Earth^{64} Data Structure V1.0.2

Authors: Toufi Saliba[1], Robert Moir PhD[1], Dann Toliver[1], and Barry Rowe PhD[1]
Contributors: Jessica Radley[2], Bryan Tran[1]

[1]Earth^{64}
[2]Earth^{64} Ecosystem

2022/11/09

Preface

An ownership management system such as (NTTs, NFTs, Tokens, Assets, Ledgers, Coins, Currency, Digital Wallets) can be optimal when its security is maximized by design while its interoperability does not depend on any third party and can truly be peer-to-peer. Additionally, its scale must be able to handle not only all existing devices and current needs but capable of expanding as many as several orders of magnitude to cover the devices and needs of the future, all while remaining fully decentralized with maximum security and minimum cost. When folks think of a truly optimal ownership management system that has all these characteristics, they must think of the ability to manage money globally and not just the NFT assets that have recently gained popularity. In this document, we picked a use case NFT but the ownership managements systems can span beyond just NFT. A true NFT management system can manage any kind of assets. For the avoidance of doubt, this is NOT an NFT art project where folks try to sell copies of paintings or whatnot, the true meaning of NFT is Non Fungible Token. Any paper money is a non-digital NFT, the smallest unit in Bitcoin referred to as satoshi aka sat is an NFT along with many other useful tools that we make reference of. Furthermore, Earth64 is NOT an ownership managements system, the same way the car transmission is not a car. Or the screws used in a construction site are not the building. Earth64 is a data structure, it is a building block and not the block

We emphasize that Earth^{64} is **not** the NFT management system. It is, however, an essential pillar, and its use can make it possible to develop an NFT management system with the desired characteristics mentioned above. Achieving this, however, requires that it is properly implemented. In this guide we do not intend to describe the implementation of the NFT management system itself, but rather how to build the data structure pillar we call Earth^{64}. This pillar is founded on its atomic, hierarchical components, which are files with the maximum size of 64 KB that some may refer to as packets. This guide is essential for any developer to ensure that the system they are building is interoperable with every other system being built anywhere on the internet that follows the same or a similar guide, with the developers neither ever having to talk to each other, nor needing to depend on any third party.

Contents

Glossary

B

bundle A (generally very large) collection of nodes corresponding to a branch of the Earth^{64} tree, each of which can hold a Sato-Server.

L

lot-space An abstract space of dimenstions $2^{32} \times 2^{32}$ used as an intermediate to map binary node addresses to GCS (latitude-longitude) coordinates to seven decimal places.

N

node A node on the Earth^{64} binary search tree (BST).

- **bundle node** An on-chain asset representing ownership of a unique node on the Earth^{64} tree and all of its descendants (i.e., an entire branch of the tree). This could be a level 13 genesis node or one of its descendants created by splitting an asset on-chain.
- **genesis node** A level 13 bundle node, one of the original 1,701 NFBs prior to activation.
- **invalid node** A node on the Earth^{64} tree such that no part of its corresponding spatial region covers a portion of the Earth's surface.
- **origin node** The starting node (node 1) when counting nodes in breadth-first search fashion, used to identify where counting starts in relative addressing. For Earth^{64} addresses, the root of the entire tree is the origin node, and for NFBs that are split on-chain, the genesis node in question is the origin node. In relative addressing, any node one likes can be treated as an origin node so that all its descendants are numbered in order starting from the origin node as 1.
- **root node** The Earth^{64} node corresponding to a bundle node after the development license has been activated (currently by burning the NFB asset on-chain).
- **valid node** A node on the Earth^{64} tree whose entire corresponding spatial region covers a portion of the Earth's surface. All of the bundle nodes and their descendants are valid nodes.
- **virtual node** A node on the Earth^{64} tree such that only part of its corresponding spatial region covers a portion of the Earth's surface.

node address A binary string with length ranging from 1 to 64 bits (inclusive), where the length minus one indicates the level of the node on the Earth^{64} tree.

non-fungible bundle (NFB) An NFT (currently on Algorand or Ethereum) that represents the ownership of a bundle node.

S

Sato-Server A 64 KB file containing cryptographic material sufficient to prove its current owner. Each such file can hold whatever information is desired or required in an implementation, including NFTs and smart contracts.

T

tilling service provider A web2 service that will make Sato-Servers viewable when the owner of a root node permits viewing and that will provide proofs of ownership of root nodes on request.

1 Introduction

Building an NFT management system using the Earth^{64} data structure can start by owning a license for a single bundle that is represented as an on-chain asset (currently on Algorand or Ethereum, and possibly others in the future). This asset is burned using the Algorand or Ethereum dApp to activate the license for the bundle. This process on Algorand, as an example, is explained in detail in a separate companion guide and is not part of the scope of this document.

The bundles in the Earth^{64} ecosystem are initially divided into 1,701 units called **genesis nodes**. Genesis nodes correspond to a unique node on level 13 of the Earth^{64} binary search tree data structure, a perfect binary tree with 64 levels and $2^{64} - 1$ nodes.[1] Genesis node are also called **bundles** because on-chain ownership of a node implies ownership of the entire branch of the tree starting from that node, which for a level 13 node is a total of $2^{51} - 1$ nodes.

Nodes that are represented as valid on-chain assets are called **bundle nodes**. These nodes can be genesis nodes on level 13, but they can also be lower level nodes because the owner of a level 13 node can split it on-chain to yield two level 14 bundle nodes, each of which represent ownership of $2^{50} - 1$ nodes on the Earth^{64} tree (essentially half the size of the original bundle). These nodes can be split again, and so on.

For this reason, tokens that determine ownership of a bundle node are called **non-fungible bundle (NFB)s** (NFBs), to distinguish them from the non-fungible tokens (NFTs) that are being managed in an NFT management system as described in this guide. Each NFB has a unique sequence number on-chain, which corresponds to a unique binary node address on the Earth^{64} tree. The entire list of the 1,701 genesis node binary addresses is readily available for any bundle node upon activation. The numbering system for assets that are split on-chain is described in Appendix D.

When a bundle node asset is burned, it activates a data structure license to enable the development of an implementation on Earth^{64}. Once the asset is burned, it leaves the blockchain environment and becomes an Earth^{64} **root node**. A root node is an Earth^{64} node that is pointed to by a burned asset, which serves to prove ownership of it and all of the nodes below it (its descendants). The value of this, then, lies in what you can do with this massive collection of nodes.

Each node in the Earth^{64} data structure can itself have attached to it a simple data structure called a **Sato-Server**, which is simply a 64 KB file containing cryptographic material sufficient to prove its current owner. As 64 KB is the maximum size of a TCP/IP packet, Sato-Servers can also be thought of as packet data, or informally as packets. It is because a branch is a (generally very large) collection of Sato-Servers, that it can be used as a key pillar in the implementation of an NFT management system in the manner illustrated in this guide.

Although running an actual implementation is a complex and serious undertaking, the big picture

[1] Note that we use the typical convention where the root of the Earth^{64} tree is level 0. So the 1,701 bundles that are level 13 nodes also have depth 13. This way, the leaf nodes of the Earth^{64} tree are nodes on level 63 yet have 64 bits in their binary addresses. This means that the number of bits in a node address is one more than the level number, and the level number is equal the number of bits of node addresses that can vary on that level, as will be explained below.

process is quite simple. This guide will provide the big picture overview of the implementation process through the consideration of an example use case of an Earth^{64} bundle to manage a loyalty NFT system for a large global company.

Following an introduction to the Earth^{64} data structure in Section 2, the main part of the guide is Section 3, which provides an illustration of how to use a burned asset as a key element in the implementation of an NFT management system using BIP32-style wallets on Earth^{64}.

At the end of the document are several appendices. Two of these are technical: Appendix A, which explains various methods of addressing nodes on the Earth^{64} data structure; and Appendix B, which explains a simple way of mapping a binary node address to the coordinates of the corners of its corresponding spatial region. Appendix C provides a detailed plot that helps to clarify the correspondence between locations on Earth and locations in the abstract "lot-space" used by Earth^{64} to map binary node addresses to geographic coordinate system (latitude-longitude) coordinates with exactly 7 decimal places. Finally, Appendix D, as indicated above, explains the procedure for mapping NFB numbers to their binary node addresses.

