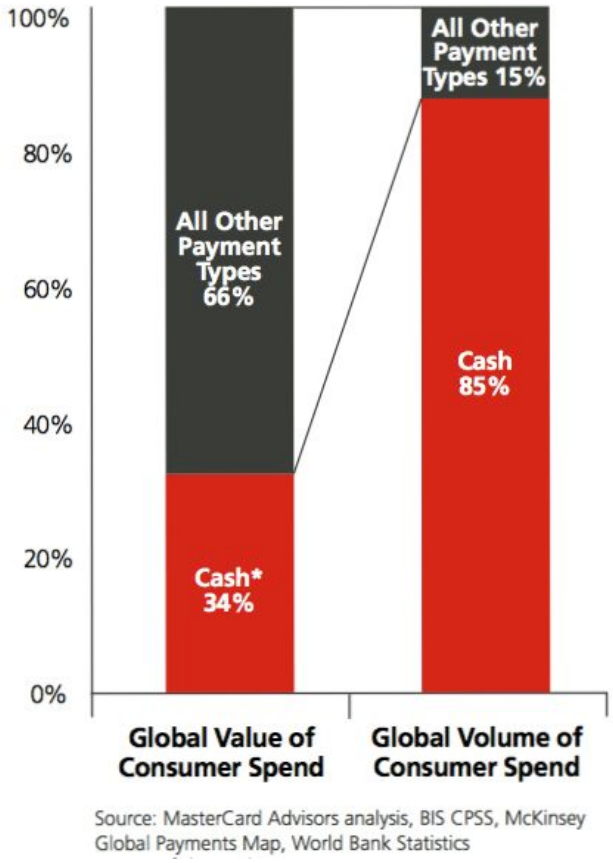
Technical Draft V 1.1

This draft is a high-level introduction about Trestor Network. The draft, however, is not the final technical white paper, but rather our first step towards it. Readers are encouraged to help us improve this draft by commenting in their suggestions. We apologise in advance, for any grammatical, flow and content errors that you may encounter, we would greatly appreciate if you can point such inconsistencies or share any other feedback that can help us improve this draft. Thank you for your time and effort.

Introduction

Cash accounts for about 85% of all global consumer transactions. While most other Fintech start-ups are focused on innovating around 15% non-cash transactions, Trestor is determined to improve cash transaction payments, especially for the highly under-banked 1 billion people at the bottom of the economic pyramid.

Our solution will not only empower individuals, it would also help nations, burden of cash usage on national economies is substantial, representing a loss of as much a 1.5% of GDP.



Source: MasterCard Advisors analysis, BIS CPSS, McKinsey Global Payments Map, World Bank Statistics

Trestor's mission is much greater than just improving cash transactions, it is to build the ***"Most efficient Money, Payment, and Market system for the world"***, especially for the under-banked 1 billion people. A poor person's inability to participate in the connected digital world and leverage its efficiency is one of his biggest handicaps.

It is fair to say that a billion people are under-banked not despite the government but because of the government. Various outdated/ineffective banking regulations restrict entrepreneurial innovation in finance.

A cryptocurrency protocol solution, if implemented correctly, has the potential to cut-through 400 years of legacy retail banking structures thereby eliminating most middlemen and money managers. It can help shape a world where money and retail banking is governed by the laws of mathematics instead of bureaucrats and politicians.

However, current crypto protocols appear to be solutions looking for a problem. Trestor Foundation has developed 'Trestor Network' aka T-Net solution by first identifying and understanding systemic global financial and economic challenges thereby working its way backwards to devise an elegant solution. At Trestor, we consider decentralization to be an excellent tool but not the end goal itself.

Key technical differences: T-Net and Bitcoin

1. Signature Algorithm Curve Selection
2. Accounts, not UTXOs
3. Distributed consensus via Node Voting not mining

Signature Algorithms

In Trestor, a user has pairs of private and public keys in order to manage his Trests. A user's balance is associated with a public key. The standard way for a user to spend his money is by creating a transaction signed with the corresponding private key.

Trestor differs from other distributed payment protocols by choosing the EdDSA signature algorithm over ECDSA with secp256k1, which is used in most other payment protocols.

ECDSA stands for Elliptic Curve Digital Signature Algorithm, which is the NIST (US government) standard for digital signatures. The secp256k1 curve was published by Ontario-based Certicom, today a subsidiary of BlackBerry.

EdDSA is a newer signature system proposed by a team of renowned experts led by the cryptographer Daniel Bernstein. It uses a Twisted Edwards curve known as Ed25519.

The Curve Matters

The specific choice of the curve is important, since many curves that have been standardized by the US government caused a lot of suspicion in the cryptographic and civil-liberties communities, especially after Snowden's publications about NSA activities. The creation of a suitable curve requires selection of several parameters. If it is unclear, where these parameters came from, it is possible, that the creator selected them based on a secret back door.

Public-key cryptography has been around for a while. The two most popular families are RSA-based systems (prime factorization) and Diffie-Hellman (discrete logarithm). Both of them are based on operations of large integer numbers, which are hard to invert.

The problem with those is that since computing power is increasing, key size and computational overhead do not scale well anymore. Elliptic-curve crypto has become popular in recent years because it yields comparable levels of security with much smaller key size (256-bit keys instead of 2048 or 4096-bit keys). Most elliptic-curve systems are similar to Diffie-Hellman systems, where the

integer numbers have been replaced by points on a geometric curve. For example, ECDSA related to the DSA signature system.

Most cryptographers agree that elliptic-curve cryptography is a good idea in general, but the crucial problem is the selection of the curve. While, in classical DSA, every user can select his own parameters at random, in ECDSA those parameters have to be selected very carefully to avoid weak curves. Because of this, standardization bodies and companies (like NIST, Certicom, telco companies ...) have published lists with many curves and users select a curve from those lists. Only the private key itself is picked by the user.

The curve secp256k1 in from one such list published by Certicom Corporation. There are many speculations why Satoshi chose this specific curve.

One good thing about secp256k1 is that it has very simple parameters. An elliptic curve is a curve that satisfies the equation $y^2 = x^3 + ax + b$, where a and b are the parameters that have to be selected. In secp256k1, these parameters are $a=0$ and $b=7$. These simple parameters make it very unlikely that they were selected with the knowledge of a backdoor as discussed before.

But the curve comes with some problems. As cryptographer Nicolas Curtois points out, the curve has some known weaknesses that make it easier to break than usual. Even though there is still no practical attack known, this is considered a bad sign by most cryptographers.

Ed25519, the curve in EdDSA, has been selected by the team with the above and other known weaknesses in mind. The parameters were generated in a way that can be reproduced by interested experts in order to show that there are no backdoors.

Accounts, not UTXO

Bitcoin, along with many of its derivatives, stores data about users' balances in a structure based on *unspent transaction outputs* (UTXOs): the entire state of the system consists of a set of "unspent outputs" (think, "coins"), such that each coin has an owner and a value, and a transaction spends one or more coins and creates one or more new coins, subject to the validity constraints:

- Every referenced input must be valid and not yet spent
- The transaction must have a signature matching the owner of the input for every input
- The total value of the inputs must equal or exceed the total value of the outputs

A user's "balance" in the system is thus the total value of the set of coins for which the user has a private key capable of producing a valid signature.

In Trestor Network, the state stores a list of accounts where each account has a balance, and a transaction is valid if the sending account has enough balance to pay for it, in which case the sending account is debited and the receiving account is credited with the value.

Distributed Consensus - Overview

Consensus is the most fundamental problem in a distributed system where all the processes/nodes agree on some data values to make some progress (liveness). Many other distributed problems such as leader election, atomic broadcasts can be solved using consensus. It is one of the hardest problem to solve as processes/nodes may crash or become unresponsive etc. FLP theorem (Fischer et. al 1985 [1]) proves that it is impossible to guarantee consensus in an asynchronous distributed system in case of even one faulty process. Practical systems are mostly asynchronous and we, therefore, try to work around with an algorithm which maximizes the probability of agreement and is enough for practical purposes.

Byzantine Faults

Nodes in a distributed system may crash. They may also behave incorrectly or maliciously which may lead to erroneous behavior of the system if not handled properly. Sometimes a node may decide to act in an undesired manner for its own selfish benefits. Achieving consensus in the presence of Byzantine Faults is, therefore, harder. Byzantine Generals' Problem [2] is one of the famous examples where it is impossible to decide on a coordinated time to attack by both attacking army units in the presence of unreliable messengers.

