TECHNICAL PAPER

Release 1.0, October 1, 2017

Written By Neeraj Murarka, CTO Bluzelle Networks

&

Pavel Bains, CEO Bluzelle Networks

1. Table of Contents

2. Key Features

2.1. Perfomance

Bluzelle's unique and proprietary swarming techniques were designed with performance in mind. Bluzelle can reduce latency by retrieving data from the nearest nodes on the leaf swarm, and/or increase speed manyfold by retrieving data in parallel from the fastest nodes on the leaf swarm. This is quite comparable to a CDN. Imagine that every Bluzelle swarm is a separate CDN. When data is requested from a swarm, it is analogous to data being requested from a CDN — the "best" node (defined by many factors including network distance) is chosen to serve this request. Bluzelle acts similarly, asking the top few nodes (to avert Sybil attacks and Byzantine faults) to respond to the request. Another good comparison is torrents and seeds. When data is requested, it is done in parallel where chunks (shards) are requested from all the different swarms that contain those shards, and these are all retrieved in parallel, resulting in desirable performance metrics.

2.2. Reliability

Using the concept of fog or swarm computing, Bluzelle follows a model where every unit of data is 100% replicated in a single leaf swarm amongst a swarm of swarms. So while the data is only in one swarm, that swarm's nodes are aplenty and are geographically dispersed, immune to localized outages caused by either natural or human-related events.

2.3. Scalability

Scalability is possible both horizontally or vertically. Bluzelle manages the various strategies and considerations around the use case of having to increase scale.

Bluzelle is a massive horizontally scaled network. In fact, horizontal scaling is a cornerstone of its architecture, where every swarm is another "unit" of horizontal scaling at the metaswarm level. Within every leaf swarm, every node acts as yet another agent of horizontal scaling, at the leaf swarm level.

3. How Does Bluzelle Work?

Using concepts taken from NoSQL (non-relational SQL), Bluzelle's first incarnation is a key-value storage that exploits horizontal data sharding partitioning concepts to split databases, irrespective of size, into smaller, faster, and more easily managed parts called data shards (shard means "a small part of a whole"). Bluzelle takes this concept and marries it with blockchain concepts to create a powerful economy of producers and consumers. In modern nomenclature, it is a crypto-economic network.

The producers (aka farmers or miners) provide resources in the form of CPU processing power, memory, permanent storage, and network connectivity to enable database shards to be stored with sufficient partial replication across a swarm of swarms of nodes (a metaswarm) to guarantee availability and redundancy. A proportional “stake” is put up as collateral by a producer as an incentive for them to guarantee reliability. The stake can be proportionally lost, if sufficient metrics are not met or the producer is deemed a bad actor.

Consumers use Bluzelle's database services provided by the collective whole (all the producers), and in turn pay for the services using the Bluzelle ERC-20 Token (BLZ) and Bluzelle Native Token (BNT). They have control over attributes that dictate how and where their data is stored, that will affect the BNT cost for these services.

Initial peer data needed for new nodes or client applications to bootstrap themselves with Bluzelle is found using the Ethereum blockchain (a pre-existing decentralized network) as a basis to get all associated data in a reliable, scalable, and low-cost way. This approach does not reduce the degree of decentralization of Bluzelle.

3.1. In More Depth

Bluzelle is truly a swarm of swarms, or as our marketing materials go, a metaswarm. In version 1.0, as stated above, Bluzelle is chiefly concerned with the storage of NoSQL key-value data

pairs belonging to data owners. More specifically, a software developer may store any data they want in Bluzelle by providing a unique key and the value to go with it. Even some users of traditional RDBMS's can use Bluzelle easily.

Each swarm has a leader, as is the design in Raft. The leader can go down and if so, a new leader is democratically elected in the swarm. This leader is only chosen from the top 5% percentile of nodes in the swarm, based on Karmic index, to greatly reduce the frequency of elections. This is a small yet important modification Bluzelle makes to the Raft consensus protocol. Accordingly, an important property of Bluzelle is that nodes can go down and new nodes can come up, but the impact and overhead to the network is minimal as the swarms themselves internally deal with membership changes. Each swarm remembers sufficient information on other swarms to be able to notify other swarms' leaders if a new leader is elected. The leaders are the mechanism by which to contact and talk to a swarm. If a swarm's population of nodes drops dangerously low where replication and redundancy guarantees could be compromised at the swarm level, nodes from other swarms can be enlisted, or the swarm in danger can be dissolved into one of the other swarms. If all the swarms are saturated and a new node joins, a new swarm can be created, where existing swarms are diluted sufficiently to populate the new swarm.

