VIBE (Blockflix) <u>a "Metaverse Network LLC" product</u>

WHITE PAPER V.1.1 Written By: Prof. Cheung WY (Producer/Distributor/Performer/MetaFi-Blockchain Specialist)



Dated: 15th December 2021

Movies and Television series has early created for more then a century. I have been in Movie production business, from creator, investment, producer, distributor, performer for more than twenty years, I was awarded for Top 10 best movie in Los Angles Golden Angle awards, also for the past five years, I invested a Crypto Exchange, decentralized wallet, with more than 4 million KYC users, ranking World's number 70 in Best Crypto Exchange platform. Thus being a expert and professional of this two cross sectors, I started to visualize the future viewing platform, both for best experience for distributor an viewers. To start, let us study the number of views rely on Television st for viewing.

TV Universe - UK, Sweden, and Germany: How people watch television today

Normally, studies of TV viewer behavior focus on a method of access, which makes it difficult to understand the entire market. In this report from nScreenMedia, we examine how viewers interact with all sources of TV content, free broadcast TV, pay TV, and online TV. The report shows how many people are using each source and all the possible combinations of sources. It explores the behavior of these viewing groups and their relationships with the TV sources used.

Data were obtained from February and March 2019 survey of TV viewers in the UK, Sweden, and Germany. Some of the information readers of the report will find include:

- A breakdown of how many viewers are using each of the major TV sources, including combinations of sources (for example, those using free and online TV)
- How much each viewing group watches and how much they pay
- How each viewing groups finds something to watch
- Device preferences of the online TV viewer
- The video buying and renting behavior of TV viewers



Free ad-supported TV is driving the increase in TV service use



User demographics for Netflix, Disney+, Tubi, and YouTube



Can the Olympics slow the acceleration in cordcutting in Q3 2021?



Two internet TV myths exposed

Making Screen Time Family Time: How kids today influence co-viewing and purchase intent in the household

At one time, it looked like the switch to online viewing might herald the death of family viewing. Early surveys of online TV viewers showed that family members were increasingly choosing to watch alone on their smart phones, tablets, and PC.

D2C is not Enough: How Pay TV Can Help Drive SVOD/AVOD Success?

Among adults U.S. TV consumers with broadband, 74% use at least one of the top three providers, and 46% use two or more. At Disney and others ramp up their marketing campaigns for there direct-to consumer (D2C) services, it will become ever harder for smaller services to break through the chatter.

It is time for SVOD and AVOD providers to start thinking outside of the D2C box to keep growing. As this paper reveals, Online TV providers of all sizes are discovering that traditional pat TV services can b great partners in helping to expand distribution and to reach potential customers.

The paper represents five key challenges faced by direct-to consumer (D2C) providers in the increasingly competitive online TV market. It also discusses how pay TV operators can provide alternative approaches to these problems. The five topics covered are:

- Accelerating growth
- Marketing in a video context
- Reducing barriers to sign up
- The emerging role of search
- Enhanced monetization opportunities

nSreenMedia spoke with executives from D2C providers to understand how they are coping with these challenges.

In order to proceed further, I would like to introduce the technology backbone of BCTV.

IPFS

The InterPlanetary File System (IPFS) is a peer-to-peer distributed file system that seeks to connect all computing devices with the same system of files. In some way, IPFS is similar to the Web, but IPFS could be seen as a single BitTorrent swarm, exchanging objects within one Git repository. In other words, IPFS provides a high throughput content-addressed block storage model, with content addressed hyper-links. This forms a generalized Merkle DAG, a data structure upon which one can build versioned file system, blockchains, and even a Permanent Web. IPFS combines a distributed hastable, an incentivized block exchange, and self-certifying name space. IPFS has no single point of failure, and nodes do not need to trust each other.

1. INTRODUCTION

There have been many attempts at constructing a global distributed file system. Some systems

have seen significant success, and others failed completely. Among the academic attempts, AFS has succeeded widely and is still in use today. Others have not attained the same success. Outside of academia, the most successful systems have been peer-to-peer file-sharing applications primarily geared toward large media (audio and video). Most notably, Napster, KaZaA, and BitTorrent deployed large file distribution systems supporting over 100 million simultaneous users. Even today, BitTorrent maintains a massive deployment where ten of millions of nodes churn daily. These applications saw greater numbers of users and files distributed than their academic file system counterparts. However, the applications were not designed as infrastructure to be built upon. While there have been successful prepossessing, no general file-system has emerged that offers global, low-latency, and decentralized distribution.

