Convex White Paper

Abstract

Decentralised networks offer the potential to enable peer-to-peer value exchange involving digital assets and services, protected at the protocol level by smart contracts and cryptographic keys that can be issued and managed on a self-sovereign basis - a vision that might well be called the "Internet of Value".

However, existing decentralised networks have notable weaknesses including poor performance, high energy consumption, long transactions confirmation times, vulnerability to "front-running" attacks and/or lack of truly decentralised security. Early Blockchain implementations, while successfully demonstrating the potential of the decentralised space, have fundamental limitations that make these weaknesses difficult or even impossible to resolve.

Convex is a decentralised platform for the Internet of Value that resolves these issues. Convex achieves consensus with a novel algorithm based upon merging Beliefs shared by peers using a function that is idempotent, commutative and associative - creating a system that provably converges to a stable consensus by forming a conflict-free replicated data type (CRDT). By including a system of economic staking, it is possible to guarantee convergence to consensus even in the presence of malicious / faulty peers (Byzantine Fault Tolerance). We call this combined scheme "Convergent Proof of Stake" (CPoS).

Convex augments the CPoS algorithm with an execution engine and storage system, building on the lambda calculus, immutable persistent data structures and content addressable storage. Combined with the consensus algorithm, this provides a fully decentralised, global computer capable of executing arbitrary smart contracts with decentralised ownership and security (the "Convex Virtual Machine").

Context

Towards a Digital Economy

Towards the end of the 20th Century, the foundational ideas were created for the Digital Economy. The start of the Information Age saw a wealth of innovation, as significant sectors of economic activity moved to the Internet:

- Online shops and marketplaces were developed, some of which became giant businesses (Amazon)
- Efficient access to information (Google)

- Entertainment, games, media and social activity (Facebook, Twitter, Instagram)

- Many business activities and tools (GitHub, Salesforce, Slack)

At the same time ideas were generated that hinted at the potential for economic value exchange *itself* to move to the Internet:

- Digital currencies were proposed and implemented

- Key concepts able to enforce terms on Digital transactions such as Smart Contracts were introduced.

- Innovations (both technical and legal) were developed to allow use of mechanisms such as digital signatures.

A problem with moving value exchange to the Internet, however, is that many parts of an economic transaction still rely on pre-Internet mechanisms: traditional fiat currencies, paper-based contracts and centuries-old legal systems for enforcement. Execution of complete transactions usually depends on trusting a single centralised entity to operate the interfaces between the Traditional and Internet worlds - handling legal issues, settling payments etc. Under such a model, economics of scale and network effects tend to favour a few commercial giants at the expense of smaller companies. This is a net loss to the economy: stifling innovation, excluding new competition, allowing monopolistic behaviour and "locking in" consumers without many practical choices.

The Internet of Value

We envision a system that enables value exchange mechanisms while solving the problems of traditional approaches - a network of economic agents that serves the needs of the digital economy.

The seven key properties that are essential to this vision are that it must be:

- **Global** - a single, global network for everybody, without artificial boundaries. Potential is maximised when everyone can transact with everyone else.

- **Open** - a decentralised, technology independent system where anyone can participate without asking for permission. No barriers to entry.

- **Automated** - able to support atomic, end-to-end transactions that are reliably executed using trusted and reliable smart contracts.

- **Secure** - protecting against all types of security threats so that users can be confident of the safety of their digital assets and transactions.

- **Extensible** - capable of supporting unlimited innovation in the types of assets and applications created. Like the Internet, it must not constrain what people can build, and it must make it easy to innovate.

- **Fast** - quick enough to confirm economic transaction in seconds, meeting the demands of consumer applications.

- **Cheap** - inexpensive to utilise, so that nobody is excluded by high costs and most kinds of applications are economically viable.

Convex has been designed from the ground up to provide these properties.

Applications

The Internet of Value's primary purpose is to enable **decentralised applications** that typically involve digital assets and value exchange. Just as anyone can create a website on the Internet, anyone can create a decentralised application for the Internet of Value. Convex is designed to make the process of building, operating and using such applications as simple and effective as possible.

There is no practical limit to the ideas that could be implemented given an open and extensible system. Some notable ideas include:

- Implementation of cryptocurrencies, utility tokens, and other forms of decentralised assets

- Economic transaction mechanisms (auctions, shops, exchanges) where terms and conditions are automatically guaranteed and executed by Smart Contracts

- Games and entertainment where rules include ownership of tradable digital assets

- Educational environments for collaborative and interactive programming

- Immutable records of documents / data provenance

- Publicly accessible databases and registries

Why is Convex needed?

Convex is needed because, despite the vast potential of the Internet of Value, a technology did not previously exist to enable it in a satisfactory way. We need a system that is *simultaneously* Global, Open, Automated, Secure, Extensible, Fast and Cheap. Previous systems fall short on one or more of these points.

Convex builds on ideas around decentralised technology popularised through blockchain innovations in recent years but was motivated by a desire to build something better than blockchains can offer. For example, Bitcoin does well on Global, Open and Secure, but is not Fast or Cheap.

Key additional motivations for the creation of Convex, while not strictly central to the mission of supporting the Internet of Value, include goals such as:

- Help the environment by supplanting systems based on Proof of Work.
- Make decentralised software development more productive, engaging and fun
- Establish new paradigms of decentralised programming around globally shared, consistent state

Prior Innovation

The space of decentralised technology has seen massive innovation in recent years, many of which have inspired the Convex project. It is not the purpose of this White Paper to chronicle all these exciting developments in detail, but some particularly significant events are worth noting:

In 2009, Bitcoin was launched by Satoshi Nakamoto, which demonstrated for the first time that a digital currency could be operated on a fully decentralised, secure network using a Proof of Work consensus algorithm. The ability to prevent "double spending" using a purely decentralised, online method was a revelation that hinted at the possibility of entire economic systems migrating to the Internet.

In 2015, Ethereum was launched, which build upon the ideas of Bitcoin but added a decentralised virtual machine (EVM) capable of executing Turing-complete smart contracts with a global state machine. This enabled a wave of innovations such as tokenisation of assets with smart contracts, and the first attempts at Decentralised Autonomous Organisations.

These innovations paved the way for significant experimentation in the space of digital currencies, tokenisation and cryptoeconomics. The space has attracted massive investment and seen vigorous innovation in recent years, hinting at the enormous opportunities presented by digital value exchange.

Technical Challenges

However, Blockchain technologies suffer from a range of issues which have proved hard to resolve. On the technical side, Ethereum founder Vitalik Buterin noted the "Scalability Trilemma" which is that is extremely hard to achieve the combination of:

- **Scalability** - Ability to offer performance comparable to traditional payment systems such as VISA
- **Security** - Resistance to attacks on assets and information integrity (such as double spending of digital currency)
- **Decentralisation** - Ability to operate free from centralised control by a single entity or group of powerful entities

Given that security is essential, and that without decentralisation blockchains offer no compelling reason to switch from centralised solutions, blockchains have generally **sacrificed scalability** to the extent that they are not practical for large scale use cases.

Other technical challenges became apparent over time. Some notable issues:

- **Energy wastage** - The use of "Proof of Work" consensus algorithms has resulted in vast and wasteful energy consumption. This is particularly apparent in the Bitcoin and Ethereum 1.0 networks, which rely on vast amounts of computing power dedicated to hashing.
- **Front-Running** is a particularly important problem in decentralised finance, where it is possible to steal value from others by quickly inserting a transaction before that of another user, and is exacerbated by the problem of long block times
- **Cross chain integration** presents a particular problem where different decentralised platforms provide different specialised capabilities but need to be integrated to form a combined solution. The problems of maintaining consensus, security, reliability etc. are magnified in such situations.
- **Latency** - The time taken for most blockchains to confirm final consensus is frequently too long to offer a positive user experience. This inability to provide quick confirmation and feedback is a significant barrier to mainstream user adoption of decentralised applications.
- **Upgradability** - Both networks themselves, and the specific implementations of smart contracts, are difficult to upgrade, in some cases requiring a

"hard fork" of the network.

- **State Growth** - Decentralised databases have an issue with *state growth*, defined as the increasing requirement for peers to store information that accumulates over time. Because on-chain data must be retained, potentially indefinitely (to satisfy future on-chain queries or smart contract operation), this imposes increasing economic costs on Peer operators. The economic costs of this often do not fall on those responsible for creating new state - leading to an instance of "the tragedy of the commons". Over time, this may make it impossible for normal machines to run a node, effectively halting decentralisation.

Convex presents a solution to these challenges, and as such we believe it allows a significant evolution "Beyond Blockchain" to deliver the Internet of Value. The remainder of this White Paper explains how we achieve this.

Convex Overview

The Convex Solution

Convex solves many of the technical challenges of Blockchains. With reference to the Scalability Trilemma, Convex offers:

- **Internet-scale performance** - Convex offers the capability to operate at transactions volumes comparable to or greater than centralised networks like VISA transaction levels, even *before* scalability solutions such as Layer 2 solutions, state sharding or optimistic lookahead approaches are applied. Early benchmarking suggests peers running commodity hardware may be able to handle in excess of 100,000 transactions per second.

- **Byzantine Fault Tolerance** - Convex meets the strongest possible threshold for security under the model of Byzantine threats. Consensus formation is guaranteed (and stable) as long as at least 2/3 of the effective voting power of the network follows the protocol honestly.

- **Full Decentralisation** - The network operates under a permissionless Peer-to-Peer model: Anyone can operate a Peer in the network, anyone can submit a transaction for execution, and transactions cannot be censored (subject to the usual security assumptions).

But Convex is not just a faster Blockchain - it is a platform for building digital economic systems. As such, it combines several capabilities that together enable construction of new classes of applications.

Some technical highlights of the Convex design that support such applications include:

- **Actors**: Programs that execute autonomously in the Convex environment with deterministic and verifiable behaviour, suitable for managing assets and enforcing Smart Contracts

- **Convex Virtual Machine (CVM)** - a fully Turing complete programming and execution environment. By designing the execution model around the Lambda Calculus, the CVM supports functional programming capabilities, with novel combination of language and runtime features to facilitate writing decentralised applications. We implement a working Lisp compiler "on-chain".

