AVES best child of Ethereum by Ibro Fazlić MCS (2022)

AVES is a hard fork of the Ethereum blockchain, with several differences:
1. The mainnet starts from the genesis block and with that the DAG file is tiny and allows everyone to mine coin.
2. The minimum Gas fee has been reduced by more than 50% compared to Ethereum, so we expect significantly lower prices for using smart contracts.
3. The snapshot syncing option has been turned off because our blockchain is very small now, and we don't want nodes to take the chain from peers instead of nodes. The advantages and disadvantages of that procedure have been discussed and it was decided that the benefits outweigh the negatives.
4. A POS upgrade is not planned in the (near) future because this blockchain has a lot of possibilities, also we are investigating implementation subprotocol aka **green protocol**, where mining software will require from miners specific hardware but resources will be used, up to 40% less.
5. The code implemented in our blockchain saves **5% of each block** to a publicly announced address and from which funds will be distributed to innovators of alternative energy sources. This will be done through legal community voting and published through the media.

Accounts

An account is similar to a bank account, for AVS instead of money, where it can be held, transferred to other accounts, and can also be used to execute smart contracts. An account is an entity that is composed of an address along with a private key. The first 20 bytes of the SHA3 hashed public key is the address. Below are the two types of Accounts:

1. **Externally Owned Account**: This is the most basic type of Aves account, it functions similarly to a bitcoin account. A private key controls the address for EOAs.(externally owned account). A person can open as many EOAs as they require. It is created whenever a wallet is created, and it is made with a private key that is required to access EOAs, check balances, send and receive transactions, and establish smart contracts.

Advantages:

Transactions from an external account to a contract account can trigger code that can execute many different actions, such as transferring tokens or even creating a new contract.
Externally Owned Accounts cannot list incoming transactions.
2. **Contract-Based Account**: Contract-based accounts can perform all of the functions of an externally owned account, but unlike EOAs, they are formed when a contract code is deployed, are governed by contract codes, and are accessed using a unique address. When one party accepts a contract, a unique account is formed which contains all of the charges associated with that contract. Each contract is granted a distinct serial number, which is referred to as a contract account.

Advantages:

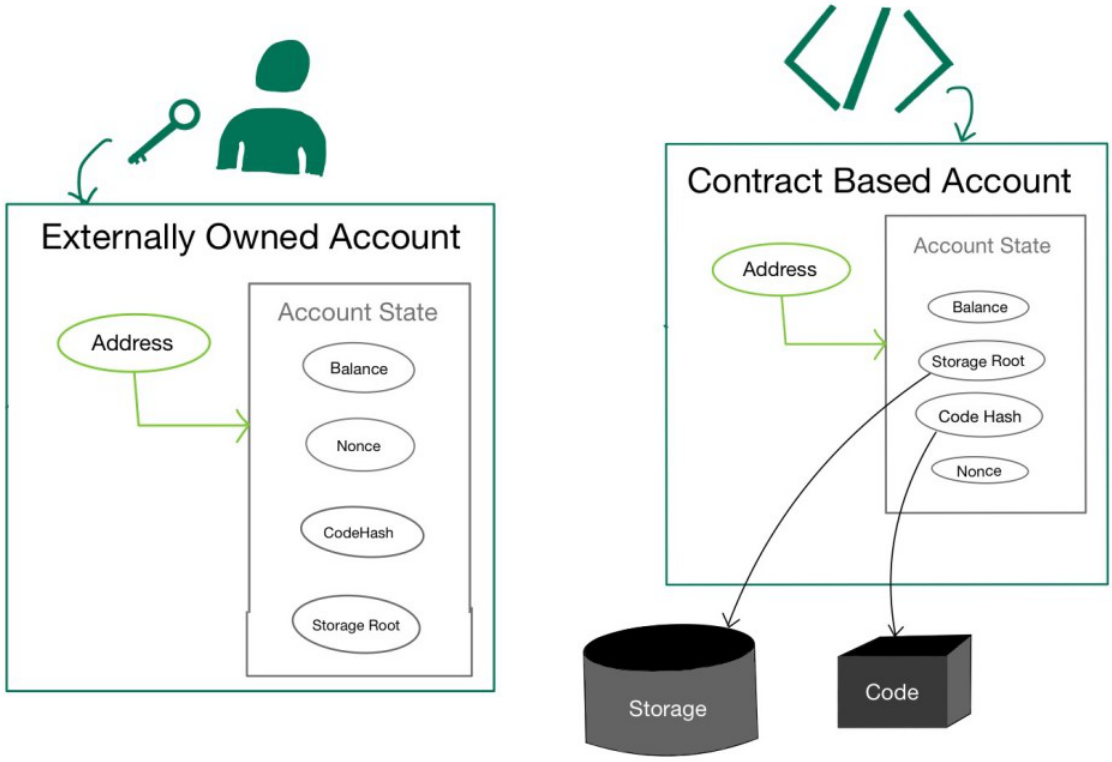
A contract account can list incoming transactions.
Contract accounts can be set up as Multisig Accounts.
A Multisig Account can be structured such that it has a daily limit that you specify, and only if the daily limit is exceeded will multiple signatures be required.
Disadvantages:

Creating contract accounts costs gas because they use the valuable computational and storage resource of the network.
Contract accounts can't initiate new transactions on their own. Instead, contract accounts can only fire transactions in response to other transactions they have received either from an externally owned account or from another contract account.
Types of Contract Accounts
Below are the three types of contract accounts:

Simple Account: The account is created and owned by a single account holder.
Multisig (multisignature) Account: A Multisig Wallet contains several owner Accounts, one of which is also the creator Account.
Simplest Account: A Multisig Wallet contains several owner Accounts, one of which is also the creator Account.
Externally Owned Accounts vs Contract Based Accounts.
Multisig (multisignature) Account: A Multisig Wallet contains several owner Accounts, one of which is also the creator.

Below are the differences between Externally owned accounts and contract-based accounts.
Different Fields in Accounts
Nonce: The nonce in an account indicated the number of transactions that have been sent from that account. This ensures that each transaction is made only once by taking count every time it takes place.
Avs Balance: The balance of an account denotes the amount of Avs present in an Avs repository of the current Avs account.

Contract Code: This is non-mandatory to fill, in case it is present since not all accounts have a contract code. But note, that they cannot be altered once executed.
Code Hash: The value of the code hash for Contract Accounts will be a hash that refers to the code present in that account and since no code is associated with externally owned accounts, therefore, code hash will be an empty string.



Externally Owned Accounts and Key Pairs
An account is a private-public key pair that may be linked to a blockchain address.