2 The Earth^{64} Binary Tree Data Structure

This section provides an explanation of the Earth^{64} binary tree data structure. Since each node of your bundle on the Earth^{64} tree corresponds to a region on the surface of planet Earth, it is important to understand how this correspondence is made. To explain this, it is necessary to understand how nodes on the tree itself are addressed and then how these addresses correspond to spatial regions.

This section will explain how each node address on the Earth^{64} tree, which is a binary string unique to each node, corresponds to a region of an abstract space called lot-space, and why only a subset of the nodes on the Earth^{64} tree correspond to regions on planet Earth using coordinates in the spherical geographic coordinate system (GCS), i.e., as latitude-longitude pairs. We can provide access to code that can inter-convert between node addresses and GCS coordinates, but the process is simple and is explained in detail below.

2.1 The Earth^{64} Data Structure

The Earth^{64} data structure is a perfect binary tree that maps the unique binary name of each node of your bundle to a unique, rectangular, geospatial region of planet Earth. It is designed so that for each node the corners of this region form a unique set of GCS coordinates with exactly seven decimal places.[2] The higher the node in the tree the larger this region is, and with each level you move down the tree the region halves in size. The design ensures that (away from the poles) the smallest divisions, those at level 63 of the Earth^{64} tree, are around 1 cm^2 in size—a relatable area for any human being since this is the approximate size of a fingertip.

2.2 Binary Node Addresses are Search Paths

Before we can explain how node addresses correspond to spatial regions, we must first explain how the node addresses are generated. The root node of the entire Earth^{64} tree has binary address `1`, and its two children have addresses `10` and `11`. Their addresses are formed by adding a `0` for the left child node and a `1` for the right. This pattern continues throughout the entire Earth^{64} tree by adding either a `0` or a `1` to the end of the parent's binary address, a `0` for the left child node and a `1` for the right (see Figure 2.1).

You will notice, then, that the binary address of each node of the Earth^{64} tree also specifies the search path taken from the root to reach that node. For example, for node `1011`, this node is reached from the root node `1` by first moving to the left child `10`, then its right child `101` followed by its right child `1011`. In this way, each binary node address is also a search path.

Note that the search path also relativizes to any given node. For example, if `11000000111001` is the address of your implementation's root node, then `1100000011100101` is reached from the root node by taking the left child and then the right, so `01` specifies the search path from the root node.

[2] Only four numbers are needed to specify this region uniquely, which can be the coordinates of opposite corners, or the coordinate of the upper-left corner and the lengths of the two sides.

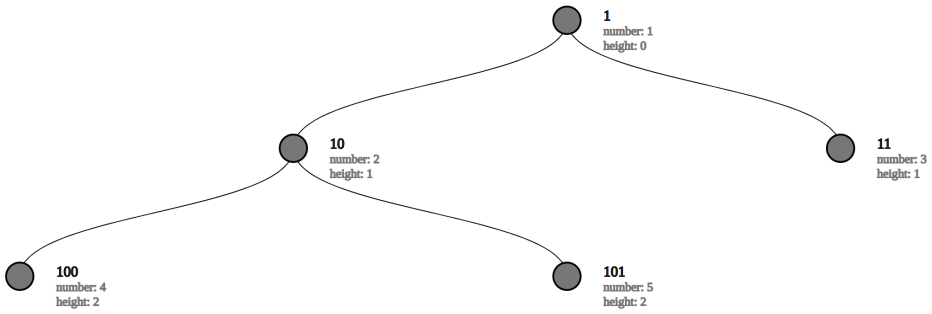


Figure 2.1: Addressing of nodes on the Earth^{64} binary tree works by adding to the end of the parent address either a `1` or a `0`, a `0` for the left child node, and a `1` for the right.

2.3 Level Numbers and Lengths of Binary Addresses

Because off-by-one errors are very easy to make in many areas of computing, it is helpful to be clear about where confusion may arise. In the case of the Earth^{64} data structure, it is important to be clear that the level number is one less than the number of bits in the binary addresses of nodes on that level.

The root of the Earth^{64} tree is level 0, and has 1 bit in its address, i.e., `1`. The genesis nodes on level 13 have 14 bits in their address, as is the case for the dummy example `11000000111001` we use throughout this guide. So the leaf nodes on level 63 have 64 bits in their binary addresses.

A result of this arrangement is that *the level number is equal to the number of bits in a node address that vary on that level.* For level 0, there is no variation (only node `1`), for level 1 there is 1 bit that can vary (yielding nodes `10` and `11`), and so on. So for level 13, that contains the genesis nodes, there are 13 bits that vary, giving rise to $2^{13} = 8192$ level 13 nodes. Similarly, there are 2^{63} leaf nodes on the last level, level 63, of the Earth^{64} tree.

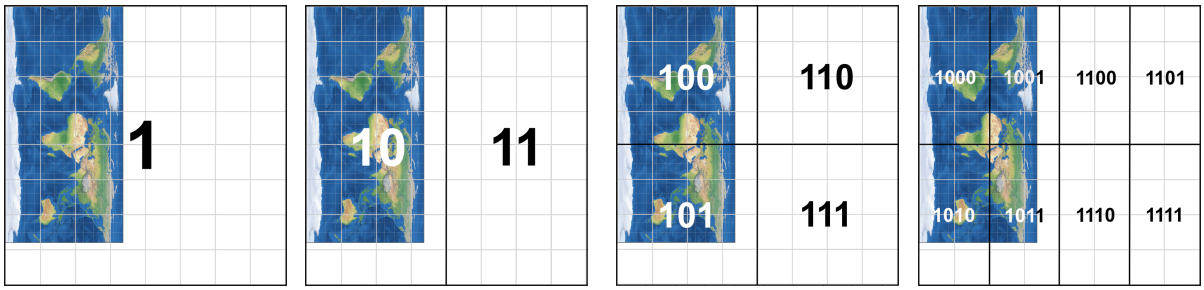
2.4 First Mapping Function: (Binary ↔ Lot-Space)

To understand the correspondence between binary addresses and regions on Earth's surface, we must introduce an abstract space, called **lot-space**, where the spatial divisions are computed. The entire lot-space has length and width 2^{32}, with the x-coordinate corresponding to the latitude and y-coordinate corresponding to the longitude.

The root node of the Earth^{64} binary tree, which has binary address `1`, represents the entire lot-space (see Figure 2.2(a)). The way it is arranged, the top-left corner has coordinate $(0, 0)$ and the bottom-right corner has coordinate $(2^{32}, 2^{32})$.[3] The two child nodes of the root represent a splitting of the root node region into two equal parts along the x-axis, which corresponds to latitude (see Figure 2.2(b)).

[3]This is done for two reasons. First, so that the first coordinate will be the latitude and the second the longitude, which is standard for GCS coordinates. Second, though making the bottom-left corner $(0, 0)$ would be more natural mathematically, understood geographically this implies a reflection and a rotation of the Earth's surface. Making the top-left $(0, 0)$, on the other hand, implies only a rotation of the Earth's surface by 90 degrees clockwise. This is evident from Figure 2.2(a), which shows the location of Earth in lot-space, but is clearer in Figure C.1.

For the splitting of the root node, its left child node `10` picks out the lot-space region with top-left corner $(0, 0)$ and bottom-right corner $(2^{31}, 2^{32})$). Similarly, the right child node `11` picks out the lot-space region with top-left corner $(2^{31}, 0)$ and bottom-right corner $(2^{32}, 2^{32})$. See Figure C.1 for the precise numerical mapping.



(a) Level 0: node `1` (b) Level 1: nodes `10` and `11` (c) Level 2: nodes `100` through `111` (d) Level 3: nodes `1000` through `1111`

Figure 2.2: The first four levels of the Earth^{64} binary tree. Level 0 (a) is all of lot-space, and each subsequent level splits each region in two, appending a `0` to the address for the left or upper sub-region, and a `1` for the right or lower sub-region. The CC BY-SA 4.0 licensed map image contained in the figure was obtained from https://map-projections.net.