Goals of Consensus in the Presence of Byzantine Faults

In the presence of Byzantine Failures and for a distributed payment system use case, a decentralized consensus protocol must satisfy the following requirements:

1. **Integrity & Validity:** If a non-faulty node decided on α, then α must have been proposed by some other non-faulty node in the system
2. **Agreement:** All non-faulty nodes must finally agree on the same value
3. **Termination & Utility:** The consensus process terminates eventually with "low" latency, time needed should be acceptable for a payment system

Constraints

1. **Byzantine Fault Tolerance (strong constraint):** It is possible that a node may behave maliciously. It may collude with other nodes in order to adversely affect the system

2. **Unknown Number of Participants (strong constraint):** Anyone can join and leave the network at any point of time. This will introduce the problem of unknown number of participants among which consensus is to be established
3. **Lightweight (weak constraint):** A consensus algorithm's hardware and bandwidth requirement should be minimal so that even old unused smartphones may be recycled to act as a Trestor node

CAP Theorem

CAP theorem [3] states that it is impossible to provide simultaneous guarantees of Consistency, Availability and Partition Tolerance in a distributed system. We can achieve consistency or availability in the presence of partition tolerance but not both at the same time.

Trestor Consensus

The Trestor network is a distributed network consisting of many independent validators operated by different parties. These validators have to agree on sets of new transactions to be added to the ledger. This is an important decision, because there can be multiple transactions which are valid each, but contradict each other (e.g. two transactions spending the same money).
In order to achieve this, T-Net implements a voting process. Any validator can propose a set of transactions, and other (honest and correct) validators vote for it, if the following requirements are met:

- The transactions are valid (i.e. have valid signatures)
- The transactions are consistent with the ledger history (enough funds for each transaction are available)
- The transactions are consistent with each other
- The validator has not voted for a contradicting transaction before

Sybil Attacks

The common problem with voting schemes based on peers is that malicious parties could set up an arbitrary number of peers to get a majority. We have to prevent this kind of manipulation to take place in order to guarantee that users can be sure about transaction confirmations.
Note that even if the majority of validators form a dishonest conspiracy, they are not able to arbitrarily spend Trests owned by users because they cannot forge arbitrary transactions without knowing the corresponding private keys. However, a dishonest majority of validators could perform:

- Censorship, by always voting against a certain transaction

- Chargeback and double spending, by voting for a ledger that invalidates a transaction which was confirmed in the previous ledger

Deposit accounts

To restrict the number of validators, T-Net will require each validator to have a deposit account. A deposit account is technically the same as a regular user account but tagged as such. T-Net will require a certain number of Trests to be deposited on that account in order to consider the validator legitimate.
In order to make it easier for the network to deal with new validators appearing on the horizon, the deposit account will have another difference to regular accounts: T-Net requires the funds on the account to stay there for a certain amount of time (about one hour to one day) before it can legitimate a validator. On the other hand, if the operator wants the funds to go back to a regular account, that transaction will take the same amount of time. Note that this is not a real limitation for the operator, since he does not pay any fees or loses any money, it only takes more time.

The private key of an account is an EdDSA private key, which can be used to sign any kind of messages. This way, the validator is able to sign messages, for example, the ones belonging to the voting process.

This solution comes with two other benefits:

- The private key of an account is an EdDSA private key, which can be used to sign any kind of messages. The validator can use it to sign messages, for example, the ones belonging to the voting process.
- With those deposit accounts, we have implicitly given a complete list of all (potential) validator nodes that are known to the network. We have no need to trust some single source for providing such a list. (We can still have list providers to make things easier, but each node is able to independently verify it based on the ledger.)

The result is that an attacker needs to control a lot of Trests in order to gain a majority. This is comparable with mining-based systems, where an attacker needs to acquire a lot of computing power in order to gain a majority of mining power. T-Net even has an advantage here: Since our process does not require expensive mining operation, we do not need to reward validators to incentivize them in the first place. In mining-based schemes, mining is rewarded, which makes the big investment for the attacker to gain a majority pay itself (at least partly), reducing the actual cost for such an attack far below the level of the hardware expense itself.

The Voting Scheme

After a validator node collected some transactions and compiled a set of non-conflicting transactions that are valid under the current ledger state, it can initiate a voting.

The voting will consist of multiple rounds. This is necessary to avoid different groups of validators agreeing on incompatible sets of transactions.

Basic Idea

1. One validator compiles a set of transactions to add to the ledger, tags it with a time stamp and proposes a voting round on this message '*t*'.
2. Every node votes "yes" if it has not voted for a conflicting set '*t'* that has not been aborted. Otherwise it votes "no".
3. If more than 80 percent of a node's peers have voted for the set '*t'*, the node will broadcast a vote for the message *confirm(t)*.
4. If more than 80 percent of peers voted *confirm(t)*, the node also votes *confirm(t)*.
5. After confirming *t*, nodes add the set to the ledger.

Note that the message *confirm(t)* is different from the message *t*. Voting for the first one does not contradict a vote against the second one.

But, unfortunately, this voting scheme does only half of the job: If only one set of transactions appears in one time, it will just find its way through the network and get confirmed. If two conflicting sets appear, there are two possibilities:

- One set will gain the necessary 80 percent first. In that case, nodes will confirm this one.
- Neither set will gain 80 percent. In that case, the voting will get stuck.

Because of that, all ballots come with a time stamp. Validators will reject any ballot with a timestamp in the future, and all ballots with a timestamp less than a certain time ago will expire. After a ballot expired, (honest) validators are free to vote on the same transactions differently.

Note that this case should not occur in normal operation, because the user app will not send contradicting transactions. However, this process is necessary to prevent attacks and avoid problems with transactions created by buggy user clients.

Randomized Peer Selection

As the number of validators will grow over time, messages from each validator to every single other one would cause a huge scalability problem. At this point, we can make use of the fact that we have an implicit list of all active validator nodes.

When a new validator node gets set up, it will select a fixed number of peers (e.g. 100) from the known set and use their votes as a basis. At the same time, there will be about 100 nodes selecting this new validator as their basis.

The selection will take place at random. Nobody should be able to influence which set of peers is trusted by a validator. Because of that, validators should only trust validators that they selected themselves. Validators should provide incoming connections with their votes, but not count votes from incoming connections.

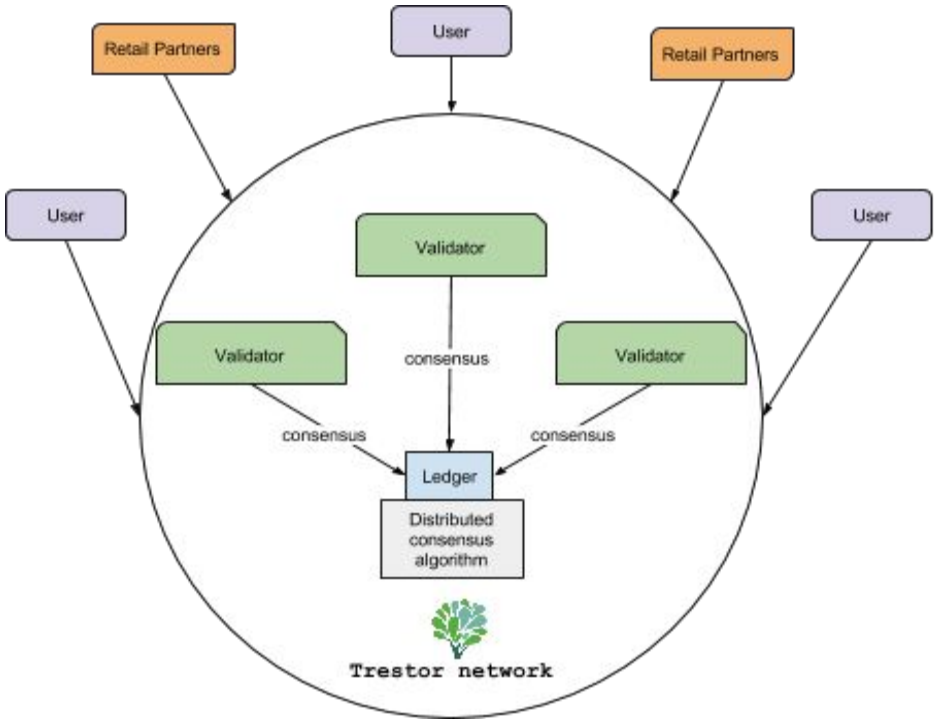
Understanding Trestor Network

T-Net circumvents synchronous communication requirement by leveraging subnetwork consensus within the larger network. Minimal trust and connectivity are required to maintain correctness and agreement throughout the whole network. Consensus algorithm will be explained in detail in our Whitepaper.

User: People who need an efficient and hassle method of value transfer

Retail partners : Businesses with a retail location who wish to trade trests, think of them as last mile broker-dealers

Validators : Nodes powering T-Net by participating in the consensus process. Every validator node holds a copy of the distributed ledger



Ledger

The ledger holds information related to all accounts and their corresponding public keys. Apart from that, every time there is a consensus, the ledger gets modified and goes to the next state.

The ledger needs to be very robust, fault-tolerant and extremely efficient. Every validator holds a copy of the ledger and it has two forms; persistent which stays in mass storages like hard disk or SSD (solid state drive) and the in-memory state i.e. when the ledger is loaded from the persistent storage to the main memory (RAM). Ledger also needs to be scalable, so that it can handle large number of users and still be able to operate very efficiently.

A. In-Memory Ledger

In-memory ledger requires an efficient structure. The In-memory structure is implemented by a 16-ary tree where the leaves contain a list of public keys.