It is a “owned” or “externally owned” account if the private key is known and controlled by someone.
Otherwise, we’re talking about smart-contract accounts if the private key is unknown and just an address exists.
Contract accounts do not have a private key connected with them, although externally owned accounts have. Control and access to one’s assets and contracts are granted through the EOA private key.

Control and access to one’s assets and contracts are granted through the EOA private key. The user keeps the private key safe.
While, as the name implies, the account’s public key is open. This key serves as the account’s identity. A one-way cryptographic function is used to produce the public key from the private key.
For example, if you create an account on , you will retain the private key to yourself while sharing the public key.
Transactions between accounts are completed using public keys.

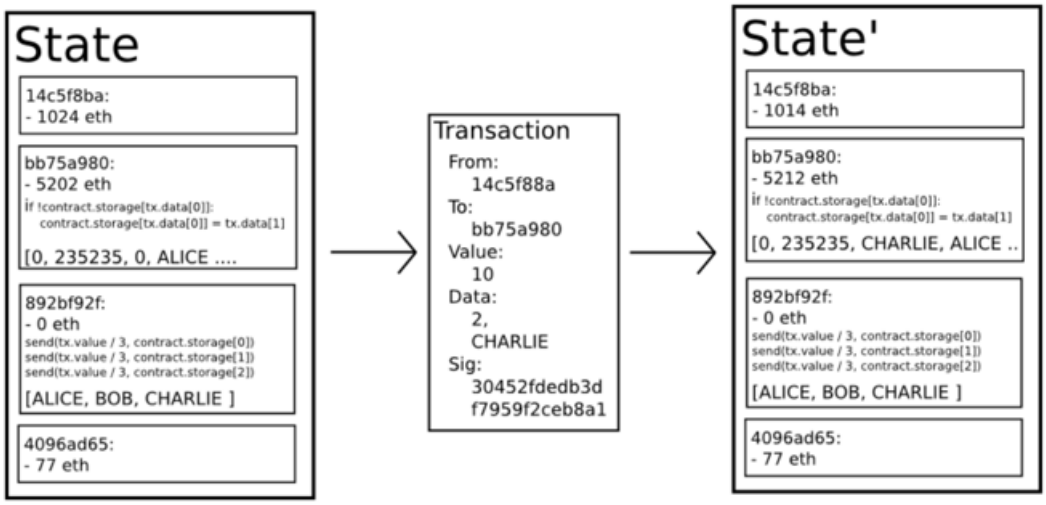
Smart contract accounts. Like EOAs,(Externally Owned Account) smart contract accounts each have a unique public address, and it’s impossible to tell them apart from EOAs by looking at an address. Smart contract accounts too can receive funds and make transactions like EOAs. Generally, the key difference is that no single private key is used to verify transactions. Instead, the logic behind how the account completes transactions is defined in the smart contract code. Smart contracts are programs that run on the Aves blockchain and execute when specific conditions are met. Their functionality within contract accounts means that such accounts, in contrast to EOAs, can, for example, implement access rights that specify by whom, how, and under which conditions transactions can be executed, as well as more complex logic.

Messages and Transactions

"Messages" in Aves are somewhat similar to “transactions” in Bitcoin, but with three important differences. First, an Aves message can be created either by an external entity or a contract, whereas a Bitcoin transaction can only be created externally. Second, there is an explicit option for Aves messages to contain data. Finally, the recipient of an Aves message, if it is a contract account, has the option to return a response; this means that Aves messages also encompass the concept of functions.
The term "transaction" is used in Aves to refer to the signed data package that stores a message to be sent from an externally owned account. Transactions contain the recipient of the message, a signature

identifying the sender, the amount of Avs and the data to send, as well as two values called STARTGAS and GASPRICE. In order to prevent exponential blowup and infinite loops in code, each transaction is required to set a limit to how many computational steps of code execution it can spawn, including both the initial message and any additional messages that get spawned during execution. STARTGAS is this limit, and GASPRICE is the fee to pay to the miner per computational step. If transaction execution "runs out of gas", all state changes revert - except for the payment of the fees, and if transaction execution halts with some gas remaining then the remaining portion of the fees is refunded to the sender. There is also a separate transaction type, and corresponding message type, for creating a contract; the address of a contract is calculated based on the hash of the account nonce and transaction data.
An important consequence of the message mechanism is the "first class citizen" property of Aves - the idea that contracts have equivalent powers to external accounts, including the ability to send message and create other contracts. This allows contracts to simultaneously serve many different roles: for example, one might have a member of a decentralized organization (a contract) be an escrow account (another contract) between an paranoid individual employing custom quantum-proof Lamport signatures (a third contract) and a co-signing entity which itself uses an account with five keys for security (a fourth contract). The strength of the Aves platform is that the decentralized organization and the escrow contract do not need to care about what kind of account each party to the contract is.