- **Decentralised Data Value Model** - A data model supporting powerful features such as orthogonal persistence, memory accounting, incremental data sharing and cryptographic verification

- **Performance**: High throughput (many thousands of transactions per second), low latency execution (zero block delay, ~1 second or below transaction confirmations)

- **Security**: Cryptographic security for control over all user accounts and assets, with byzantine fault tolerance at the level of the decentralised network.

The main sections of this White Paper describe the key subsystems of Convex that make all this possible:

- The **Consensus Algorithm** which enables the Convex network of Peers to agree on a consistent, replicated state of the world - essential to provide reliable confirmation of transactions in a decentralised system

- The **Execution Engine** performs computations necessary to implement secured transactions on the Convex Network - it is the mechanism that updates the Global State and enforces Smart Contracts.

- The **Storage System** enables Peers to store and manage data volumes with a level of scale and performance necessary to support a high throughput of transactions and data volume.

- The **Memory Accounting** model keeps track of data sizes stored and transferred by users, allowing Convex to correctly align economic incentives to make optimal use of on-chain memory.

We summarise these areas below:

Consensus Algorithm

Convex, like other decentralised systems, depends upon a consensus algorithm to ensure that everyone agrees on a single version of the truth - this is a precondition for any decentralised economic system that needs to enforce ownership of digital assets.

Like blockchain technology, Convex allows **Blocks** consisting of multiple transactions to be submitted to the network. In contract to blockchains however, the blocks are not linked to previous blocks - there is no "chain" as such. Relaxing this requirement enables Convex to handle new block submission concurrently, significantly improving performance.

The role of the consensus algorithm is to create an **Ordering** of blocks. A stable order solves the famous "double spend" problem by ensuring that only the first transaction is able to spend any given funds or assets. Any later transaction that attempts to spend the same funds will fail.

The algorithm operates by implementing a novel variant of a **Conflict-free Replicated Data Type** (CRDT), which can be proven to converge to a stable consensus through a few rounds of random gossip between Peers. This approach is efficient, robust to temporary failures, and provably secure even in the presence of malicious or faulty peers (i.e., it is "Byzantine Fault Tolerant" under reasonable security assumptions).

The Convex consensus algorithm also makes use of **Proof of Stake**, a mechanism by which peers are required to deposit an economically significant stake to ensure their good behaviour and be granted participation rights in the consensus protocol. This avoids the wasteful use of resources and energy that plagues systems based on "Proof of Work". As well as offering substantially improved performance, this means that Convex presents an environmentally friendly alternative to previous models such as Bitcoin or Ethereum.

Execution Engine

Convex implements a full virtual machine for smart contracts, the **Convex Virtual Machine (CVM)**, designed to facilitate digital economic transactions. Given an initial State and an ordering of Blocks (and therefore transactions) from the consensus algorithm, the CVM can process the transactions and compute a new updated State. The latest state contains information of interest to users of the Convex network, in particular the record of ownership of digital assets.

The CVM has the capability to execute arbitrary, Turing-complete **smart contracts** which in turn can be used to express the logic of digital assets and decentralised applications.

Importantly, the CVM operates on a **Global State** - execution of Blocks and Transactions (once these are in consensus) is equivalent to creating a new state through a state transition function.

Some particular innovations of interest to facilitate the development of decentralised applications:

- **Decentralised Data Values (DDVs)** - A system of data types and structures enabling efficient and secure replication of data across the Convex network, and supporting the implementation of the CVM. The CVM works with a wide variety of data types enabling construction of powerful applications with optimised performance.

- **Convex Lisp** - A powerful language where CVM code is itself expressed as Decentralised Data Values. The compiler itself executes on-chain - giving developers and Actors the power to construct, compile and deploy new actors on-chain without external tools. This enables systems of on-chain MetaActors - actors who can autonomously create and manage other actors.

- **Scheduled Execution** - The protocol allows for deterministic execution of Actor code at any future point in time. This allows for more advanced, time-based processes to be implemented on chain (without such a feature, smart contracts would need external systems and events to trigger execution at specific times, such as the Ethereum Alarm Clock)

- **Execution Worlds** - Each account on the network (external user or Actor) is granted a secure, scriptable code execution environment with its own database. This enables highly interactive use of the CVM by advanced users.

Storage System

Convex implemented a novel storage scheme, specifically designed to support the requirements of Convex DDVs. Key features of this system include:

- **Content addressable storage (CAS)** - The key for every value in the database is the cryptographic hash of its encoding. This means that given a single root storage hash, an entire Directed Acyclic Graph (DAG) of values is directly reachable by recursively fetching nested values.

- **Smart References** - references to data that can be lazily loaded and verified (via CAS), allowing just a small, required subset of data to be accessed on demand.

- **Orthogonal Persistence** - Decentralised Data Values used in Convex (such as the CVM state) are stored in a virtual database which may be much larger than main memory and is completely transparent to the user. This opens up interesting opportunities for future scalability and

sophisticated Actors capable of working with large databases.

- **Novelty Detection** - The design of the storage system enables Convex to detect *novel* information when it is written to storage. This is important to reduce bandwidth requirements: only novel information will typically need to be broadcast to the Peer network.

- **Proofed Persistence** - Certain proofs relating the the validation of data are persisted along with the data itself. This is an important optimisation: Entire large data structures can be verified in O(1) time by checking the cached proof.

An important feature *excluded* from the storage system is that of "update". Once written, data values are immutable and cannot be changed. This limitation is appropriate given that keys are cryptographic hashes of value encodings: finding a different data value that maps to the same key would require breaking SHA3-256. However, this exclusion is also an advantage: it reduces the need for more complex database features such as index updates

The database engine itself is called Etch. Etch is an embedded database engine optimised for these specific requirements. We believe that building a customised engine is a worthwhile investment, because of the specific feature requirements and performance improvements possible - at least an order of magnitude faster than using more traditional, general purpose databases.

The Storage System supports optional garbage collection for Peers that wish to compact storage size. A Peer is only required to maintain the current state, and a short history sufficient to participate in the consensus algorithm. Of course, Peers may choose to retain additional information for historical analysis.

Memory Accounting

Decentralised databases have an issue with *state growth*, defined as the increasing requirement for peers to store information that accumulates over time. Because on-chain data must be retained, potentially indefinitely, to satisfy future on-chain queries or smart contract operation. There is no option to discard data arbitrarily: a good peer cannot do so and maintain correct participation in the consensus protocol.

This growing demand for storage space presents a significant problem.

- It creates a requirement for peers to incur increasing storage costs over time, for data that must be retained indefinitely

- There are perverse incentives at work: a user might pay a one-off code to store data on-chain, but does not bear the cost of indefinite future storage

(which falls on peer operators)

- Over time, careless use of on-chain storage may make it impractical for a typical individual to operate a peer with normal computer hardware and storage capabilities

- This problem might be manageable for low-transaction-volume platforms, but is clearly unacceptable for systems such as Convex that are designed to handle high volumes of transactions for sustained periods of time

Convex implements a novel solution of Memory Accounting to help manage the problem.

- Each user is given a Memory Allowance, which is a fully fledged second native currency on the Convex Network

- Memory Allowance is consumed when on-chain storage is allocated, and released when stored objects are deleted (this can be efficiently tracked by careful integration with the storage subsystem)

- A automated Memory Exchange Actor is provided that maintains a pool of liquidity with memory available for users to purchase. This regulates the maximum size of the on-chain state based on supply and demand. This pool can be increased over time to allow for reasonable state growth, improvements in technology and to discourage hoarding.

- By this mechanism, a fair market price for memory is established that creates an economic incentive for efficient memory usage.

Technical Description

Design Rationale

It is worth reflecting on the logic behind the design of Convex: this logic has driven the majority of key design decisions in the construction of the Convex system.

Convex works on the principle of proving a globally shared state on a permissionless decentralised network which executes instructions (transactions) on behalf of users: a "public computer" that everyone can access but where nobody has absolute control.

The motivation for the development of a system of this nature is because it can act as the foundation for the Internet of Value - a system of decentralised value exchange built in the open spirit of the original Internet. But is this necessary?

Could it not be done in a simpler way? In this section we argue that these capabilities are necessary and sufficient (given the additional obvious assumption of adequate performance).

It is important to consider why the Internet itself does not already function as an "Internet of Value". The key insight here is that the original Internet is primarily a **stateless** system that enables communication between different participants. A system of digital value exchange requires **state** - at the very minimum it must be able to record the ownership of digital assets (in Bitcoin, for example, this state is manifested in the set of UXTOs). While clients and servers on the Internet individually store and manage their own state, such state is inadequate for decentralised value exchange because it is susceptible to corruption or arbitrary modification by the party that controls the relevant network nodes. At scale, such centralised state cannot be trusted.

If we are unable to trust centralised actors as the sole arbiters of truth regarding the state of assets, the solution must therefore involve decentralised verification. It must furthermore ensure **consensus** in order that the whole network sees the same verified state - assets cannot reliably be used for value exchange if there is ambiguity over ownership (the "double spend" problem). To provide the basis for a global, open network, the consensus itself must be **global** and available to all participants.

Because we wish to ensure openness and avoid the issue of centralised control over the system, it must also be **permissionless** in the sense that any actor can participate on an equal basis and not be excluded or censored by other participants wielding excessive power.

However, This ideal of decentralisation presents the problem that some actors in the system may be malicious and actively attempting to defraud other actors - a significant problem when valuable digital assets are at stake. Furthermore, even if not malicious, software or hardware failures can potentially disrupt the system. We therefore require the property of **Byzantine Fault Tolerance** - which can be informally characterised as resistance of the consensus mechanism to malicious attacks and failures. Theoretical results (e.g. the seminal work by Leslie Lamport) have demonstrated that such consensus requires 2/3 of participants to be good actors - that is, byzantine fault tolerance is possible to achieve as long as no more than 1/3 of actors are malicious or faulty. We would like the Internet of Value to operate with a consensus algorithm that achieves this theoretical optimum level of security.

We now consider the nature of Digital Assets, and what is necessary to implement them. A key observation is that for assets to be meaningful, they must obey rules. The most obvious example is that of ownership: only the owner of an asset should be able to use it in an economic transaction. But other rules may also apply: for example a financial option includes a right to "exercise" the option in exchange for some other underlying asset. Since assets only have value if owners trust that their rules will be enforced, we need a system of encoding and

executing these rules in an automated, verifiable way as part of the decentralised protocol - such rules must be implemented as **smart contracts**.