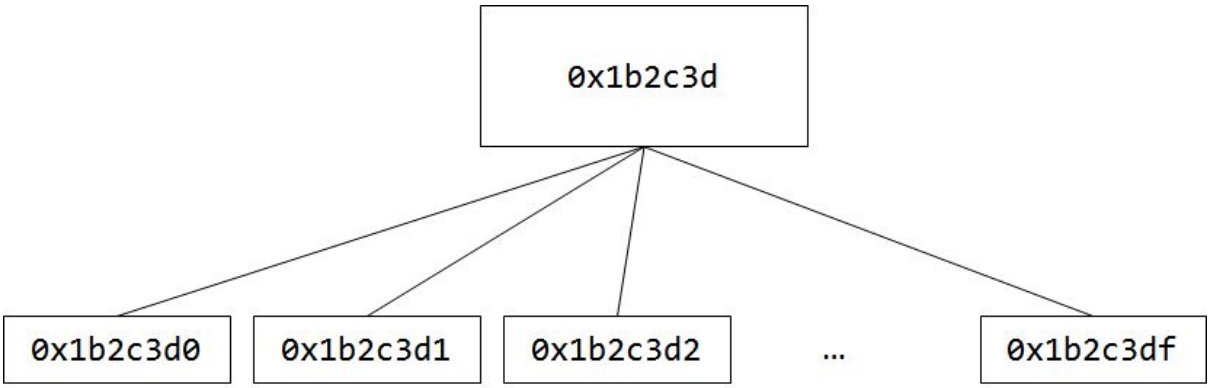


Figure. A node in a tree

Every node in the tree can have at max 16 children and depth of 8. This in-memory structure represents the public key which is 32 byte and can be represented as
(0x at the front denotes base 16)

0x1b2c3d14cffda57313cdad35cd870c251b2c3d14cffda57313cdad35cd870c25

Here each byte is two hex character and the 8 depth (4 bytes) of the tree represent first 8 hex character of the public key i.e. **0x1b2c3d14**

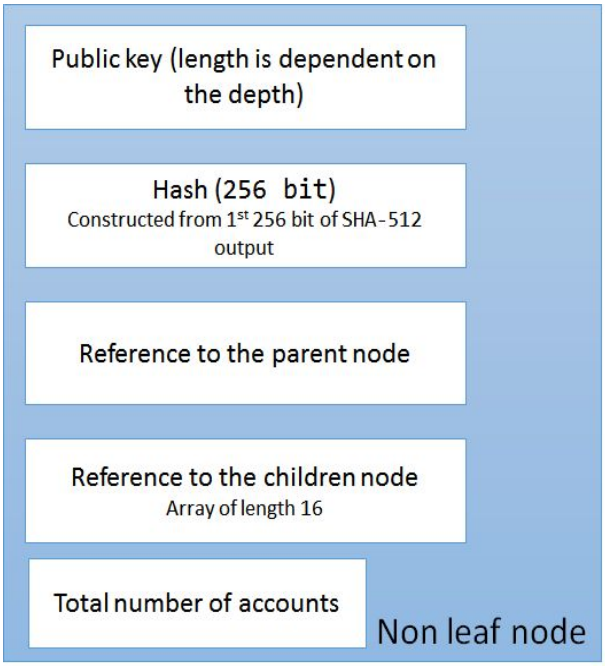
At depth i (0<=i<=8), all the nodes represent the first floor(i/2) byte of the public key (PKi), and the kth child represents the first i+1 byte which is (Pki | k-1).

Example: At depth 6 one node represents **0x1b2c3d**, the 1st child represents **0x1b2c3d0**, the 2nd child represents **0x1b2c3d1**, and the 15th child represents **0x1b2c3df**.

There are 2 types of Nodes:

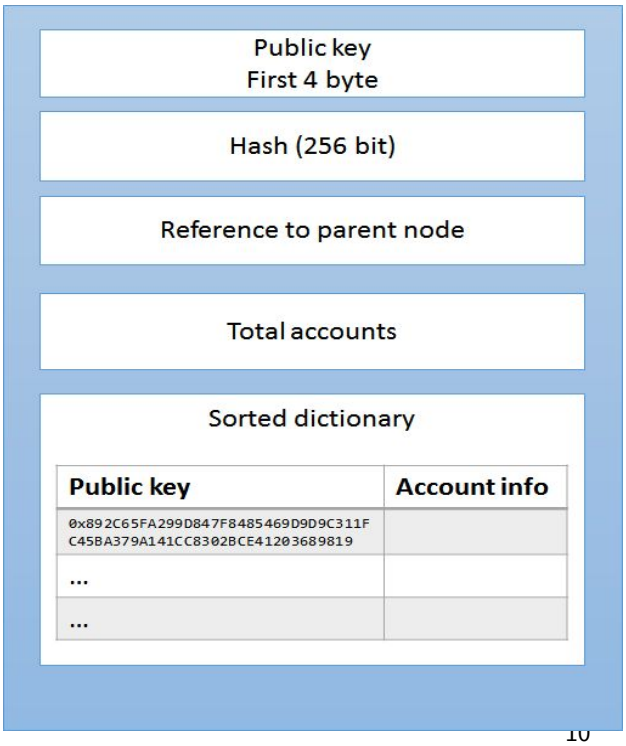
i. Non-Leaf node

1. **Public key:** The part of the public key dependent on the depth as described previously.
2. **Hash:** Cryptographic hash function to check consistency. The procedure of calculating the hash function is described in the later section.
3. **Reference to the parent node:** A pointer is kept to go to the ancestor nodes.
4. **Reference to the children:** An array of 16 pointers to traverse to the children nodes.
5. **Total number of accounts:** Denotes to the sum of the number of accounts in the leaves under this particular node.



ii. Leaf node

1. **Public key:** First 4 bytes of the public keys which it going to store.
2. **Hash:** Cryptographic hash function to check consistency. The procedure of calculating the hash function is described in the later section.
3. **Reference to the parent node:** A pointer is kept to go to the ancestor nodes.
4. **Total accounts:** Total number of accounts (public keys) the node holds.
5. **Sorted dictionary:** This contains the list of all accounts along with the information of those accounts. The list is sorted by public keys for fast searching. The dictionary is a key-value pair where the key is the public key and the value is a object called account info. One example of account info is



```
"AccountInfo":
     {
     "PublicKey":"F92C65FA299D847F8485469D9D9C341FC45BA379A141CC8302B
                  CE41203689819",
        "Money": 10000000000000,
        "Name": "BruceWayne",
        "Address": "TNpHEr3kNEQdwqwh3mYHUcJRmg44kYPdgX8",
        "AccountState": 0,
        "NetworkType": 14,
        "AccountType": 217,
        "LastTransactionTime": 130680355669128949
      }
```

Fig. Sample account info

Hash Calculation

The in memory ledger also features a Merkle tree by keeping cryptographic hash values in all the nodes in such a way that change in any of the node reflects to all of its ancestors up to the root node of the tree for easy detection. The hash value is calculated differently in case of leaf nodes and non-leaf nodes.

Hash calculation in leaves

At the time of calculating hash in a leaf node, all the account information is taken to make sure the slightest change in one account reflects a drastic change. An accountInfo class has 7 fields:

a. Public key
b. Money
c. Name
d. Address (Derived from public key - one way)
e. account type
f. network type
g. account state

All of them are appended and passed to the SHA-512 hash function and from the result first 32 bytes are taken and considered as an internal hash for that particular accountInfo object.

All the internal hashes of the accounts are appended and passed to the SHA-512 hash function and from the result first 32 bytes are taken and considered as the hash for the leaf node.

```
private void updateInternalHash()
{
    List<byte> data = new List<byte>();
    data.AddRange(publicKey.Hex);
    data.AddRange(Conversions.Int64ToVector(money));
    data.AddRange(Encoding.GetEncoding(28591).GetBytes(name));
    data.Add((byte)accountState);
    data.Add((byte)networkType);
    data.Add((byte)accountType);
    data.AddRange(Conversions.Int64ToVector(lastTransactionTime));

    internalHash = new Hash(((new SHA512Cng()).ComputeHash(data.ToArray())).Take(32).ToArray());
}
```

Fig. Hash calculation for leaf nodes

Hash calculation for non leaves

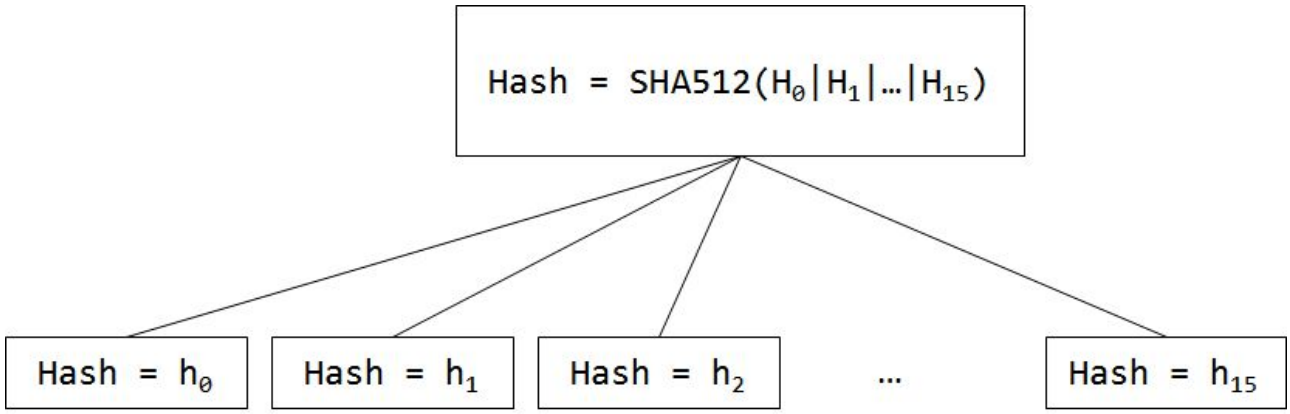


Fig: Hash calculation for non-leaf nodes

For a non-leaf node, the hash of all its children is appended and passed to the SHA512 hash function. From the output first 32 bytes are taken and considered as the hash for the current node.