Aves State Transition Function



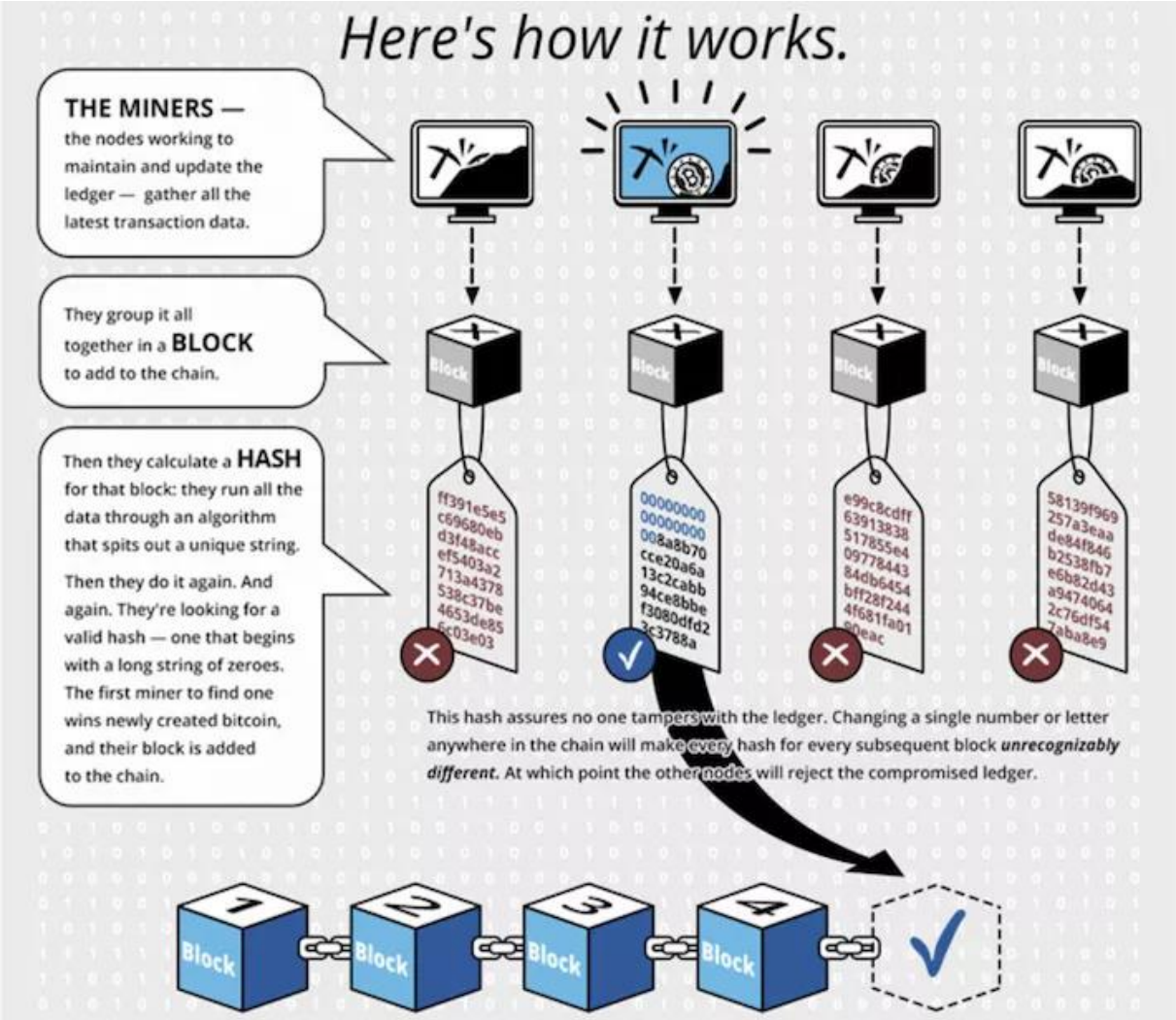
The Aves state transition function, APPLY(S,TX) -> S' can be defined as follows:
1. Check if the transaction is well-formed (ie. has the right number of values), the signature is valid, and the nonce matches the nonce in the sender's account. If not, return an error.
2. Calculate the transaction fee as STARTGAS * GASPRICE, and determine the sending address from the signature. Subtract the fee from the sender's account balance and increment the sender's nonce. If there is not enough balance to spend, return an error.
3. Initialize GAS = STARTGAS, and take off a certain quantity of gas per byte to pay for the bytes in the transaction.
4. Transfer the transaction value from the sender's account to the receiving account. If the receiving account does not yet exist, create it. If the receiving account is a contract, run the contract's code either to completion or until the execution runs out of gas.
5. If the value transfer failed because the sender did not have enough money, or the code execution ran out of gas, revert all state changes except the payment of the fees, and add the fees to the miner's account.
6. Otherwise, refund the fees for all remaining gas to the sender, and send the fees paid for gas consumed to the miner.
For example, suppose that the contract's code is:
if !contract.storage[msg.data[0]]:
contract.storage[msg.data[0]] = msg.data[1]
Note that in reality the contract code is written in the low-level EVM code; this example is written in Serpent, our high-level language, for clarity, and can be compiled down to EVM code. Suppose that the contract's storage starts off empty, and a transaction is sent with 10 Avs value, 2000 gas, 0.001 Avs gasprice, and two data fields: [2, 'CHARLIE'][3]. The process for the state transition function in this case is as follows:
1. Check that the transaction is valid and well formed.
2. Check that the transaction sender has at least 2000 * 0.001 = 2 Avs. If it is, then subtract 2 Avs from the sender's account.
3. Initialize gas = 2000; assuming the transaction is 170 bytes long and the byte-fee is 5, subtract 850 so that there is 1150 gas left.
4. Subtract 10 more Avs from the sender's account, and add it to the contract's account.

5. Run the code. In this case, this is simple: it checks if the contract's storage at index 2 is used, notices that it is not, and so it sets the storage at index 2 to the value CHARLIE. Suppose this takes 187 gas, so the remaining amount of gas is 1150 - 187 = 963
6. Add 963 * 0.001 = 0.963 Avs back to the sender's account, and return the resulting state.
If there was no contract at the receiving end of the transaction, then the total transaction fee would simply be equal to the provided GASPRICE multiplied by the length of the transaction in bytes, and the data sent alongside the transaction would be irrelevant. Additionally, note that contract-initiated messages can assign a gas limit to the computation that they spawn, and if the sub-computation runs out of gas it gets reverted only to the point of the message call. Hence, just like transactions, contracts can secure their limited computational resources by setting strict limits on the sub-computations that they spawn.

Code Execution

The code in Aves contracts is written in a low-level, stack-based bytecode language, referred to as "Ethereum virtual machine code" or "EVM code". The code consists of a series of bytes, where each byte represents an operation. In general, code execution is an infinite loop that consists of repeatedly carrying out the operation at the current program counter (which begins at zero) and then incrementing the program counter by one, until the end of the code is reached or an error or STOP or RETURN instruction is detected. The operations have access to three types of space in which to store data:
● The **stack**, a last-in-first-out container to which 32-byte values can be pushed and popped
● **Memory**, an infinitely expandable byte array
● The contract's long-term **storage**, a key/value store where keys and values are both 32 bytes. Unlike stack and memory, which reset after computation ends, storage persists for the long term.
The code can also access the value, sender and data of the incoming message, as well as block header data, and the code can also return a byte array of data as an output.
The formal execution model of EVM code is surprisingly simple. While the Aves virtual machine is running, its full computational state can be defined by the tuple (block_state, transaction, message, code, memory, stack, pc, gas), where block_state is the global state containing all accounts and includes balances and storage. Every round of execution, the current instruction is found by taking the pc-th byte of code, and each instruction has its own definition in terms of how it affects the tuple. For example, ADD pops two items off the stack and pushes their sum, reduces gas by 1 and increments pc by 1, and SSTORE pushes the top two items off the stack and inserts the second item into the contract's storage at the index specified by the first item, as well as reducing gas by up to 200 and incrementing pc by 1. Although there are many ways to optimize Aves via just-in-time compilation, a basic implementation of Aves can be done in a few hundred lines of code.