Bluzelle's swarms interact with each other by means of swarm leaders initiating conversations that can then be delegated to nodes within each swarm that are best qualified to perform the activities in question. In fact, each swarm and the other swarms it directly is aware of as a result of Kademlia finger tables are considered a virtualized metaswarm [1].

3.2. What is a Swarm?

A swarm is a collection of other units. There are three kinds of swarms:

- **The singleton metaswarm**. Bluzelle has subordinate units called leaf swarms. It is important to remember that the metaswarm is a symbolic construct. There is no state

[1] A new term coined here by Bluzelle

machine or data that is ever directly shared or processed on a metaswarm level. In other words, there is NO data that all the leaf swarms have in common -- this is a vital property that emphasizes how decentralized Bluzelle is. The metaswarm is a symbolic grouping of all swarms.

- **The leaf swarms**. Each subordinate unit here is a node. One of the nodes is the democratically elected leader of that swarm. Every node in the swarm stores the exact same data [2]. A leaf swarm can be considered one and the same as a horizontal database partition, and is identified uniquely by a logical shard number. The leaf swarm contains many logical shards of data, each mapping to the same leaf swarm id number.

- **The virtualized metaswarms**. A virtualized metaswarm is a collection of leaf swams that are all related to exactly one of the collection's swarms in a mathematical way. More specifically, all the subordinate swarms are related to a "root" swarm because they are each within a specific exponential (base 2) XOR distance interval [3] from the root swarm, in terms of swarm id XOR distance. Every Bluzelle swarm participates in approximately O(log(n)) virtual metaswarms (where n is the number of swarms in Bluzelle).

  For a given root swarm, each of the subordinate swarms in its virtualized metaswarm can also be the root swarms for their own virtualized metaswarms, each of which the root swarm itself may play a role in as a subordinate swarm [4]. It can be correctly inferred that every virtualized metaswarm has approximately O(log(n)) members. A root node stores its virtualized metaswarm information by means of a Kademlia finger table [5]. Root swarms are responsible for talking to all their subordinate swarms whenever a change of leadership occurs. The virtualized metaswarm enables all the subordinate

---

[2] Referring to the database data belonging to data owners.

[3] The precise XOR distance interval is $[2^i..2^{i+1}-1]$, for a given value of i uniquely identifying that root swarm's i^{th} virtualized metaswarm.

[4] This is thanks to the fact that the XOR distance function is symmetric.

[5] Although Kademlia DHT's do not typically refer to the Chord DHT's "finger table" terminology, it is a convenience that this paper employs.

swarms to update their finger tables when the root swarm has a change of leadership. The root swarm's finger table remains unchanged.

3.3. Bluzelle Harvesting Agent

Bluzelle is ultimately powered by a single piece of software that resides on every node in the network, irrespective of which swarm that node is in or if the node is a leader or not. That software is one of the key deliverables from the Bluzelle project. It is a C++-based software that drives all the logic in Bluzelle and that evolves regularly throughout the lifespan of Bluzelle. Since the key work that this software does is harvesting of data for customers, it is called the Bluzelle Harvesting Agent (BHA).

3.4. Tokenization

Bluzelle is powered by two tokens:

- **Ethereum ERC-20 external token: BLZ.**

  This externally-tradable token bridges the Bluzelle network's native BNT token with Ethereum's own native ETH token.

- **Bluzelle Network Token: BNT.**

  This is a token native to the Bluzelle network. It is internal to Bluzelle alone, and enables the Bluzelle crypto-economy, where consumers pay in BNT tokens and producers earn in BNT tokens.

The Ethereum blockchain, while universally accepted as a standard crypto-currency blockchain that bridges many other ERC-20 tokens like BLZ, operates within the constraints of a blockchain. It is limited by block confirmation times, as well as mounting transaction costs every time a transaction is sent on Ethereum. The latency in transactions "completing" [6] and the

[6] A transaction is deemed completed once both the database (CRUD) transaction has completed AND the payment has been confirmed/completed.

high cost for each such transaction makes the ERC-20 token unsuitable as a direct representation of value for Bluzelle's network transactions.

More succinctly:
Ethereum and its ERC-20 tokens are too slow and expensive for real-time database accounting.
The BNT internal native token exists to enable high-speed, zero-cost, and real-time database accounting.

3.4.1. Bluzelle Token Gateway

The Bluzelle token gateway bridges the Bluzelle network with the Ethereum network, enabling the flow of value between the two networks. In Ethereum nomenclature, the gateway is an oracle that connects the two networks together. There is absolutely no need for human intervention, for the gateway to serve traffic in both directions.