The hash calculation for the in memory ledger tree is a recursively backward process, from leaves to the root node. The slightest change in one account reflects drastically in the hash value of the nodes on the path from that particular leaf node to the root node. This property is used for the ledger synchronization and in-memory structural consistency checking process.

Ledger Synchronization

Ledger synchronization must be optimized so that it requires minimal computational and network bandwidth.

Sync Procedure

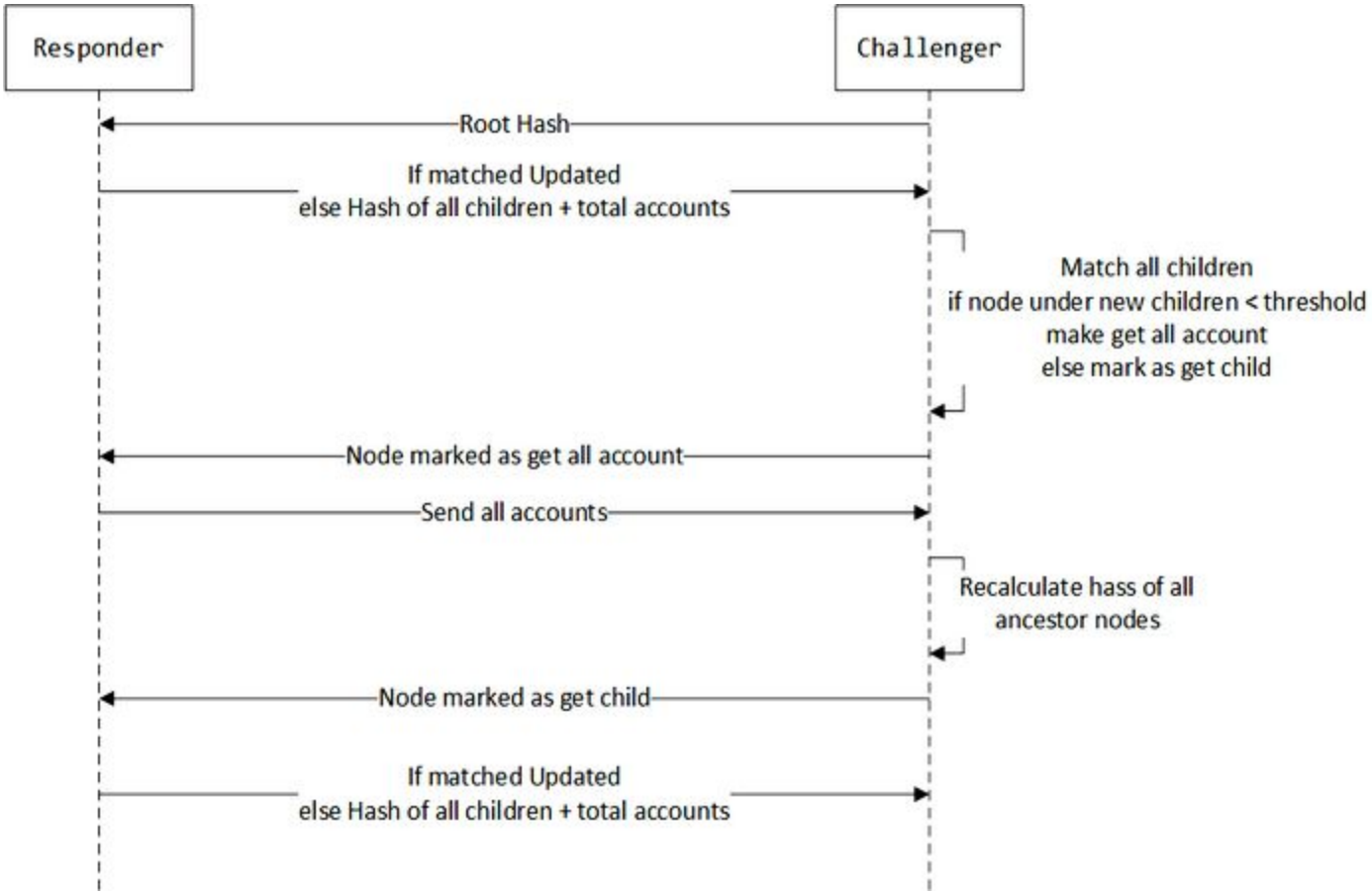


Fig: Sequence diagram of ledger synchronization

Tree synchronization algorithm is very critical as it makes sure that all the validator nodes in the trestor network are in sync and have an identical copy of the ledger to participate in the consensus process.

There are two functions involved in this synchronization algorithm, one of which runs in the node which have the backdated ledger and another one on the node containing the most recent ledger.

Selecting the nodes for the most recent ledger is tricky because the node with the backdated ledger may contact some malicious node which may supply a wrong version of the ledger.

To make sure the probability of such an event is very low, the node will randomly query a sufficiently large number of nodes to demand the most recent version of the ledger. The validator will then select nodes which have trees with most number of identical root hash.

In the following algorithm, we denote the node with the backdated ledger as the challenger and the node with updated ledger as responder. The synchronization algorithm consists of 3 functions:

Challenger_Initiate()

This function will be called only one at the start of the synchronization process.

Responder()

This function will be called every time there is a request from the challenger

Challenger()

This function will be called every time there is a response from the responder. It has a stop condition to make sure the synchronization process does not continue indefinitely.

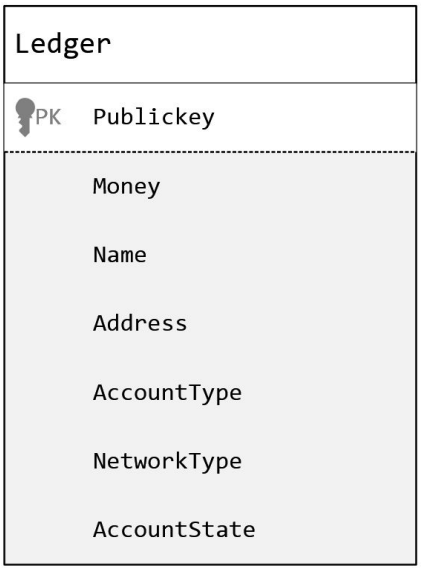
```
Function Challenger_Initiate()
   1. Start
   2. Send root hash to Responder
   3. End

Function Responder (List of nodes)
   1. Start
   2. address = address of a node from the list
   3. From address derive depth
   4. Initiate list of nodes to send = out
   5. For all node addresses
         a. Check id node hash == node hash in the own tree
         b. If same then identical
         c. Else
             i.   If node is marked ad get all children then retrieve all the
                  account information from all the leaves under this node and add
                  to out
            ii.   Else add the all children of node to out
   6. Send out to challenger
   7. End
```

```
Function Challenger (List of nodes)
   1. If all the nodes are leaves, commit changes to the leaves and goto step 4
   2. Initiate list of nodes = out
   3. For all node in nodes
         a. If the node is not in the tree
             i.   If number of children are < threshold value add the node to out
                  and mark it as get all children
         b. Else
             i.   Add all the children of node to out
   4. Send out to Responder
```

B. Persistent Ledger

The persistent version of the ledger stores it on a mass storage like a hard disk or SSD. We have used the sqlite database to store all the account information. We have also provided mechanism to convert the In-memory ledger tree to save in the database and also load the tree from the database.

When the Trestor network validator program is initiated, it will load the database to memory and construct the tree and try to synchronize the tree with the most updated version of the ledger from the network.



Transaction

User wallet proposes transactions to the validators, who validates and votes on transactions in the consensus process. To understand how the transaction works, we have to understand the construction of the transaction entity itself.

A. Transaction status type

Denotes the current status of a transaction. An enum TransactionStatusType is there to define various states.