Blockchain and Mining



The Aves blockchain is in many ways similar to the Bitcoin blockchain, although it does have some differences. The main difference between Aves and Bitcoin with regard to the blockchain architecture is that, unlike Bitcoin, Aves blocks contain a copy of both the transaction list and the most recent state. Aside from that, two other values, the block number and the difficulty, are also stored in the block. The block validation algorithm in Aves is as follows:

1. Check if the previous block referenced exists and is valid.
2. Check that the timestamp of the block is greater than that of the referenced previous block and less than 15 minutes into the future
3. Check that the block number, difficulty, transaction root, uncle root and gas limit (various low-level Aves-specific concepts) are valid.
4. Check that the proof of work on the block is valid.
5. Let S[0] be the STATE_ROOT of the previous block.
6. Let TX be the block's transaction list, with n transactions. For all in in 0...n-1, setS[i+1] = APPLY(S[i],TX[i]). If any applications returns an error, or if the total gas consumed in the block up until this point exceeds the GASLIMIT, return an error.
7. Let S_FINAL be S[n], but adding the block reward paid to the miner.
8. Check if S_FINAL is the same as the STATE_ROOT. If it is, the block is valid; otherwise, it is not valid.

The approach may seem highly inefficient at first glance, because it needs to store the entire state with each block, but in reality efficiency should be comparable to that of Bitcoin. The reason is that the state is stored in the tree structure, and after every block only a small part of the tree needs to be changed. Thus, in general, between two adjacent blocks the vast majority of the tree should be the same, and therefore the data can be stored once and referenced twice using pointers (ie. hashes of subtrees). A special kind of tree known as a "Patricia tree" is used to accomplish this, including a modification to the Merkle tree concept that allows for nodes to be inserted and deleted, and not just changed, efficiently. Additionally, because all of the state information is part of the last block, there is no need to store the entire blockchain history - a strategy which, if it could be applied to Bitcoin, can be calculated to provide 5-20x savings in space.

Applications

In general, there are three types of applications on top of Aves. The first category is financial applications, providing users with more powerful ways of managing and entering into contracts using their money. This includes sub-currencies, financial derivatives, hedging contracts, savings wallets, wills, and ultimately even

some classes of full-scale employment contracts. The second category is semi-financial applications, where money is involved but there is also a heavy non-monetary side to what is being done; a perfect example is self-enforcing bounties for solutions to computational problems. Finally, there are applications such as online voting and decentralized governance that are not financial at all.

Token Systems

On-blockchain token systems have many applications ranging from sub-currencies representing assets such as USD or gold to company stocks, individual tokens representing smart property, secure unforgeable coupons, and even token systems with no ties to conventional value at all, used as point systems for incentivization. Token systems are surprisingly easy to implement in Aves. The key point to understand is that all a currency, or token systen, fundamentally is is a database with one operation: subtract X units from A and give X units to B, with the proviso that (i) X had at least X units before the transaction and (2) the transaction is approved by A. All that it takes to implement a token system is to implement this logic into a contract.

The basic code for implementing a token system in Serpent looks as follows:

```
from = msg.sender
to = msg.data[0]
value = msg.data[1]
if contract.storage[from] >= value:
contract.storage[from] = contract.storage[from] value
contract.storage[to] = contract.storage[to] + value
```

This is essentially a literal implementation of the "banking system" state transition function described further above in this document. A few extra lines of code need to be added to provide for the initial step of distributing the currency units in the first place and a few other edge cases, and ideally a function would be added to let other contracts query for the balance of an address. But that's all there is to it. Theoretically, Aves-based token systems acting as sub-currencies can potentially include another important feature that on-chain Bitcoin-based meta-currencies lack: the ability to pay transaction fees directly in that currency. The way this would be implemented is that the contract would maintain an Avs balance with which it would refund Avs used to pay fees to the sender, and it would refill this balance by collecting the internal currency units that it takes in fees and reselling them in a constant running auction. Users would thus need to "activate" their accounts with Avs, but once the Avs is there it would be reusable because the contract would refund it each time.

Financial derivatives and Stable-Value Currencies

Financial derivatives are the most common application of a "smart contract", and one of the simplest to implement in code. The main challenge in implementing financial contracts is that the majority of them require reference to an external price ticker; for example, a very desirable application is a smart contract that hedges against the volatility of Avs (or another cryptocurrency) with respect to the US dollar, but doing this requires the contract to know what the value of AVS/USD is. The simplest way to do this is through a "data feed" contract maintained by a specific party (eg. NASDAQ) designed so that that party has the ability to update the contract as needed, and providing an interface that allows other contracts to send a message to that contract and get back a response that provides the price.

Given that critical ingredient, the hedging contract would look as follows:

1. Wait for party A to input 1000 Avs.
2. Wait for party B to input 1000 Avs.
3. Record the USD value of 1000 Avs, calculated by querying the data feed contract, in storage, say this is $x.
4. After 30 days, allow A or B to "ping" the contract in order to send $x worth of Avs (calculated by querying the data feed contract again to get the new price) to A and the rest to B.

Such a contract would have significant potential in crypto-commerce. One of the main problems cited about cryptocurrency is the fact that it's volatile; although many users and merchants may want the security and convenience of dealing with cryptographic assets, they many not wish to face that prospect of losing 23% of the value of their funds in a single day. Up until now, the most commonly proposed solution has been issuer-backed assets; the idea is that an issuer creates a sub-currency in which they have the right to issue and revoke units, and provide one unit of the currency to anyone who provides them (offline) with one unit of a specified underlying asset (eg. gold, USD). The issuer then promises to provide one unit of the underlying asset to anyone who sends back one unit of the crypto-asset. This mechanism allows any non-cryptographic asset to be "uplifted" into a cryptographic asset, provided that the issuer can be trusted.

In practice, however, issuers are not always trustworthy, and in some cases the banking infrastructure is too weak, or too hostile, for such services to exist. Financial derivatives provide an alternative. Here, instead of a single issuer providing the funds to back up an asset, a decentralized market of speculators, betting that the price of a cryptographic reference asset will go up, plays that role. Unlike issuers, speculators have no option to default on their side of the bargain because the hedging contract holds their funds in escrow. Note that this approach is not fully decentralized, because a trusted source is still needed to provide the price ticker, although arguably even still this is a massive improvement in terms of reducing infrastructure requirements (unlike being an issuer, issuing a price feed requires no licenses and can likely be categorized as free speech) and reducing the potential for fraud.

Identity and Reputation Systems

The earliest alternative cryptocurrency of all, Namecoin, attempted to use a Bitcoin-like blockchain to provide a name registration system, where users can register their names in a public database alongside other data. The major cited use case is for a DNS system, mapping domain names like "bitcoin.org" (or, in Namecoin's case, "bitcoin.bit") to an IP address. Other use cases include email authentication and potentially more

advanced reputation systems. Here is the basic contract to provide a Namecoin-like name registration system on Aves:

```
if !contract.storage[tx.data[0]]:
contract.storage[tx.data[0]] = tx.data[1]
```

The contract is very simple; all it is is a database inside the Aves network that can be added to, but not modified or removed from. Anyone can register a name with some value, and that registration then sticks forever. A more sophisticated name registration contract will also have a "function clause" allowing other contracts to query it, as well as a mechanism for the "owner" (ie. the first registerer) of a name to change the data or transfer ownership. One can even add reputation and web-of-trust functionality on top.

