OYSTER
WEB STORAGE

Whitepaper v0.6

September 2017

Bruno Block

brunoblock@oyster.ws

oyster.ws

The Oyster Protocol enables websites to silently generate traffic revenue as visitors perform Proof of Work for a decentralized storage ledger.

Introduction

Despite the exponential growth of the internet, mechanisms for monetizing web content have remained stagnant. Advertisements intrude on privacy, distract from the intended content, and break design continuity in websites. Due to a general disregard and negative sentiment towards online advertisements, ad blockers have become mainstream. They have become so mainstream that content publishers are pushing back by blocking and limiting viewers from content if ad blockers are detected. Publishers are losing either a large amount of money from ad blockers, or a large amount of viewers from ad block retaliation mechanisms. Therefore the entire advertising scene gradually morphed into an ineffective, inefficient and intrusive ordeal without foresight and wholistic solution deployment.

In parallel, there is currently no storage service that is both convenient and private. If you choose convenience, then you are opting for a standard cloud storage company which precludes privacy and anonymity. Closed source software means you can never truly take what they say for granted. If you choose privacy, then you will seldom find an accessible and straight forward web interface with a simple ‘upload’ button.

The Oyster Protocol is a true two-birds-one-stone proposition. The Protocol introduces a radically different approach to getting content publishers and content consumers to reach equilibrium and cooperation. As a consequence, anyone with a web browser can store and retrieve files in a decentralized, anonymous, secure, and reliable manner.

The following is a list of all the parties that make up the Oyster Ecosystem:

Storage User - A user that spends Oyster Pearls to upload a file

Responsibilities
- Pay the correct amount of Oyster Pearls to two Broker Nodes.
- Despite automation, has final discretion in choosing which two Broker Nodes to use.
- Encrypt and split file locally in browser, then send parts to chosen Broker Nodes.
- Verify the integrity of the Data Map installed by Broker Nodes.
- Share Broker Node contracts via the Distributed Reputation System.
- Securely store the Oyster Handle to retrieve the file from the Tangle at a later time.

Reward
- Their file is securely, reliably, and anonymously stored.

Website Owner - An organization or individual that runs a website

Responsibilities
- Provide content/goods/services to Web Nodes.
- Add the Oyster Protocol script to their website HTML.

Reward
- Get paid in Oyster Pearls that have been discovered by Web Nodes.

Web Node - A web browser that is visiting a web site

Responsibilities
- Search through Treasure Maps via Proof of Work to discover embedded Oyster Pearls.
- Submit discovered Treasure to a Broker Node for claiming on behalf of Website Owner.
- Perform Proof of Work for Broker Nodes to get Web Node identities and new Treasure Maps.
- Perform Proof of Work for Web Nodes to get Web Node identities and old Treasure Maps.
- Send Web Node identities and old Treasure Maps to Web Nodes that have performed the adequate Proof of Work.
- Share Broker Node contracts via the Distributed Reputation System.

Reward
- Granted access to content/goods/services from the corresponding Website Owner.
- Pass on Proof of Work burden to other Web Nodes where applicable.

Broker Node - A network device with access to the Tangle and Blockchain

Responsibilities
- Maintain connectivity to the Tangle via mutual neighboring Nodes.
- Provide Web Nodes and Storage Users with access to the Tangle.
- Perform Proof of Work for new file uploads where applicable.
- Submit the Storage User's Pearls to a buried state in the Blockchain Contract.
- Unlock discovered treasure for which the Proof of Work has been correctly performed.
- Maintain a positive balance of ETH to unlock discovered treasure.
- Build a reputation score on the Distributed Reputation System.
- Broker peer-to-peer connection initiations between Web Nodes.
- Send new Treasure Maps to Web Nodes that perform Proof of Work.

Reward
- Earn Oyster Pearls by collecting leftovers from newly buried treasure.
- Earn Oyster Pearls by collecting fees from newly discovered treasure.
- Pass on Proof of Work burden to Web Nodes where applicable.

IOTA Tangle - A distributed ledger known as a Directed Acyclic Graph

Responsibilities
- Retain data for which the Proof of Work has been performed.
- Geographically distribute redundant copies of data.
- Load balance storage burden, such as with Swarm Intelligence.

Reward
- Network experiences increased resistance against attack vectors.
- Faster average confirmation time for transactions.

Ethereum Blockchain - A distributed ledger with Smart Contract capabilities

Responsibilities
- Provide Smart Contract Framework that produces the properties inherit in Oyster Pearls (as tokens).

Reward
- Blockchain miners receive fees paid in ETH from the Broker Nodes.

Mechanics of the Tangle

The IOTA Tangle is a Directed Acyclic Graph, which means it is a blockless distributed ledger. A live visualization of the IOTA Tangle can be seen here. Each submitted transaction must perform Proof of Work for two prior transactions, therefore confirming them. These two transactions are contextually referenced as the branch and trunk. See here for more details concerning broadcasting a transaction to the Tangle. Each transaction has a payload capacity which is used to retain the data that is uploaded by the Storage User. Transactions are propagated throughout a mesh-net of Nodes that have mutually peered with each other, whilst each Node retains a redundant copy of the transactions. This leads to a great redundancy of data copies, therefore heavily mitigating the risk of data-loss whilst not relying on a centralized hosting provider.

Tangle Nodes are designed so that they automatically delete old data once they reach saturation of their physical storage limits (it is called Automatic Snapshots and it is not yet live on the Tangle). This means that, eventually, transaction data is deleted from the Node. Therefore Web Nodes perform Proof of Work to re-attach the transaction data to the Tangle as they search for embedded Oyster Pearls. This ensures that the data is maintained across the topology of the Tangle Nodes and never becomes irrevocably deleted.

The IOTA Tangle contains many innovations in it's roadmap, specifically Swarm Intelligence. Swarm Intelligence is relevant to the Oyster Protocol because it removes the bottleneck of each Tangle Node being required to maintain the entire ledger. It is similar to transitioning from a RAID 1 drive array setup to a RAID 10 setup. Implementation of Swarm Intelligence further strengthens the scalability merits of the Oyster Protocol.