3.5. Database 101 - CRUD API

CRUD stands for "create, read, update, and delete". It is a commonly used acronym in software engineering and stands for the four basic functions pertaining to databases and permanent storage. CRUD covers the functionality of relational databases, where each of create, read, update, and delete can be mapped to corresponding SQL and HTTP methods.

A password of the user's own choosing is also required, and it is up to the user to protect this password and keep it available for later. All the data stored in key value pairs are encrypted, with the password being used as the initialization vector in AES 256 [7] symmetric key encryption. This password is only ever used locally and never travels on the network in any way, shape, or form [8].

[7] Symmetric encryption is faster than asymmetric and harder to break for the same key length.

[8] A derivative of the password will be temporarily shared with a trusted 3rd party proxy if the user elects to participate later in optional symmetric re-encryption.

3.6. Sharding

Shard stands for "System for Highly Available Replicated Data". Large databases often are hard to work with due to the size and memory constraints they come with. By breaking up the database along logical lines (partitioning), the database becomes much easier to work with.

A logical shard is an atomic unit of storage - it is the smallest unit in Bluzelle's engine and contains individual units of data that all share the same partition key. A partition key is a unique identifier that allows the shard to be accessed for the retrieval of information. In Bluzelle, the partition key is the leaf swarm number. A leaf swarm contains many logical shards, all having the same logical shard identifier, each of which maps back to data shards.

In Bluzelle nomenclature, groups of logical shards are stored on leaf swarms, and it is the amalgamation of these units (leaf swarms) that makes up the entirety of the Bluzelle database. As such, logical shards cannot span multiple leaf swarms and the size of any one shard is an important consideration.

Partition keys [9], the identifiers of logical shards (synonymous with leaf swarm id's), allow the application to store and retrieve data from the correctly identified leaf swarm efficiently. These keys relate to the data stored within the shard using either dynamic or algorithmic categorization.

Through algorithmic categorization, a function requiring the identifiers for the actual stored data is used to determine the logical shard identifier it maps to. A simple example [10] of this would be using the modulus operator to split all data evenly among fifty shards:

hash(key) % 50

[9] Not to be confused with document partitions.

[10] This is just an example. Bluzelle uses a more complex function defined in the Jump Consistent Hash algorithm

The remainder would then determine which logical shard identifier to use to store and retrieve information for data that is uniquely identified by 'key' [11].

3.7. Jump Consistent Hashing

Jump consistent hashing (JCH) was first described in a white paper by John Lamping and Eric Veach at Google [12]. It is an elegant algorithm that only takes about 5 lines of code in a language like C++. JCH does not have a state machine, and therefore requires no storage. It is a simple algorithm without lookups in memory and is therefore much faster as well. In fact, it even divides the keyspace more uniformly amongst buckets. Furthermore, when the number of buckets changes, it is able to uniformly re-distribute the keys in a way where only a minimum number of keys actually move. Its chief requirement is that the buckets themselves have to be identifiable in a sequential number order.

More specifically, JCH satisfies the following two properties:

1. Each bucket gets approximately the same number of keys.
2. When the number of buckets changes, most keys do not move. Only the small proportion of all keys that have to move are affected.

Bluzelle uses JCH to map from the key (in key value pairs in a NoSQL table) to the id of the swarm that the key is replicated in. Once that id is found, it is easy (using Kademlia hashing) to find the means to reach that swarm even if that specific swarm is not running.

Example:

```
F(key) = swarmId
```

[11] The value of the "key" can be the literal value of the key, or some simple hash of the key if the key itself is not a scalar value (i.e., a tree or array or document)

[12] https://arxiv.org/pdf/1406.2294.pdf

The function F above is the JCH function. It takes as input the key value, and outputs a swarm identifier number, which is a number in the range 1..n, where n is the number of swarms. If n changes, the value of `swarmId` for the vast majority of values of `key` does NOT change.

3.8. Kademlia Hashing

In a DHT, a peer (node) contains a local table known as a routing table. This table is important when performing a lookup, because it contains the useful information to traverse the local topology around the node within the network (like a map that a courier uses when delivering packages in a localized area).

Kademlia is an advanced form of a typical peer-to-peer distributed hash table which has been structured in a way to make particular use of the special symmetric and geometric properties of the bitwise XOR function.