This spatial division process continues for each new level of the tree, and alternates between a vertical division (splitting on x-axis, latitude) and horizontal division (splitting on y-axis, longitude), dividing the region in two at each step. The left child node corresponds to the left or upper half of the parent's region, and the right child node corresponds to the right or lower half. This pattern is apparent in Figures 2.2(a) through 2.2(d).

The correspondence between the binary node address and lot-space regions allows the mapping from binary to lot-space to be given as a simple function where the lot-space coordinates are specified in terms of the even bits (for X or latitude) and odd bits (for Y or longitude) of the node address, where the leading `1` of every node address, considered bit 1 (i.e., odd), is not used in the mapping. Details of how this works are provided in Appendix B.

2.5 Second Mapping Function: (Lot-Space ↔ GCS Coordinates)

The mapping from lot-space (X, Y) to GCS latitude-longitude coordinates (λ, ϕ) is very simple:[4]

$$(X, Y) \mapsto (\lambda, \phi) = \left(\frac{X}{10^7} - 90, \frac{Y}{10^7} - 180 \right). \tag{2.1}$$

This means that the point $(0, 0)$ in lot-space is mapped to $(-90, -180)$, one of the points at the South Pole. It also implies that more than half of lot-space is mapped to invalid GCS coordinates. See Figure 2.2(b), which shows how over half of lot-space does not correspond to a location on

[4]Note that for every Earth^{64} node, there is always an exact correspondence between the lot-space coordinates of the corners of the region picked out by that node and the GCS coordinates of those points to the 7^{th} decimal place. If, however, floating point arithmetic is used in (2.1), ensuring an exact result will require rounding to the nearest 7^{th} decimal place on account of floating point error.

Earth. Figure C.1(a) provides the precise numerical correspondence.

The actual surface of the Earth corresponds to the lot-space region from $(0, 0)$ on the top-left, to $(180 \cdot 10^7, 360 \cdot 10^7)$ on the bottom-right, showing how lot-space is tracking GCS coordinates to the 7^{th} decimal place as integers (see Figure C.1 for a detailed plot). This is done because GCS coordinates to the 7^{th} decimal place are universally agreed upon as valid across the globe, providing globally unique addresses searchable in a perfect binary search tree.

Given that many (more than half) of the Earth^{64} nodes do not correspond to actual geographical regions on the surface of the planet, we need to distinguish nodes that include territory on Earth from those that do not.

The **valid nodes** on the Earth^{64} tree are those that correspond to regions in lot-space that map fully into some region of the planet's surface. For example, node `1000` and all of its children are valid nodes (see Figure 2.2(d)).

Nodes that map partially into a region of the surface are called **virtual nodes**, which are distinguished by the fact that they may have valid descendants. The root of the Earth^{64} tree is such a node, as are nodes `10`, `100` and `101` (see Figures 2.2(a) through 2.2(c)).

The **invalid nodes** are those that correspond to lot-space regions with no overlap with the surface of the Earth, which includes node `11` and all its descendants (see Figure 2.2(b)).

Please note that not all of the nodes that map into the surface of planet Earth are utilized by Earth^{64}. Only those nodes that cover international territories are part of this project, while the remaining valid nodes are either land being used by the Land^{32} project or are disputed territories without global consensus on their ownership.

2.6 Available Helper Code

Some example code, including the resources referenced below, can be used to navigate the Earth^{64} tree.

Code to convert binary node addresses to GCS coordinates is available in JavaScript as the file `sato-data.js`. In particular, the function `getPosCoordinates` takes as input a binary string (the node address) and returns an object specifying the GCS coordinates of the top-left and bottom-right corners of the region picked out by the node. There are also helper functions that map between GCS coordinates and lot-space, and that inter-convert binary addresses and their equivalent decimal values.

We can also provide Python code to inter-convert node addresses, lot-space bounding boxes and GCS coordinates using the bit manipulation techniques described in Appendix B.

Another resource that is useful for illustrating the structure of the Earth^{64} tree is the `graph-view` tool, which is a web page interface that allows you to click on a node to split it and see the node address and the GCS coordinates of the region picked out by that node. This information is color-coded in red, yellow or green, depending respectively on whether the node is valid, virtual or invalid.

This tool uses `sato-data.js` in the background to do the conversions, and was used to generate the image of the Earth^{64} tree in Figure 2.1 above. A JavaScript object that contains information about all of the nodes up to level 13 is also included in `tree_data.js`.

3 Using Earth^{64} to Implement an NFT Management System

Once an asset has been burned on Algorand or Ethereum, the license to use the corresponding Earth^{64} bundle is granted, where the burned asset itself functions as the cryptographic proof of ownership of the development license and ownership of all the Sato-Servers the bundle contains.

The possible use cases of Earth^{64} bundles are limited only by the imagination of those developing the NFT management system.

As an illustrative example use case we will show you how a large company with a global footprint, such as Walmart, could use a single Earth^{64} bundle to create around 1.1 billion (2^{30}) hierarchical deterministic (HD) wallets that each contain up to around 1 million (2^{20}) NFTs.

3.1 Proof of Ownership of Earth^{64} Assets

First we must provide a bit more detail on how proof of ownership is handled for Earth^{64} assets. This is actually quite simple, but depends on whether the bundles are still being managed on-chain (on Algorand or Ethereum) as bundle nodes or whether they have proceeded to the implementation stage as root nodes and Sato-Server files have been created.

For bundle nodes, i.e., NFBs that are still being managed on-chain, ownership is determined by the private key for the account that purchased or received the asset. Since the same account retains ownership of any asset that is split on-chain, there is no question about who owns the split assets.

On the other hand, once a NFB is burned on-chain to become a root node, activating the development license, the owner of the burned asset as registered on-chain becomes the owner of all of the Sato-Servers in the corresponding branch of the Earth^{64} tree. Moreover, this burned asset becomes the proof that the last owner of that asset before it was burned is the owner of all the Sato-Servers. This is the manner in which the burned asset becomes the development license.

To indicate ownership of the Sato-Servers, i.e., the up to 64 KB files corresponding to the nodes of a bundle, the Sato-Server files (including a header and body) need to be signed by the private key corresponding to the public key of the burned asset, or to another signing key that the root node has delegated signing privileges to. As will be explained below, ownership of the Sato-Servers themselves are fixed to the owner of the burned asset, but the contents of Sato-Servers (contained in the body) can be assigned to different private keys by granting permission for the contents to be signed by those private keys.

In our scenario, the wallets and NFTs are both digital entities that are stored in the body of a Sato-Server. If the wallets are to be managed entirely by the company (e.g., Walmart), then the wallets (each contained inside a Sato-Server) would be cryptographically signed by a private key owned by the company. If instead the company wished a given wallet to be owned by a particular user, or some third party, then that wallet would be signed over to a wallet owned by that party, who can then sign from that point forward (multi-signature wallets could be handled in the same way).

3.2 Creating Your Sato-Servers I: Design Phase

The first high-level step of implementing your NFT management system is the creation of the Sato-Server files. This can be done in a variety of different ways, depending on what is desirable for your business. The main constraints are that all nodes must be properly indexed with the correct binary address on the Earth^{64} tree, the Sato-Server files must be no larger than 64 KB (the maximum size of a TCP/IP packet),[5] and each Sato-Server file and its body must be signed with the appropriate key.

In the scenario described above, imagine you wish to create up to 1.1 billion (2^{30}) hierarchical deterministic (HD) wallets (each defined by a token contained within an internal node Sato-Server), each of which can contain up to 1 million (2^{20}) NFTs (each contained in its own leaf node Sato-Server). You would almost certainly not wish to create all of these wallets at once, creating instead, say, only a few million for an initial release. And for the NFTs "inside" each wallet,[6] you would not require all 1 million NFT containers immediately either.