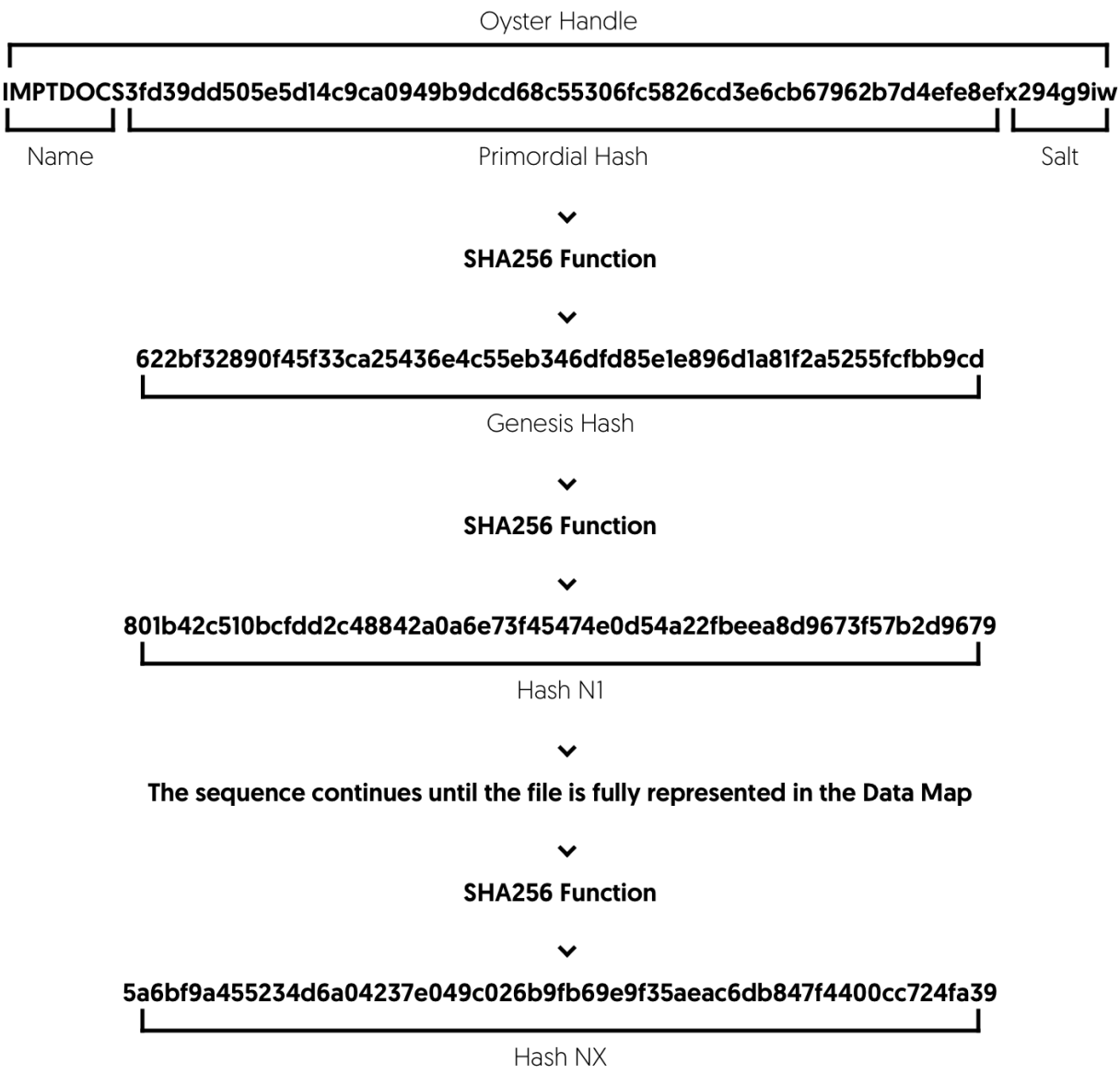
Network bandwidth is the most scarce resource concerning reliably committing data across the Tangle. A Tangle Node is inherently restricted by it's network interface bandwidth. Since the Oyster Protocol aims to commit data more reliably to the Tangle, bandwidth access to the Tangle is rewarded with a share of the processed Oyster Pearls. Tangle Nodes that adhere to the Oyster Protocol specifications are called Broker Nodes. Broker Nodes act as a bridge from the Tangle to Web Nodes and Storage Users.

Data is stored on the Tangle in ~1 KB parts within the transaction payload. A SHA256 hash is the referenced basis for storing and retrieving data on the Tangle. When a SHA256 hash has been selected to represent data, it is converted into it's trinary form to represent the recipient address of the transaction. To retrieve the data from the Tangle the hash is again converted into it's trinary form to produce the recipient address, and then all of the transactions under the address are recovered. The transaction with the oldest issuance timestamp contains the payload data that represents the selected hash.

Initial File Storage on the Tangle

When a Storage User wants to upload a file via the Oyster Protocol, the file is split into parts and encrypted locally in the browser. This isolation ensures the impossibility of a malicious actor retrieving the data since it can only be accessed with the corresponding encryption key, which is known as the Oyster Handle.

The first 8 characters of the Oyster Handle represents the name of the file. This is usually copied from the filename that was uploaded to the browser, but can also be customized by the Storage User for their own references. The Primordial Hash is a 64 character long SHA256 hash of random input that is generated from within the Storage User's browser with as much entropy as possible.



The last 8 characters of the Handle is the cryptographic salt that differentiates the Primordial Hash from the overall encryption key. The salt is used to further protect the data in case the Primordial Hash is found because of a future weakness in a hash function or a rainbow table attack on the Genesis Hash. Therefore the entire 80 character long Handle is the entire encryption key used to encrypt and decrypt the split parts of the data. The Oyster Protocol also supports adding a passphrase to the encryption scheme. The Primordial Hash initiates a sequence of SHA256 hashes that represents split parts of the data. The data is first split in ~1 KB parts, then each part is individually encrypted with the entire Handle as the key. Each part is sequentially represented by a Hash iteration (Genesis, N1, N2 etc.), and is eventually submitted as a Tangle transaction each by two Broker Nodes.

The Distributed Reputation System amongst Web Nodes and Storage Users tracks the best performing Broker Nodes, therefore automatically selecting the most appropriate two Broker Nodes on behalf of the Storage User. The selection of exactly two Broker Nodes is stipulated by the Oyster Protocol so as to induce a competitive race between the two for which Broker Node can install the most Pearls into the Data Map. The better performing Broker Node receives less Pearls for the specific session, yet gains more reputation and hence Pearl revenue in the future.

The value of the Genesis Hash is submitted to both Broker Nodes by the Storage User. One Broker Node is designated to commit the Data Map working downwards from the Genesis Hash (Alpha Node), whilst the other (Beta Node) is designated to commit the Data Map upwards from the NX Hash (X represents the last iteration of the sequence). The correct amount of Oyster Pearls are sent to the Alpha-designated Broker Node. The Alpha Node is sent the full Pearl amount and the Ethereum address of the Beta Node. The Alpha Node sends half of the Pearls to the Beta Node when it receives them along with a cryptographically signed statement that reveals it's identity. Any defections by either Node will be reported via the Distributed Reputation System, which would exponentially degrade their reputation to Web Nodes and Storage Users across the Oyster Network. The amount of Pearls which the Storage User pays is half of the amount that gets eventually embedded in the Data Map. Broker Nodes are allowed to keep any leftover Pearls after the Data Map has been correctly installed.

By default, the selected Broker Nodes are responsible for performing the Proof of Work to attach each data part to the Tangle. The Tangle address used to send the transaction is the trinary form of the corresponding Hash iteration (Genesis, N1, N2 etc.) of the sequence. However, Broker Nodes are able to delegate the Proof of Work tasks to Web Nodes if there is sufficient demand for peer-to-peer connection brokerage and new Genesis Hashes.