State	Enum
Unprocessed	0x00

Proposed	0x11
InPreProcessing	0x12
InProcessingQueue	0x13
VoteInProgress	0x14
Failure	0x20
Processed	0x40
Success	0x50

B. Transaction processing result

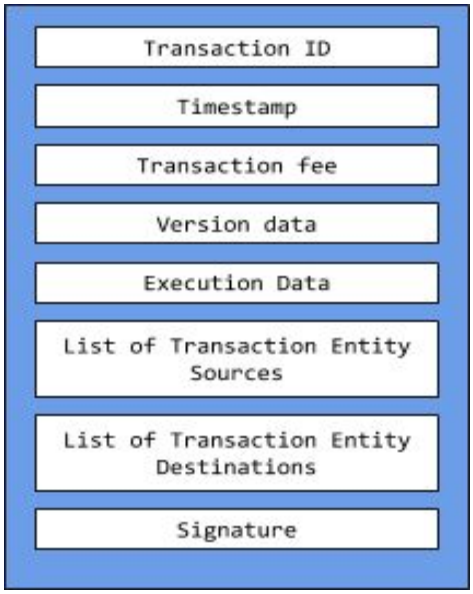
Denotes to the type of result after the transaction gets processed by a validator.

Result Type	Enum	Description
Unprocessed	0x00	The transaction is yet to be processed
Accepted	0x01	Initial integrity checks passed Queued for further processing
InsufficientFunds	0x02	Insufficient funds in sources. This is an integrity check, not considering the actual accounts
SourceSinkValueMismatch	0x03	The value in sources and destinations don't match
SignatureInvalid	0x04	Invalid signature. Mayday Mayday!!!
InsufficientSignatureCount	0x05	All the sources don't have their associated signatures
InsufficientFees	0x06	Insufficient network fees
InvalidTime	0x07	Proposal time for the transaction is invalid (Should be in limits)
InvalidTransactionEntity	0x08	Invalid Source/Destination Entity or Main/Test Net Mismatch
NoProperSources	0x09	A Source providing less number of trest's than network minimum for any transaction
NoProperDestinations	0x0A	A Destination having less number of trest's

		than network minimum for any transaction
InvalidVersion	0x0B	Invalid transaction packet version
InvalidExecutionData	0x0C	Invalid execution Data for any transaction
SourceDestinationRepeat	0x0D	The Source entity is also present as one of the Destinations
PR_SourceDoesNotExist	0x20	Processing Result: Source does not exist
PR_BadAccountName	0x21	Processing Result: Invalid / Banned account name in destination
PR_BadAccountAddress	0x22	Processing Result: Destination account address validation failure
PR_BadAccountCreationValue	0x23	Processing Result: Insufficient amount to create new account
PR_BadAccountState	0x24	Processing Result: Invalid Account state banned/disabled
PR_BadTransactionFee	0x25	Processing Result: Invalid transaction fee
PR_BadInsufficientFunds	0x26	Processing Result: Not enough funds in account or double spending
PR_Validated	0x40	Processing Result: OMG !!! Its all good !!!
PR_Success	0x50	Processing Result: The transaction is successfully processed and account balances reflect the result of the transaction

Transaction content:

1. **Transaction ID:** Hash of all the elements (2 - 8).
2. **Timestamp:** The current timestamp of the user when the transaction was proposed.
3. **Transaction fee:** This is a mandatory amount which is to be added with the transaction amount as a fee. Currently we kept the transaction fee as 0, so no extra charge.
4. **Version data:** Future feature field.
5. **Execution data:** Future feature field.
6. **Sources:** Transaction entity array containing all the sources.
7. **Destinations:** Transaction entity array containing all the destination.



8. **Signature:** Sign concatenation of the elements (2-7). This is also an array in case there is many to one or many to many transactions (Explained in the later paragraphs).

Transaction content is a general definition of a financial transaction which is a many to many mapping. The most common form of transaction is a one-one transaction where a user Bruce sends some Trests to Clark. In such a case, both the source and destination transaction entity array will hold only one transaction entity object (construction of Transaction Entity object is described in the next section).

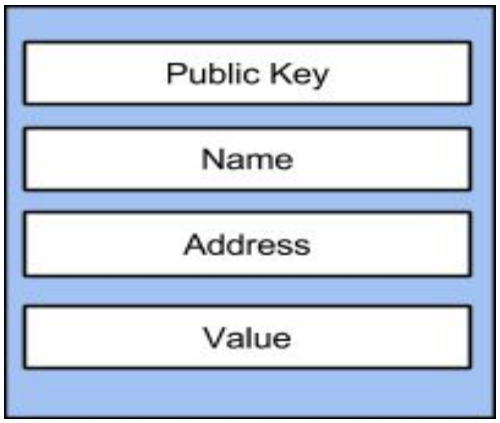
In case it is a one-many transaction, Example: Bruce wants to send money to Clark, Peter, Selina and Logan, then the source array will hold one object and destination array will have 4 transaction entity objects. Moreover, the value (trests) from Bruce's transaction entity is equal to the sum of Clark, Peter, Selina and Logan's.

The many-to-many transaction is a bit tricky. Consider a situation Bruce and Tony together wants to send Trests to Peter, Clark and Berry. Tony will prepare the transaction content object with source array containing 2 transaction entity objects and the destination having 4. The total Trests in the source transaction entity should be equal to the sum of the Trests in the destination transaction entity objects. Tony will sign the transaction and send to Bruce for his signatures.

Therefore, the signature field of the transaction content object will hold both of Bruce and Tony's signature to make it valid as both of them are together spenders. Many-one is also similar to the former with one one transaction entity object in the destination array.

Transaction Entity:

1. **Public key:** Public key of the sender/receiver
2. **Name:** Name of the sender/receiver
3. **Address:** Address of the sender/receiver
4. **Value:** In case the transaction entity is meant for the sender, the value is the amount of Trests the sender will spend and for the receiver it will denote the amount of Trests the receiver will receive after the consensus round



Trestor Line Protocol (TLP)

TLP is the lowest level protocol for serializing data for communication between the peers. It is also used to serialize data for storage. It supports multiple nesting using the **vector** data type. TLP supports the following types of data:

ID	Type	Length (in bytes)
0	Vector (Array of bytes)	Variable (< 2^{32} bytes)
1	Byte (Unsigned 8 bit integer)	1
2	Int16 (Signed Short)	2
3	Int32 (Signed Int)	4
4	Int64 (Signed Long)	8
5	Float (IEEE 754 Single Precision)	4
6	Double (IEEE 754 Double Precision)	8
7	Boolean (True = 1, False = 0)	1
8	String (ISO-8859-1 Encoded)	Variable (< 2^{32} bytes)
9	Varint (Variable length integer)	Variable (1-9)

Figure: Line Protocol Data Types

TLP: Data Format

1. **NameType**: To represent an ***array***, multiple frames can be encoded with the same *NameType*. The decoder will create an array out of the multiple frames having the same *NameType*. It is important that the *DataType* corresponding to a single *NameType* is constant
2. **DataType:** The type of the data being stored as the Payload of the frame. The value is ID of the enumeration specified in the 'Line Protocol Data Types' table above.
3. **PayloadLength**: Length of the payload encoded as a Varint
4. **Payload:** The actual payload in the frame

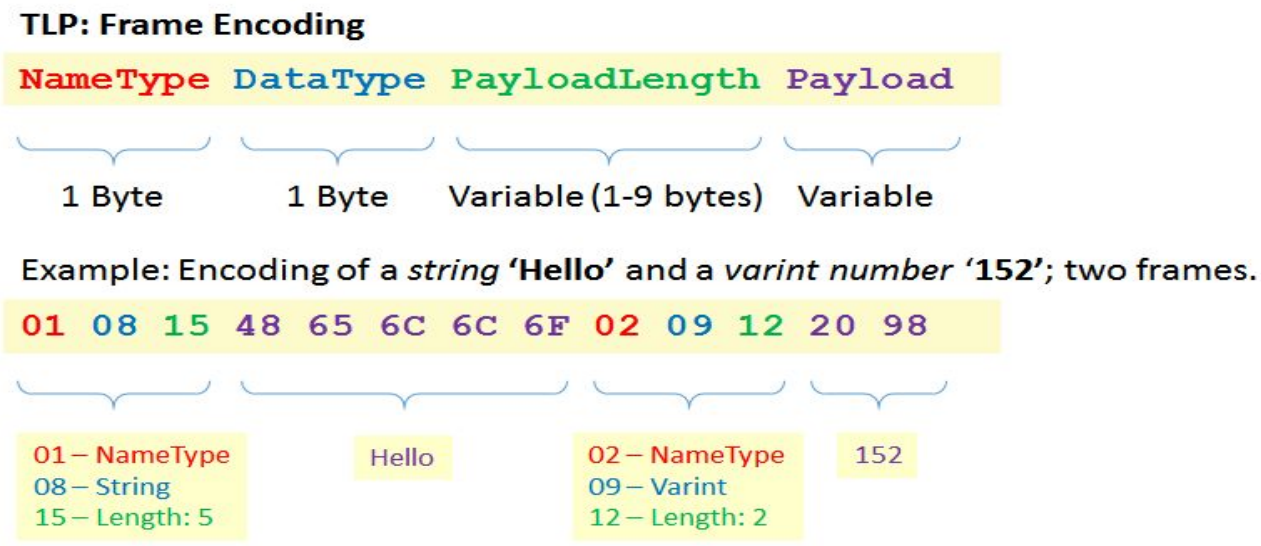


Figure: TLP: Frame Encoding

TLP encodes data by appending four fields in together to give an array of bytes.