Thus, the strategy would be to decide how the wallets and NFTs will be encoded as contents of a Sato-Server, and then create the Sato-Server files for the first million or so wallets on level 43 ($= 13+30$) of the Earth^{64} tree, and then "inside" each of these wallets you would create on level 63 ($= 43 + 20$) as many Sato-Servers as NFTs you wish to disburse, with each of these Sato-Servers containing a single NFT.

To work out the addresses of these wallets you need to know the binary address of your root node, the different levels (in a relative sense) on which you are creating Sato-Servers (here level 30 below the root and 20 levels below that) and how many Sato-Servers you are creating on each level. The addresses for the Sato-Server can then be specified in the relative notation described in Appendix sections A.2 and A.4 below.

For example, suppose that your level 13 root node has binary address `1100000111001`. This is equivalent to 12345 in decimal. Suppose that we want to create $N + 1$ wallets on level 43 and $M + 1$ NFT containers on level 63. In this case, the $N + 1$ wallets on level 43 (30 levels down from level 13) would have large absolute decimal addresses from 13255342817280 to $13255342817280 + N$. In the relative notation, however, these are more compactly expressed as the range from 12345/30:0 to 12345/30:N, but correspond to the same binary addresses.

In turn, the $M + 1$ NFT containers on level 63 (20 levels down from wallet level 43) would have even larger absolute decimal addresses (now 64 bit numbers), but in the compact, iterated relative notation (see Appendix A.5 for details) would have addresses from 12345/30:Y/20:0 to 12345/30:Y/20:M, where Y is the (zero-indexed) wallet number, between 0 and N (inclusive). Thus, you would create the $(N + 1)(M + 1)$ Sato-Servers at these addresses, ensuring that body content is signed with the appropriate signing key, depending on your ownership model, and that

[5] A smaller Sato-Server may be desirable for optimal network efficiency, since large TCP or UDP packets are split before being sent on the network. For optimal efficiency, the Sato-Server size (plus header overhead) should be less than the network path MTU. Also, to fit inside a large TCP or UDP packet, the Sato-Server should be slightly less than 64 KB so that the packet including the TCP/UDP and IP headers is less than or equal to $65,536$ bytes.

[6] Although from a user-experience perspective the NFTs are stored inside the wallet, in terms of the implementation the NFTs are actually contained inside Sato-Servers of nodes that are (level 63) descendants of the Sato-Server of the level 43 node containing the given wallet token.

the files are signed with the root node key or delegate.

3.3 Creating Your Sato-Servers II: Implementation Phase

To properly automate the process of file creation in this scenario, you will need to develop a tool that allows you to create a certain number of Sato-Server files on specific levels of the 64 level Earth^{64} BST. The reason for this is that in our scenario we only need Sato-Servers on the wallet container (level 43) and NFT container (level 63) levels.

We will consider in detail only the case of wallet container creation, as the NFT container case is similar. The majority of this subsection, however, will involve explaining how to construct the root node header.

Before you do anything else, you will need a function `create_sato_server`, that will create the files for each Sato-Server, starting with the root node. This file will need to take many arguments, enough to fill all the fields of the header and body, as explained below.

Since the root node is a special case, the root node Sato-Server will require sufficient information to permit the retrieval of a proof-of-ownership from the appropriate blockchain. Retrieving this proof is generally a task that will be performed by a tilling service, as will be explained below.

The format of the data in any Sato-Server, be it the root node or otherwise, will be required to conform to a standard across the Earth^{64} ecosystem. We provide an example format below for illustrative purposes; a formal standard Sato-Server specification is forthcoming.

We provide an example Sato-Server structure using Python `dict` types, which can be exported to a JSON file. The overall Sato-Server structure consists of a header, body and file signature, as such

```
sato_server = {'header': sato_server_header,
               'body': sato_server_body,
               'signature': signature_of_file_content},
```

where `sato_server_header` is a dictionary with a specified structure that will be described below and the body has the high-level structure

```
sato_server_body = {'content': body_content_dict,
                    'signature': signature_of_body_content}.
```

You will notice that the body contains a signature of its content, while the header is only signed together with the body. It is important to be clear that the header content is fixed by the Earth^{64} specification, but the content of the body can vary depending on the needs of the implementation. Also, the two signatures have different functions and requirements. The signature of the file must be produced by the signing key of the owner, or an approved delegate,[7] of the burned

[7] A recommended protocol for delegating file signing privileges to non-leaf subnodes of the root node will be included in the forthcoming Earth^{64} Sato-Server Specification, but this process can be handled in whatever way is optimal for your implementation. It is important to understand that although it is possible to do this, for security reasons a production system must not use the root node key to sign all of the Sato-Server files unless this process can be handled in an extraordinarily secure way. Consideration of delegation protocols, however, is beyond the scope of this document.

on-chain asset, while the body signature can vary depending on the needs of the implementation (as already indicated above).

In this example format, the header content of a generic Sato-Server must include the following data:

```
sato_server_header = {'address': binary_address_of_this_sato_server,
                      'counter': file_version_number},
```

where `binary_address_of_this_sato_server` is the correct Earth^{64} node address of the Sato-Server in question, and `file_version_number` is a number that is incremented each time the file is updated (to prevent use of old files and file replay attacks). Recall that the address of the Sato-Server included here also functions as a record of the search path to reach that node.

Again, for this example format, the header content of a root node Sato-Server must include three additional fields, to look like the following

```
sato_server_header = {'address': binary_address_of_this_sato_server,
                      'counter': file_version_number,
                      'transaction_id': id_of_tx_proving_ownership,
                      'algorithm_id': id_of_signature_algorithm_used,
                      'public_key': public_key_of_root_owner},
```

where `id_of_tx_proving_ownership` is the transaction ID of the transaction proving ownership of the asset that is burned, `id_of_signature_algorithm_used` will be an integer or string constant indicating the signature algorithm used by the root node throughout your implementation, and `public_key_of_root_owner` is the public key of the owner of the burned asset.

With the information contained in the root node header, it is possible to verify that this node is owned by the owner of the burned asset by searching a blockchain explorer for the transaction and verifying the signatures of the transaction and the Sato-Server header with the supplied public key.[8] This proof, once obtained, is then a proof of ownership of all of the Sato-Servers below the root in this branch of the Earth^{64} tree.

For illustrative purposes, we will provide a dummy example of a signed root node header using the Python `bitcoin` library for simplicity. Let us suppose that the burned asset has the transaction hash

`0c32de59227d6db2a255ddec768c1ec28a0d814237af1cc560e14d0df2c04e4a`

and has

`02ca6c6c6fc220c3ed3fa50e89625ee4d3113010ec6ce0e94417777ab4edb55cf1`,

as the compressed public key, then this `dict` would look something like the following:

```
root_sato_server_header = {'address': '11000000111001',
                           'counter': 0,
                           'transaction_id':
```

[8] The full proof also requires tracing the chain of ownership from a genesis node to the given root node.

```
'0c32de59227d6db2a255ddec768c1ec28a0d814237af1cc560e14d0df2c04e4a',
                            'algorithm_id': 'ECDSA',
                            'public_key':
'02ca6c6c6fc220c3ed3fa50e89625ee4d3113010ec6ce0e94417777ab4edb55cf1'},
```

where we have used the same level 13 root node address as in previous examples (decimal address 12345).

The body will be empty so its content dictionary will simply be the empty dictionary. To obtain the signature of the body content dictionary we first call `repr` to get a string representation of it and then pass the result to `bitcoin.ecdsa_sign` to obtain the following result for the root Sato-Server body:

```
root_sato_server_body = {'content': {},
                         'signature':
                           'G7i9rsxrNwvECg03v7ZxLb/prCZ+GjowKHHSUSMM4Pcc'+\
                           'RHs7V++ONxaCl4iCINkdV786TIfvZJPK97218HlnJ3Y='}.
```

To obtain the signature of the entire file we call `repr` on the tuple `(sato_server['header'], sato_server['body'])` and then pass the result to `bitcoin.ecdsa_sign` to obtain the following result for the root Sato-Server header:

```
root_sato_server = {'header': root_sato_server_header,
                    'body': root_sato_server_body,
                    'signature':
                      'GyZJ1Cw5/fLd9XVaTJjUH5YWwx9lAc9yK/Q/lVGhKoqk'+\
                      'VagjESI9hJbsfrmh/GWWR9vjIcA0UzIyFDxjjHdoUIU='}.
```

A call to `bitcoin.ecdsa_verify` using `repr(sato_server['body']['content'])` as the first argument, followed by the file `signature` and header `public_key` fields will show that the body signature is valid. A similar call on the tuple specified above will validate the file signature.[9]

Note that a configuration file will be needed that specifies the signature algorithms used in your implementation (the function corresponding to `'ECDSA'` in this example), unless this is included as part of the forthcoming Sato-Server standard.