While it would be possible to create a simple system of smart contracts that tackle many useful applications without full programmability, the Internet of Value calls for extensibility to new forms of assets that may not be originally anticipated by the designers of the network. We therefore require a **Turing complete** smart contract execution model (capable of performing any computation) if we are to avoid the risk of future limitations preventing important digital asset classes from being created.

To ensure that only valid transactions are executed on digital assets with the authorisation of the owner, we need a secure way to validate the authenticity of transactions. This is fortunately a well-studied problem that can be solved with cryptographic techniques, and in particular **digital signatures**, where the authenticity of a transaction can be validated through the use of a secret private key held by users, and a public key that is visible to all.

Maintenance of the network consensus invariably requires resources: powerful servers with compute capability, significant storage and network bandwidth are necessary to operate the consensus algorithm and execute transactions 24/7 at global scale. These resources are ultimately not free, and to compensate their operators for the economic cost of their services there is a need to impose **transaction fees** for usage. Without some economic cost of transacting, the network could be swamped by low-value or malicious transactions that consume excessive resources. The need to charge a transaction fee naturally leads to the conclusion that some form of **native currency** is required, being a digital currency that is valid to pay for the usage of network services (which can be enforced at the protocol level).

Finally, we note some practical considerations. Information will need to be durably maintained and frequently communicated as part of the consensus algorithm and in serving users of the network, which requires systems for **storage** and **transmission** of data. Ideally, such systems need to be **efficient** to provide necessary storage, and they must ensure **integrity** to allow recovery from faults and malicious tampering.

The remainder of this White Paper describes the technical implementation of Convex, which implements all the above capabilities in order to provide a foundation for the Internet of Value.

A note on Values

As a distributed information system, Convex must deal with the representation of data. We therefore rely frequently on the definition of a Decentralised Data Value (DDV) which has the following properties:

- It is immutable: the information content cannot be changed once constructed

- It has a unique canonical Encoding as a sequence of bytes, which is used for both network transmission and storage

- It can be assigned a unique Value ID (VID) which is defined as the SHA3-256 cryptographic hash of the value's Encoding. This serves multiple purposes, most importantly for cryptographic verification and as a key for various data structures (e.g. hash maps and Content Addressable Storage).

- Other DDVs can be recursively nested inside - in essence forming a Merkle Tree, which becomes a Merkle DAG given structural sharing of identical children.

Where we refer to "value" or "data value" in this document, we are generally referring to DDVs.

Consensus Algorithm

Peers and Stake Convex defines the set of actors that participate in the consensus algorithm as **Peers** in the Network.

Anyone may operate a Peer by providing an economic **Stake** in Convex Coins. The size of the Peer Stake determines the relative voting power of the Peer in the consensus algorithm, but it is locked up and at risk of forfeiture in the case that bad behaviour is provably detected. Stake could also be appropriated by malicious actors if the Peer does not maintain strong security for their system (in particular the security of the Peer's private key). Requiring a Stake is therefore a key aspect of the economic incentive for Peers to maintain the security of the network.

As a reward for helping to operate and secure the network, Peers are entitled to a share of fees for transactions executed on the network, plus other incentive pools provided by the network. This reward is proportional to Peer stake, again creating a positive incentive for Peers to provide more Stake and greater security.

State The key task of the Peer Network is to securely store and update the **Global State**.

The State is a representation of the complete information in the CVM at any point in time. The Convex Network operates as a globally replicated state machine, where new updates cause changes to the current State. Updates are defined on a globally consistent basis according to the sequence of transactions confirmed through the Consensus algorithm.

The latest State of the CVM network after all verified Transactions in consensus have been executed is called the **Consensus State**.

The State is represented as an immutable Decentralised Data Value (DDV) that includes:

- All Account information and balances
- All Actor code, static information and current state
- All information relating to active Peers and staking
- Global information (such as the latest Block timestamp)

Since it is a DDV, it follows that a State is a Merkle DAG, and has a unique Value ID. This means that if two Peers compute a State update and announce the VIDs, it is possible to immediately validate if they computed the same resulting State.

Transactions and Blocks A **Transaction** is an instruction by any network participant (typically users of client applications) to affect an update in the Consensus State. Transactions are digitally signed to ensure that they can only cause effects in the Consensus State that are authorised for the holder of the corresponding private key.

Transactions are grouped into **Blocks**, which contain an ordered sequence of Transactions and some additional metadata (most importantly a Block timestamp).

The inclusion of Transactions in Blocks is at the discretion of the Peer to which they are submitted. Users may choose any Peer, but normally should prefer to submit transactions to a Peer that they trust to behave correctly, since this discretion could be abused by the receiving Peer:

- The Peer could ignore a Transaction and neglect to propose it for consensus
- The Peer could insert its own Transaction(s) before the user's transaction, potentially executing a front running attack

Reduction to the Ordering Problem Consensus in a decentralised state machine can trivially be achieved with the combination of:

- Agreement on some initial genesis State: `S[0]`

- Consensus over the complete ordering of Blocks of transactions `B[n]`

- A deterministic State Transition Function, which updates the State according to the Transactions contained in a Block: `S[n+1] = f(S[n],B[n])`

This construction reduces the problem of generalised consensus to the problem of determining consensus over Block ordering. The CVM execution environment provides the State Transition Function, which is orthogonal to the consensus algorithm but provides the deterministic computations to compute the new Consensus State (given any Ordering of Blocks in consensus).

We define the **Consensus Point** to be the number of Blocks confirmed by the consensus algorithm in the Ordering, and the Consensus State is correspondingly the State obtained after applying the State Transition Function up to the Consensus Point.

The hard problem that the consensus algorithm needs to solve is determining the **Ordering** of Blocks given the potential presence of malicious actors who may seek to profit by changing the order of transactions (e.g., a "Double Spend" attack).

Block proposals Traditional blockchain solutions have focused on mechanisms to determine which participant gains the right to propose the next block, which includes a hash of the previous block in order to extend a linked chain of blocks. This was the basis for the original Bitcoin Proof of Work algorithm (which used the ability to mine cryptographic hashes as the basis for allowing a miner to publish a block and claim the corresponding block reward).

Approaches that involve selecting a "leader" to publish a new block have some notable problems:

- It is a complex task to determine which participant should be the next leader, at least in a way that simultaneously works efficiently, provides security in the presence of potential malicious actors, and is guaranteed to make progress in cases such as leaders becoming unavailable.

- Including the hash of the previous block in a chain creates an inherent data dependency that limits the ability to propose blocks in parallel and increases latency - each leader must build upon the work of the previous leader sequentially, which implies a minimum lower bound on the block time (given fundamental physical constraints).

- It is necessary to make sure that the leader possesses the transactions that should be included in a Block. This implies the need for a mechanism to transmit transactions across peers prior to block creation (e.g., with the "mempool" used by Bitcoin to share pending transactions), which

in turn implies additional communication costs and many opportunities for attackers to launch "front running" attacks on transactions that they observe.

Convex therefore eschews the idea of selecting a leader. We maintain the principle that **any Peer may propose a new Block at any time**. This "leaderless" approach has some important desirable consequences:

- Blocks can be **instantaneously proposed**, without a Peer having to become a leader, perform any expensive computation, or wait for confirmation on any previous Block.
- Users can **select a trusted Peer** to publish their transactions, rather than having to forward them to a leader that may not be trustworthy. As previously noted, this is an important mitigation against censorship and front-running attacks.
- Blocks can be **independent of all previous Blocks** - they do not necessarily form a "chain" linked by cryptographic hashes.
- It is possible for multiple Peers to **concurrently** propose Blocks for inclusion in consensus at the same time, removing a major bottleneck compared to systems that require on some form of sequential leadership.

A Peer proposed a Block for consensus simply by appending it to its current Ordering, where this Ordering includes all Blocks up to the Consensus Point, plus any additional Blocks still to be confirmed in consensus.

Convergent Consensus Convex uses a variant of Convergent Replicated Data Types (CRDTs) ensure that the Network converges to Consensus. CRDTs have the provable property of Eventual Consistency - which might be informally defined as a situation where all nodes eventually agree on the same value (in this case, the value of interest is the Ordering up to an agreed Consensus Point).

The CRDT is implemented through:

- A data structure called a **Belief**, which represents a Peer's view of consensus across the whole Network, including the latest Block Orderings from other Peers
- A **Belief Merge Function**, which:
- Combines any two (or more) Beliefs to create an updated Belief

- Is idempotent, commutative and associative with respect to the merging of other Beliefs

This effectively forms a *join-semilattice* for each Peer, and satisfies the conditions required for a CRDT. Repeated applications of the Belief merge result to Beliefs propagated by Peers in the network automatically results in convergence to a stable consensus.

Digital signatures ensure that Peers are only able to (validly) update the part of the overall Belief structure that represents their *own* proposals. This ensures that full cryptographic security is maintained throughout the operation of the consensus algorithm since nobody can impersonate a Peer.

Malicious Peers could remove the Ordering of one or more other Peers from their Belief, perhaps in an attempt to censor transactions. However, this will not be an effective attack against the Network since the Ordering will also be relayed via other Peers and will ultimately be merged into Consensus.

Stake-weighted voting During the process of convergence, it may be necessary to resolve conflicts in proposed block Orderings from different Peers. This is achieved by a system of stake-weighted voting.

At each Belief Merge step, if there is a conflict between Orderings the Peer computes the total share of stake voting for each proposed Block in the next position after the current Consensus Point. It is able to do this because it has a view of the Orderings proposed by all other Peers.

This voting is applied iteratively to Blocks in following positions, but only counting the votes by Peers that have supported the winning Ordering up to this point (supporting a minority Block causes Peers to be temporarily excluded from the considered vote in following Blocks). Peers that vote for anything inconsistent with the majority therefore cannot influence the Ordering of any subsequent Blocks.

Once the overall winning Ordering has been determined, the Peer appends any new Blocks it wishes to propose, then adopts this Ordering as its own proposal. This Ordering is signed and incorporated into the Peer's own Belief, which is then propagated onwards to other Peers.