Decentralized File Storage

Over the past few years, there have emerged a number of popular online file storage startups, the most prominent being Dropbox, seeking to allow users to upload a backup of their hard drive and have the service store the backup and allow the user to access it in exchange for a monthly fee. However, at this point the file storage market is at times relatively inefficient; a cursory look at various existing solutions shows that, particularly at the "uncanny valley" 20-200 GB level at which neither free quotas nor enterprise-level discounts kick in, monthly prices for mainstream file storage costs are such that you are paying for more than the cost of the entire hard drive in a single month. Aves contracts can allow for the development of a decentralized file storage ecosystem, where individual users can earn small quantities of money by renting out their own hard drives and unused space can be used to further drive down the costs of file storage.

The key underpinning piece of such a device would be what we have termed the "decentralized Dropbox contract". This contract works as follows. First, one splits the desired data up into blocks, encrypting each block for privacy, and builds a Merkle tree out of it. One then makes a contract with the rule that, every N blocks, the contract would pick a random index in the Merkle tree (using the previous block hash, accessible from contract code, as a source of randomness), and give X Avs to the first entity to supply a transaction with a simplified payment verification-like proof of ownership of the block at that particular index in the tree. When a user wants to re-download their file, they can use a micropayment channel protocol (eg. pay 1 szabo per 32 kilobytes) to recover the file; the most fee-efficient approach is for the payer not to publish the transaction until the end, instead replacing the transaction with a slightly more lucrative one with the same nonce after every 32 kilobytes.

An important feature of the protocol is that, although it may seem like one is trusting many random nodes not to decide to forget the file, one can reduce that risk down to near-zero by splitting the file into many pieces via secret sharing, and watching the contracts to see each piece is still in some node's possession. If a contract is still paying out money, that provides a cryptographic proof that someone out there is still storing the file.

Decentralized Autonomous Organizations

The general concept of a "decentralized organization" is that of a virtual entity that has a certain set of members or shareholders which, perhaps with a 67% majority, have the right to spend the entity's funds and modify its code. The members would collectively decide on how the organization should allocate its funds. Methods for allocating a DAO's funds could range from bounties, salaries to even more exotic mechanisms such as an internal currency to reward work. This essentially replicates the legal trappings of a traditional company or nonprofit but using only cryptographic blockchain technology for enforcement. So far much of the talk around DAOs has been around the "capitalist" model of a "decentralized autonomous corporation" (DAC) with dividend-receiving shareholders and tradable shares; an alternative, perhaps described as a "decentralized autonomous community", would have all members have an equal share in the decision making and require 67% of existing members to agree to add or remove a member. The requirement that one person can only have one membership would then need to be enforced collectively by the group.

A general outline for how to code a DO is as follows. The simplest design is simply a piece of self-modifying code that changes if two thirds of members agree on a change. Although code is theoretically immutable, one can easily get around this and have de-facto mutability by having chunks of the code in separate contracts, and having the address of which contracts to call stored in the modifiable storage. In a simple implementation of such a DAO contract, there would be three transaction types, distinguished by the data provided in the transaction:

- [0,i,K,V] to register a proposal with index i to change the address at storage index K to value V
- [0,i] to register a vote in favor of proposal i
- [2,i] to finalize proposal i if enough votes have been made

The contract would then have clauses for each of these. It would maintain a record of all open storage changes, along with a list of who voted for them. It would also have a list of all members. When any storage change gets to two thirds of members voting for it, a finalizing transaction could execute the change. A more sophisticated skeleton would also have built-in voting ability for features like sending a transaction, adding members and removing members, and may even provide for Liquid Democracy-style vote delegation (ie. anyone can assign someone to vote for them, and assignment is transitive so if A assigns B and B assigns C then C determines A's vote). This design would allow the DO to grow organically as a decentralized community, allowing people to eventually delegate the task of filtering out who is a member to specialists, although unlike in the "current system" specialists can easily pop in and out of existence over time as individual community members change their alignments.

An alternative model is for a decentralized corporation, where any account can have zero or more shares, and

two thirds of the shares are required to make a decision. A complete skeleton would involve asset management functionality, the ability to make an offer to buy or sell shares, and the ability to accept offers (preferably with an order-matching mechanism inside the contract). Delegation would also exist Liquid Democracy-style, generalizing the concept of a "board of directors".
In the future, more advanced mechanisms for organizational governance may be implemented; it is at this point that a decentralized organization (DO) can start to be described as a decentralized autonomous organization (DAO). The difference between a DO and a DAO is fuzzy, but the general dividing line is Avs the governance is generally carried out via a political-like process or an “automatic” process; a good intuitive test is the “no common language” criterion: can the organization still function if no two members spoke the same language? Clearly, a simple traditional shareholder-style corporation would fail, whereas something like the Bitcoin protocol would be much more likely to succeed. Robin Hanson’s futarchy, a mechanism for organizational governance via prediction markets, is a good example of what truly “autonomous” governance might look like. Note that one should not necessarily assume that all DAOs are superior to all DOs; automation is simply a paradigm that is likely to have have very large benefits in certain particular places and may not be practical in others, and many semi-DAOs are also likely to exist.
Further Applications

1. Savings wallets. Suppose that Alice wants to keep her funds safe, but is worried that she will lose or someone will hack her private key. She puts Avs into a contract with Bob, a bank, as follows:
● Alice alone can withdraw a maximum of 1% of the funds per day.
● Bob alone can withdraw a maximum of 1% of the funds per day, but Alice has the ability to make a transaction with her key shutting off this ability.
● Alice and Bob togAvs can withdraw anything.
Normally, 1% per day is enough for Alice, and if Alice wants to withdraw more she can contact Bob for help. If Alice's key gets hacked, she runs to Bob to move the funds to a new contract. If she loses her key, Bob will get the funds out eventually. If Bob turns out to be malicious, then she can turn off his ability to withdraw.