The headers for the generic Sato-Servers can be created in a similar way, and the function to do so will only depend on the node address and file counter value. The greater variation will come in the contents of the wallet and NFT containing Sato-Servers. However you decide to specify the data constituting a wallet, it will need to indicate what is the valid signing key for that wallet.

This can be done, for example, by including a wallet NFT in the content of each level 43 Sato-Server. This wallet NFT would need to contain the public key for the wallet signing key, and have its data signed by the root node (or delegate) signing key. The presence of this NFT in the wallet node Sato-Server indicates that the descendents of this node contain assets that belong to the

[9] The signatures themselves can be generated using the private key

144c2e006fde1f0e3e79011d198c8fdfb126611429c0d83edf89e9ce2ce905b3.

signing key of the wallet (that are in the custody of the company). In keeping with this, the body content of the Sato-Servers below this node would be signed with the wallet signing key. The wallet Sato-Server is then signed with the root node signing key to indicate that the root node approves of this state. There are, of course, other ways in which ownership could be handled here.

A similar procedure can then be used to disburse NFT rewards to the Sato-Servers inside each of the wallets you create. Once the details of the structure and content of the wallet and reward NFTs have been specified, it is straightforward to write a function to automate file creation.

With a function to create the files in hand, you can then write a function `create_level_files` that creates files on one level. This should take (at least) three arguments:

1. `start_node`: the node at which you start splitting (for wallet creation this is the root node),
2. `level_delta`: the number L of levels below your start node on which you want to create your Sato-Servers (for wallet creation this is 30),
3. `number_of_files`: the number of Sato-Servers to create on the given level (for wallet creation $N + 1$).

The idea here is, after creating the Sato-Server file for the root node, you then only create Sato-Server files on the levels where you need them. Assuming that your branch is implemented as a binary tree,[10] what `create_level_files` would do is to split the root node to the left L times to reach the left-most node on level $13 + L$ (precisely how depth-first search would navigate down L levels).

At this point, you create a Sato-Server on that node by calling `create_sato_server` with the appropriate arguments, including the binary string address of the node. You would then use the breadth-first search algorithm to move to the next node to the right on the same level and call `create_sato_server` there, performing this shift-to-the-right a total of N times on that level.

3.4 Tilling: Making Your Sato-Servers Available

Once you have created all of the Sato-Server files you require for the initial release, you then need to make them available on a network. This can be done in a variety of ways, including using on-premise server banks, cloud service providers, a CDN service, or in a decentralized manner using IPFS. The simplest way to accomplish this, however, is to use a **tilling service provider**.

A tilling service provider, in return for a fee, will make each Sato-Server file available to anyone on the web to view. Similar to a CDN, you may choose to have many tillers. In general, tillers will compete to have fast, accurate and up to date information for each file.

Changes of state are handled in an exceptionally easy way, by simply making the appropriate changes to the Sato-Server files and signing them with the appropriate private key(s). This provides users with easily verifiable assurance of the authenticity of the state change. Then, whenever you want to provide users with this updated information, you can simply inform the

[10] Note that this need not be the case. For example, it could be possible have a tilling service (described below) responsible for maintaining the tree state and create, read, update and delete files directly through an API. In this case you would simply make requests to the tiller for the Sato-Servers you want to create.

tillers of the change.

The typical implementation will involve first developing a file management system for your branch of the Earth^{64} tree and then making those files available for tilling. You may choose to use a tilling service or to do the tilling yourself. Once a tilling ecosystem has developed, however, another possibility emerges, that of using the tilling services to manage the files for you via a REST API, for example using JSON and HTTPS, and issuing requests to the tiller for CRUD operations. Using this approach you would not need to create the files first and then till them, you could do both at once by relying on the tilling service for file management.

In any case, tilling will require the specification of a REST API for Earth^{64}, which is currently a work in progress.

3.5 Creating Secure Wallets

Securing your wallets and the NFTs they contain requires the establishment of a key management system.[11] For our scenario we will set up this system based on the Hierarchical Deterministic wallet design of BIP32, and loosely based on its refinement BIP44, relying on elliptic curve secp256k1.[12]

BIP32 provides a method whereby one can build a tree of wallets or accounts with an unbounded number of nodes, such that the (public and private) keys for all of them can be deterministically recovered from a single secret seed S, or in our case a root key m for reasons explained below.[13] A Python implementation of this BIP is available as `bip32`.

We consider here a way of implementing this in an Earth^{64} bundle using a structure based loosely on the BIP44 wallet structure (of course, BIP44 could be followed strictly if it was deemed desirable to do so).

The procedure begins optimally with the generation of a truly random 256 bit seed S or 512 bit root key m, which should be produced on a hardware security module (HSM) so that no human or network ever has access to it.[14] The reason for this is that we are managing an enterprise-scale system of value, consisting of all of the NFTs managed by the system, which requires a very high security system where information access is need-to-know, and no human needs to know the key to be able to generate wallet, account and address keys from it.[15] There should also not

[11] Note that this guide is for illustration purposes only, and that a real implementation would require the definition of appropriate policies and procedures for key management according to a defined standard, such as one based on NIST Special Publication 800-57. Please refer to our companion document [forthcoming, link to be provided] for details on our recommendations for key management.

[12] For other curves, SLIP10 extends the BIP32 protocol to curves NIST256P1 and Ed25519, but using this protocol, the derivation of public keys from an extended public key is not supported for curve Ed25519. It is, however, possible to restore this capability for Ed25519 (and other curves with a non-linear key space, such as Ed448) using a different protocol described in BIP32-Ed25519.

[13] The key m is often referred to as a 'master' key, but our preferred term to refer to the same object is 'root key', the root of the tree of keys, a term we use throughout.

[14] BIP32 allows for the generation of the root key from a 128 bit to 256 bit seed, but a truly random 512 bit root key will have higher entropy than one generated from an up to 256 bit seed.

[15] If the signing key used to purchase the bundle on Algorand or Ethereum was not generated in this way, ownership of the assets should be formally transferred to a key that was so generated before a high-value system such as this goes live.

be a single point of failure, so there should be a secure backup of S or m once it has been generated.

Once the seed S or root key m has been generated, it will need to be stored in a secure location, treated with a similar level of security as a bank's root payment card key, protected from physical and electromagnetic access. The root key itself will need to be accessible only when it is being used to generate wallet keys,[16] should never touch a networked machine, and any intermediate keys produced that are not needed in production should be destroyed after use to be regenerated from the seed whenever needed.

Using this root key, the secret keys for each of the 1.1 billion wallets, or any desired subset of these, can be generated (using the so-called hardened method in BIP32). The wallet keys would be generated with the hardened method (requiring access to the root key m) so that only you can generate or recover the public keys of the wallets.

We will continue the relative addressing example from the previous section and adopt a key generation path notation similar to BIP44. Thus, for the Earth^{64} node with address 12345/30:Y, we will denote its wallet derivation path by m/Y', where m is the root key and the apostrophe indicates that key generation uses the hardened method for wallet Y.[17] In this way the 1.1 billion wallet public-private key pairs can be recovered when needed.[18]

In terms of a potential implementation approach, the `bip32` Python package allows you to generate keys from a path defined by a list of tuples. For example, the wallet path for m/Y' would simply be `PATH = [(Y, True)]`, where `True` indicates that the hardened method is being used (`False` indicates the non-hardened method is being used). There is no need to include m because the path proceeds from the root key `root_xpriv` and the public key for m/Y' is generated with the method call `root_xpriv.derive_path(PATH)`.