Line Protocol: Varint (Variable Length Integer)

Even though there are many possible techniques to encode variable length integers, like the *varint* used in google's protobuf, or sqlite database, TNet uses a special variant which is both space efficient than the others, but, also simpler and faster to encode and decode.

The idea is to store the number of required extra bytes using 4 MSB's of a byte and then store the rest of the bits after it.

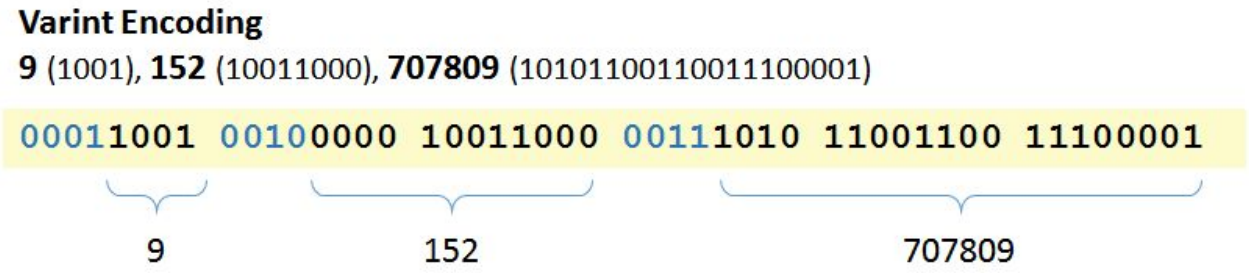


Figure: Varint Encoding of three integers.

Using this scheme, variable length integers of up to 16x8 = 128 bits can be encoded. It is interesting to note that we are not doing any pre-processing for negative integers, this will cause negative numbers to take the maximum length (depending on type, short, long etc.); this is not an issue as all the integers being sent through the network would be positive, the benefit is extra performance. The choice of 4 bits to represent the number of bytes is also optimal as most of the numbers transferred by TLP are very large integers.

Trestor Wire Protocol (TWP)

TWP is used to connect to other peers in the network and secure transfer packets. The said packets contain data serialized using TLP. TWP is used to transfer packets over TCP.

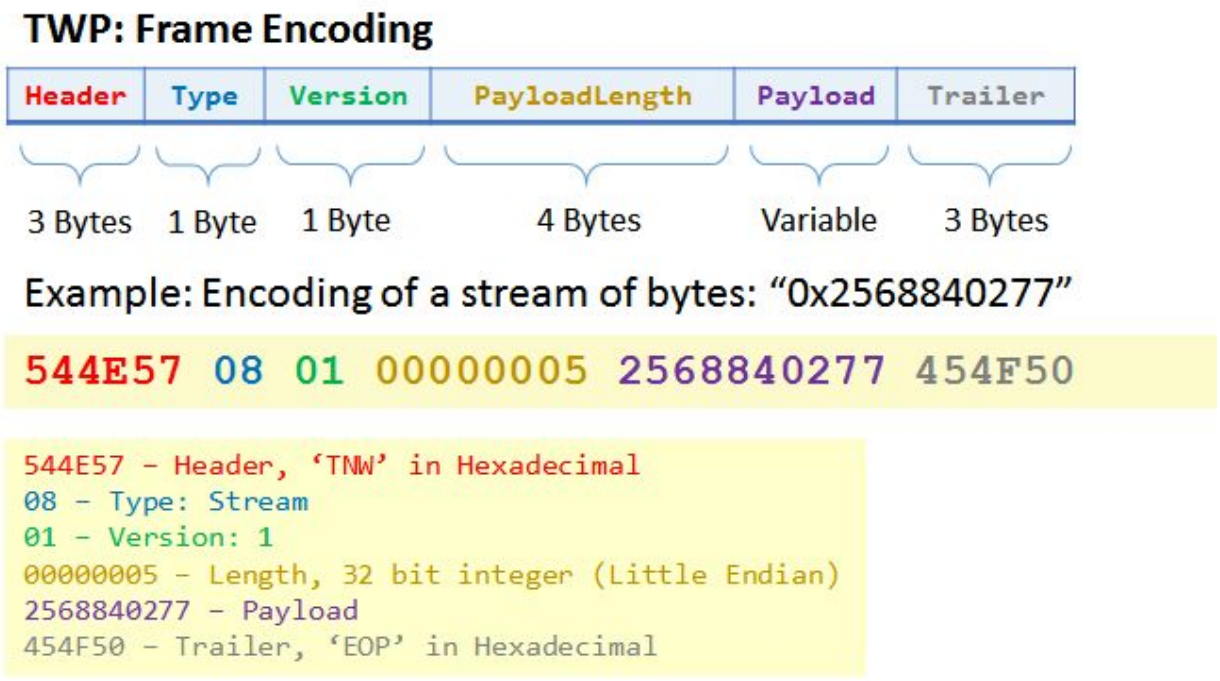


Figure: TWP Frame Encoding.

TWP: Connection and Key-Exchange

TCP connection is established securely without transmitting any identifiable information before key-exchange. This also prevents MITM (man-in-the-middle) attacks. Only after secure key exchange and validation, data packets are sent. Because of the nature of TCP, one connection allows bidirectional transfer, but one peer has to initiate the connection.

This makes the connection establishment process asymmetric even though all peers have similar capabilities. Let's consider two peers to be **S** (server) and **C** (client), where **C** initiates a connection to **S**. After the connection is established and the keys are exchanged and verified, both behave the same.

ID	Type	Dir	Description
0	Initialize	C->S	This packet is used to start a new connection.
1	Control	Both	The control packet is used to transfer information about the capabilities of the device and also issue lower level commands. *(Not implemented yet)*

2	WorkProofRequest	S->C	After the server receives the *Initialize* packet it sends this packet to the client. It contains the *ProofOfWork* data.
3	WorkProofKeyResponse	C->S	See Text Below.
4	ServerPublicTransfer	S->C	See Text Below.
5	KeyExComplete_1	C->S	See Text Below.
6	KeyExComplete_2	S->C	See Text Below.
7	DataCrypted	Both	All data is encrypted and sent using this packet.
8	Stream	Both	N/A
9	StreamCrypted	Both	N/A
10	InvalidAuthDisconnect	Both	Disconnects when anything invalid or illegal happens.
11	KeepAlive	Both	Packets to make sure, the connection is alive.

Table: TWP Packet Command Types.

Keys involved in the Key-Exchange process:

Key Name	Type	Len	Description	Lifetime
ServerPublic ClientPublic	EdDSA Public	32	This is the same as the public key for the Trestor account.	Permanent
ServerPrivate ClientPrivate	EdDSA Private	64	This is the same as the private key for the Trestor account.	Permanent
DH_ServerPublic	ECDH Public	32	Generated by the server and is used to create secure connections with multiple clients.	One Time
DH_ServerPrivate	ECDH Private	32	Generated by the server for every new connection. This is kept for the duration of the connection.	One Time
DH_ClientPublic	ECDH Public	32	Generated by the client for every new connection.	One Time
DH_ClientPrivate	ECDH Private	32	Generated by the client for every new connection. This is kept for the duration of the connection.	One Time
TransportKey	Symmetric	32	Generated by the server by using Diffie-Hellman *Key-Exchange* and	One Time

			Hashing the *SharedSecret* using *SHA512*. All data is encrypted using this key.	
Authentication Key	Symmetric	32	Generated by the server by using Diffie-Hellman *Key-Exchange* and Hashing the *SharedSecret* using *SHA512*. All data and stream HMAC authentication are done using this key.	One Time

Figure: TWP Keys.

Handling and interpretation of TWP packets

0. Initialize (C -> S)

This packet is the first packet sent after the TCP connection is established. It tells the server to start the Key-Exchange process, by sending the ProofOfWork request. It should be ignored and the connection disconnected if this packet is received more than once. The packet contains no data fields. As a response, the server generates a random ProofRequest and sends it to the client using the WorkProofRequest packet.

The format of the data fields for WorkProofRequest is:

PowRandom$_{[16]}$	**CurrentTime$_{[8]}$**

WorkProofRequest Format

1. Control (Both)

The control packet is used to send information about the device capabilities and limitations. Used to transfer low-level information like cryptographic methods used and other optional fields (not implemented in TNetD v1).

2. WorkProofRequest (S -> C)

After a server receives an *Initialize* packet. It randomly generates a 24 byte *ProofRequest* and sends it.

PowRandom$_{16}$ is generated using a cryptographically secure generator and CurrentTime$_{8}$ is obtained by converting the current time to a 64-bit integer (windows file time as long integer).