Perhaps this is because a "good enough" system for most use cases already exists: HTTP. By far, HTTP is the most successful "distributed system of files" ever deployed. Coupled with the browser, HTTP has had enormous technical and social impact. It has become the de facto way to transmit file across the internet. Yet, it fails to take advantage of dozens of brilliant file distribution techniques invented in the last fifteen years. From one perspective, evolving Web infrastructure is near-impossible, given the number of backwards compatibility constraints and the number of strong

Industry has gotten away with using HTTP this long because moving small files around is relatively cheap, even for small organizations with lots of traffic. But we are entering a new era of data distribution with new challenges: (a) hosting and distributing petabyte datasets, (b) computing on large data across organizations, (c) high-volume high definition on-demand or real-time media streams, (d) versioning and linking of massive datasets, (e) preventing accidental disappearance of important files, and more. Many of these can be boiled down to "lots of data, accessible everywhere." Pressed by critical features and bandwidth concerns, we have already given up HTTP for different data distribution protocols. The next step is making them part of the Web itself.

Orthogonal to efficient data distribution, version control systems have merged to develop important data collaboration work flows. Git, the distributed source code version control system, developed many useful ways to model and implement distributed data operations. The Git tool chain offers versatile versioning functionality that large file distribution systems severely lack. New solutions inspired by Git are emerging, such as Camlistore, a personal file storage system, and Dat a data collaboration toolchain and dataset package manager. Git has already influenced distributed filesystem design, as its content addressed Merkle DAG data model enables powerful file distribution strategies. What remains to be explored is how this data structure can be influence and design of high-throughput orientated file systems, and how it might upgrade the Web itself. This paper introduces IPFS, a novel peer-to-peer version-controlled filesystem seeking to reconcile these issues. IPFS synthesizes learning s from many past successful systems. Careful

For example, Linux distributions use BitTorrent to transmit disk images, and Blizzard, Inc. Uses it to distribute video game content. (War of Warcraft)

Parties invested in the current model. But from another perspective, new protocols have emerged and gained wide use since the emergence of HTTP. What is lacking is upgrading design: enhancing the current HTTP web, and introducing new functionality without degrading user experience.

interface-focused integration yields a system greater the sum of its parts. The central IPFS principle is modeling *all data* as part of the same Merkle DAG.

2. BACKGROUND

This section reviews important properties of successful peer-to-peer systems, which IPFS combines.

2.1 Distributed Hash Tables

Distributed Hash Tables (DHTs) are widely used to coordinate and maintain metadata about peer-to-peer systems. For example, the BitTorrent MinlineDHT tracks sets of peers part of a torrent swarm.

2.1.1 Kademila DHT

Kademlia is a popular DHT that provides:

1. Efficient lookup through massive networks: queries on average contact $[log_2(n)]$ nodes. (e.g. 20 hops for a network of 10,000,000 nodes).

2. Low coordination overhead: it optimize the number of control massages it sends to other nodes.

3. Resistance to various attacks by preferring long-lived nodes.

4. Wide usage in peer-to-peer applications, including Gnutella and BitTorrent, forming networks of over 20 million nodes.

2.1.2 Coral DSHT

While some peer-to-peer filesystems store data blocks directly in DHTs, this "wastes storage and bandwidth, as data must be stored at nodes where it is not needed". The Coral DSHT extends Kademlia in three particularly important ways:

1. Kademlia stores values in nodes whose ids are "nearest" (using XOR-distance) to the key. This does not take into account application data locality, ignores "far" nodes that may already have the data, and forces "nearest" nodes to store it, whether they need it or not. This wastes significant storage and bandwidth. Instead, Coral stores addresses to peers who can provide the data blocks.