It may be desirable to add another layer of keys for each wallet, to create a separation of the core wallet keys, and a particular iteration of the wallet. This would ensure that if the active private key for the wallet is compromised, then a new active key can be generated and used instead (similar to the generation of new payment cards when one is lost or stolen). For simplicity, we will ignore this nicety here.

[16] An astute reader will notice that in this scenario the root key is also needed to sign all of the Sato-Servers of the branch. As this is usually inconsistent with good security practices for reasons already explained, most implementations will want to delegate node signing privileges to other signing keys, to reduce the consequences of signing key exposure. For simplicity, however, the delegation process is not covered in this document, as was indicated above

[17] There is a subtle issue with this system because the way that BIP32 works, not all indices Y will yield valid keys. To keep track of this, then, it will be necessary to construct a map from the Earth^{64} relative wallet address to the BIP32 index used to create the keys. This can be accomplished efficiently by simply keeping track of the indices that do not yield valid keys, as this will indicate when the BIP32 index needs to be incremented relative to the current Earth^{64} relative wallet address.

[18] Note that in practice at this scale it is usually better to generate wallet keys from a set of intermediate keys rather than directly from the root key. This allows geographical distribution of wallet generation capability and can yield an increase in security because leakage of intermediate keys has less severe consequences than leakage of the root key. On the other hand, the presence of a large number of intermediate keys in persistent storage presents a much larger attack surface, so as always there are trade-offs between security and ease of use. What is best in your case will depend on the requirements of your implementation, but again it is important to follow industry best practices for key management.

From here an additional layer of accounts could be created that correspond to the different types of NFTs that a wallet could contain. These accounts could be generated with the wallet if the types are defined by the system, or by the wallet app if they are defined dynamically or by the user. We will assume that the accounts are created at the same time as the wallet, and have the derivation path $m/Y'/A'$, where A is a code for the account type.

The BIP32 path for each account would then simply be `PATH = [(Y, True), (A, True)]`, and the private keys are generated in the same way as before.

At this point the company can mint NFTs by including in the NFT data whatever information meets the specification of the company's design and then signing the NFT data with some high-security private key with an exposed public key to prove authenticity. The minted NFT can then be transferred to one of the NFT containers that correspond to the wallet with Earth^{64} address 12345/30:Y (the NFT container addresses are then 12345/30:Y/20:Z), by signing the body of the NFT container with the key for the wallet account $m/Y'/A'$. This action can be performed by the company, say Walmart, whenever it wishes to assign NFTs to a particular account.

For clarity, in terms of the implementation discussed above, the NFT would be included as an item in the `body_content_dict` that holds the content of the `sato_server_body`. A string representation of the `body_content_dict` would then be signed by the signing key of the wallet account $m/Y'/A'$, indicating that all the NFTs it contains, which are the NFTs contained in that Sato-Server, are held in the wallet account $m/Y'/A'$. They cannot be moved directly by the account holder, however, because doing so would require changing the Sato-Server, requiring an increment of the counter and a fresh file signature. Thus, this is a case of custodial assets being held for account holders by the company.

3.6 Moving and Redeeming Assets

Because ownership of Earth^{64} assets is retained fully by the company, the user will not be able to move assets out of the NFT containers without the company initiating the transfer (because the company must sign the Sato-Server file contents). The NFTs can only be owned by the user if the NFTs themselves can be transferred to a wallet owned by the user, but again the user would need the company's permission to move them. Assuming conditions of trust, this is from the user's perspective simply having a trusted party hold your assets for you.

If the company wishes to grant the user with the right to securely request transfer of ownership of NFTs assigned to a wallet, the extended private keys for the accounts in the wallet could be distributed to the user (securely using the wallet app)[19] to allow the user to sign the requests. The user may then request that the transferred NFTs be moved to a location owned by the recipient of the NFTs, which can be handled securely by having the user sign the request with the wallet's private key, which the company can easily verify because they have the public keys for the wallet.

With this system of management, the user's account is easily and fully recoverable in case the user's keys become lost or stolen, because the account relies on keys in the BIP32 derivation path. Assuming the NFTs have serial numbers and a database of these is maintained by the company

[19] Note that transmitting and storing keys securely requires different encryption schemes to be set up to protect key material in transit and at rest, beyond the scheme used for digital signatures.

(possibly as older versions of Sato-Server files), the company could invalidate any stolen NFTs and issue new ones to restore the assets of the user by creating a fresh BIP32 wallet, or simply issue the next iteration of the wallet if the core wallet keys were separated from its instances, and placing newly minted NFTs in the new wallet.

For privacy and security reasons, transactions with NFTs as described above can use a new public key for each disbursement or request, using the non-hardened version of BIP32 key generation protocol. This would allow a unique address to be used when disbursing NFTs to a user account, and allow users to sign requests using a unique address each time.

A Notational Systems for Earth^{64} Node Addresses

There are different ways to denote Earth^{64} addresses that can be useful for different purposes. This appendix presents different notational systems that can be useful for written communication and that will have simple implementations in code.

In particular we present notation that allows one to identify arbitrary nodes on the Earth^{64} binary tree as well as several modified notations to identify specifically those nodes that you can develop on after burning the representation on Ethereum, Algorand and/or other chains, such as Cardano.

A.1 Absolute Numbering for All Nodes

As a result of the address generating algorithm described above, where a `1` or `0` is appended to the parent address to generate child node addresses, each node in the tree is numbered in sequence from left to right from one level to the next (just as in search order in breadth-first search). Level 0 contains only the root node number `1`, level 1 contains the nodes `10` and `11`, level 2 the nodes `100`, `101`, `110`, `111`, and so on (see Figures 2.1 and 2.2 above).

You will then notice that we can refer to each node of the tree using its binary value, or its corresponding integer. For example, node `101` is also node 5, and its child nodes `1010` and `1011` are nodes 10 and 11.

In this way each node of the Earth^{64} tree receives a unique binary number address, which is equal to one of the numbers from 1 to $2^{64} - 1$ in decimal, since there are $2^{64} - 1$ nodes in the 64-level Earth^{64} tree.

A.2 Relative Numbering for Nodes that You Own

Ownership of any node, in particular your root node, gives you the ability to develop on all of its descendants, all the way down to the leaves. Consequently, it is useful to have a convenient way of addressing the child nodes you control.

You will notice from the previous example that the child nodes of node 5 are nodes 10 and 11 (not 6 and 7), which shows that the child node addresses are not obtained by simply adding 1 to the parent node address. It can therefore be helpful to address the descendant nodes of your root node in counting order, which can be accomplished with relative numbering.

Now suppose, as above, that the binary address of your level 13 root node is `11000000111001` (node 12345 in decimal). To count the descendant nodes by increasing by 1, we use the notation `11000000111001`/Y, where Y is the counting number of the descendant node. In the context of relative addressing we will refer to the address on the left of the '/' sign (here your root node) as the **origin node**, because it is the origin from which the descendants are being counted in order.

The binary value of Y is determined in the same way as the absolute addressing algorithm, except that rather than the root of the Earth^{64} tree being node `1`, the origin node counts as node `1`. In this way, the numbers Y then increase in the counting sequence from 1 up to the total number $(2^{51} - 1)$ of Sato-Servers contained in your bundle.

Note that there is a simple algorithm to convert from the relative address to the absolute Earth^{64} tree node address. For example, consider the node `11000000111001/110111100`. Since the decimal value of `110111100` is 444, this is the address of the 443$^{\text{rd}}$ descendant node of your root node (the 444$^{\text{th}}$ node of the subtree including the root node). To get the full absolute number of this node we remove the leading `1` from Y = `110111100` and append the resulting binary string (i.e., `10111100`) to X = `11000000111001`, the address of the origin, to obtain `1100000011100110111100` as the Earth^{64} node address of your 443$^{\text{rd}}$ descendant node, which is the 3,160,508$^{\text{th}}$ node of the Earth^{64} tree.