As an illustration, consider three Peers that are initially in consensus with respect to an Ordering of Blocks `XXXX` but peers `A` and `B` propose new Blocks `Y` and `Z`:

```
Peer A: (stake 20) ordering = XXXXY
Peer B: (stake 30) ordering = XXXXZ
Peer C: (stake 40) ordering = XXXX
```

Peer `C` observes the orderings of Peer `A` and `B` (after propagation of Beliefs). It sees two conflicting proposals, but because Peer `B` has the higher stake it takes this Ordering first. It then appends the other Block it has observed:

```
Peer A: (stake 20) ordering = XXXXY
Peer B: (stake 30) ordering = XXXXZ
Peer C: (stake 40) ordering = XXXXZY (updated)
```

Peer `A` now observes the Orderings of the other Peers. Since there is still a conflict, it calculates the vote for each ordering and sees that there is a 70-30 vote in favour of having block `Z` first (and a 40/0 vote in favour of block `Y` next). It therefore adopts same the same Ordering as proposed by Peer `C`.

```
Peer A: (stake 20) ordering = XXXXZY (updated)
Peer B: (stake 30) ordering = XXXXZ
Peer C: (stake 40) ordering = XXXXZY
```

Peer `B` now observes the Orderings. It sees everyone agreed on block `Z`, and a 60-0 vote in favour of Block `Y` next. It therefore adopts this winning Ordering as its own:

```
Peer A: (stake 20) ordering = XXXXZY
Peer B: (stake 30) ordering = XXXXZY (updated)
Peer C: (stake 40) ordering = XXXXZY
```

This procedure naturally converges to a single Ordering: Any situation where Peers are voting for different Blocks (in any position) is unstable and will collapse towards one outcome, since Peers will switch to whichever Ordering is observed to have a slight majority. After a few rounds of Belief propagation, all good Peers will align on the same Ordering.

Stability The Belief Merge procedure outlined above has many desirable stability properties even in the presence of some proportion of malicious adversaries (Bad Peers). Some of these are listed below:

51% Stability with Good Peer Majority If more than 50% of Peers adopt the same Ordering, this majority consists entirely of Good Peers and they are mutually aware of each other's agreement, then the Ordering is provably stable no matter what any adversaries subsequently attempt, since the adversaries cannot cause any Good Peer to change their vote.

51% Stability with Rapid Propagation If more than 50% of Peers (some of which may be Bad Peers) adopt the same ordering, less than 50% of all Peers are Bad Peers, and these Beliefs are propagated quickly to all other Peers (at least before the next round of Belief Merges), then the Ordering will be provably stable since a majority of Good Peers will adopt the same Ordering during the next Belief Merge.

67% Stability vs. Irrelevant Alternatives Assuming that:

1. At least 2/3 of all Peers are aligned in proposing the same Ordering, and are aware of each other's Orderings

2. Less than 1/3 of Peers (by Staked voting weight) are Bad Peers, the remainder (>2/3) are Good Peers

3. Bad Peers may collude arbitrarily, but do not have a Belief propagation speed advantage (on average) relative to Good Peers

4. Non-aligned Good Peers do not initially have any significant support for any conflicting ordering.

Then the Ordering is provably stable, since:

- By (1) and (2), the number of Good Peers within in the aligned set of Peers must strictly outweigh the Bad Peers.

- Whatever the Bad Peers do (including switching from being in the aligned group to supporting a new conflicting Ordering), their new Ordering will still be outweighed by the Good Peers which are already in alignment

- Therefore, the initially aligned Good Peers will win 50%+ majority with Good Peers, since they will win a propagation race by sharing their Beliefs which will bring the majority of remaining non-aligned Good Peers as per assumption (3)

75% Stability vs. powerful adversaries Assuming that 75% of Peers are aligned in proposing the same Ordering, and are aware of each other's orderings, the Ordering is stable as long as less than 25% of Peers are Bad Peers.

This hold true even against powerful adversaries with capabilities such as:

- Ability to temporarily isolate and trick non-aligned Peers into adopting a conflicting Proposal

- Ability to censor or delay arbitrary messages on the network (as long as at least one Belief propagation path eventually exists between any pair of Good Peers)

- Ability to delay the activity of any Good Peer

Determining consensus Even after a stable Ordering is observed, it is necessary to confirm Consensus. This is achieved through what is essentially a decentralised implementation of a 2-phase commit.

Once a 2/3 threshold of Peers are observed by any Peer to be aligned on the same Ordering up to a certain Block number, the Peer marks and communicates this number as a **Proposed Consensus Point (PCP)**. The Peers propagate this PCP as part of their next published Belief, attached to their Ordering.

Once a 2/3 threshold of Peers are observed to have the same Proposed Consensus Point with the same Ordering, this value is confirmed by the Peer as the new **Consensus Point (CP)**. From this point on, Good Peers will consider the Consensus final.

Consensus is **guaranteed** providing:

- A stable Ordering is reached where a majority of Peers consistently propose the same Ordering
- At least 2/3 of Stake is held by Good Peers that are active in the network

This follows from the fact that given a majority for a stable Ordering, all Good Peers will eventually adopt the same Ordering and therefore the Network will pass both the Proposed Consensus and Consensus thresholds.

Illustration Consider a case where all peers A, B, C, D and E initially agree on a Consensus Ordering (labelled `o`). At this point, peer B receives a set of new Transactions, composes these into a Block and produces a Belief with an updated Ordering (`x`), including the new proposed Block. Initially, this is unknown to all other Peers.

We can visualise this initial situation as a Matrix, where each row is the Belief help by one peer, and each column represents the latest signed Ordering observed by each peer from another peer. Each Peer also has knowledge of the current Consensus defined by `o`, which is also its Proposed Consensus.

```
  ABCDE  Consensus   Proposed Consensus

A ooooo  o           o
B oxooo  o           o
C ooooo  o           o
D ooooo  o           o
E ooooo  o           o
```

Because it has a new Belief which represents novelty to the Network, Peer B propagates this Belief to other Peers. The other Peers observe that Peer B has

proposed a new Ordering `x`, and incorporate this into their Belief regarding Peer B:

```
  ABCDE  Consensus   Proposed Consensus

A oxooo  o           o
B oxooo  o           o
C oxooo  o           o
D oxooo  o           o
E oxooo  o           o
```

With this information, all Peers are aware of a new Ordering. They validate that this is consistent with the previous Consensus Ordering `o`, and because it is a simple, non-conflicting extension of `o` (just one new Block appended) they automatically adopt it as their own proposed Ordering (the diagonal of the matrix).

```
  ABCDE  Consensus   Proposed Consensus

A xxooo  o           o
B oxooo  o           o
C oxxoo  o           o
D oxoxo  o           o
E oxoox  o           o
```

Another round of Belief propagation is performed. Now each peer is aware of the latest Ordering `x` being communicated by all other Peers. Since each Peer can now observe 100% of Stake proposing the same Ordering, it meets the threshold to be considered as the Proposed Consensus (the start of the 2-phase commit).

```
  ABCDE  Consensus   Proposed Consensus

A xxxxx  o           x
B xxxxx  o           x
C xxxxx  o           x
D xxxxx  o           x
E xxxxx  o           x
```

Finally, another round of propagation is performed. Peers now observe 100% of Stake supporting the same Proposed Consensus, so can confirm the Ordering `x` as the new Consensus (the completion of the 2-phase commit)

```
  ABCDE  Consensus   Proposed Consensus
```

```
A xxxxx  x           x
B xxxxx  x           x
C xxxxx  x           x
D xxxxx  x           x
E xxxxx  x           x
```

The network is now in a new quiescent state, with the Consensus Point advanced to include the full Ordering `x`, and ready to process the next proposed Block(s).

In this simple case, the new Consensus is confirmed within just three rounds of Belief propagation:

- Peer B communicates the new Block to other Peers
- Other Peers communicate their adoption of the new Block
- All Peers communicate Proposed Consensus (after which individual Peers can independently confirm Consensus)

In more complex cases:

- Multiple Peers may propose Blocks at the same time. In this case, stake weighed voting would be used to resolve conflicts and determine which Blocks are included first. It may take an additional round or two to resolve such conflicts into a stable Ordering, though overall this is actually more efficient since multiple Blocks are being brought into Consensus in a similar overall number of rounds.
- The network might not reach a quiescent state before further new Blocks are added. This is not an issue: consensus will be confirmed for the initial Block(s) while the new Blocks are still being propagated at earlier stages.
- Some Peers might misbehave or be temporarily unavailable. Again, this is not a problem as long as a sufficient number of Good Peers are still operating and connected, since the consensus thresholds can still be met. Temporarily disconnected or offline Peers can "catch up" later.
- The Peer Network may not be fully connected, potentially adding `O(log(number of peers))` additional rounds of propagation assuming that each Peer propagates to a small constant number of other Peers in each time period. However, in practice, these additional rounds may not all be needed because a smaller number of highly staked and well-connected Peers will be able to confirm consensus without waiting for the rest of the Network.

Important note on complexity At first glance, the Convex consensus algorithm might be considered impractical because of the scale of data structures being shared. Consider a plausible high volume operating scenario:

- n = 1,000 Peers active in the network
- r = 10 new Blocks per second
- s = 10k of data for each Block (around 100 Transactions)
- o = 1,000,000,000 Blocks of transactions in the each Ordering (a few years of blocks)

Each Peer would theoretically be holding ~100 *petabytes* of information for their Belief, which would need to be transmitted in each propagation round, requiring a bandwidth in the order of many *exabytes* per second. Clearly this is not practical given current hardware or network capacity.

However, we exploit some powerful techniques to minimise this:

- Beliefs are represented as Decentralised Data Values that support **structural sharing**: identical values or subtrees containing identical values need only be stored once. Since Orderings will be identical up to the Consensus Point, these can be de-duplicated almost perfectly.
- Peers are only required to actively maintain Block data for a limited period of time (e.g. 1 day of storage would be less than 10GB in this case)
- The Decentralised Data Values support usage where only the **incremental change** (or "Novelty") can be detected.
- The number of outgoing connections for each Peer is **bounded** to a small constant number of Peers that they wish to propagate to (typically around 10, but configurable on a per-peer basis)
- Beliefs can be **selectively culled** to remove Orderings from Peers that have very low stakes and are irrelevant to consensus. This can be performed adaptively to network conditions if required: Peers may only need to consider the "long tail" of low staked Peers in rare situations where these are required to hit a consensus threshold or decide a close vote.

With these techniques, Peers only need to propagate the novelty they receive (in this example around 100k of Block data per second, plus some accounting and structural overhead) to a small number of other peers. Bandwidth required is therefore on the order of 1-10MB/s (allowing for overheads and a reasonable

number of Peer connections), which is certainly practical for any modern server with decent network connectivity.