2. Coral relaxes the DHT API from get_value(key) to get_any_values(key) (the "sloppy" in DSHT). This still works since Coral users only need a single (working) peer, not the complete list. In return, Coral can distribute only subsets of the values to the "nearest" nodes, avoiding hot-spots I overloading *all the nearest nodes* when a key becomes popular).

3. Additionally, Coral organizes a hierarchy of separate DSHTs called *clusters* depending on region and size. This enables nodes to query peers in their region first, "finding nearby data without querying distance node" and greatly reducing the latency of lookups.

2.1.3 S/Kademlia DHT

S/Kademlia extends Kademlia to protect against malicious attacks in two particularly important ways:

1. S/Kademlia provides schemes to secure **Nodeld** generation, and prevent Sybill attacks. It requires nodes to create a PKI key pair, derive their identity from it, and sign their messages to each other. One scheme includes a proof-of-work crypto puzzle to make generating Sybills expensive.

2. S/Kademlia nodes lookup values over disjoint paths, in order to ensure honest nodes can connect to each other in the presence of a large fraction of adversaries in the network. S/Kademlia achieves a success rate of 0.85 even with an adversarial fraction as large as half of the nodes.

2.2 Block Exchanges-BitTorrent

BitTorrent is widely successful peer-to-peer filesharing system, which succeeds in coordinating networks of untrusting peers (swarms) to cooperate in distributing pieces of files to each other. Key features from BitTorrent and its ecosystem that inform IPFS design include:

1. BitTorrent's data exchange protocol uses a quasi tit-for-tat strategy that rewards nodes who contribute to each other, and punishes nodes who only leech others' resources.

2. BitTorrent peers track the availability of file pieces, prioritizing sending rarest pieces first. This takes load off seeds, making non-seed peers capable of trading with each other.

3. BitTorrent's standard tit-for-tat i vulnerable to some exploitative bandwidth sharing strategies. PropShare is a different peer bandwidth allocation strategy that better resists exploitative strategies, and improves the performance of swarms.

2.3 Version Control Systems-Git

Version Control Systems provide facilitate to model files changing over time and distribute different versions efficiently. The popular version control system Git provides a powerful Merkle DAG₈ object model that captures changes to a filesystem tree in a distributed-friendly way.

1. Immutable objects represent File (blob), Directories (tree), and Changes (commit).

2. Objects are content-addressed, by the cryptographic hash of their contents.

3. Links to other objects are embedded, forming a Merkle DAG. This provides many useful integrity and work-flow properties.

4. Most versioning metadata (branches, tags, etc.) are simply pointer references, and thus inexpensive to create and update.

5. Version changes only update references or add objects.

6. Distributing version changes to other users is simply transferring objects and updating remote

references.

2.4 Self-Certified Filesystems-SFS

SFS proposed compelling implementations of both distributed trust chains, and (b) egalitarian shared global namespaces. SFS introduced a technique for building *Self-Certified Filesystems*: addressing remote filesystems using the following scheme

/sfs/<Location>:<HostID>

where Location is the server network address, and :

```
HostID = hash(public_key | | Location)
```

Thus the *name* of an SFS file system certifies its server. The user can verify the public key offered by the server, negotiate a shared secret, and secure all traffic. All SFS instances share a global namespace where name allocation <u>is cryptographic, not gated by</u> any centralized body. ₂Merkle Directed Acyclic Graph-similar but more general construction than a Markle Tree. Deduplicated, does not need to be balanced, and non-leaf nodes contain data.

3. IPFS DESIGN

IPFS is a distributed file system which synthesizes successful ideas from previous peer-to-peer systems, including DHTs, BitTorrent, Git, and SFS. The contribution of IPFS is simplifying, evolving, and connecting proven techniques into a single cohesive system, greater than the sum of its parts. IPFS presents a new platform for writing and deploying applications, and a new system for distributing and versioning large data. IPFS could even evolve the web itself. IPFS is peer-to-peer; no nodes are privileged. IPFS nodes store IPFS objects in local storage. Nodes connect to each other transfer objects. These objects represent files and other data structures. The IPFS Protocol is divided into a stack of sub-protocols responsible for different functionality:

1. Identities-manage node identity generation and verification. Described in Section 3.1.

2. **Network**-manages connections to other peers, uses various underlying network protocols. Configurable. Describe in Section 3.2.