2. Crop insurance. One can easily make a financial derivatives contract but using a data feed of the weather instead of any price index. If a farmer in Iowa purchases a derivative that pays out inversely based on the precipitation in Iowa, then if there is a drought, the farmer will automatically receive money and if there is enough rain the farmer will be happy because their crops would do well.
3. A decentralized data feed. For financial contracts for difference, it may actually be possible to decentralize the data feed via a protocol called "SchellingCoin". SchellingCoin basically works as follows: N parties all put into the system the value of a given datum (eg. the AVS/USD price), the values are sorted, and everyone between the 25th and 75th percentile gets one token as a reward. Everyone has the incentive to provide the answer that everyone else will provide, and the only value that a large number of players can realistically agree on is the obvious default: the truth. This creates a decentralized protocol that can theoretically provide any number of values, including the AVS/USD price, the temperature in Berlin or even the result of a particular hard computation.
4. Smart multi-signature escrow. Bitcoin allows multisignature transaction contracts where, for example, three out of a given five keys can spend the funds. Aves allows for more granularity; for example, four out of five can spend everything, three out of five can spend up to 10% per day, and two out of five can spend up to 0.5% per day. Additionally, Aves multisig is asynchronous - two parties can register their signatures on the blockchain at different times and the last signature will automatically send the transaction.
5. Cloud computing. The EVM technology can also be used to create a verifiable computing environment, allowing users to ask others to carry out computations and then optionally ask for proofs that computations at certain randomly selected checkpoints were done correctly. This allows for the creation of a cloud computing market where any user can participate with their desktop, laptop or specialized server, and spot-checking with security deposits can be used to ensure that the system is trustworthy (ie. nodes cannot profitably cheat). Although such a system may not be suitable for all tasks; tasks that require a high level of inter-process communication, for example, cannot easily be done on a large cloud of nodes. Other tasks, however, are much easier to parallelize; projects like SETI@home, folding@home and genetic algorithms can easily be implemented on top of such a platform.
6. Peer-to-peer gambling. Any number of peer-to-peer gambling protocols, such as Frank Stajano and Richard Clayton's Cyberdice, can be implemented on the Aves blockchain. The simplest gambling protocol is actually simply a contract for difference on the next block hash, and more advanced protocols can be built up from there, creating gambling services with near-zero fees that have no ability to cheat.
7. Prediction markets. Provided an oracle or SchellingCoin, prediction markets are also easy to implement, and prediction markets with SchellingCoin may prove to be the first mainstream application of futarchy as a governance protocol for decentralized organizations.
8. On-chain decentralized marketplaces, using the identity and reputation system as a base.

Precautions

Modified GHOST Implementation
The "Greedy Heaviest Observed Subtree" (GHOST) protocol is an innovation first introduced by Yonatan Sompolinsky and Aviv Zohar in December 2013.
The Ghost Protocol is a development in the cryptographic protocol behind Bitcoin that allows for transactions to be processed without broadcasting them. It is an end-to-end encryption protocol that provides authentication without having to rely on centralized trust authorities. It can be either symmetric or asymmetric, depending on how it’s used. The

principle of GHOST is that the sender only sends a ghost (or dummy) packet to the receiver, which can then reply with as many packets as it needs.

The sender creates a digital signature by encrypting the packet with the receiver's public key.
The receiver decrypts it using his private key (the public key is used to encrypt).
If the decryption was done correctly, the sender is assumed to be who he claims to be, and the transaction is accepted.
He may also send this ghost packet to other receivers (i.e., the transaction is broadcasted) using the same procedure.
Since there may be more than one receiver, this protocol is called "GHOST", which stands for " Greedy Heaviest Observed Sub-Tree", as a reference to how it routes packets through other nodes in addition to its direct route between sender and receiver.

Need For GHOST Protocol

The transactions in blockchain can be published from anywhere. In PoW blockchains like Bitcoin, Aves, etc due to the random nature of hashing two miners can be working on the same transaction producing two blocks.

Only one of these transactions can be added to the main blockchain.
This means that all the work done by the second miner on verifying the second block is lost (orphaned).
The miner does not get rewarded. These blocks are called uncle blocks in Aves.
GHOST protocol is a chain selection rule that makes use of previously orphaned blocks and adds them to the main blockchain and partially rewards the miner also. This increases the difficulty of an attack on the network as now winning miner is not the only one who owns the computing power. More nodes retain the power and discourage the need for centralized mining pools on larger chains.

Implementation of GHOST Protocol
Bitcore, a bitcoin development team implemented the GHOST Protocol. This is also the first public implementation of the GHOST protocol.

They can be used to Avs in various ways to maximize their effectiveness.
For example, a GHOST channel can be used to exchange coins or other digital assets that do not benefit from the benefits of Bitcoin's block verification times and consensus process (e.g., coins that require trustless processing such as stablecoins).
How Does the GHOST Protocol Work?

GHOST works by sending dummy/empty packets or 'ghosts' to the receiver.
A sender sends a ghost packet with a header and encrypted payload, but no block reward (i.e., no transactions), and waits for an empty packet from the receiver.
If an empty packet is received, then it means that the receiver received the ghost, so he can send up to 2*pendingtxns to the network without broadcasting them.
When more than one node has a pending transaction in the queue, then there must be some sort of protocol in place for deciding which node will broadcast its block (i.e., which node will win).

Pros of GHOST Protocol

Scalability: GHOST protocol was designed and built with scalability and security in mind so that it can easily handle thousands.
Easy transactions: In a world where cryptocurrency transactions can be completed within seconds from anywhere in the world, GHOST Protocol allows individuals to make transactions with ease through efficient use of computing power.
Freedom to developers: If a developer doesn't want to take on the responsibility of maintaining their own infrastructure, they can utilize GHOST-powered smart contracts which run on top of it instead.
S time and effort: It s them time and effort. Smart contracts are much quicker and easier than writing applications from scratch. It allows for more people to get involved in the dApp space. This is a great thing for new developers and entrepreneurs to get involved in.
Better transparency: It provides better transparency than Aves's ERC20 standard (which platforms like **MyEthWallet** and **MetaMask** still use). It allows developers to accept payments while being completely anonymous. A non-anonymous or pseudonymous payment system is much preferred by hackers and online phishers, preventing them from either targeting you or stealing your funds.
Cons of GHOST Protocol
Hampers adoption growth: It hampers adoption growth.
When not in use over-complicated: If no one wants to use the GHOST protocol, it will remain an over-complicated means of paying users in their tokens or Avs.
Not viable option: It's not a viable option for certain platforms. Blockchain-based games are the first thing that comes to mind.
Makes dApps expensive: It makes dApps more expensive.
Gas costs for all transactions: dApps utilizing this protocol need to pay the gas costs of all transactions, even those that don't involve them.

Fees

A gas fee is the amount of Avs (AVS) required for an Aves blockchain network user to conduct a transaction on the network.

Gas fees are used to compensate Aves miners for their work in verifying transactions and securing the network. Gas fees also help keep the network from becoming bogged down by malicious users spamming the network with transactions. Aves gas fees vary because the formula used to calculate them is dynamic. Large fees and relatively slow speeds are common criticisms of the Aves network. Gas fees are paid in Avs and are denominated in gwei or gigawei. Each gwei is equal to 0.000000001 AVS.
The gas fee is dependent on two factors. Gas units and gas price.