A.3 Compressing Binary Addresses

The binary addresses are important because they correspond to the splitting of nodes (required for rapid lookup) and the geographical splitting of area on the planet's surface. But long binary numbers are not always the most convenient to deal with, so it can be helpful to use any other convenient base to produce a notation that is easier to work with. Using the previous example, the 443$^{\text{rd}}$ descendant node of the 12,345$^{\text{th}}$ node of the Earth^{64} tree, can be addressed by 12345/444 in decimal to become more meaningful, or `3039/1bc` in hex to become more compact.[20]

A.4 Relative Numbering for Nodes on a Specific Level

Using the decimal notation it is easy to express the relative addresses of different levels of nodes. Consider again that your root node address is `11000000111001` at level 13, 12345 in decimal. In the relative notation this node is 12345/1. Then the first two child nodes can be denoted 12345/2 on the left and 12345/3 on the right. Then on the next level, the nodes are addressed from left to right as 12345/4-7. This pattern continues, such that the relative addresses of the nodes on the level L levels below your root node, are addressed in numerical order from $12345/2^L$ on the far left to $12345/(2^{L+1}-1)$ on the far right.

This observation leads to another kind of relative addressing that can be useful, namely one that gives all of the nodes on a specific level in counting order starting from 0. If L is the level number relative to your root node (i.e., treating the root node as level 0), then we we can denote the nodes of your branch on that level more conveniently by replacing the notation X/Y with the notation $X/L{:}Z$. The number Z here is obtained from Y by subtracting 2^L, which leaves a number Z between 0 and 2^L-1 that picks out the address of each node on level L in order from left to right. This notation can be useful when developing wallets or NFTs that are all on the same level of the Earth^{64} tree, which is precisely why we employed this notation in Section 3.

Note that in the binary representation of Y the conversion from Y to Z involves only stripping the leading one (and removing leading zeros), or flipping a single bit.

A.5 Iterated Relative Numbering

One final sort of relative numbering that can be useful is when we iterate the method of relative numbering. This can be useful when you use your bundle to implement iterated containers with large numbers of containers on the same level, as in the above case of many wallets on level 43 containing many NFT containers on level 63.

[20] Note that we omit the prefix '`0x`' for hex numbers because it is clear from the context whether binary or hex is being used in `teletype` font. This expression could be written `0x3039/1bc` if there is any chance of ambiguity.

In this case, we have a numbering scheme of the form $X_1/X_2/\ldots/X_n$, where X_1 is your root node, X_2 is the address of one of your top-level containers relative to X_1, and so on, to X_n, the address of one of your most nested containers relative to $X_1/\ldots/X_{n-1}$.

Consider the case of a particular wallet, a Sato-Server on level $13+30=43$. With $X=12345$ as your level 13 root node as before, suppose the wallet address is 12345/1073809714. This is then the same address as 12345/30:67890, indicating that this is the address of wallet number 67,890, the 67,891$^{\text{st}}$ wallet on that level. We recover the relative address of the wallet simply by adding 2^{30} to 67890 to obtain 1073809714.

Now since this wallet can contain many NFT containers on level $43+20=63$, treating node 12345/1073809714 as the new origin node (node 13255342885170 on the Earth^{64} tree), we can number its descendant nodes using the same relative numbering methods already described.

The level 63 NFT container addresses then lie in the range from 12345/1073809714/1048576 up to 12345/1073809714/2097151 (since $2^{20}=1048576$ and there are 2^{20} containers in wallet number 1073809714). In the level-relative notation this is then identical to the range from 12345/30:67890/20:0 to 12345/30:67890/20:1048575.

The level-relative notation has the advantage of clarifying the numbers of both the wallet (67890) and the NFT containers (0 to 1048575), while still allowing the absolute and full relative addresses to be easily recovered.

A.6 More on Address Compression

Hex can be useful for notational compression here, since it is easy for both human and machine to recover the binary values from hex values. In hex, the wallet addresses in this example are in the range `3039/1e:0` to `3039/1e:3fffffff`. Considering wallet 12345/30:67890 in particular, which is `3039/1e:10932` in hex, its NFT container addresses are in the range `3039/1e:10932/14:0` to `3039/1f:10932/14:fffff`, since 20 is `14` in hex.

Of course, if it is simply machine compression that is desired, other bases, such as Base58 and Base64, could be used.

B Simple Mappings: Binary ↔ Lot-Space ↔ GCS

B.1 Forward Mapping: Binary → Lot-Space → GCS

It turns out that there is a simple correspondence between the (non-leading) bits of the binary address of a node and the location of the upper-left corner of the corresponding lot-space region. This is a consequence of the fact that the address of a child node appends a single bit to the end of the parent's address, and the geographical splitting that corresponds to a move from parent to child alternates between by-latitude and by-longitude.

Recall that the leading bit of an Earth^{64} address indicates the root node of the Earth^{64} tree, and each subsequent bit indicates a search path following either the left (for `0`) or right (for `1`) child. Since the first split is a split by latitude, the second bit in the address indicates which half (left or right) of lot-space the corresponding region lies. Similarly, the third split, also by latitude, is the fourth bit in the address, which indicates which half (left or right) of the lot-space region of the parent. So all of the even bits indicate by-latitude splits, with `0` representing the left half and `1` the right half.

Since the second split is a split by longitude, the third bit in the address indicates which half (upper or lower) of the lot-space region of the parent corresponds to that node. Similarly, the fourth split, also by longitude, is the fifth bit in the address, which indicates which half (upper or lower) of the lot-space region of the parent. So all of the odd bits (except the leading `1`) indicate by-longitude splits, with `0` representing the upper half and `1` the lower half.

Since the width (for latitude splits) and height (for longitude splits) divides by 2 each time there is a split, and the width and height of lot-space are both 2^{32} (a power of 2), the length (width or height) of an edge of a node region is always a power of 2. This follows because the latitude is split up to a maxiumum of 32 times, and longitude is split up to a maximum of 31 times.

Recall that for node `1` the coordinate of the upper-left corner of the corresponding lot-space region is $(0, 0)$. For each split the bit added to the address determines where (and whether) the upper-left corner point moves in the transition to the lot-space region of the child.

If the bit added is `0`, the upper-left corner of the child is the same as that of the parent. If the bit added is `1`, however, then the X coordinate (latitude) increases for splits from an even level parent to an odd level child (level shifts $0 \to 1$, $2 \to 3$, $\ldots$, $62 \to 63$) and the Y coordinate (longitude) increases for splits from an odd level parent to an even level child (level shifts $1 \to 2$, $3 \to 4$, $\ldots$, $61 \to 62$). The size of the coordinate increase starts from 2^{31} for the level $0 \to 1$ and level $1 \to 2$ splits, and decreases by a factor of 2 for each subsequent pair of levels.

These observations lead to the following result. Let $A = 1b_2b_3 \ldots b_\ell$, $b_i \in \{\mathtt{0}, \mathtt{1}\}$, be the binary value of a node address, with $\ell - 1$ being the level number of the node. Let $E = b_2b_4 \ldots b_{2j}$ be the the even bits, and $O = b_3b_5 \ldots b_{2k+1}$ be the odd bits. Then the coordinates of the upper-left corner of the lot-space region corresponding to A are

$$(X_{\text{ul}}, Y_{\text{ul}})_A = (E << (32 - j), O << (32 - k)) \,,$$

where $m << n$ indicates a left bit-shift operation on m by n bits and if either E or O is empty, then j or k, respectively, is 0 and the respective coordinate remains 0. Similarly, the width ΔX

and height ΔY of the region are given by

$$(\Delta X, \Delta Y)_A = (1 << (32 - j), 1 << (32 - k)),$$

with the same conditions on j and k when E or O is empty.

Using these formulas it is straightforward to compute the remaining three corners of the lot-space region corresponding to the node with address A. Then, the GCS coordinates are easily computed using equation (2.1).