3. **Routing**-maintains information to locate specific peers and objects. Responds to both local and remote queries. Defaults to a DHT, but is swappable. Described in Section 3.3.

4. **Exchange**-a novel block exchange protocol (BitSwap) that governs efficient block distribution. Modeled as a market, weakly incentivizes data replication. Trade Strategies swappable. Described in Section 3.5.

5. **Objects**-a Merkle DAG of content-addressed immutable objects with the links. Used to represent arbitrary datastructures, e.g. file hierarchies and communication systems. Described in Section 3.5.

6. Files-versioned file system hierarchy inspired by Git. Described in Section 3.6.

7. Naming-A self-certifying mutable name system. Described in Section 3.7.

These subsystems are not independent; they are integrated and leverage blended properties. However, it is useful to describe them separately, building the protocol stack from the bottom up. Notation: data structures an functions below are specified in Go syntax.

3.1 Identities

Nodes are identified by Nodeld, the cryptographic $hash_3$ of a public-key, created with S/Kademlia's static crypto puzzle. Nodes store their public and private keys Encrypted with a passphrase). Uers are free to instantiate a "new" node identity on every launch, though that loses accrued network benefits. Nodes are incentivized to remain the same.

type Nodeld Multihash type Multihash [] byte // self-describing cryptographic hash digest

type PublicKey [] byte

3 Throughout this document, hash and checksum refer specifically to cryptographic hash checksums or data.

type PrivateKey [] byte // self-describe keys

type Node struct { Nodeld NodelD PubKey PublicKey PriKey PrivateKey }

S/Kademlia based IPFS identity generation:

```
difficulty = <integer parameter>
n = Node{ }
do {
    n.PubKey, n.PrivKey = PKI.genKeyPair()
    n.Nodeld = hash(n.PubKey)
    p = count_preceding_zero_bits(hash(n.Nodeld))
} while (p < difficulty)</pre>
```

Upon first connecting, peers exchange public keys, and check: hash(other.PublicKey) equals other.Nodeld. If not, the connection is terminated.

Note on Cryptographic Functions.

Rather than locking the system to a particular set of function choices, IPFS favors self-describing values. Hash digest values are stored in multihash format, which includes a short header specifying the hash function used, and the digest length in bytes. Example:

<function code><digest length><digest bytes>

This allow the system to (a) choose the best function for the use case (e.g. stronger security vs faster performance), and (b) evolve as function choices change. Self-describing values allow using different parameter choices compatibly.

3.2 Network

IPFS nodes communicate regularity with hundreds of other nodes in the network, potentially across the wide internet. The IPFS network stack features:

- **Transport**: IPFS can use any transport protocol, and is best suited for WebRTC DataChannels (for browser connectivity) or UTP(LEDBAT).
- **Reliability**: IPFS also uses the ICE NAT traversal techniques.
- Integrity: optionally checks integrity of messages using a hash checksum.
- Authenticity: optionally checks authenticity of messages using HMAC with sender's public key.

3.2.1 Note on Peer Addressing

IPFS can use any network; it does not rely on or assume access to IP. This allows IPFS to be used in overlay networks. IPFS stores addresses as **multiaddr** formatted byte strings for underlying network to use. **multiaddr** provides a way to express addresses and their protocols, including support for encapsulation. For example:

an SCTP/IPv4 connection /ip4/10.20.30.40/sctp/1234/

an SCTP/IPv4 connection proxied over TCP/IPv4/ip4/5.6.7.8/tcp/5678/ip4/1.2.3. 4/sctp/1234/

3.3 Routing

IPFS nodes requires a routing system that can find (a) other peers' network addresses and (b) peers who can serve particular objects. IPFS achieves this using a DSHT based on S/Kademlia and Coral, using the properties discussed in 2.1. The size of objects and use patterns of IPFS are similar to Coral and Mainline, so the IPFS DHT makes a distinction for values stored based on their size. Small values (equal to or less than 1KB) are stored directly on the DHT. For values larger, the DHT stores references, which are the **Nodelds** of peers who can serve the block. The interface of this DSHT is the following:

type IPFSRouting interface {

FindPeer(node Nodeld) // gets a particular peer's network address

SetValue(key []bytes, value []bytes) // stores a small metadata value in DHT

GetValue(key []bytes) // retrieves small metadata value from DHT

ProvideValue(key Multihash) // announces this node can serve a large value

FindValuePeers(key Multihash, min int) // gets a number serving a large values}

Note: different use cases will call for substantially different routing systems (e.g. DHT in wide network, static HT in local network). Thus the IPFS routing system can be swapped for one that fits users' needs. As long as the interface above is met, the rest of the system will continue to function.