To understand how the increase in overall Proof of Work being performed across the Tangle lowers transaction confirmation times and increases the general security of the network, more information can be found here.

Burying Pearls with Broker Nodes

Oyster Pearls are designed to be embedded within the Data Map that defines the structure and contents of the uploaded file. Instead of Storage Users embedding the Pearls into the Data Map themselves, Broker Nodes have been allocated this task for several reasons:

- Broker Nodes have access to ETH balances, and therefore can pay the Gas fees needed to move all the Pearls into the correct designations. Every 1 GB worth of user data requires a transaction on the Ethereum Blockchain.

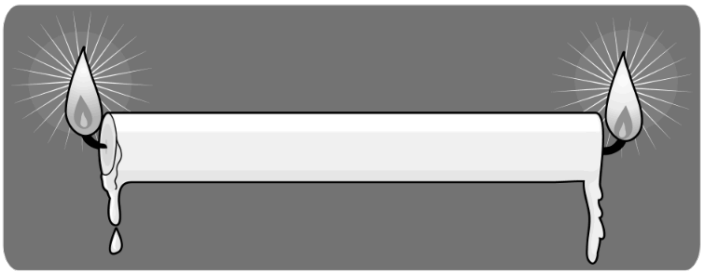
- It would be impractically complicated for the Storage User to have to perform a large amount of complex Blockchain transactions, which includes invoking custom Smart Contract functions. When this complexity is passed onto Broker Nodes, the Storage User need only send the Pearls once to the Alpha-designated Broker Node via a typical Ethereum Wallet.

- Having Broker Nodes embed the Pearls in the Data Map heavily mitigates the red herring attack vector.

A red herring attack is when a malicious Storage User pretends to embed Pearls into the Data Map but does not actually embed them. If they were to upload garbage data without any Pearls in it, it would waste the time of Web Nodes that are searching for treasure (that doesn't exist). Eventually the Web Nodes would realize that the Data Map is not within the bounds of the Oyster Protocol specification, but by then the energy spent by the Web Nodes is expected to be greater than the energy put in by the malicious attacker. Therefore this would cause the attack to be successful and potentially profitable. However, because Web Nodes rely on Broker Nodes to receive Genesis Hashes (which define the entire Data Map), red herring attacks are heavily mitigated. This is because if a Broker Node began to give out Genesis Hashes that represented an invalid Data Map (that doesn't have the correct amount of Pearls in the correct places) it would be easy for Web Nodes to report the Broker Node and therefore ruin it's reputation and future traffic. Whilst Broker Nodes have consistent identities that have

associated reputation scores, Storage Users and Web Nodes are much more dynamic. Not only is it already difficult to establish consistent cryptographic identities for Web Nodes and Storage Users, but the Oyster Protocol defines that Web Nodes need to reset their identities every X amount of treasure hunts. Storage Users do not have a discernible identity except on a session basis to negotiate with Broker Nodes.

Once a Storage User submits Pearls for payments, approximately half are embedded in the Data Map and the other half are collected as compensation by the two Broker Nodes. The two Broker Nodes installing the Data Map can be likened to a candle that burns on both ends.



The candle wax represents the Data Map and the two flames each represent a Broker Node. A Broker Node has the right to retain any leftover Pearls remaining after the Data Map has been completely installed (or the candle completely burns out). The default economic pressures would indicate that it is advantageous for a Broker Node to install the Data Map (burn one side of the candle) very slowly or not at all. If the Alpha Node burns the candle at 10 units per second, and the Beta Node burns the candle at 2 units per second, they will still eventually meet at some stage but the Beta Node will have a lot more leftover Pearls, which it is entitled to keep. The logical extension of this economic situation is that both Nodes would not burn the candle at all, both trying to keep the most Pearls for themselves.

The Distributed Reputation System inverses the economic incentives. Broker Nodes are assigned cryptographic identities which initially debut with a score of zero, the lowest possible. Web Nodes and Storage Users seek to perform transactions with the Broker Nodes that have the highest available reputation scores (whilst also factoring in latency and other selection restrictions). As the average candle burning speed of a Node increases linearly, it's reputation score increases exponentially. This causes Broker Nodes to race in burning more of the candle themselves, despite there being less Pearl revenue in the short term. A Broker Node that intends to burn the candle as quickly as possible will earn less in the short term whilst earning exponentially more in the long term. Therefore the economic incentive for a Broker Node to defect has been nullified.

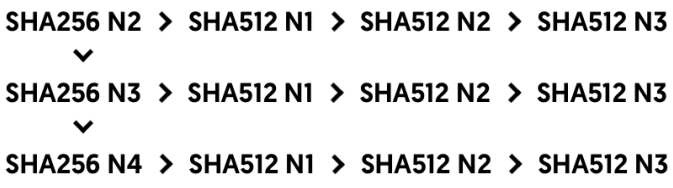
When a Broker Node embeds Oyster Pearls into a Data Map, a special bury function of the Oyster Contract is invoked. A Sector of a Data Map represents 1,000,000 hashes since the chosen hash (Genesis - N999,999, N1,000,000 - N1,999,999 etc). Therefore a Sector holds 1 GB of user data. Each Sector must have at least one embedded treasure of Pearls inside, whilst it is possible that sometimes a Sector will contain two due to non-perfect calibration between both Broker Nodes. The location of the Pearls within the Sectors is randomly chosen by both Broker Nodes. Therefore the amount of Pearls per Sector defines how long the file should be retained in the Tangle. 1 PRL will ensure 1 GB is maintained on the Tangle for 1 year. Therefore the Oyster Contract locks up the Pearls for the duration of the intended storage time. During this time, Web Nodes perform Proof of Work to find the embedded Pearls.

Pearls must already be in a buried state before they can be claimed by a Web Node. Pearls are also claimed in separate time zones named Epochs. The Oyster Protocol defines an Epoch as being 1 year long in duration. This means that if there is a Sector of the Data Map that contains treasure of 2 PRL (4 years storage time), then there are 4 available Epochs: year 1, year 2, year 3 and year 4. The Oyster Contract allows exactly 0.5 PRL to be claimed per Epoch. Web Nodes claim Pearls on behalf of the Website Owner that invoked them.

Treasure Hunting for Oyster Pearls

Web Nodes search through Sectors of Data Maps to find embedded Oyster Pearls. A Data Map is defined by a single SHA256 hash known as the Genesis Hash. Web Nodes get Genesis Hashes from Broker Nodes and other Web Nodes. Genesis Hashes don't come for free, a Web Node must perform specific Proof of Work tasks that are defined by the Counterparty Node. The Proof of Work task is defined by quoting two unconfirmed transactions from the Tangle, one that is the designated branch and the other that is the designated trunk. Once the Proof of Work has been completed, the Web Node responds to the Counterparty Node with the identity of the transaction it just submitted. The Counterparty Node then checks the Tangle to verify that the quoted transaction matches the previously specified branch and trunk identities, and also that the transaction references the data that was meant to have Proof of Work performed on it. Once the Counterparty Node verifies the Proof of Work was performed, the Genesis Hash is sent in exchange.