Gas fee= gas units X gas price

Gas units is a number that depends on the amount of computation required for a transaction. For e.g., if you send some Ave to someone, it requires 11,000 gas units. It's the minimum number of units required for any transaction. On the other hand, Gas price is determined by the demand for making transactions. The more traffic, the higher the price. This is why you don't pay the same gas fee each time you transact.
In our case, we have a lower gas fee than Ethereum, so we don't expect too high prices in case of huge demands for smart contracts
Our minimal gas fee is:
TxGas uint64 = 11000
TxGasContractCreation uint64 = 23000

Computation And Turing-Completeness

What is Turing completeness?

Turing-completeness is a word defined by Alan Turing which it describes the idea that some computing machines are capable of performing any task a computer can perform.

The concept of Turing-completeness is one at the heart of software and application development, where it allows code to be written without having to check beforehand if it will work or not.
In other words, one can write your program without worrying about what else is allowed for it to do.
This is essential in determining usability as well as many other aspects of the software. It is also important to know what "Turing-complete" means and how it relates to Aves.
In Turing's paper, the concept of Turing-complete machines is used to disprove the possibility of true artificial intelligence.
For instance, a machine can eventually imitate the behaviors of a human. In practical terms, this means that "Turing-complete" allows programmers to write code that can be used by any computer to achieve any result.
It is necessary for introducing new techniques and ideas into software programming such as functional programming or even for understanding ideas about universal computation with regard to general computing.

One of the main obstacles that cryptocurrency runs into is reliance on a third party, typically an entity such as a bank.

These companies are responsible for ensuring that a cryptocurrency can be used in everyday transactions because it must be compatible with traditional banking services. Turing completeness is a characteristic of a programming language.
A language is Turing complete if it can be used to simulate a Turing machine, which means that an appropriately designed program can solve any problem that a Universal Turing Machine (UTM) can solve.
In order for this to be feasible, programs must be free from restrictions, such as halting and infinite loops.
Theoretically, Turing's completeness enables the development of highly advanced programs in one language and allows other projects or companies to create highly advanced applications using the same tools.

Key Points

Aves can be built on the blockchain that has been built right now with no need to upgrade. So, one gets a cheap and scalable Blockchain offering storage and processing power that is almost limitless and at the same time evolving.
Aves will change the whole Blockchain ecosystem by making it possible to do many more things on it. All thanks to its Turing-complete language Solidity.
The whole concept of cryptocurrencies and smart contracts is based on the idea of Turing-complete languages.
Smart contracts are being used for all kinds of applications including common transactions like payments, buying a car, or even licensing music or software. Smart contracts can be used in an efficient way to run state channels between users and to make payments transparently.
ERC20 is the standard document that provides a structure for tokens (works on Aves) and it is compatible with most tokens projects like City Coin, Request Network, and others.
Initial coin offering (ICO) which is a new way to raise money for a project – uses ERC20 as the monetary unit without issuing tradable ERC20 tokens before the ICO launch. Aves as Turing Completeness.

Since it relies on programmable smart contracts, Aves is not reliant on third-party services to function.
This means that, theoretically, one could buy a house or make other major purchases on the Aves blockchain through the use of a smart contract.
However, there are concerns regarding or not this is feasible due to the high costs associated with Turing complete systems and their ability to run continuously without human intervention.

Theoretically, there are several challenges associated with Turing complete cryptocurrencies. As the cryptocurrency industry continues to grow and expand, new ideas are being brought forth constantly — many of which would not have been possible without Turing completeness.

Does a System Have To Be Turing Complete To Be Useful in Blockchain?

A system has to be Turing complete in order to be useful in blockchain, but it can have all the other desirable properties of a blockchain, such as decentralization and trustless transactions.

A system is Turing complete if it can simulate an arbitrary computer program. Turing complete systems have to be able to run any possible computation, which includes the most complex types of computation such as those found in blockchain. This type of system also often has better performance than other systems because it can use a set of rules which are more efficient when solving problems with many steps.

In addition to being Turing complete, a system must also be decentralized and allow trustless transactions according to consensus in order for it to be useful in the blockchain. These properties are necessary for the security and consistency that a blockchain needs in order for its data records or “blocks” to have value and meaning.

Aves is a very new technology with many possibilities that can disrupt our lives in the future. In short, it is a complex network of computers that allows one to create own currency and it is totally decentralized and free to use.

Mining Centralization

The Bitcoin mining algorithm basically works by having miners compute SHA256 on slightly modified versions of the block header millions of times over and over again, until eventually one node comes up with a version whose hash is less than the target (currently around 2^{190}). However, this mining algorithm is vulnerable to two forms of centralization. First, the mining ecosystem has come to be dominated by ASICs (application-specific integrated circuits), computer chips designed for, and therefore thousands of times more efficient at, the specific task of Bitcoin mining. This means that Bitcoin mining is no longer a highly decentralized and egalitarian pursuit, requiring millions of dollars of capital to effectively participate in. Second, most Bitcoin miners do not actually perform block validation locally; instead, they rely on a centralized mining pool to provide the block headers. This problem is arguably worse: as of the time of this writing, the top two mining pools indirectly control roughly 50% of processing power in the Bitcoin network, although this is mitigated by the fact that miners can switch to other mining pools if a pool or coalition attempts a 51% attack.

The current intent at Aves is to use a mining algorithm based on randomly generating a unique hash function for every 1000 nonces, using a sufficiently broad range of computation to remove the benefit of specialized hardware. Such a strategy will certainly not reduce the gain of centralization to zero, but it does not need to. Note that each individual user, on their private laptop or desktop, can perform a certain quantity of mining activity almost for free, paying only electricity costs, but after the point of 100% CPU utilization of their computer additional mining will require them to pay for both electricity and hardware. ASIC mining companies need to pay for electricity and hardware starting from the first hash. Hence, if the centralization gain can be kept to below this ratio, (E + H) / E, then even if ASICs are made there will still be room for ordinary miners. Additionally, we intend to design the mining algorithm so that mining requires access to the entire blockchain, forcing miners to store the entire blockchain and at least be capable of verifying every transaction. This removes the need for centralized mining pools; although mining pools can still serve the legitimate role of evening out the randomness of reward distribution, this function can be served equally well by peer-to-peer pools with no central control. It additionally helps fight centralization, by increasing the number of full nodes in the network so that the network remains reasonably decentralized even if most ordinary users prefer light clients.