Overall complexity is therefore (factoring out constants):

- $O(r \times s)$ bandwidth, scaling with the rate of new transaction data size
- $O(r \times s)$ storage, scaling with the rate of new transaction data size
- $O(\log n)$ latency, scaling with the logarithm of number of Peers (based on standard analysis of gossip networks)

We believe this is optimal for any decentralised network that maintains consensus over a global state. Note that Lower latency can be achieved by communicating to all peers simultaneously, but at the cost of significantly higher bandwidth.

A note on Front Running Front running is difficult for an adversary to perform against the Convex consensus algorithm. While theoretically possible, it would require a sophisticated and well-resourced attacker.

The main reason for this is that Transactions are not visible to any untrusted participants in the Network until *after* a new Block has been proposed by a Peer and propagated as part of a Belief, by which point it is already well on its way to being included in consensus.

A user concerned about front-running attacks should submit vulnerable transactions exclusively via a well connected, well-staked Peer that is trusted not to be malicious, i.e. this Peer must not itself be helping to facilitate a front-running attack.

In this scenario a front running attack would need to:

- Listen to vulnerable transactions broadcast on the Network
- Quickly generate a new Block with the Transaction(s) needed to execute the front-running attack
- Have sufficient influence over consensus formation to ensure that the new Block is somehow re-ordered *before* the original Block (that is already approaching consensus)

Practically, this attack would require the attacker to have more resources (Stake and network connectivity) than the original Good Peer *and all the Good Peers it is connected to*, since the original Block would already be ahead by at least one round of propagation by the time the attacker can observe it. Furthermore,

the attack would be publicly visible and traceable to the Peer(s) that launched it: so even if successful the attacker would be vulnerable to blacklisting and/or real world legal repercussions.

Assuming Good Peers are well-staked, and connect preferentially to other well-staked, trusted Peers with a known legal identity (which would be good practice, and should be configured as default behaviour), we believe such front running attacks will be highly difficult to execute and generally impractical from an economic perspective.

Execution Engine

The Convex execution engine is referred to as the Convex Virtual Machine (CVM). This is a general-purpose computational environment that is used to execute the State Transitions triggered by Transactions.

Accounts The fundamental control mechanism for the CVM is via Accounts. There are two main types of Accounts, which differ primarily in the means that they can be controlled:

- **User Accounts**: Accounts that are controlled by external users, where access is secured by Ed25519 digital signatures on Transactions.
- **Actor Accounts**: Accounts that are managed by an autonomous Actor, where behaviour is 100% deterministic according to the defined CVM code. Actor functionality may be called directly or indirectly within an externally submitted Transaction, but only if this is initiated and validated via a User Account.

It is important to note particular the two types of Account share a common abstraction. Both User Accounts and Actor Accounts may hold exclusive control over assets, allowing for decentralised value exchange mediated by smart contracts. This common abstraction is useful, because it makes it simple to write code that does not need to distinguish between assets controlled directly by a user and assets managed by a Smart Contract.

User Accounts are **protected by digital signatures**. A transaction which uses a specific account is only considered valid if accompanied by a valid digital signature. Without access the corresponding private key, it is computationally infeasible for an attacker to submit a fake transaction for an Account. No external transactions are permitted on Actor Accounts - they operate purely according to the rules expressed in their code.

Environments A novel feature of the Convex Account model is that each Account receives it's own *programmable environment* where variables, data structures and code can be dynamically defined and updated. Definitions held within different accounts cannot collide since they have independent environments.

- For User Accounts, this behaves like a computer completely under the control of the user. Each user receives the equivalent of a fully functional "Lisp Machine", which can modify its own definitions and has read-only access to the environments of other Accounts.

- For Actor Accounts, this can be used to store Actor code and state required for the operation of the Actor. Deployment of an Actor is equivalent to creating an Account and initialising the Actor's environment, with subsequent changes to the environment strictly controlled by a set of exported functions that can be externally called.

We believe this is a powerful model to encourage rapid development and innovation: for example, a developer can easily experiment with code in their own user account, then capture the same code in the definition of a deployable Actor for production usage.

Optionally, Actor Accounts can be utilised as **Libraries** of code for use by other Accounts. Since it is possible to create an immutable Actor Account (i.e., Any actor that lacks externally accessible code to change its own environment), this means that you can create Libraries that are provably immutable, and can therefore be relied upon from a security perspective never to change.

Environments also support **Metadata** which can be optionally attached to any definition. This innovation is particularly useful to allow custom tags and documentation to be attached to library definitions in a way that can be inspected and utilised on-chain. For example, the metadata for a core function might look like:

```
{
     :doc {:description "Casts the argument to an Address. Valid arguments include hex Strin
         :examples [{:code "(address 451)"}]
         :type :function
         :signature [{:params [a]
                      :return Address}]
         :errors {:CAST "If the argument is not castable to a valid Address."}}
}
```

Information Model Convex requires a standard information model because for consensus to be useful, it must be agreed precisely what the information in consensus represents, and it is necessary for any smart contract to operate on well-defined data with clear semantics.

Design decisions regarding the information model have been driven by a combination of theoretical and pragmatic considerations:

- Representing types that are theoretically sound and fundamental to computing such as vectors, maps and lambda functions
- Providing types that are generally useful for developers of decentralised asset systems
- Supporting all the capabilities required for a Lambda Calculus
- Using types that are conveniently represented in modern computing platforms (e.g., the use of 64-bit Long integers and IEEE 754 double precision floating point values)
- Ensuring that information can be efficiently encoded to minimise storage and bandwidth requirements

Convex implements a relatively comprehensive set of data types, which are utilised both within the CVM and in the broader implementation of a Convex Peer (including the consensus algorithm and communication protocols).

All data types available in the CVM might be considered as Decentralised Data Values (DDVs) - immutable, persistent and structured for efficient network communication of information.

Primitive types Several basic primitive types are supported, consistent with a typical modern language and broadly equivalent to those available on the JVM:

- `Byte` - an 8-bit unsigned integer
- `Long` - a 64-bit signed integer
- `Double` - an IEEE754 double-precision floating point value
- `Character` - a UTF16 character
- `Boolean` - `true` or `false`

These behave generally as expected, with the important proviso that arithmetic is implemented exclusively using long and double types (other types are automatically upcast to long and double as required).

There is also a set of primitive value types generally useful for programming on the CVM:

- `Keyword` - a named value, most often used for map keys (`:foo`)
- `Symbol` - a name generally used to refer to a value in an Environment (`bar`)
- `String` - an arbitrary length sequence of Characters (`"Hello"`)
- `Address` - a 64-bit identifier for an Account (e.g., `#1234`)

Data values that are sufficiently small, including most of the above, have compact encodings that are **embedded** directly within the encoding of larger Data Values that contain them. This is an internal implementation detail, but important to reduce the overhead of storing and communicating many small values independently, which is transparent to CVM code.

Blobs A `Blob` is an arbitrary-length sequence of Bytes and is considered a first class value on the CVM. e.g.

`0xa0b1c2d3e4f5`

Cryptographic values such as Hashes are generally treated as small fixed-length Blobs.

Internally, Blobs are stored as a Merkle tree of chunks of up to 4096 bytes in length. Blobs may exceed the size of working memory: they can technically be up to 2^63-1 bytes in length.

Blobs could also be used as a basis for decentralised file storage, perhaps as a Layer 2 solution like IPFS.

Data Structures Convex supports a range of first-class data structures, primarily:

- `Vector` - a sequence of values (e.g., `[1 2 3]`)
- `List` a sequence of values usually used to represent code (e.g., `(foo bar baz)`)
- `Map` a mapping of keys to values (e.g., `{:bar 1, :baz 2}`}
- `Set` a set of values (e.g., `#{1 2 3}`)

All data structures are immutable, functional data structures that support structural sharing based on an underlying tree representation (in fact, a Merkle tree). Critically, these provide efficient `O(log n)` operations for append, access,

update etc. without requiring expensive "copy on write" operations to preserve immutability. These data structures are similar to the data structures frequently found in modern functional programming languages such as Clojure, Scala or Haskell.

A moderately high branching factor (typically 16) is used. This is important because:

- It facilitates faster lookups (less nodes to traverse by a factor of 4 vs. a binary tree)
- It reduces the number of new node allocations required to update a path to a leaf node.
- It reduces the number of hashes that need to be computed (a performance bottleneck in some cases)
- There is a certain elegance and minor performance benefit in being able to index the tree using hex digits

CVM data structures are used widely throughout the whole Convex implementation: For example the ordering of blocks in the CPoS algorithm is internally implemented as a Vector.

Syntax Objects `Syntax` objects are wrapped values that contain both a raw value and optional metadata.

Metadata may be an arbitrary Map, but typically would include such things as:

- Source code references
- Documentation
- Information generated through macro expansion

Syntax Objects are inspired by Racket, and are generally used for code generation and compilation, although they are also available for use in regular CVM code if desired. They are marginally more efficient than storing a value and metadata as two separate fields in a Map, for example.

The primary usages of Syntax Objects within the CVM are:

- Allowing metadata to be attached to values (e.g., documentation for Actor functions)

- Supporting the implementation of the Convex Lisp compiler and macro system

NOTE: In the future Syntax Objects may be extended to implement a gradual type system such as seen in Typed Racket. Racket has demonstrated the value of Syntax Objects in helping to support future language evolution.

Nil Convex supports the value `nil` as a first-class value (which can be considered the sole member of the type `Nil`).

By convention, core runtime functions generally return `nil` to indicate the absence of a value, for example looking up a value in a map with a key that is not present.

`nil` values are considered as being "falsey" (equivalent to `false`) in conditional operations, which facilitates the technique of "nil-punning" popular in languages such as Clojure and Common Lisp. For example, the following is a typical pattern:

```
(if (lookup-optional-value ...)
  (true-branch ...)
  (false-branch ...))
```

There is no direct equivalent of a `NullPointerException` since CVM objects do not implement methods, however careless use of `nil`s may result in type cast errors (e.g. `(+ 2 nil)`).

Records Certain CVM structures are defined as built-in record types, e.g.

- `AccountStatus`
- `PeerStatus`
- `State`

These are primarily used internally by the Convex Peer and CVM implementations, though for convenience they may be accessed and treated as Maps from field names to values in CVM code.