To search for treasure, Web Nodes select a random Sector of the Data Map that is derived from the newly earned Genesis Hash. The Web Node then checks to see if Proof of Work was performed by another Web Node within the current Epoch by referencing the Tangle. If the Proof of Work was performed, then the Web Node abandons the Sector and tries again at another Sector. This is because even if the Web Node were to find the treasure of that Sector, it is highly likely that another Web Node already claimed the Pearls for the Epoch at the Oyster Contract. If Proof of Work was not performed during the current Epoch, then the Web Node scrolls through each consecutive hash of that Sector. For example: if Sector 5 was chosen, the Web Node scrolls through Hashes N5,000,000 to N5,999,999, which represents 1 GB of data of the uploaded file. Over time, new Web Node strategies can be deployed which do not conform to this behavior, such as bluffing Web Nodes. A Web Node bluffs when it performs Proof of Work on a Sector that it does not expect to contain claimable treasure in the near future. This way, other non-bluffing Web Nodes are prevented from accessing it, which causes the Sector to become exclusively available to the bluffing Web Node. Therefore the bluffing Web Node hopes to keep the Sector exclusively for themselves until the next Epoch becomes available. The game theory mechanics concerning Web Node interactions can become much more complex and advanced as more sophisticated strategies for effectively finding treasure are produced and refined, therefore producing a Prisoner's Dilemma.



For each hash that is encountered, the Web Node first performs the Proof of Work on the Tangle for the corresponding transaction, which utilizes the GPU of the device (using WebGL2). Thereafter, the Web Node retrieves the ~1 KB payload of the corresponding Transaction and calculates the SHA512 hash of the current SHA256 hash. The Web Node then attempts to unlock the payload using the SHA512 hash as the decryption key. If it unlocks, then this means it is treasure that contains Pearls. If it does not unlock, then the SHA512 hash of the current SHA512 is calculated, therefore scrolling through to the next link in the sequence known as the hashchain. If the Web Node reaches the upper limit of the hashchain size defined by the Oyster Protocol, it moves on to the next SHA256 hash in the Data Map and repeats the process.

The Oyster Protocol defines that treasure must exist exactly once per Sector. If no treasure was found in an entire Sector, then the Data Map is declared invalid and participants of the Distributed Reputation System are warned. This would lead to the degrading of the reputation

of the Broker Node that initially introduced the Genesis Hash for that invalid Data Map. It could take days for a Web Node to scroll through a single SHA512 hashchain, and therefore many months for an entire Sector. This prevents Web Nodes from consuming large amounts of data in sudden spikes, as the 1 GB data consumption would be spread out over those months (< 5 MB per day). Therefore Web Nodes do not undertake bandwidth intensive tasks so as to prevent burdening limited/expensive data connection plans. Performing the SHA256, SHA512, and decryption functions all use CPU instructions. This means that a Web Node can perform Proof of Work negotiations with other Web Nodes and Broker Nodes via the GPU, whilst simultaneously searching for treasure in the Sector of a Data Map via the CPU.

Pearls are embedded in the Data Map by including the private seed key of the Ethereum address that holds the Pearls. Therefore when a Web Node discovers treasure, it stores and guards the private seed key via the HTML5 localstorage directive. Despite the discovered treasure, the Web Node faces two major dilemmas:

- The Web Node does not have direct access to the Ethereum Blockchain, and would find difficulty in invoking complex Contract functions.

- The Ethereum address contains Pearls but no ETH for Gas. Therefore ETH must first be sent to the address from somewhere else to allow the private seed key to generate a transaction that will be accepted by the Ethereum miners.

Because of these two dilemmas, the Oyster Protocol defines that a Web Node collaborates with a Broker Node to unlock the treasure. The Web Node securely sends over the private seed key, and the Broker Node checks that there are indeed Pearls inside. Once the presence of the Pearls are confirmed, the Broker Node sends a very small amount of ETH to the address to function as Gas for the transaction. Then the Broker Node submits the transaction to the Blockchain. The transaction invokes the claim function of the Oyster Contract to claim the Pearls to the Ethereum address of the Website Owner that corresponds with the Web Node that discovered the treasure. However there are two immediate concerns:

- The Web Node is concerned that the Broker Node will not claim the Pearls with the Ethereum address of the corresponding Website Owner, therefore stealing the Pearls.

- The Broker Node needs to send ETH to the address that is derived from the private seed key to fuel the transaction, which the Web Node claims contains treasure. It could be a malicious actor pretending to be a Web Node who wants to trick the Broker Node into sending a small amount of ETH to an address that he controls. As soon as the Broker Node would send the ETH, the malicious actor steals a marginal amount after the minimum Gas payment is made. The profit margin may be very small, but it could be repeated indefinitely to cause significant financial losses for the Broker Node.

To solve the concern of the Web Node, the Distributed Reputation System would quickly degrade the reputation of the Broker Node. Broker Nodes have consistent cryptographic identities upon which they build reputation and use to convince Web Nodes and Storage Users to use them for services. Web Nodes will seek to unlock discovered treasure with only the most reputable of Broker Nodes. If a Broker Node were to steal the Pearls of a single Epoch, within a single Sector, within a single Data Map, it would cause much more damage than profit. The Broker Node would conceivably lose thousands of dollars worth of potential future revenue in exchange for a fraction of a dollar. Therefore the Web Node has confidence in dealing with reputable Broker Nodes.

To solve the concern of the Broker Node, the Broker Node does not accept the request of a third party to unlock treasure until it verifies that the Proof of Work for the entire Sector was

recently completed on the Tangle. This means that the malicious actor would have to perform Proof of Work on the entire Sector before convincing a Broker Node to unlock it's treasure. To illustrate the futility of the attack vector; in order for a malicious actor to make a profit of 1 cent they would have to spend $5 worth of electricity in completing Proof of Work puzzles. Therefore if a Broker Node witnesses that the Proof of Work has been completed for the Sector from which the claimed treasure originates, it becomes economically viable for it to send ETH to the treasure address to fuel the Oyster Contract claim function.

Web Node and Broker Node Collaboration

One of the major interactions that occurs within the Oyster Ecosystem are Web Nodes that perform a lot of Proof of Work as payments towards a purchase of information from other Web Nodes and Broker Nodes. Therefore Web Nodes need consistent access to the Tangle in order to operate correctly. Whilst it is technically possible that one day Web Nodes will be able to access the Tangle directly, current library implementations and hardware/bandwidth limitations restrict Web Nodes to being light clients of the Tangle network. This means they need need an intermediary light client host to serve Tangle requests and submissions. Tangle light client hosts already exist independently of Broker Nodes, but the vast majority of them cannot be used by Web Nodes because they don't serve the requests via SSL. The Oyster Protocol requires that Broker Nodes serve all Tangle requests via SSL. This is due to the expectation that most of the websites that will run the Oyster Protocol will be hosted via SSL, therefore the Web Node operation logic must be loaded via SSL and any outgoing or incoming communications must be via SSL.