Via recursive indirection, Bluzelle uses Kademlia hashing to efficiently enable nodes to know about every other swarm on the network. Using Kademlia's own form of "finger tables" [13], each node in the network only needs to know information about how to reach O(log(n)) other leaf swarms, where n is the total number of leaf swarms on the network. It can be proven that with this information, every node can reach every other leaf swarm, by doing multiple lookups until the desired swarm is found. This means that irrespective of how large the network ever becomes, every node can reach every other leaf swarm within O(log(n)) tries, by only storing O(log(n)) data [14]. This is an important fact:

Bluzelle is able to handle exponential growth.

[13] We use the term "finger tables" here loosely for convenience, as an analog to a Kademlia routing table.

[14] This also implies that any node in any leaf swarm x can reach every other leaf swarm y in O(log(n)) tries, and that each such leaf swarm x only needs to collectively store O(log(n)) data to do so, via its consensus protocol.

3.9. Partial Replication

Replication of shards across multiple nodes in a swarm offers a number of unique benefits to the Bluzelle engine. Note that partial replication means that not every node in the network has a copy of the data -- only the nodes within the leaf swarm delegated to that data replicate it. This is one of the key differences between Bluzelle and a traditional "blockchain". Blockchains are inherently slow and do not scale well, as every shard of data (transactions or blocks) is 100% replicated everywhere, putting severe vertical scaling limitations on the network. On the other hand, Bluzelle by design only stores the data amongst a strategic subset of the nodes, statistically providing an effective guarantee that the data is always available and still achieving the benefits of boundless horizontal scaling. An important property of a leaf swarm is there is 100% replication within the leaf swarm. But on the whole, partial replication exists because only ONE swarm amongst all the swarms in the network replicates a given piece of data. An interesting incumbent technology that can be compared to this is the content delivery network (CDN).

3.9.1. Load Balancing

A benefit of having a logical shard stored on multiple physical nodes is speed - by having the same data accessible through different hardware resources at various geographical locations, the system may load-balance queries to retrieve data from nearby nodes that are least taxed at any given moment in time. This permits Bluzelle to dynamically perform queries and retrieve data in the most efficient way possible, maximizing use of the shared resources spanning across multiple nodes.

3.9.2. Redundancy

As replicated data is stored across different nodes with unique infrastructure, there is a severely reduced causation between single-node failure and loss of the shard. This method of mirroring serves to secure the availability of data in an efficient manner by ensuring any single point of failure is inconsequential.

3.10. Consensus

Bluzelle deals with consensus differently from blockchains, doing away with any concept of a network-wide universal state. There is no need for a single state for the whole network, so Bluzelle applies the consensus model on a swarming level, ensuring that leaf swarms of nodes storing data shards are each reaching localized consensus, using our customized forms of consensus and proof algorithms [15].

A swarm with consensus appears to clients interacting with that swarm (or other swarms interacting with the swarm) as a single, atomic, indivisible unit that stores a set of data reliably. Any node in that leaf swarm can accurately service requests pertaining to that data.

In the Bluzelle model, we have a swarm of swarms (metaswarm), where each of the "leaf swarms" can be regarded metaphorically as a single, black-boxed node. How each such swarm achieves consensus is abstracted away, meaning each swarm could use any one of the consensus protocols mentioned in this paper, or even each be a private blockchain onto itself.

4. Karma

Every producer (farmer) operator on the network is typically entitled to run one or more nodes on the network as farming nodes. Each such operator will use their Ethereum address as the "key" that identifies them. This identifier is unique to that farmer and is tied 1:1 with their Ethereum address. The farmer also has a "Karma Index", which is a score that dictates how well-behaved the farmer is. The karmic index can go up and down depending on the farmer's activities and decisions, intentional or not, autonomous or not, and spans all the nodes the farmer operates. If one such node misbehaves, the karmic index typically drops and this applies to all the farmer's nodes. Furthermore, the farmer is required to put up a stake (in BNT tokens) that is proportional to the number of nodes and inversely proportional to the karmic index.

[15] For the purposes of this document, the algorithms are briefly introduced in their generalized form.

5. Sybil Attacks

Some blockchain networks, like Bitcoin, allow anyone to add their node to the network. That brings the concern that a malicious organization could potentially add so many nodes that they disproportionately control the network.

This is referred to as a Sybil attack. This is an attack on the reputation of a distributed system where an attacker creates a large number of nodes with the intent of disrupting the network by hijacking or dropping messages.

Bitcoin and Ethereum obviate Sybil attacks by making them prohibitively expensive via proof of work. In other networks, such as BigchainDB, the governing organization controls the member list, so Sybil attacks are not an issue. However, in this latter case, it is not truly decentralized.