NOTE: Supporting user-defined, row-polymorphic record types is under consideration for future implementation (probably in V2).

Functions Functions are first class objects suitable for use in functional programming. Convex implements functions in this way because they are fundamental and powerful constructs that allow the construction of effective programs without having to simulate them with lower-level constructs (e.g. a stack based model).

The decision to emphasise first-class functions and functional programming is justified by the strong theoretical foundations of the Lambda Calculus.

Important features of functions include:

- Support for variable arity function application like `(+ 1 2 3 4)`
- Full lexical closures (capturing values in the lexical environment at the point of creation).
- Explicit tail-recursion support (recursively calling functions without consuming stack space)

Many functions are provided as part of the runtime environment, generally available to users in the standard library `convex.core`. These functions provide the foundation for construction of higher-level functionality.

In addition (adopting an idiom that has proved convenient in the Clojure language), data structures may be used in place of functions in some defined circumstances, e.g.:

- Maps may be used as functions that implement map lookup: `({:foo 1 :bar 2} :bar) => 2`
- Sets may be used as functions to test membership: `(#{1 2 3} 4) => false`
- Vectors may be used to perform indexed lookup: `([1 2 3] 2) => 3`

Macros and Expanders Convex supports the use of macros, in the manner of most Lisps. Macros provide powerful code generation and templating facilities, allowing users to extend the language to add new programming constructs. In fact, a large proportion of the `convex.core` library itself is implemented using macros.

Macros are also useful for generating efficient smart contracts, since they enable many computations to be performed once at compile time, reducing the cost of subsequent executions. For example, mathematical values required for liquidity curve calculations can be compiled into constants when liquidity curve Actors

are deployed, eliminating wasteful computation when the Actor is subsequently called by users.

Macros are implemented using the lower-level construct of Expanders, which are Functions that generate code at expansion time (i.e. just before compilation).

The idea of Expanders as a fundamental language construct is covered in the 1988 Paper "Expansion-passing style: A general macro mechanism" (R. Kent Dybvig, Daniel P. Friedman & Christopher T. Haynes). Interested readers are encouraged to read this article to understand the detailed rationale for this approach, but perhaps the most important point is that Expanders are strictly more powerful and flexible than traditional Lisp macros.

Macros and expanders present powerful possibilities for decentralised application, including automated code generation for new actors and smart contracts.

Ops Ops are low level, programmatic constructs that represent individual instructions on the CVM. All CVM code is compiled to a tree of Ops. They can be considered as the "machine code" instructions on the CVM. Currently the key Ops supported are:

- `cond` - conditional evaluation
- `constant` - load a constant value
- `def` - modify environment (map of symbols to values)
- `do` - sequential composition of operations
- `invoke` - execution of a Function
- `lambda` - instantiation of a Function
- `let` - definition of values in a lexical scope
- `lookup` - lookup of a value in the environment
- `special` - access to special values in the CVM state and/or execution context

These Ops resemble the basic primitives frequently found in an implementation of the lambda calculus - in particular `invoke` and `lambda` are direct implementations of "Application" and "Abstraction".

It should be noted that certain other important constructs e.g., `cons`, `quote`, `=` etc. are currently implemented as functions in the runtime environment, so

the base language of the CVM can be regarded as the combination of the Ops and runtime functions (accessed via the `invoke` Op), applied to arbitrary CVM values.

Execution constraints Since the CVM supports Turing-complete computation, it is necessary to place constraints upon code execution to prevent erroneous, badly written or malicious code from consuming excessive resources. This is particularly important in a decentralised system because such resources are a global, shared cost.

The CVM therefore constrains **time**, **space** and **depth**.

Time Convex constrains time by placing a "juice cost" on each CVM operation performed. Any transaction executed has a "juice limit" that places a bound on the total amount of computational work that can be performed within the scope of the transaction.

The originating account for a transaction must reserve juice by paying an amount `[juice limit] x [juice price]` at the start of the transaction. Any unused juice at the end of the transaction is refunded at the same rate. The juice price a dynamically varying price that adjusts with amount of execution performed per unit time by the Convex network as a whole: this is a cryptoeconomic mechanism to disincentivise transactions from being submitted at peak periods, and as a protection from DoS attacks by making it prohibitively expensive to flood the compute capacity of the network for a sustained period of time.

If the juice limit has been exceeded, the CVM terminates transaction execution with an exception, and rolls back any state changes. No juice is refunded in such a situation - this penalises users who attempt excessive resource consumption either carelessly or maliciously.

Space Convex performs a complete, deterministic analysis of space usage by each Transaction, defined as the delta in the size of the Global State caused by the Transaction.

This is an important execution constraint, without which there would be poor incentives for developers to be efficient with CVM memory usage (beyond paying the initial juice cost). Juice costs alone cannot be accurately used to constrain memory usage, because they are fundamentally a one-off "flow" cost that is immediately incurred, whereas space is an ongoing "stock" cost that is incurred by all Peers over time.

This constraint is described in more detail in the "Memory Accounting" section of the White Paper.

Depth Convex places a limit on "stack" depth within Ops and Functions. While not strictly necessary (execution time constraints will at some point halt infinite recursion) a maximum depth is useful for two reasons:

- Unbounded recursion should be discouraged in CVM code. The kinds of situations where it might be useful (heavy computations, or traversing large data structures, for example) should probably not be running on the CVM itself - this generally belongs in client or server code outside the CVM.

- It makes the CVM implementation simpler and more performant, since the depth limit allows the underlying JVM stack to be safely used without the risk of `StackOverFlowError`s, and therefore removes the need to explicitly handle these.

Currently the depth limit is 256. This could be relaxed if needed, but we currently do not see any realistic smart contract use cases that are likely to require this much stack depth, especially considering that the CVM supports techniques like tail recursion (which avoid consuming stack depth).

Runtime environment The CVM defines a small core runtime system that provides CVM capabilities to CVM programs. These include:

- Standard language control structures (loops, conditionals, error handling etc.)

- Basic numerical functions (focused on 64-bit integer and IEEE 764 double precision floats)

- Functions to manipulate and manage immutable persistent data structures (vectors, lists, maps sets)

- Control of assets native to the Convex network, such as balances and stake

- Ability to interact with Actors (deploying Actors, calling Actor functions)

- Functionality useful for managing CVM state updates, e.g. transaction rollback

- Language constructs necessary to support Convex Lisp (see below)

The core system is designed so that these low-level capabilities can be easily composed to create higher level capabilities, through composition of data and functions. Runtime functions are generally exposed to the CVM in the `convex.core` library.

At the same time, the capabilities of the runtime system are constrained so that they cannot break the rules of CVM execution necessary for deterministic state updates. There is no external IO capability, no ability for non-deterministic behaviour, and no ability to affect CVM state in a way that breaks the security model. CVM code is therefore fully "sandboxed" from the perspective of the overall Convex system.

In many cases, the runtime system is optimised for performance - for example, methods to update CVM data structures are implemented in efficient, low level, compiled code. Emulating such operations in pure CVM code would be many orders of magnitude slower, so this approach allows us to provide sophisticated, immutable data structures and higher-level language features without compromising performance.

Transparent persistence The Convex execution engine implements a system of transparent (sometimes also known as orthogonal) persistence. In this model, the CVM state size may exceed the working memory capacity of a Peer, and necessary parts of the State tree are loaded in from persistent storage on demand.

This presents a significant conceptual benefit for the developer: there is no need to write any code to load or unload data from storage in normal CVM code. There is some additional implementation complexity for the CVM itself, but this is considered a worthwhile trade-off, especially since it simplifies the logic of other parts of the Convex Peer implementation (e.g., eliminates the need to explicitly handle the memory consumption growth of long Block Orderings generated by the CPoS consensus algorithm over time).

In the current implementation, this is achieved with judicious reliance upon the very efficient JVM automatic memory management. This enables the following lifecycle for in-memory data values:

1. Values are initial created with strong (RefDirect) references, which ensure that they are held in memory for as long as they are potentially needed

2. At certain checkpoints (most importantly, after the successful processing of each Block) the current State is *persisted*. All Cells which are reachable but not yet persisted are written to storage, and references to them are converted from strong references to soft (RefSoft) references. This happens as an atomic operation. This is made efficient by the system of Novelty Detection which can identify the `n` new Cells to be persisted in `O(n)` time.

3. From this point onwards, the persisted objects may be garbage collected at any time by the JVM if memory pressure occurs.

4. If an attempt is made to access a value that has been garbage collected, the reference automatically fetches the associated data value from storage.

This is guaranteed to succeed assuming that the previous persistence step was successfully completed.

5. Over longer time periods, it is possible to perform garbage collection on the storage itself by compacting the store to remove data that is no longer required by the current consensus state. Peers may choose to do this at their own discretion based on their operational requirements, or alternatively they may decide to preserve all data (for example in order to perform historical analysis)

Convex Lisp The CVM includes a small, dynamically typed, embedded Lisp suitable for general purpose programming within the CVM environment. Convex Lisp draws inspiration from Common Lisp, Racket and Clojure. It is designed as primarily a functional language, with fully immutable data structures, as it is our belief that functional programming forms a strong foundation for building robust, secure systems.

Convex Lisp was chosen as the first language implementation in Convex for the following reasons:

- Experience with Lisp as a highly productive language for developers, particularly when manipulating data structures (as seen in data-driven development approaches with Clojure, for example).

- It can be constructed using a very small number of simple, well-defined axiomatic primitives, which in turn are based on the Lambda Calculus. This provides a robust logical and mathematical foundation, suitable for the type of verifiable, deterministic computations that the CVM must support.

- Lisp has a very simple regular syntax, homoiconic nature of code and ability to implement powerful macros. We hope this provides the basis for innovative new languages and domain-specific languages (DSLs) on the CVM.

- Lisp compilers are small enough and practical enough to include as a capability within the CVM, avoiding the need for external compilers and tools to generate CVM code.

- It is comparatively simple to implement, reducing the risk of bugs in the CVM implementation (which may require a protocol update to correct).

- Lisp is well suited for interactive usage at a REPL prompt. This facilitates rapid prototyping and development of Actors in a way that we believe is a significant advantage for decentralised application builders looking to test and prototype new ideas.