Broker Nodes also enable Web Nodes to interact with other Web Nodes directly in a peer-to-peer connection. The PeerJS Library is used which relies on the WebRTC Standard. Therefore a Broker Node runs the PeerJS server software to enable Web Nodes to communicate directly to each other.

Web Nodes are constantly in demand for Genesis hashes, especially new ones, due to the implications that the embedded Oyster Pearls of the first Epoch are unclaimed. Once a Broker Node concludes a file upload session with a Storage User, the Broker Node retains the Genesis Hashes. When the Oyster Network is in a typical state of equilibrium; Broker Nodes always retain an excess supply of new Genesis Hashes, whilst Web Nodes are in a constant state of excess demand for Genesis Hashes.

Instead of freely giving away new Genesis Hashes, Broker Nodes stipulate a large amount of Proof of Work that must be performed in exchange. This is primarily done to prevent malicious actors from easily attaining Genesis Hashes and retrieving the embedded treasure. Therefore the added Proof of Work burden makes it even more uneconomical for malicious actors to seek Genesis Hashes from Broker Nodes. The Proof of Work requirement also removes reliance on altruistic behavior, which the Oyster Protocol avoids.

The Proof of Work tasks that the Broker Node burdens the Web Nodes with are the same tasks that the Storage User burdened the Broker Node with. Therefore if the equilibrium of the Oyster Network were to be perfectly saturated, Broker Nodes would never perform Proof of Work as the tasks would get constantly offset to Web Nodes. The exchange sequence that occurs between Web Nodes and Broker Nodes is the same as the one that occurs between Web Nodes themselves. The Exchange Sequence is as follows:

- A Web Node asks the Broker Node if they have new Genesis Hashes or reliable Neighbor identities available (only one information type is requested per sequence).

- The Broker Node responds, and in this case indicates that there is availability. The Broker Node also indicates the requested Proof of Work burden magnitude. Burden magnitude fluctuates according to the state of the Oyster Network economy due to supply/demand constraints.
- If the Web Node agrees with the Proof of Work burden magnitude, then it responds with acceptance of the job.
- The Broker Node sends three references of three transactions on the Tangle. One transaction contains the relevant Storage User data which was formerly a burden to the Broker Node. The last two are unconfirmed transactions that are recommended for confirmation by the IOTA Algorithm. They are each specified for becoming the branch and trunk transactions.
- The Web Node performs the replayBundle function on the Tangle, therefore manually setting the branch and trunk of the transaction exactly as specified by the Broker Node.
- Once the Web Node completes the Proof of Work and the entire Tangle transaction is submitted, it sends the identity of the newly submitted transaction to the Broker Node.
- The Broker Node verifies that the quoted transaction identity represents the correct data on the live Tangle, and has the correct branch and trunk allocations that were specified earlier.
- If there is more Proof of Work to be completed to satisfy the agreed upon burden magnitude, then it is also completed as described above.
- Once the agreed upon burden magnitude has been satisfied, the Broker Node delivers the Genesis Hash or Neighbor identity to the Web Node in exchange. The Broker Node is pressured to deliver the agreed upon information by the Distributed Reputation System.

Overtime, Genesis Hashes migrate from Broker Nodes to the collective consciousness of Web Nodes. Web Nodes will only intentionally forget of a Genesis Hash if all of the Epochs of all of the Sectors have already been claimed. This would indicate data is intended to expire, and therefore no longer guaranteed by Proof of Work exhibitions, unless the Storage User were to add more Pearls to the treasure thereby extending the guaranteed lifespan of the data. Web Nodes use the HTML5 localstorage directive to retain data, including Genesis Hashes that are known by them. If the space afforded by the localstorage directive were to be saturated, the Web Node would begin pruning the data by deleting Genesis Hashes that have the smallest prospects for being profitable to the Website Owner.

The migration of Genesis Hashes from Broker Nodes to Web Nodes is vital. When an upload session is completed with the Storage User, the Genesis Hashes exist momentarily in only both Broker Nodes of the session. The positive consequence of Web Nodes seeking Genesis Hashes is that it secures the existence of the Genesis Hashes amongst the Oyster Network, therefore removing the initial risk that exists when Broker Nodes exclusively hold the Genesis Hashes. If a Genesis Hash were to be forgotten by the collective consciousness of the Oyster Network, it would be no longer maintained with Proof of Work. Therefore the Tangle would no longer be responsible for retaining the data for an extended period of time within it's Node topology.

Web Node to Web Node Interaction

Web Nodes perform peer-to-peer interactions with each other due to the demand/supply constraint motivations that exist within the Oyster Network economy. Peer-to-peer connections are made via the PeerJS Library which relies on the WebRTC Standard. For Web Nodes to be able to communicate with each other, they need to be able to identify each other across the Oyster Network. Therefore each Web Node adopts for itself a cryptographic pseudo-persistent identity. This means that identities are meant to be reliable and consistent until a Web Node reaches a stage in it's treasure seeking career when it needs to wipe away it's memory and start from scratch; as if it were recently introduced to the Oyster Network. This is done to induce dynamic turnover cycles within the Web Node topology of the Network. If Web Nodes were to indefinitely persist in communicating with the same Neighbors, the Network would become too static and unresponsive to environment changes. As long as the majority of Web Nodes follow the Oyster Protocol stipulation of identity refreshing, it will compel the minority to not expect Neighbor relations to persist for an extended period of time.

When a Web Node is either first introduced to the Oyster Network or has recently reset it's identity, it has no Neighbors and must therefore build a list of Neighbors. The Web Node is therefore faced with a Catch 22 dilemma. Neighbor identities are shared by other Web Nodes, but it cannot ask because it doesn't known anyone initially. The initial solutions seems to be to ask a Broker Node, but even Broker Node identities are shared via the Distributed Reputation System that Web Nodes participate in. Therefore Web Nodes assume an initial trusted Broker Node as a default reference. Website Owners are able to change their defaults to which ever Broker Nodes they consider to be trustworthy, therefore maintaining the Network's compliance with decentralization principles.

Broker Nodes are used to broker the initial connections between Web Nodes that already know each other's identities. Therefore a Broker Node retains a list of recently active Web Nodes and their identities. A new Web Node will purchase these identities from Broker Nodes in exchange for performing Proof of Work as defined in the Exchange Sequence. A new Web Node keeps purchasing identities from Broker Nodes, whilst also purchasing identities from other Web Nodes that it has just formed new relations with. Therefore the Neighbor list of the new Web Node expands exponentially until it reaches a saturation point of pursuit due to the principle of diminishing returns. A Web Node is discouraged from pursuing too many neighbors because the energy spent on Neighbor-seeking (via Proof of Work) could have been used to purchase Genesis Hashes and seek treasure. Since the Neighbor list has expanded, the Web Node can now sufficiently receive reputation statements from the Distributed Reputation System. Therefore the Web Node is able to communicate with new Broker Nodes and also keep the original default Broker Nodes in check. Because Web Node identities reset periodically, Web Nodes are compelled to consistently yet gradually seek new Neighbors.