B.2 Reverse Mapping: Binary ← Lot-Space ← GCS

The reverse mapping, from GCS to lot-space to binary, is also easy to compute. The GCS coordinates of the corners of the geographical region corresponding to a node are all specified exactly as decimal numbers with seven decimal places. Multiplying these numbers by 10^7 and adding 90 (latitude) or 180 (longitude) then yields a pair of integers specifying the exact integer lot-space coordinates (X, Y).

In particular, the lot-space coordinates of P_{ul} and P_{lr}, the points at the upper-left and lower-right corners, can be determined in this way. The width ΔX and height ΔY of the region can then be computed from the difference

$$(\Delta X, \Delta Y) = P_{\text{lr}} - P_{\text{ul}}.$$

The binary address of the corresponding node can then be determined from the upper-left corner lot-space point $P_{\text{ul}} = (X_{\text{ul}}, Y_{\text{ul}})$ and the width and height $(\Delta X, \Delta Y)$ of the region in the following way.

The lengths j and k of the even and odd bit strings, respectively, can be obtained from the following equation

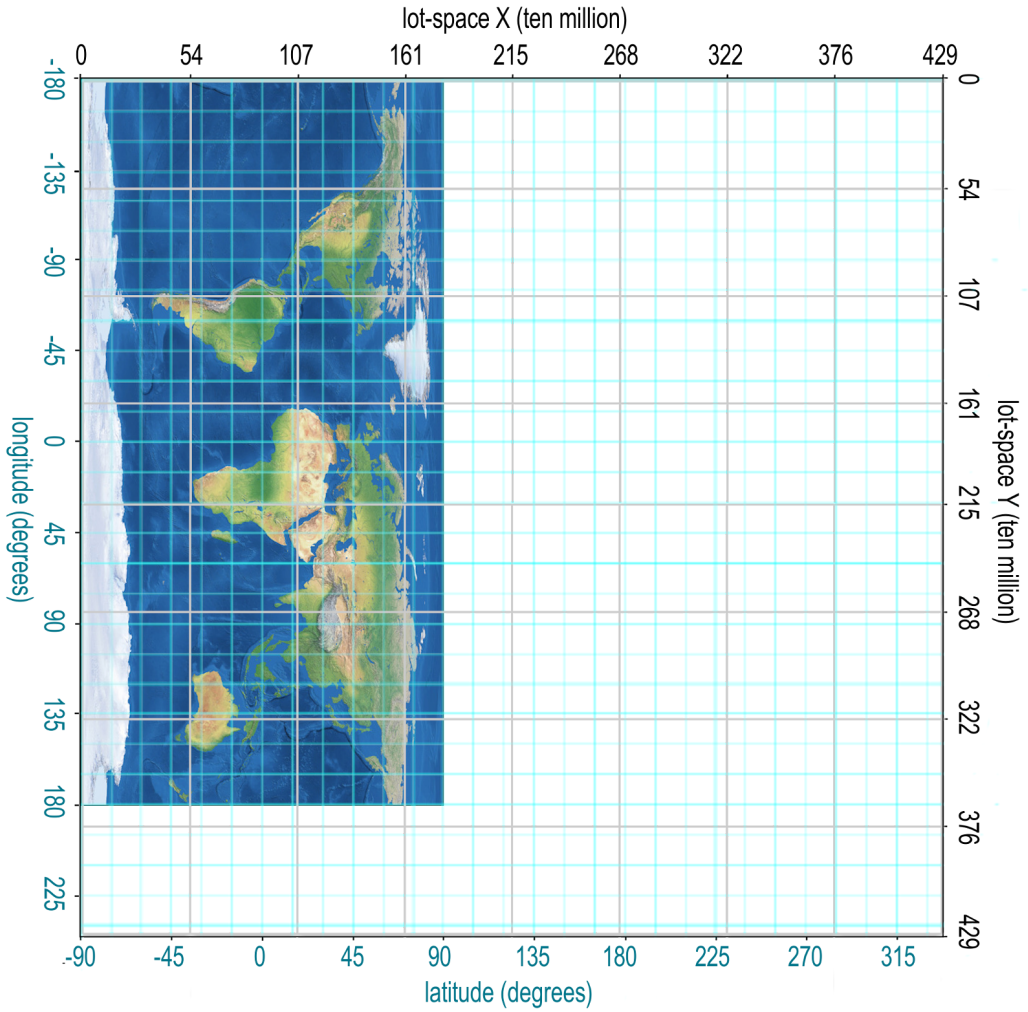
$$j = 32 - \lg(\Delta \text{X}),$$
$$k = 32 - \lg(\Delta \text{Y}),$$

where lg indicates logarithm base 2. The even $E = b_2 b_4 \dots b_{2j}$ and odd $O = b_3 b_5 \dots b_{2k+1}$ bit strings are then recovered using

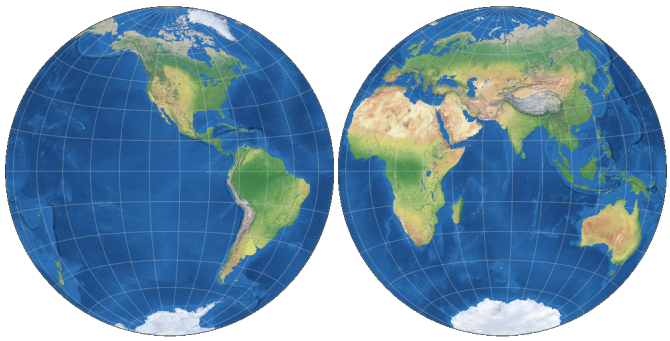
$$(E, O) = (X_{\text{ul}} >> (32 - j), Y_{\text{ul}} >> (32 - k)),$$

where $>>$ is the (logical) right bit-shift operator. The binary node address $A = 1 b_2 b_3 \dots b_\ell$ is then recovered from these bit strings.

C Lot-Space-GCS Correspondence Plot



(a) Lot-space and GCS coordinates on the plate carrée projection.



(b) Lambert azimuthal equal-area projection

Figure C.1: (a) A dual plot of lot-space coordinates and GCS coordinates. The lot-space coordinates are in units of ten million (10^7) so that the value corresponds to a latitude or longitude (tick values have been rounded to the nearest integer). (b) A quasi-spherical projection for comparison. The CC BY-SA 4.0 licensed plate carrée and Lambert azimuthal equal-area projections contained in the figure were obtained from https://map-projections.net.

D On-Chain Bundle Numbers and Their Binary Addresses

Bundle nodes sold as NFTs on Ethereum and Algorand are referred to as non-fungible bundles (NFBs). Each NFB has a unique number associated with it and this number corresponds to a unique binary address on the Earth^{64} BST. This section explains how an NFB number and the binary address of the corresponding genesis node can be used to recover the binary address of any bundle node. The NFB numbers for the level 13 genesis nodes and their corresponding Earth^{64} binary addresses are available upon activation of your bundle node.

Genesis nodes that have not been split have NFB numbers N-0, where N ranges from 1 to 1,701. We can drop the '-0' when it is clear that we are referring to the genesis node. Bundle nodes that have been split have NFB numbers N-M, where N is as before but M is some number greater than or equal to 2 that indicates the splitting path.

The NFB numbers of bundle nodes that have been split are numbered using the relative addressing scheme described in Section A.2 but using a slightly different notation (using a '-' rather than a '/').

To illustrate, suppose that your NFB number is 4444-9, which would be a level 16 bundle node, meaning that the level 13 genesis node 4444-0 was split three times to reach NFB 4444-9. The '4444' in 4444-9 means that the genesis node your bundle node originated from has NFB number 4444, which we suppose has binary address `11000000111001` (12345 in decimal).

The '-9' in 4444-9 indicates that the relative address of your NFB is 9, treating the genesis node as the origin. In the relative notation introduced in Appendix A.2, the bundle node address is 12345/9 in decimal. The procedure to recover the absolute binary address runs as follows. 9 in binary is `1001`, so we strip the leading `1` and append the resulting bit string to the genesis node binary address to obtain `11000000111001001` as the Earth^{64} address of your level 16 bundle node.