Scalability

One common concern about Aves is the issue of scalability. Like Bitcoin, Aves suffers from the flaw that every transaction needs to be processed by every node in the network. With Bitcoin, the size of the current blockchain rests at about 437 GB, growing by about 1 MB per hour. If the Bitcoin network were to process Visa's 2000 transactions per second, it would grow by 1 MB per three seconds (1 GB per hour, 8 TB per year). Aves is likely to suffer a similar growth pattern, worsened by the fact that there will be many applications on top of the Aves blockchain instead of just a currency as is the case with Bitcoin, but ameliorated by the fact that Aves full nodes need to store just the state instead of the entire blockchain history.

The problem with such a large blockchain size is centralization risk. If the blockchain size increases to, say, 100 TB, then the likely scenario would be that only a very small number of large businesses would run full nodes, with all regular users using light SPV nodes. In such a situation, there arises the potential concern that the full nodes could band and all agree to cheat in some profitable fashion (eg. change the block reward, give themselves BTC). Light nodes would have no way of detecting this immediately. Of course, at least one honest full node would likely exist, and after a few hours information about the fraud would trickle out through channels like Reddit, but at that point it would be too late: it would be up to the ordinary users to organize an effort to blacklist the given blocks, a massive and likely infeasible coordination problem on a similar scale as that of pulling off a successful 51% attack. In the case of Bitcoin, this is currently a problem, but there exists a blockchain modification suggested by Peter Todd which will alleviate this issue.

In the near term, Aves will use two additional strategies to cope with this problem. First, because of the blockchain-based mining algorithms, at least every miner will be forced to be a full node, creating a lower bound on the number of full nodes. Second and more importantly, however, we will include an intermediate state tree root in the blockchain after processing each transaction. Even if block validation is centralized, as long as one honest verifying node exists, the centralization problem can be circumvented via a verification protocol. If a miner publishes an invalid block, that block must either be badly formatted, or the state S[n] is

incorrect. Since S[0] is known to be correct, there must be some first state S[i] that is incorrect where S[i-1] is correct. The verifying node would provide the index i, along with a "proof of invalidity" consisting of the subset of Patricia tree nodes needing to process APPLY(S[i-1],TX[i]) -> S[i]. Nodes would be able to use those nodes to run that part of the computation, and see that the S[i] generated does not match the S[i] provided.
Another, more sophisticated, attack would involve the malicious miners publishing incomplete blocks, so the full information does not even exist to determine or not blocks are valid. The solution to this is a challenge-response protocol: verification nodes issue "challenges" in the form of target transaction indices, and upon receiving a node a light node treats the block as untrusted until another node, the miner or another verifier, provides a subset of Patricia nodes as a proof of validity.

Decentralized Applications

The contract mechanism described above allows anyone to build what is essentially a command line application run on a virtual machine that is executed by consensus across the entire network, allowing it to modify a globally accessible state as its “hard drive”. However, for most people, the command line interface that is the transaction sending mechanism is not sufficiently user-friendly to make decentralization an attractive mainstream alternative. To this end, a complete “decentralized application” should consist of both low-level business-logic components, whether implemented entirely on Aves, using a combination of Aves and other systems (eg. a P2P messaging layer, one of which is currently planned to be put into the Aves clients) or other systems entirely, and high-level graphical user interface components. The Aves client’s design is to serve as a web browser, but include support for a “Ave” Javascript API object, which specialized web pages viewed in the client will be able to use to interact with the Aves blockchain. From the point of view of the “traditional” web, these web pages are entirely static content, since the blockchain and other decentralized protocols will serve as a complete replacement for the server for the purpose of handling user-initiated requests. Eventually, decentralized protocols, hopefully themselves in some fashion using Aves, may be used to store the web pages themselves.

Conclusion

The Aves protocol was originally conceived as an upgraded version of a cryptocurrency, providing advanced features such as on-blockchain escrow, withdrawal limits and financial contracts, gambling markets and the like via a highly generalized programming language. The Aves protocol would not "support" any of the applications directly, but the existence of a Turing-complete programming language means that arbitrary contracts can theoretically be created for any transaction type or application. What is more interesting about Aves, however, is that the Aves protocol moves far beyond just currency. Protocols and decentralized applications around decentralized file storage, decentralized computation and decentralized prediction markets, among dozens of other such concepts, have the potential to substantially increase the efficiency of the computational industry, and provide a massive boost to other peer-to-peer protocols by adding for the first time an economic layer. Finally, there is also a substantial array of applications that have nothing to do with money at all.
The concept of an arbitrary state transition function as implemented by the Aves protocol provides for a platform with unique potential; rather than being a closed-ended, single-purpose protocol intended for a specific array of applications in data storage, gambling or finance, Aves is open-ended by design, and we believe that it is extremely well-suited to serving as a foundational layer for a very large number of both financial and non-financial protocols in the years to come.

References

Ethereum White paper by Vitalik Buterin 2014 https://ethereum.org/en/whitepaper/

GeeksforGeeks https://www.geeksforgeeks.org/

Further Reading

1. Intrinsic value: https://tinyurl.com/BitcoinMag-IntrinsicValue
2. Smart property: https://en.bitcoin.it/wiki/Smart_Property
3. Smart contracts: https://en.bitcoin.it/wiki/Contracts
4. B-money: http://www.weidai.com/bmoney.txt
5. Reusable proofs of work: http://www.finney.org/~hal/rpow/
6. Secure property titles with owner authority: http://szabo.best.vwh.net/securetitle.html
7. Bitcoin whitepaper: http://bitcoin.org/bitcoin.pdf
8. Namecoin: https://namecoin.org/
9. Zooko's triangle: http://en.wikipedia.org/wiki/Zooko's_triangle
10. Colored coins whitepaper: https://tinyurl.com/coloredcoin-whitepaper
11. Mastercoin whitepaper: https://github.com/mastercoin-MSC/spec
12. Decentralized autonomous corporations, Bitcoin Magazine: https://tinyurl.com/Bootstrapping-DACs
13. Simplified payment verification:https://en.bitcoin.it/wiki/Scalability#Simplifiedpaymentverification
14. Merkle trees: http://en.wikipedia.org/wiki/Merkle_tree
15. Patricia trees: http://en.wikipedia.org/wiki/Patricia_tree

16. GHOST: http://www.cs.huji.ac.il/~avivz/pubs/13/btc_scalability_full.pdf
17. StorJ and Autonomous Agents, Jeff Garzik: https://tinyurl.com/storj-agents
18. Mike Hearn on Smart Property at Turing Festival: http://www.youtube.com/watch?v=Pu4PAMFPo5Y
19. Aves RLP: https://github.com/Aves/wiki/wiki/%5BEnglish%5D-RLP
20. Aves Merkle Patricia trees: https://github.com/Aves/wiki/wiki/%5BEnglish%5D-Patricia-Tree
21. Peter Todd on Merkle sum trees:http://sourceforge.net/p/bitcoin/mailman/message/31709140/