A Web Node can perform the Exchange Sequence with both other Web Nodes and Broker Nodes. This allows the Web Node to perform Proof of Work in exchange for valuable information such as Genesis Hashes or Neighbor identities. The Exchange Sequence is described as the following:

- A Web Node asks the Neighbor if they have new Genesis Hashes or Neighbor identities available (only one information type is requested per sequence).

- The Neighbor responds, and in this case indicates that there is availability. The Neighbor also indicates the requested Proof of Work burden magnitude. Burden magnitude fluctuates according to the state of the Oyster Network economy due to supply/demand constraints.

- If the Web Node agrees with the Proof of Work burden magnitude, then it responds with acceptance of the job.
- The Neighbor sends three references to three transactions on the Tangle. One transaction contains the relevant Storage User data which was formerly a burden to the Neighbor. The last two are unconfirmed transactions that are recommended for confirmation by the IOTA Algorithm. They are each specified for becoming the branch and trunk transactions.
- The Web Node performs the replayBundle function on the Tangle, therefore manually setting the branch and trunk of the transaction exactly as specified by the Neighbor.
- Once the Web Node completes the Proof of Work and the entire Tangle transaction is submitted, it sends the identity of the newly submitted transaction to the Neighbor.
- The Neighbor verifies that the quoted transaction identity represents the correct data on the live Tangle, and has the correct branch and trunk allocations that were specified earlier.
- If there is more Proof of Work to be completed to satisfy the agreed upon burden magnitude, then it is also completed as described above.
- Once the agreed upon burden magnitude has been satisfied, the Neighbor delivers the Genesis Hash or Neighbor identity to the Web Node in exchange.

Typically, burden magnitudes for Genesis Hashes from Broker Nodes are greater than if they were from other Web Nodes. This is because Broker Nodes hold relatively new Genesis Hashes, whilst Web Nodes hold relatively older Genesis Hashes. The going rate for a new Genesis Hash is expected to be higher because there is a greater expectation of unclaimed treasure. This also implies that Web Nodes ask Broker Nodes for Genesis Hashes before they ask other Web Nodes. This ensures that Genesis Hashes from Broker Nodes are constantly, effectively, and quickly migrating towards the collective consciousness of Web Nodes where they are highly impervious to data loss.

When Web Nodes communicate with each other, they measure the connection latency for consistent communication types. This means that, for all communications/transactions that have a consistent size of payload exchange, the time between the connection initiation and completion is measured. Therefore a Web Node will be able to deduce the approximate relative distance of it's Neighbors. This connection latency information is retained so that, gradually over time, a Web Node will favor nearby Neighbors over far away Neighbors. The result of this behavior is that over time the Neighbor list of a Web Node will become optimized so that it primarily communicates with Web Nodes that are nearby. The same optimization is applied for a Web Node's Broker Node list, despite more credence being given to a Broker Node's reputation rather than latency.

The fruits of the latency optimization is that the Oyster Network becomes a decentralized low-latency mesh-net with efficient Node hop pathways upon which third party applications can be built. For example, any small group of skilled programmers could write a decentralized Javascript telephone service that extends the core Web Node Protocol logic and uses it's own Ethereum Token. Therefore the extension can be published as open source code and shared within the Oyster Community. Therefore the Website Owners can add the extension in pursuit of the extra revenue afforded by the telephone service sub-economy. This further enables the monetization of the web for creative content publishers, therefore accomplishing the Oyster Protocol's goal. The phone call mechanism would simply run on top of the API calls that the Web Nodes provides, thereby sending audio packets efficiently across the Oyster Network topology to the desired recipient. Therefore the Oyster Protocol is designed as an Extension

Platform that becomes the bedrock for decentralized code development and deployment by providing optimized mesh-net Node topology and automated Node Hop Logic interaction via a simple API.

Content Consumption Entitlement

The basic social contract of the internet is an exchange of information. Website Owners make investments to produce/obtain/deliver original content and/or services. They must also bear the burden of hosting costs. There must be an economic bridge to justify the investments made by the Website Owner; as there is no such thing as a free lunch.

Whilst in need of this economic bridge, the internet as a whole has resorted to the mediocre solution of advertisement exchanges. Advertisements are consistently distracting, tangential, privacy-invasive, and break the design continuity of websites everywhere. Therefore advertisements have produced unanimous disdain throughout the greater internet community. In due response, ad blockers have become mainstream to the detriment of the internet economy. Therefore creative content publishers are left stranded, as they still need to justify the costs of producing and hosting content. Overtime, creative content publishers are left at the mercy of the policies, decisions and whims of the centralized advertisement exchanges.

The Oyster Protocol is a radical new departure from the old advertisement paradigm, which allows creative content publishers to gain full autonomy over the monetization of their content. Visitors are able to pay the price of admission whilst not being burdened by off-putting and tangential distractions. As money flows back to the already suffering creative content publishers, content quantity and quality can cease receding and increase once again. This, in turn, entices visitors to continue visiting and spending their computational resources via the Oyster Protocol.

The Oyster Protocol is very simple for Website Owners to enable. They need only add a single line of code to their website HTML to fully enable the Oyster Protocol and receive automatic payments in Pearls, like so:

```
<script id="o.ws" data-payout="ETH_ADDRESS"
src="https://oyster.ws/webnode.js"></script>
```

The Oyster Protocol is also very simple for visitors to disable, should they not consent to spend their computational resources in exchange. In such a case, a blocking flag would be installed at the HTML5 localstorage area of the disabled Web Node. The visitor's device would then not perform any Proof of Work or treasure hunting tasks, but a javascript flag would be enabled on the Website Owner's site to mark the abstention. Therefore a Website Owner can opt to easily block sending out content to anyone who doesn't consent to work for them to find treasure.

Because Web Nodes use the HTML5 localstorage directive to retain data, they retain the same identity and work queue whilst being invoked by different Website Owners. Once a treasure hunting session has been initiated, the Web Node permanently associates the Ethereum address of the invoking Website Owner as the claimant.

For example: a person is browsing four of their favorite websites on their laptop, two of which have the Oyster Protocol enabled. When the Oyster enabled website A is visited, the visitor's laptop will become a Web Node and attribute any active treasure hunts to the Website Owner of website A. Therefore, any discovered Pearls would be claimed in the Oyster Contract under the Ethereum address of Website A. Thereafter the person visits the Oyster Protocol enabled website B. The laptop still operates as a Web Node and retains the same cryptographic

identity, collection of Genesis Hashes, identities of other Web Nodes and Broker Nodes, as well as any pending Data Maps that it is working on. If any new treasure hunts are initiated under the jurisdiction of website B, then any discovered Pearls are attributed to the Website Owner of website B.