The client then generates the ProofOfWork response using the algorithm given below and replies to the server with a WorkProofKeyResponse packet with the $WorkProof_{[8]}$, $ClientPublicDH_{[32]}$ and $AuthRandom_{[24]}$ as data fields. The fields $ClientPublicDH_{[32]}$ and $AuthRandom_{[24]}$ are generated for each new connection. In order to generate the DH public keys, the private keys need to be generated before.

The format for the data fields for WorkProofKeyResponse is as follows:

WorkProof$_{[8]}$	**ClientPublicDH$_{[32]}$**	**AuthRandom$_{[24]}$**

WorkProofResponse Format

3. WorkProofKeyResponse (C -> S)

Check if the WorkProof is valid by the proof validation algorithm and then perform the following operations:

1) Use ***Curve25519*** functions to generate the $DH\_ServerPublic_{[32]}$ and $DH\_ServerPrivate_{[32]}$ keys.
2) Use ***Curve25519*** to get $SharedSecret_{[32]}$ from $DH\_ClientPublic_{[32]}$ and $DH\_ServerPrivate_{[32]}$.
3) Hash the $SharedSecret_{[32]}$ using SHA512 to get a 64 byte value. The first 32 bytes of the hash is the $TransportKey_{[32]}$ and the last 32 bytes is the $AuthenticationKey_{[32]}$.
4) Sign the $AuthRandom_{[24]}$ with $ServerPrivate_{[32]}$ using ***Ed25519*** to get $ServerAuthSign_{[64]}$.
5) Encrypt $ServerAuthRandomSign_{[64]}$ using ***SymmetricCipher*** (***Salsa20*** *stream cipher* is *default*) to get $CryptedServerAuthSign_{[64]}$ with *Key* $TransportKey_{[32]}$ and zero *Nonce.*
6) Perform **HMACSHA256** on $CryptedServerAuthSign_{[64]}$ using $AuthenticationKey_{[32]}$ as HMAC key to get $ServerAuthSign\_MAC_{[32]}$.
7) Append $DH\_ServerPublic_{[32]}$, $ServerAuthSign\_MAC_{[32]}$ and $CryptedServerAuthSign_{[64]}$ and reply to the client as a ServerPublicTransfer packet.

   The data format for ServerPublicTransfer is:

DH_ServerPublic$_{[32]}$	**ServerAuthSign_MAC$_{[32]}$**	**CryptedServerAuthSign$_{[64]}$**

ServerPublicTransfer Format

4. ServerPublicTransfer (S -> C)

The following operations are performed after this packet is received:

1) Calculate the $SharedSecret_{[32]}$ from $DH\_ServerPublic_{[32]}$ and $DH\_ClientPrivate_{[32]}$ using Curve25519.

2) Hash the *SharedSecret*$_{[32]}$ using SHA512 to get a 64-byte value. The first 32 bytes of the hash is the *TransportKey*$_{[32]}$ and the last 32 bytes is the *AuthenticationKey*$_{[32]}$. (Now both the peers have common Symmetric Keys for transferring data.)

3) Calculate **HMACSHA256** on *CryptedServerAuthSign*$_{[64]}$ using *AuthenticationKey*$_{[32]}$ as the HMAC key to get *ServerAuthSign_EXP*$_{[32]}$.

4) Continue if *ServerAuthSign_MAC*$_{[32]}$ and *ServerAuthSign_MAC_EXP*$_{[32]}$ are the same, disconnect otherwise.

5) Decrypt the encrypted signature *CryptedServerAuthSign*$_{[64]}$ using ***SymmetricCipher*** and *TransportKey*$_{[32]}$ as Key to get *ServerAuthSign*$_{[64]}$.

6) Verify the signature *ServerAuthSign*$_{[64]}$ using AuthRandom[24] as message and ServerPublic$_{[32]}$ as public-key.

7) If signature verification in the previous step goes well, continue with client authentication, else disconnect the connection.

8) Sign the WorkTask$_{[24]}$ using ClientPrivate$_{[24]}$ using **Ed25519** to get WorkSign$_{[64]}$.

9) Append WorkSign$_{[64]}$ and ClientPublic$_{[32]}$ to get WorkSigPK$_{[96]}$.

10) Encrypt WorkSigPK$_{[96]}$ using ***SymmetricCipher*** and *TransportKey*$_{[32]}$ as Key to get *CryptedSigPK*$_{[96]}$.

11) Perform **HMACSHA256** on *CryptedSigPK*$_{[64]}$ using *AuthenticationKey*$_{[32]}$ as the HMAC key to get *CryptedSigPK_MAC*$_{[32]}$.

12) Append *CryptedSigPK_MAC*$_{[32]}$ and *CryptedSigPK*$_{[96]}$ and reply to the server as a KeyExComplete_1 packet.

The data format is:

CryptedSigPK_MAC$_{[32]}$	**CryptedSigPK$_{[96]}$**

KeyExComplete_1 Format

5. KeyExComplete_1 (C -> S)

The following operations are performed after this packet is received:

1) Calculate **HMACSHA256** on *CryptedSigPK*$_{[96]}$ using *AuthenticationKey*$_{[32]}$ as the HMAC key to get Crypted*SigPK_MAC_EXP*$_{[32]}$.

2) Continue if *CryptedSigPK_MAC*$_{[32]}$ and Crypted*SigPK_MAC_EXP*$_{[32]}$ are the same, disconnect otherwise.

3) Decrypt CryptedSigPK$_{[96]}$ using ***SymmetricCipher*** to get WorkSign$_{[64]}$ and ClientPublic$_{[32]}$.

4) Verify the signature WorkSign$_{[64]}$ using WorkTask$_{[24]}$ as message and ClientPublic$_{[32]}$ as public-key.

5) If the signature in the previous step is valid, that means the connected client holds the private key for the public key being advertised. Even though we don't have a way to check for correctness of the provided public key in the p2p setting, we assume it to be correct.

6) This marks the successful end of the Key-Exchange from the server side. The server then adds the client to the list of authenticated connections. A KeyExComplete_2 packet is sent to the client to inform this.

6. KeyExComplete_2 (S -> C)

This marks the successful end of the Key-Exchange from the client side. The client adds the server to the list of authenticated connections.

7. DataCrypted (Both)

This packet is used to send serialized data in an encrypted and authenticated form over the network.

The data format for DataCrypted packet is:

Nonce$_{[8]}$	**HMAC$_{[16]}$**	**Counter$_{[4]}$**	**Data$_{[variable]}$**

DataCrypted Format

Every single data packet should be encrypted with a different *nonce*. The nonce should be generated using a cryptographically secure random source.

We use EtM (Encrypt then MAC) in TWP, as a result, the data is first encrypted using the **SelectedCipher** and then HMACSHA256 is performed. A 32-bit little-endian counter is appended before the data field before encryption. This counter is incremented for every packet sent or received, the purpose is to prevent packet replay attacks.

8. Stream

Reserved for streaming mode data (Not implemented in this version)

9. StreamCrypted

Reserved for streaming mode data (Not implemented in this version)

10. InvalidAuthDisconnect

When the Key-Exchange fails due to any of the MAC or signature validation failures, or received data cannot be authenticated. The connection is closed, after sending this packet with an empty data field.

11. Keepalive

Sent every minute to make sure the peer is still connected. The connection is disconnected if no reply is received in 15 seconds.

WHAT IS BITCOIN'S KILLER APP?

As a by-product of Bitcoin mining (proof-of-work) incentives, bitcoin network is now a distributed supercomputer; and that too, not just any supercomputer - **its the fastest supercomputer in the world**!

You can watch this 90 seconds video for a quick overview of bitcoin mining.

Even though Bitcoin protocol has created a highly contentious one task supercomputer, the question still remains - **Where is the Killer app?**

Before we get to the killer app, let's quickly understand 'Public goods' and 'Free Rider problem'.

'Public Goods' - *A commodity or service that is provided without profit to all members of a society, either by the government, a private individual or an organization.*

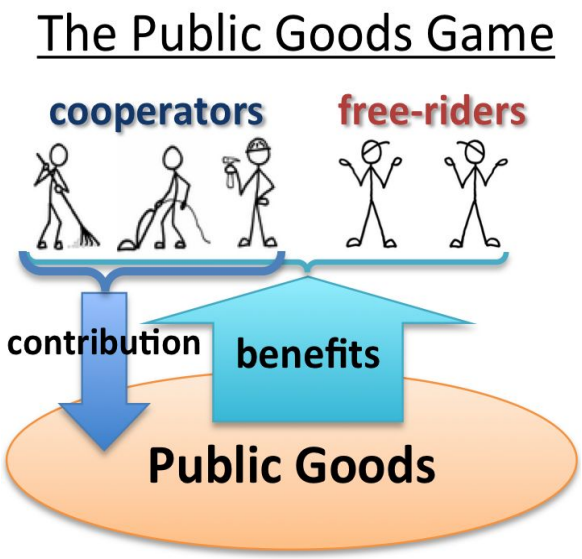
'Free Rider Problem' - *Refers to a situation where some individuals in a population either consume more than their fair share of a common resource, or pay less than their fair share of the cost of a common resource.*

A commonly used example of the economic notion of the free rider problem is found in National Defense. All citizens of a country benefit from being defended. However, individuals who evade taxes are still protected by the same common resource of National Defense, even though they did not pay for their fair share of the resource.