There are several vectors Bluzelle employs to prevent Sybil attacks or statistically drive the probability of successfully executing one down to the point where it is for all intents and purposes impossible to carry one out.

When a bad actor is caught, they can be blacklisted, and economically penalized. Other nodes on Bluzelle powered by the same operator as the bad acting node will also be affected and ultimately could all be removed from the network, if the operator's karmic index goes below a minimum threshold. Following are the anti-Sybil measures taken by Bluzelle:

- Farmers (producers) are required to put up a BNT stake to participate in the network. This stake serves as a requirement for participation and as a strong economic deterrent from bad behaviour.
- The Kademlia distributed hash table is used as it relies on message redundancy and the XOR distance function. Neighbours are selected from a uniformly distributed list, and messages are redundantly sent to multiple neighbors of the intended node for anti-Sybil verification purposes. Using this approach and with multiple parallel efforts to find a swarm, it is statistically improbably that a successful Sybil attack can be performed that impedes or

misleads swarm location efforts. Nodes that mislead the swarm location effort will be systematically tracked down and caught and treated as bad actors.

- A request to a swarm for CRUD functionality is done with redundancy, where multiple nodes in the same swarm all perform the request. These nodes are typically chosen using metrics comparable to how a CDN chooses nodes to serve out a request. Given the node->swarm membership rules for Bluzelle, it is statistically unlikely that multiple such nodes chosen to perform a given transaction are colluding bad actors that deliver bad yet consistent data. A bad actor will be caught.
- Swarm membership is determined by the network and cannot be chosen by nodes. This means that short of a massive attack on the order of a huge proportion of the network (ie: a "Google" attack), a would-be Sybil attacker who attempts to join the network with n nodes or masquerading with the identity of n nodes will not be able to gain a critical mass of memberships into any single swarm. Due to how requests are made redundantly to multiple nodes in a swarm, bad actors who infiltrate a swarm will be caught.
- Nodes can be posed a challenge request to participate in a proof of storage test. This test is performed in cooperation with the consumer on either a random network-initiated basis or by the consumer directly and forces the targeted node to prove they have the correct data. Failure to pass this test can result in the aforementioned penalties for being a bad actor.

6. Byzantine General's Fault

Byzantine failures in a system are a result of not being tolerant to Byzantine faults.

What then is a Byzantine fault?

Simply, it is any fault that manifests itself with different symptoms, depending on who the observer is. This makes tolerance to Byzantine faults particularly difficult. The goal of a Byzantine-fault tolerant (BFT) system then is for the system to be able to continue to operate correctly even if some such faults exist.

The Byzantine General's problem can be described as follows: You have a collection of generals, each leading their own division of the Byzantine army. They wish to sack the city they have surrounded. The decision now is whether to attack or retreat. Some generals might want to attack. Some might want to retreat. In fact, some generals are traitors and intentionally make a decision that undermines the others, giving each other camp (retreat or attack) the impression that the majority has chosen that option. This might happen if the traitors are in the position to break a stalemate. By selectively making multiple votes, the traitors can mislead both sides and cause catastrophic loss. Adding insult to injury, the messengers between the generals themselves can be killed, lost, or could forge the messages.

One of the key ways to protect against Byzantine is to have a default understanding of what to do if there is no information. In the case of Bluzelle, if misleading or corrupted or inconsistent information is detected, the default is to do absolutely nothing. Misleading information might be data that does not agree with the consensus state machine. Corruption can be detected if a CRC32 checksum fails. Thanks to the aforementioned redundancy in CRUD requests made to a swarm, inconsistency is caught, whether intentional or not. In any case, if such a fault is detected, Bluzelle nodes are instructed to ignore the transaction and do nothing. Only authenticated transactions with proper credentials and checksums are accepted and transacted upon. By this way, Bluzelle is Byzantine Fault Tolerant by design.

7. What is the Bluzelle pricing model?

For the purposes of the MVP, the pricing model is a straightforward one chargeable in BNT tokens. Costs are expressed on a scale of 1-10, where functions with cost 1 are free and functions with cost 10 are extremely expensive.

The mapping from "cost" to Bluzelle tokens should be considered exponential and not linear, where higher costs become prohibitively higher amounts of tokens, numerically, and therefore should be functions that are used sparingly.

Absolute prices are calculated dynamically and are not 100% deterministic. Maximums are baked into Bluzelle, where the network adjusts the number of swarms and distribution of nodes as prices start to approach the maximums, getting increasingly aggressive with network adjustments as the maximums draw nearer, until expected average prices start to trend again.