Oyster Pearl Token Functionality

Because the Oyster Network is a fully decentralized system, it requires a trustless mechanism to manage it's referenced value token; the Oyster Pearl. The Oyster Pearl is an ERC20-compliant token on the Ethereum Blockchain that contains custom designed properties that enable the functionality of the Oyster Protocol.

A custom function specific to the Pearl token is the bury function. Burying an Ethereum address blocks Pearls from being withdrawn whilst still permitting deposits. Deposits to a buried address are still enabled to allow Storage Users to potentially extend the lifespan of their data to prevent intentional data expiration. Broker Nodes invoke the bury function of the Oyster Contract when initially uploading a file to the Tangle. The Pearls that are embedded in the Data Map by the Broker Node are withheld by the Oyster Contract and are therefore unspendable.

As Web Nodes seek treasure they will encounter private seed keys of Ethereum addresses that have had the bury function invoked on them, therefore locking the Pearls from being withdrawn all at once. Therefore anyone who retrieves the private seed key, Web Node or not, will not be able to withdraw any Pearls via the typical transfer function that is invoked on all ERC20-compliant tokens. A Web Node must ask a Broker Node to invoke the claim function on behalf of the Website Owner's Ethereum address. The claim function can only be invoked on an Ethereum address that is in a buried state. The Oyster Contract calculates the Epochs for the specified Sector, and allocates only an Epoch's worth of Pearls to the claimant. If two or more Epochs worth of Pearls are unclaimed within a Sector, then the claimant is rewarded all of them. The Oyster Contract doesn't need to factor in the metrics of Sectors, because each embedded Ethereum address already represents exactly one Sector each.

The Broker Node invokes the claim function with the Ethereum address of the claimant Website Owner which invoked the Web Node that discovered the treasure. The claim function also defines a fee address variable. When the Broker Node invokes the claim function, it submits it's own Ethereum address as the fee variable. Therefore the Oyster Contract automatically assigns the due fee that the Broker Node has earned for unlocking the treasure. Therefore the fees which Broker Nodes receive are unanimously agreed upon and auditable. Once the claim function is executed, any claimable Pearls are directly sent to the Ethereum address of the Website Owner, whilst an agreed upon percentage is reserved for the Broker Node as a brokerage fee.

Therefore the Oyster Pearl is the essential medium of exchange that bridges the economic motivations of Website Owners, Web Nodes, Broker Nodes, and Storage Users.

File Verification and Retrieval

To upload a file via the Oyster Protocol, the Storage User's client selects two Broker Nodes to commit the data to the Tangle. The data is processed from the beginning and end of the file, one Broker Node performing each, similar to a candle that burns on both ends. At some stage the Broker Nodes will have met somewhere near the middle of the Data Map. However, in the unlikely scenario that one of the Broker Nodes was defecting by not performing the Proof of Work (and keeping the Oyster Pearls for themselves), the non-defecting Broker Node will have

completed the entire Data Map. The Storage User's client would have noted the defection and cryptographically reported the defecting Node via the Distributed Reputation System.

Once the file has been fully committed to the Tangle, the client begins downloading the entire Data Map in order to verify it's integrity. The client uses Broker Nodes other than the first two (that performed the upload) to access the Tangle and download the Data Map. This verification stage is technically skippable, but recommended so that in the very unlikely scenario that two Broker Nodes conspired against the Storage User or both individually defected, the Storage User's client would be able to cryptographically report the offenses via the Distributed Reputation System. Honest Broker Nodes would then elect to perform the task that the dishonest Broker Nodes didn't do, whilst not seeking any Pearl payments from the Oyster User. Honest Broker Nodes seek to perform this act of data rectification because they would get a large boost to their cryptographic reputation, which would in turn boost their prospects for increased future revenue.

Whilst the verification process guards against dishonest Broker Nodes, it also guarantees that there is no flaw in the Data Map due to a programming error or execution bug. The download sequence that is used for both upload verifications and candid data retrievals is defined as such:

- The client retrieves the Primordial Hash from the Oyster Handle and submits it as input to the SHA256 function to produce the Genesis Hash.
- The client calculates the trinary data of the selected hash of the SHA256 hashchain (Genesis Hash for first iteration).
- The client retrieves the payload data of the Tangle transaction that correlates with the trinary data from the previous step. The retrieval is performed via a Broker Node that is selected according to the Distributed Reputation System (which is used by Web Nodes and Storage Users). If a verification process is occurring, the Broker Nodes used to access the Tangle cannot be of the same Broker Nodes that performed the initial upload.
- Once the payload data is retrieved, the client attempts to unlock it with the entire Oyster Handle as the encryption key. The Oyster Protocol also allows for a passphrase to be used in the encryption scheme, yet caution is advised due to the risk of the Storage User forgetting the passphrase. If either the Oyster Handle or the optional passphrase are lost, then the file is permanently lost.
- If the payload data unlocks, then it is part of the data sequence that makes up the uploaded file. If it does not unlock, then it is a reference to the SHA512 hashchain that contains the embedded treasure. As the client progresses through the Data Map it checks that at least one SHA512 hashchain reference exists per Sector (1,000,000 SHA256 hashes), or else it declares the Data Map as invalid and performs the appropriate procedure with the Distributed Reputation System.
- Once the payload data of a single hash has been retrieved, it is stored on the Storage User's persistent storage device and is freed from the corresponding memory usage allocation.
- The client calculates the next iteration in the SHA256 hashchain, by submitting the current SHA256 hash into the SHA256 function. The resultant hash is the next iteration of the hashchain. (e.g. The N1 hash comes after the Genesis Hash).
- The process repeats itself until the entire file has been retrieved from the Tangle.

- The individual file parts are glued back together, and the entire contents are compared to the embedded checksum to guarantee data integrity.

The Storage User is able to access their files through any client, even from Tangle Nodes that are not Broker Nodes. The only two things that are needed to retrieve that data are the Oyster Handle and generic access to the IOTA Tangle.

Distributed Reputation System

Broker Nodes are needed by Storage Users to bury Pearls, and by Web Nodes to claim them. In both instances, valuable Pearls are sent to the Broker Node for processing. Therefore a monitoring system is used by Storage Users and Web Nodes to keep the Broker Nodes in check. This system is known as the Distributed Reputation System and operates in a similar manner to eBay. Broker Nodes are like eBay sellers and Web Nodes/Storage Users are like eBay buyers. Web Nodes/Storage Users will only conduct transactions with the highest reputation Broker Nodes that they can find. The Broker Node selection algorithm considers other criteria such as network latency, traffic constraints, and Protocol prohibitions (e.g. Storage Users cannot use the same Broker Node for both the upload and verification processes). Despite these separate criteria, a relatively small range of the highest reputation Broker Nodes will receive the vast majority of the traffic, and hence Pearl revenue. Therefore it is critical for Broker Nodes to be honest in order for them to become profitable.

All reputation scores debut at zero. There are no negative reputation scores, or else a malicious Broker Node could remove it's negative reputation by generating a brand new cryptographic identity that debuts at score zero. The flawed strategy of switching reputations to perform scams is comparable to a rogue eBay seller that keeps generating new accounts. Little to no attention will be given to these new accounts that have zero reputation. Instead, the honest sellers with high reputation will get the vast majority of the business.