This video explains public goods and other related concepts.

Now, the killer app of Bitcoin protocol, but even more so applicable to a well designed crypto-economic protocol where the tokens are pre-mined is that it has the potential to enable geographically disconnected people, groups and organizations, collaborate and invest time and capital to create products and services which can be utilized as **Public Good**, this is especially true in the case of anti-rival goods, ie. the more people share anti-rival goods, the more utility each person gets.

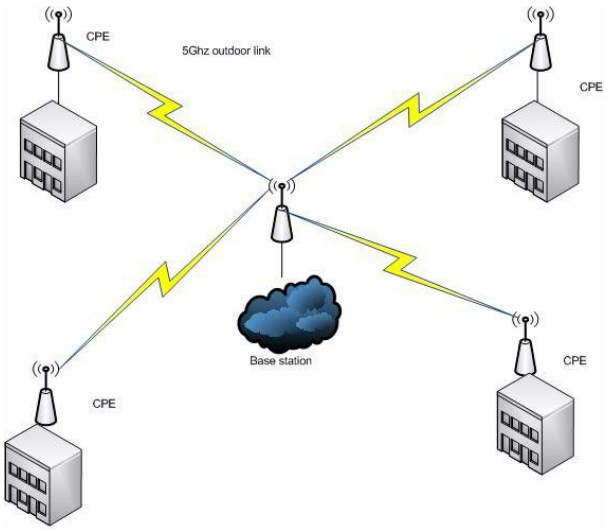
For example, open-source coding projects for most part are an anti-rival, as the number of people interested in using the open-source code increases, contribution to the codebase increases, a community is formed around the codebase and everyone is better off in the process.



A crypto-economic protocol is rightly positioned to unlock the true potential of anti-rival goods thereby utilising them as public good because it beautifully solves the free-rider problem, In fact free riders miss out on the emotional and economic dividends that participants earn by contributing their time and other resources in furthering the mandate of the crypto-economic network.

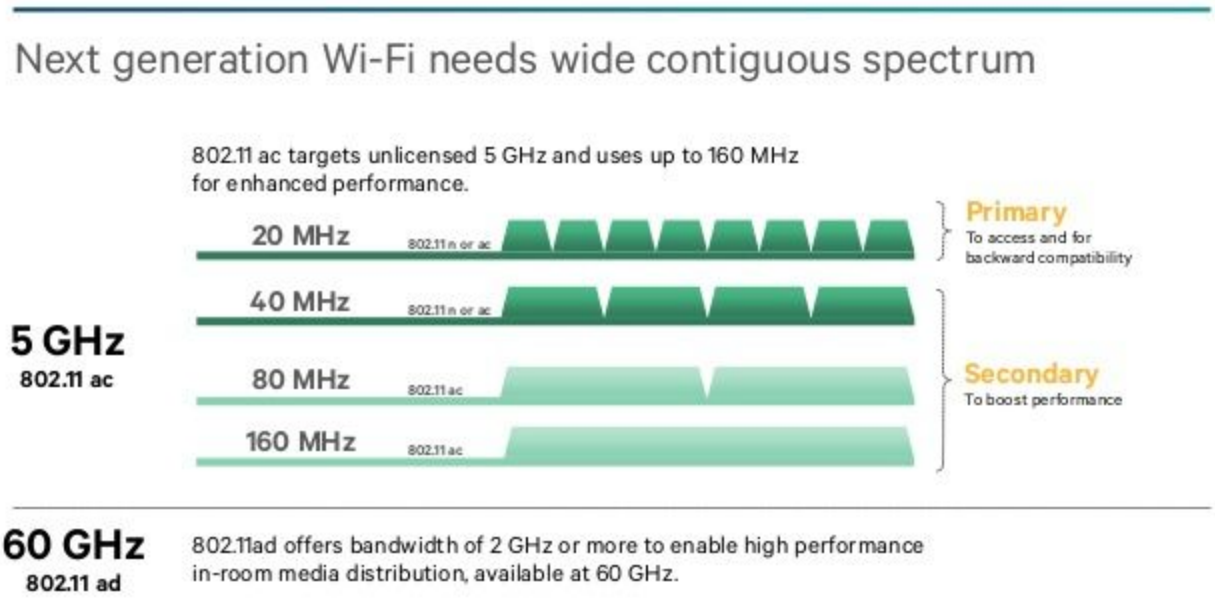
So far, Bitcoin protocol has produced the world's biggest Sha256 solving distributed supercomputer, whose yearly maintenance cost is about half a billion dollars. All this seems justified, because bitcoin mining is instrumental in making the entire bitcoin network always agree on the same set of transactions.

Some other crypto-economic protocols have demonstrated alternatives to proof-of-work mining, their network consumes a fraction of resources while providing comparable security.

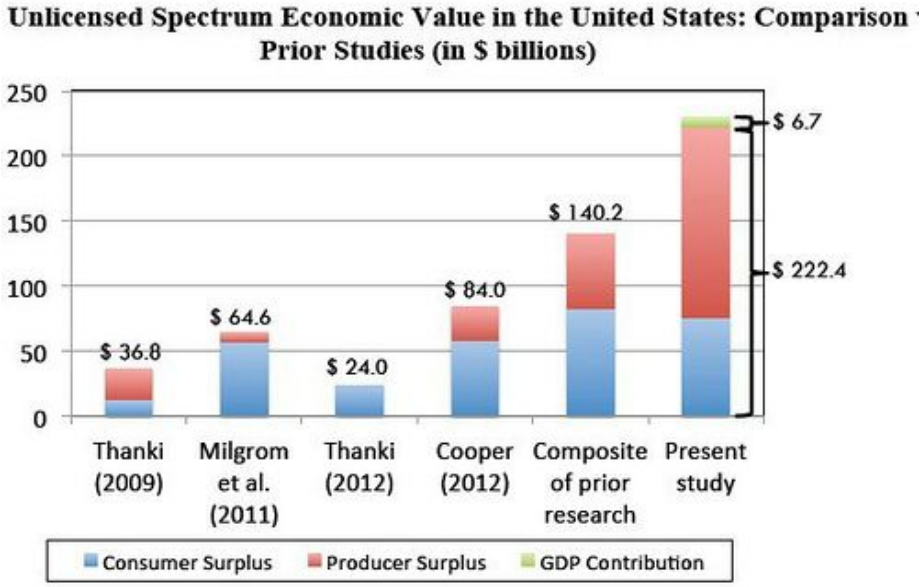


Although these competing protocols may have other shortcomings, the above statement makes us think very hard about the possibilities of what a well-designed crypto-economic protocol could possibly achieve. What if a new crypto protocol were to give out incentives in such a way that anyone can can run a node by purchasing a $1000 base station (which includes a trusted a computing module (TCM)) with an omnidirectional antenna and after downloading the appropriate software, it starts broadcasting free wireless backhaul connectivity and Internet

connectivity at around 5 GHz unlicensed frequency bands to their neighborhood, such nodes get to earn Bitcoins, payout being a function of publicly verifiable metrics of quality of service(QoS) of the wireless connection.



If people who mine bitcoins are willing to invest their money in mining hardware and burn electricity, contribute to global warming and absorb price volatility in anticipation to generate some economic profit, they definitely would be willing to provide free access to quality Wi-Fi to their community. $500 Million spent in mining annually, can potentially be invested in giving out free Wi-Fi access as Public goods.



Buying Internet in bulk (wholesale rate) is far cheaper than buying a retail internet plan - **1 Gigabyte download retail cost is about 10 times more than the wholesale cost.**

Assuming most of these mining entrepreneurs, who were earlier running Bitcoin mining operations and are now giving out free community Wi-Fi, negotiate a sweet wholesale deal with their ISP, the total benefit to public would be about of $5 Billion annually ($500 million * 10 (buying at wholesale rate)). This utility incentive would start a value creation feedback loop, in which more and more people can enjoy the utility created by Bitcoin protocol and would invest some part of their savings in Bitcoin. As the price of Bitcoin goes up, more and more entrepreneurs would be incentivized to broadcast free Wi-Fi, thereby growing the overall free Wi-Fi zone.

If Bitcoin is able to maintain a price around $250 despite burning half a billion dollar worth electricity, imagine what its price would be if it is helping deliver $5 Billion worth of internet annually to thousands of communities and neighbourhoods all across the globe!

A crypto-economic system's biggest use-case is that it can elegantly solve the free-rider problem because no one needs to pay explicitly - the value arises out of the emergent value of the crypto-economic network itself.

Such a protocol would be equivalent to mining gold without digging big holes in the ground, but by creating anti-rival goods and services. Such a protocol would constitute a free lunch indeed, and will gain very high user acceptance.