Developers using the Convex system are not required to use Convex Lisp: It is perfectly possible to create alternative language front ends that target the CVM (e.g. by constructing trees of Ops directly). Convex has experimental support for a JavaScript-like language (Scrypt) and community members are encouraged to innovate further in this space.

Scheduled execution The CVM includes a specialised data structure called the Schedule that references CVM code to be executed under a specific Account at a defined future timestamp.

The main purpose of the Schedule is to allow Actors to implement autonomous behaviour without the need to wait for an external transaction to trigger execution. This could be used to finalise a decentralised auction, to distribute the prize from a random lottery, to trade on a periodic basis, or to unlock assets that have been frozen for a specified period of time.

The schedule is **unstoppable**, in the sense that once the consensus timestamp advances based a scheduled execution time, the associated code is automatically executed according to protocol guarantees. This execution is guaranteed to happen before any other transactions in a block submitted on or after the same timestamp.

Scheduled executions currently cannot be cancelled, but this is not a serious limitation: Actors can simply implement code to ignore the event if it is no longer relevant.

Garbage collection The CVM automatically garbage collects objects to which references are no longer maintained. The choice of including garbage collection in the CVM is motivated by the following factors:

- **Convenience** - developers need not be concerned with manual memory management. In general, this is a significant productivity gain.
- **Performance** Given a focus on immutable data, garbage collection offers significant performance advantages because references can be shared internally within the CVM implementation, as opposed to relying on expensive "copy on write" approaches.
- **Security** - Mistakes in memory management are one of the most common defect types, often resulting in significant security issues (e.g. "buffer overruns"). Such risks are generally unacceptable for smart contract code securing significant digital assets.

Short lived objects are garbage collected by the host runtime (the JVM). This will happen for most temporary objects created during the execution of CVM code.

For those data values that are persisted to long term storage (e.g. because they become part of the updated CVM state), the host runtime may garbage-collect the in-memory copy.

Peer operators may also choose to either garbage collect old data from long term storage, or alternatively maintain old data for historical analysis. Peers are only required to maintain object information necessary to execute the consensus algorithm (belief structures plus the proportion of CVM state relating to Peer information and stakes). For more details, see the section on Convergent Immutable Storage.

Storage System

Convex makes use of a specialised storage system that complements the design of the CVM. This provides significant performance advantages, since the format of data in storage aligns directly to the usage patterns and data structures utilised in the CVM.

The storage system is also used to facilitate serialisation and transport of data across the network in communication between peers and clients.

Cells Storage is constructed out of Cells.

In most cases, a Cell is an entity that represents an Value in the CVM Informational Model. Normally there is a 1-1 mapping between Cells and CVM Values, however there are some exceptions:

- For larger data structures a tree of Cells may be necessary - this is because we need to place a fixed upper bound on the size of each cell.
- Small data values do not require a whole Cell, since it is more efficient to embed them directly within a larger Cell.
- Some special data structures used in Convex are technically implemented as Cells for the purpose of storage and serialisation but are unavailable for use within the CVM since they are used externally to the CVM State - for example the `Belief` data structure used in the CPoS consensus algorithm.

Encoding All Cells have a unique Encoding.

The Encoding is designed to provide the following properties:

- A bounded maximum encoding size for any Cell (currently 8191 bytes)

- Very fast serialisation and deserialisation, with minimal requirements for temporary object allocation.

- Uniqueness of encoding - there is a 1:1 mapping between Cell values and valid encodings. This means, among other useful properties, that Value equality can be determined by comparing hashes of encodings.

- Self describing format - given a valid Cell Encoding, the Data Value can be reconstructed without additional context

The same encoding is utilised in both durable storage and in network transmission.

Value IDs as storage keys The cryptographic hash of the Cell encoding is used an an identifier (the "Value ID" or "VID") to refer to a Cell, and as a key for addressing data in the storage system.

This has the important property that it requires all values in the storage system *immutable* - the data value cannot change for a given key, or else the VID will no longer be valid. This restriction may seem limiting at first, but in fact provides significant advantages for the Convex storage implementation:

- No need to re-size values once written: the database can be accumulated in an "append-only" manner. The prevents storage fragmentation.

- No need for cache invalidation or synchronisation of replicas: values cannot change

- Rapid verification: if a hash exists in the store, and the data has already been validated, it must still be valid.

Embedding Small Data Values can usually be Embedded within the Encoding of another Cell (typically a Cell representing part of a larger data structure). In most cases, this avoids the need to construct and store separate cells for small primitive values, and often small data structures themselves can be fully embedded.

For example the vector `[1 2]` is encoded as a 6 byte sequence (`0x800209010902`) which can be seen to embed the values `1` (`0901`) and 2 (`0902`).

Currently, Cells with an Encoding size of up to **140 bytes** are automatically embedded. This heuristic may be modified based on further testing and profiling, but it seems reasonable: per-Cell storage overheads make it inefficient to separately store such small objects, and by compressing many small objects into a single Cell we avoid the need to compute separate SHA3-256 VIDs for each, which we have observed to be a bottleneck in some cases.

Convergence Given the above design features, we can implement a system of immutable storage that is Convergent: Additional storage information may be merged into the store in a manner analogous to a CRDT.

It is a well-known result that taking the union of sets in a purely additive manner (a Grow-only Set) is a valid CRDT. The storage system can be regarded as a growing set of immutable (key, value) pairs, and hence satisfies the CRDT property.

This convergence property is particularly beneficial when combined with the structured of Merkle trees used throughout the CVM: data structures with identical branches are automatically de-duplicated when they are stored, since the existing storage entry can simply be re-used. If effect, the Merkle trees become Merkle DAGs with guaranteed sharing of identical children.

Monotonic Headers In addition to Value IDs and Encodings, the storage system allows header information to be attached to each Cell.

As we require the storage system to be convergent, we require each field of the header to be *monotonic*, i.e., there is a simple function that can compute the new header as the maximum value of any previous header values. This ensure that the headers themselves satisfy the convergence property.

The current Convex implementation utilises Monotonic Headers for the following fields:

- Lazily computed memory size (any value is considered to replace an empty value)
- Status tagging (see below)
- Marking Cells to be pinned for purposes of durable persistence or garbage collection

Unlike the Encoding, Monotonic Headers associated with each Value ID are essentially *transient* in nature, i.e., they can be reset or discarded without affecting the Cells themselves. This allows the Monotonic headers to be used locally by Peers independently of the general functioning of Convex as a decentralised network. For example, rebuilding the database during garbage collection may safely unmark pinned Cells providing the Peer has ensured that it has retained all the information it needs.

Status tagging In order to support efficient Peer operation, the storage system implements a system of status tagging, used to indicate the level to which a data value has been verified by the Peer.

Status tagging is monotonic in nature (increases in value) and hence can be included in part of the Monotonic Header

The basic status levels are:

- **UNKNOWN** - The Peer has an identifier (Hash), but does not know yet if this is consistent with any encoded data
- **STORED** - The Peer has encoded data in storage which is validated to be a well-formed Cell (ignoring children), and consistent with the Hash.
- **PERSISTED** - The Peer has validated the structure of the Cell completely, including recursively validating all its children. At this point, we can rely on a Data Value represented by the Cell to be usable in CVM execution
- **ANNOUNCED** - The Peer has included the data in a publicly broadcast Belief

Some special status levels are also possible, including:

- **EMBEDDED** - A Cell is able to be embedded within other Cells and does *not* need to be individually stored.
- **INVALID** - A Cell has been proven to be inconsistent with some validation rules. Such values cannot be used in the CVM, but caching the invalid status can be helpful to avoid the need to repeat the validation.

This status tagging is monotonic and compatible with being included in the storage CRDT, since:

- The status level can never go backwards: once verified, the result is known to be true forever. If the status was reset (e.g., in the case of storage failure), the only real loss would be the Peer having to repeat certain calculations to re-verify the status.
- Where there are two possible outcomes (valid or invalid, embedded or non-embedded) all Peers that perform correct validation must agree (i.e., it is effectively monotonic for any given data value)

Novelty detection A key feature of the storage system is the ability to detect and apply special handling to Novelty. Novelty is defined as a stored value that is moving up to a higher status level for the first time.

Novelty detection is important for the following reasons:

- When information needs to be shared on the network, only the incremental information needs to be transmitted. This is especially important for the consensus algorithm, for example: the transmission of a new Belief need only include the additions to the proposed Ordering, without communicating the complete Ordering (which may be very long, but is already likely to held by all other Peers)

- When validating data, it avoids the need to re-compute validation on parts of the data that have already been validated. This is particularly important when the data structures to be validated are large, but have only a few small changes in comparison with a previously validated data structure (e.g., the entire CVM State)

Most importantly, when a Belief data structure is produced an determined to be Novelty, Peers utilise this fact to trigger the propagation of the Belief to other Peers - however they only need to transmit the small subset of Cells in the Belief that are new, since most of the Belief data structure will not be novel and a Peer can safely assume that other Peers will already have access to such data in memory or storage.

Garbage Collection Given infinite cheap storage, we could just keep accumulating values in the database forever. However, practical storage limits or costs will make this infeasible or undesirable for many Peers operators.

Peers are only strictly required to maintain:

- Enough information regarding Beliefs to participate in the consensus algorithm (about one day of orderings and transactions - exact limit TBC)

- The current Consensus State for the CVM

The storage system therefore allows garbage collection to be performed on a periodic basis, so that storage space containing data that is no longer required can be reclaimed. Garbage collection is done on a mark+copy basis, where currently used storage is copied to a new data file, and after which the old data file can be safely discarded. This could theoretically be performed concurrently with ongoing Peer operation in a future version.

This behaviour is of course configurable by Peer Operators - we expect some will want to maintain and index all historical data for analytical purposes, or in order to provide their clients with additional historical query capabilities.

Memory Mapped Implementation The Convex reference implementation implements the storage system using a specialised memory-mapped database

called Etch, which is heavily optimised for the storage of Cells as described here. Assuming sufficient available RAM on a Peer, Etch effectively operates as an in-memory database.

In performance tests, we have observed millions of reads and writes per second. This compares favourably to traditional approaches, such as using a relational database or a more generalised key-value store.

Memory Accounting

In order to address the problem of economic and storage costs of state growth, Convex performs continuous memory accounting calculations to ensure that participants pay appropriate costs for resources that they consume.

Motivation A significant but often overlooked problem facing a global, decentralised database that provides a commitment to preserve data indefinitely is the problem of state growth: if not constrained, the size of the CVM state might grow excessively large.