From a score of zero, honest Broker Nodes make slow and gradual progress by granting Web Nodes and Storage Users with access to the Tangle. Genuine access to the Tangle is verifiable and fault tolerant, by referencing more reputable Broker Nodes to confirm that the real Tangle is being accessed. When an honest Broker Node gathers enough initial reputation it begins to receive requests from Web Nodes/Storage Users for performing value based transactions (burying and claiming Pearls). They will initially be assigned as the Beta Node since Beta Nodes handle less Pearl value. If the value based transactions are performed correctly, then the Broker Node's reputation will begin to increase exponentially. In turn, this would cause more Web Nodes/Storage Users to request it for value based transactions. Dishonest Broker Nodes will seldom reach the exponential reputation boost ramp.

Each Broker Node's identity is based off of a generated PGP key, which must be kept secret. If the PGP key where to be leaked it would quickly degrade the Node's reputation (and future revenue prospects) because malicious actors would quickly use it for short term gain. A Broker Node's ability to safeguard their PGP keys precedes and confirms their ability to safeguard the treasure coordinates of a Data Map and the private seed keys of treasure.

Broker Nodes and Web Nodes/Storage Users agree to perform a value based transaction after negotiating two terms: a minimal state of the Tangle and a minimal state of the Blockchain. The minimal state of a ledger defines the minimum scope of transactions that should exist after the contract is successfully executed and before the contractual deadline. A Storage User's client defines all of the branch and trunk transactions for each SHA256 hash of the Data Map as the minimum scope of what the Tangle should contain. In response the Broker Node sets the minimum scope of the Blockchain to contain a transfer of Pearls to an address that it controls.

If the proposed contract conforms with the Oyster Protocol and the environmental context of both parties, then both parties digitally sign the contract with their PGP signatures.

Broker Nodes always install Data Maps in pairs whilst racing each other to embed the most Pearls. In the contract, each Node is assigned a branch/trunk designation by the Storage User's client. Once the Data Map installation is complete the observing Web Nodes can estimate the performance of each Broker Node by comparing the trunk/branch designations that were defined in the contract with the actual trunks and branches that were referenced on the Tangle. Therefore there is unanimous agreement amongst participants of the Distributed Reputation System for who performed the best. The Broker Node that performed the majority Proof of Work receives a reputation score promotion, whilst the Broker Node that performed the minority Proof of Work either gets no reputation change or a demotion (depending on the degree of performance). Therefore it is in a Broker Node's long term interest (concerning revenue generation) to perform the Proof of Work as quickly as possible. This economic pressure ensures a fast and efficient experience for the Storage User to upload a file.

Broker Nodes are able to offset the Proof of Work burden to Web Nodes that want to purchase Genesis Hashes or Web Node identities. Therefore a Broker Node with a strong reputation can typically install the Data Map faster because it has more Exchange Sequences occurring with more Web Nodes, therefore offsetting more of the branch/trunk Proof of Work designations to them. If a zero or near-zero amount of branch/trunk designations from the signed contract match with the Tangle after the contract deadline, then this will be considered a defection and the Broker Node's reputation will be severely demoted.

When a Web Node needs to unlock discovered treasure, it defines in the contract the minimal state of the Blockchain. The minimal state defines that the Pearls from the treasure address should be claimed at the Oyster Contract for the Website Owner's Ethereum address. No minimal state of the Tangle is defined for this contract. If the Broker Node agrees with the terms, it digitally signs the agreement with it's PGP keys. Only once the Web Node receives the signed copy of the contract will it send the private seed key of the treasure to the Broker Node. If the minimal Blockchain state has not been fulfilled before the contractual deadline, then the Web Node notifies other Distributed Reputation System participants of the failure by quoting the signed contract. Therefore the mesh-net of Web Nodes gradually build consensus that the Broker Node did not perform the task it was meant to, therefore demoting it's reputation and future income prospects.

Intrinsic Storage-Pegged Value

PRL is an ERC20 token and the heart of the Oyster economy that bridges the cash flow of storage users and website owners. 1 PRL will always represent 1 gigabyte of anonymous data retention for 1 year. This is fixed within the protocol just like how Bitcoin has a fixed supply of ~21 million BTC. This leads to interesting implications for the price evaluation of PRL.
PRL will always have a natural price discovery mechanism that calibrates with market rates of digital data storage. There will inevitably be minor price fluctuations that are not synchronized with storage prices, but PRL constantly inclines towards them. This means that PRL cannot go above or below storage prices by a factor of 100x after the initial calibration of the Oyster network. It has the inherent properties of a stable-coin as it is pegged to a commodity yet without requiring a reserve to match it's aggregate value.

Consider if a whale where to dump 60 million PRL on an exchange, thereby slashing the price in half. The price would then be well below the market rate for digital storage. This would inevitably attract individuals and companies that are looking for storage solutions, especially those that are seeking to reduce their budgets and save money. As more users buy PRL to

benefit from the inexpensive storage, the price automatically corrects back into the fluctuation zone of storage prices.

Imagine a volleyball being released from the bottom of the ocean. It shoots up violently to the surface, puncturing the waves and becoming suspended in mid-air for several seconds. It then finds buoyancy equilibrium as it rests on the ocean waves, elegantly levitating upwards and downwards along with them. This is the perfect analogy for the price discovery mechanism of PRL.

The pre-product price of PRL is offered low as a protocol function to later jumpstart activity in the Oyster network by luring website owners to PRL's coveted value. They install the one line Javascript code to enable Oyster on their website and earn PRL, thereby cementing the Oyster network's capacity to retain data on the Tangle. When the network is live the price initially shoots up as people load up on affordable data storage, until it eventually reaches the surface of the digital ocean. Whales and small fish alike can proceed to dump the token on exchanges for massive profit-taking without moral hazard, as they are being rewarded for having initialized the topology of the network. The price would simply climb back up to it's parity with digital storage due to the commodity demand curve.

Conclusion

The Oyster Protocol is designed to solve revenue generation, anonymous and accessible storage, and decentralized application development and deployment. In the same way that the Ethereum Blockchain provides a straightforward framework for token creation, the Oyster Protocol provides a straightforward framework for accessing a decentralized mesh network.

Web Nodes are everyday computers, smartphones, cars, fridges - anything with a modern web browser. They communicate with each other directly, only needing occasional connection brokerage from Broker Nodes. They automatically choose neighbors that have lower latencies over time, leading to an overall optimized Node topology. Extensions can be built in Javascript, a popular and easy to use language, therefore developers can gain access to a world-spanning mesh-net. This creates the perfect breeding ground for decentralized applications to become easily built and access a capable, latency optimized mesh net.

The Oyster Protocol unlocks the dormant revenue potential of millions of websites, the storage dilemmas of individuals and corporations, and the mesh-net platform that developers need.