of all the various block segments available over the entire network. If one segment is hosted by fewer hallmarked nodes than another segment, the PoP reward mechanism automatically makes it theoretically more profitable to host that more rare segment.

The reward algorithm knows how many segments there are on the network. Initially there is one segment, this is the segment that contains the genesis block and the first set of blocks after that. When this segment is completed there will be two segments: One segment which is completed and one which is the active not yet completed segment.

During the period of the first segment, when no other segment has yet been completed, the protocol uses relatively small random sections of the active segment to create challenges for those parts of the blockchain. Through an evenly random lottery mechanism of all hallmarked accounts we select the account that gets to solve the challenge. From a point when the first segment is completed and gets archived, a different algorithm will be applied.

For scalability reasons, the PoP rewards are aggregated in batches of average 10 blocks, and rewarded to the finder of every 10th block (random, in average). The algorithm that runs at random intervals will start by randomly selecting one of the completed segments. This ensures that each segment is entitled to the same amount of POP reward.

When a segment is selected the protocol randomly selects a hallmarked account that says it has that segment. For this hallmarked account a challenge is created and published on the blockchain through a transaction.

If the hallmarked account solves the challenge correctly (if its response matches the question) the protocol rewards this account with the HEAT PoP reward. If it doesn't solve it (false answer or non-response) then that hallmarked account has to skip one next turn before it can solve a challenge again.

When configuring the HEAT server with access to your private key so it can sign transactions and setup your hallmark, all this challenge solving and responding happens automatically.

NOTE: To maximize secure usage of the private key with a locally installed HEAT server, it is possible to assign the POS weight of your cold wallet account through offline balance leasing. This way you could use a second blank account as the hallmark account and signature account, without ever exposing your main account’s private key to an online node.

PoP Challenges

Fair beyond fair

Challenges are always provably fair. Anyone can verify that the correct, random challenge was indeed correctly generated by the generator account and written properly to the blockchain.

You could run the same challenge creator algorithm and will see the challenge creator not only created the correct challenge with the correct solution as its answer, but also that the challenge truly questions the correct part of the blockchain and not some section the challenge creator just happens to have.

A challenge itself consists of a challenger account writing a challenge to the blockchain. The idea of the challenge is that anyone knows what the answer should be.

But what is not known is a secret included inside the challenge. The challenged account must decrypt the secret before getting access to the actual answer.

If the expected answer and the decrypted data are the same, this indicates the challenged account must have known the secret.

The challenger account

In early phases of the HEAT network, Heat Ledger Ltd developers will operate the single challenger account. Because this account is bound to strict protocol rules it could not be viewed as being autonomous or granting undue power to the developers. All it can do is do what the protocol allows it to do, or decline from its assigned tasks and do nothing. If nothing is done this immediately becomes clear to anyone on the network.

When the early operational phase has been passed, we will change the way the challenger account works. It will become a more decentralized process by allowing network users to also play the role of challenger. Being completely rule bound this will not impose a threat but make the challenger process more durable and better suited for a decentralized system.

It will for one allow for better storage scaling through segment distribution over multiple challenger accounts, so the single challenger does not have to hold all segments in order to create challenges.

Challenge Generation

There are several parts to the challenge generation algorithm. All parts are however under active development and might change based on future insights.

1. Selection of the challenge block (when do we do the challenge)
2. Selection of the challenge segment (what are we creating a challenge for)
3. Selection of the challenged account (who has to solve the challenge)
4. Selection of the challenge seed (random input to prevent pre-calculations)

1. Selection of the challenge block

Challenges are created on a random interval, every 10 blocks on average a new challenge block is selected. The selection of a challenge block is based on the block id, which is a number. A block is a challenge block if:

```
(BLOCK_ID_NUMBER % 10) == 0
```

This gives on average 10 hits but randomly distributed.

2. Selection of the challenge segment

Each challenge is supposed to select an evenly selected segment of all existing segments, the idea being that by rewarding for segments in an equal way we are sure of an equal distribution of segments over all nodes. This method guarantees that no matter how large the blockchain grows, there will always be high incentive to archive the most rare parts on the network, ensuring an always equal distribution of all past segments.

To select a segment we look at the number of possible segments to select from in the first place. If there are for instance 10 segments, each numbered from 0 to 9, we have to randomly select all with equal chance. For this we will also use the current block id number and select the closest segment based off that, again using modulo arithmetic.

3. Selection of the challenged account

Every possible account is registered as hallmarked account and also acknowledged whether they are hosting the selected segment. From the list of registered accounts one will be selected to be the challenged account.

To find the selected account we take a SHA256 hash of the current block signature and add to that the previous block signature. To the resulting hash we add the selected segment.

We then check each registered account and for each account create a SHA256 hash of its public key, added to that is the block signature. We take the last 8 bytes of each of these resulting hashes; whichever account has the highest number wins and is the challenged account.

4. Selection of the challenge seed

The challenge seed is a random unpredictable piece of data. For this we again create a SHA256 hash, we feed it the block signature, we feed it the challenger account public key, and we feed it the public key of the challenged account. The resulting digest is the challenge seed.

The actual challenge

At this point we know *when* to create the challenge, *what* the challenge will be about, *who* is challenged and what the *answer* of the challenge should be (the answer being the seed). Now it is time to create the challenge and write that challenge to the blockchain.

First the challenger account creates a checksum of the segment or part of that. The result is a 32 byte array (a very large number).

Now the challenger creates a SHA256 hash of the block signature, the challenged account public key and the checksum obtained in previous phase. The result is the secret that has to be published by the challenged account.

Next the challenger creates a SHA256 hash of the secret and together with the challenge details (what segment, what account) publishes this to the blockchain as a transaction.

If the challenged account responds with a transaction that contains the secret - which when put through SHA256 gives the same hash as the published hash - the protocol would consider that a challenge solved correctly.

CONCLUSION

The HEAT decentralized crypto ledger platform is an ambitious business project bringing together several dramatic server level changes, client side features and middleware system arrangements to form the basis to software solutions suitable for financial applications of today and the future.

Extensive in scope and going boldly where no crypto geek has gone before, we the HEAT Team look forward to bringing the maximum technological capacity of modern commodity hardware to the use of everyday distributed blockchain applications.

Eventually, this feat will be made possible only through the engagement of a supportive community. By careful technical and structural design - together with the enthusiastic users involved with the distribution of the HEAT system and token - we strive to achieve the status of HEAT being rightfully recognized a proper, first “Gen 3.0” cryptocurrency platform.

White Paper Version history

2016-08-01	v1.0	Initial publication
2016-08-02	v1.01	Minor wording changes, improved pdf formatting