This is an economic problem: The participants who create additional state are not necessarily the same as those who must bear the cost of ongoing storage. This can create a "Tragedy of the Commons" where participants are careless about creating new state. This could quickly lead to a situation where the state grows too large to be feasible for normal computers to participate as Peers in the Convex networks, which will in turn cause centralisation towards a few large and powerful nodes.

This problem cannot be solved by charging at execution time alone. There is no way to determine at execution time how long a particular set of data will be stored for - it might be deallocated in the very next transaction, or it might need to persist forever. Any "average" estimate will end up penalising those who are being efficient only need the storage briefly, and subsidising those who are careless and consume space forever.

Overall Design To solve the state growth problem, Convex implements **Memory Accounting** with the following features:

- Every change to the state tracks the impact on **State Size**, measured in bytes, which is (to a close approximation) the amount of memory that would be required to write out the byte encoding of the entire state tree.
- Each account has an allocation of **Memory Allowance** to utilise.

- When a transaction is executed, the **change in State Size** is computed. An increase in state size reduces the accounts free memory, while a decrease in state size increases the account's free memory.

- If at the end of a transaction the incremental space exceeds free memory then the transaction will fail and be rolled back.

- Accounts may *temporarily* exceed their memory allocation during the course of transaction execution - perhaps by constructing temporary data structures. We can safely allow this because the maximum amount of temporary object allocation is bounded to a constant size by juice limits.

Note 1: that in practice, the actual storage size of the CVM state will be significantly smaller than the tracked state size, because the use of immutable persistent data structures allows many equal tree branches to be shared. The effectiveness of this structural sharing needs to be observed over time, but we anticipate perhaps a 2-3x reduction in state size may be possible in the long term.

Note 2: Observant system hackers may notice that the memory accounting mechanism means that if Account A causes some memory to be allocated, and Account B causes that same memory to be de-allocated (e.g., through the use of calls to an Actor), then Account B will gain memory from A. We consider this a feature, not a bug: It incentivises participants to clean up state wherever possible and incentivises the writers of Actor code to consider their memory allocations and deallocations carefully.

To ensure correct economic behaviour, it is necessary for free memory to have an economic cost. Therefore, Convex provides a **Memory Exchange** though which memory allocations may be traded. This has the following features:

- An automatic market maker enabling accounts to buy and sell memory at any time, placing an effective price on memory

- A total cap on available memory set at a level that constrains the total state size to an acceptable level

- Ongoing governance to regulate changes in the total cap, which can be adjusted to allow for additional state growth as technology improves average Peer resources, without risking a loss of decentralisation.

For convenience, memory purchases happen automatically if additional allocations are needed within a transaction. This means that in most cases, users need not be concerned with the specifics of managing their memory allowance.

The overall cryptoeconomic design of Memory Accounting and the Memory Exchange offers a number of important benefits to the Convex ecosystem:

- A guaranteed cap on state growth, that can safeguard against the growth of storage requirements driving centralisation
- A general incentive for all participants to minimise and manage memory usage. This incentive increases as total state size grows towards the cap.
- A specific incentive for coders to write memory-efficient code and provide the ability for unused data to be deleted, if they want their Actors to be considered high quality and trustworthy.
- A partial disincentive to hoard memory allocations (since expected future cap additions may devalue large memory holdings).
- When space becomes scarce, there is an incentive for less economically viable applications to wind up operations and sell their freed memory allocation.

Memory Size Each Cell (in memory or storage) is defined to have a "Memory Size" which approximates the actual storage requirement (in bytes) for the object

The Memory Size includes:

- The size of the Encoding of the Cell in bytes
- The total Memory Size of referenced child Cells, (e.g., if the object is a data structure)
- An allowance for indexing and storage overheads (currently set to a fixed estimate of 64 bytes)

Lazy computation Memory Size for a Cell is only calculated when required (usually at the point that the State resulting from a transaction is persisted to storage).

This minimises the computational costs associated with memory accounting for transient in-memory objects.

Memory Allowance Each Account on the Convex network is given a Memory Allowance which is a quantity of memory that may be consumed by that Account before incurring additional costs.

Consumption Whenever a Transaction is executed on the CVM, Memory Consumption is calculated based on the total impact of the Transaction on the size of CVM state (the State Size).

Memory Consumption is computed at the end of each transaction, and is defined as:

```
Memory Consumption = [CVM State Size at end of Transaction] -
[CVM State Size at start of Transaction]
```

If a transaction has zero Memory Consumption, it will complete normally with no effect from the Memory Accounting subsystem

If a transaction would complete normally, but has a positive Memory Consumption, the following resolutions are attempted, in this order:

1. If the user has sufficient Allowance, the additional memory requirement will be deducted from the allowance, and the transaction will complete normally

2. If the transaction execution context has remaining juice, and attempt will be made to automatically purchase sufficient memory from the Memory Exchange. The maximum amount paid will be the current juice price multiplied by the remaining juice for the transaction. If this succeeds, the transaction will complete successfully with the additional memory purchase included in the total juice cost.

3. The transaction will fail with a MEMORY Error, and any state changes will be rolled back. The User will still be charged the juice cost of the transaction

If a transaction has negative Memory Consumption, the memory allowance of the user will be increased by the absolute size of this value. In effect, this is a refund granted for releasing storage requirements.

Allowance transfers It is permissible to make an allowance transfer directly between Accounts. This is a practical decision for the following reasons:

- It enables Actors to automate management of allowances more effectively

- It enables Accounts controlled by the same user to shift allowances appropriately

- It avoids any need for resource-consuming "tricks" such as allocating Memory from one Account, and deallocating it from another to make an allowance transfer

- It creates a potential for Memory Allowances to be handled as an asset by smart contracts

Actor Considerations All Accounts, including Actors, have a Memory Allowance. However, in most cases Actors have no need for a Memory Allowance because and memory consumption will be accounted for against a User account that was the Origin of a transaction.

The exception to this is with scheduled execution, where an Actor itself may be the Origin for a transaction.

Actor developers may include a capability to reclaim Memory allowances from an Actor (e.g. transferring it to a nominated User Account). This is optional, but without this there may be no way to ever utilise an allowance held within an Actor (either because a scheduled transaction obtained a Memory refund, or because an allowance transfer was made to the Actor).

Memory Exchange trading The Memory Exchange is a simple Automated Market Maker, allowing users to buy and sell memory allowances at any time from a Memory Pool. The price of memory in the Pool will automatically adjust to find an equilibrium between supply and demand.

New Memory Release It is expected that advantages in storage technology over time will allow memory constraints to be gradually relaxed. Furthermore, it would be unwise to have memory be priced too cheaply at the beginning but continuously increase as State size grows and more memory is bought from the Pool.

Methods are therefore implemented to allow the gradual release of new memory into the Pool. Knowledge that additional memory will be released (and subsequently reduce memory prices) is a useful incentive against deliberate hoarding of memory allowances.

The current Convex design anticipates two such mechanisms:

- A protocol based, automatic addition of new memory into the Pool (based on Consensus timestamps).
- Network Governance roles specially authorised to create and release new Memory into the Pool.

The first of these two methods is strongly preferred in order to minimise potential centralisation, complexity and risk innate to allowing any privileged governance controls over the Network. However it may be necessary given the high probability of technological shocks which cannot be predicted in the protocol.

Size persistence The memory size is persisted in the Storage System as part of the header information for a Cell. Persisting this value is important to ensure that memory sizes can be computed incrementally without re-visiting the complete tree of child Cells.

Memory Accounting impact The memory accounting subsystem is designed so that it always has a minimal effect on CVM state size, even though it causes changes in the CVM state (consumption of allowances etc.). This limits any risk of state growth size from the Memory Accounting itself.

This is achieved mainly by ensuring that state changes due to Memory Accounting cause no net Cell allocations: at most small embedded fields within existing Cells are updated (specifically balances and allowances stored within Accounts).

Performance characteristics Memory Accounting is `O(1)` for each non-embedded Cell allocated, with a relatively small constant. This would appear to be asymptotically optimal for any system that performs exact memory accounting at a fine-grained level.

This achievement is possible because:

- The Memory Size is computed incrementally and cached for each Cell.
- The number of child cells for each Cell is itself bounded by a small constant
- Memory Size computation is usually lazy, that is it is not performed unless required
- The immutable nature of Convex Cell values means that there is never a need to update Memory Sizes once cached

Accounting for computational costs The direct computational cost of performing this Memory Accounting is factored in to the juice cost of operations that perform new Cell allocations. This compensates Peer operators for the (relatively small) overhead of performing Memory Accounting.

The storage cost is, of course, handled by the general economics of the Memory Accounting model and pool trading.

Cryptographic Primitives

Convex uses cryptographic primitives for the following functions:

- Digital Signature (Ed25519)
- For every Transaction submitted by a Client
- For every Block proposed by a Peer for consensus

- For every Ordering constructed and shared by a Peer
- For every Belief shared by a Peer on the gossip network
- Cryptographic Hashes (SHA3-256)
- For every Cell which forms part of a Decentralised Data Value, a hash of its byte encoding is computed for storage, identity, indexing and verification purposes. This is effectively equal to the VID.
- For every key value used in a map data structure, its hash is computed (if necessary)
- Standard approaches used to store and protect keys in common key file formats (e.g. .pem, .pfx)

As an engineering principle, Convex only uses trusted implementations of cryptographic algorithms in well tested libraries (currently Bouncy Castle, and the cryptographic routines available as standard in the JVM). There is no need to "roll our own" with respect to fundamental crypto algorithms.

Conclusion

Convex presents a new approach to programmable economic systems that provides a powerful combination of scalability, security and decentralisation - suitable for building applications for the Internet of Value.

At the same time, it maintains a certain degree of simplicity. Simple, composable systems offer a more stable and secure foundation to build upon, and Convex therefore features:

- Functional programming on the CVM based on the lambda calculus
- Immutable values for all data structures
- A surprisingly simple consensus algorithm based on CRDTs

We hope that the innovations in Convex and the careful engineering decisions made in its implementation will provide a practical, high performance system for a new generation of decentralised applications and economic value creation.

Contact and Links

To learn more and experiment with the live Convex test network: Convex World

For discussion of this White Paper and other topics relating to Convex, you are very welcome to join our public Discord Server.

Email: info@convex.world