3.4 Block Exchange - BitSwap Protocol

In IPFS, data distribution happens by exchanging blocks with peers using BitTorrent inspired protocol: BitSwap. Like BitTorrent, BitSwap peers are looking to acquire a set of blocks (want_list), and have another set of blocks to offer in exchange (have_list). Unlike BitTorrent, BitSwap is not limited to the blocks on one torrent. BitSwap operates as a persistent marketplace where node can acquire the blocks they need, regardless of what files those blocks are part of. The blocks could come from completely unrelated files in the filesystem. Nodes come together to barter in the marketplace.

While the notion of a barter system implies a virtual currency could be created, this would require a global ledger to track ownership and transfer of the currency. This can be implemented as a BitSwap Strategy, and will be explored in a future paper.

In the base case, BitSwap nodes have to provide direct value to each other in the form of blocks. This works fine when the distribution of blocks across nodes is complementary, meaning they have what the other wants. Often, this will not be the case. In some cases, nodes must **work** for their blocks. In the case that a node has nothing that its peers want (or nothing at all), it seeks the pieces its peers want, with lower priority than what the node wants itself. This incentivizes nodes to cache and disseminate rare pieces, even they are not interested in them directly.

3.4.1 BitSwap Credit

The protocol must also incentivize nodes to seed when they do not need anything in particular, as they might have the block other want. Thus, BitSwap nodes send blocks to their peers optimistically, expecting the debt to be repaid. But leeches (free-loading nodes that never share)

must be protected against. A simple credit-like system solves the problem:

1. Peers track their balance (in bytes verified) with other nodes.

2. Peers send blocks to debtor peers probabilistically, according to a function that fails as debts increases.

Note that if a node decides not to send to a peer, the node subsequently ignores the peer for an ignore_cooldown time-out. This prevents senders from typing to game the probability by just causing more dice-rolls. (Default BitSwap is 10 seconds).

3.4.2 BitSwap Strategy

The differing strategies that BitSwap peers might employ have widely different effects on the performance of the exchange as a whole. In BitTorrent, while a standard strategy is specified (tit-for-tat), a variety of others have been implemented, ranging from BitTyrant (sharing the least-possible), to BitThief (exploiting a vulnerability and never share), to PropShare (sharing proportionally). A range of strategies (good and malicious) could similarly be implemented by BitSwap peers. The choice of function, then, should aim to:

- 1. maximize the trade performance for the node, and the whole exchange
- 2. prevent freeloaders from exploiting and degrading the exchange
- 3. be effective with and resistant to other, unknown strategies
- 4. be lenient to trusted peers

The exploration of the space of such strategies is future work. One choice of function that works in practice is a sigmoid, scaled by a *debt ratio*:

Let the debt ratio *r* between a node and its peer be:

Given *r*, let the probability of sending to a debtor be:

$$P(\text{ send}/r) = 1 - 1 + exp(6 - 3r)$$

As you can see in Figure 1, this function drops off quickly as the nodes' *debt ratio* surpasses twice the established credit.



Figure 1 : Probability of Sending as r increases

The debt ratio is a measure of trust: lenient to debts between nodes that have previously exchanged lots of data successfully, and merciless to unknown, untrusted nodes. This (a) provides resistance to attackers who would create lots of new nodes (sybill attacks), (b) protects previously successful trade relationships, even if one of the nodes is temporarily unable to provide value, and (c) eventually chokes relationships that have deteriorated until they improve.

3.4.3 BitSwap Ledger

BitSwap nodes keep ledgers accounting the transfers with other nodes. This allows nodes to keep track track of history and avoid tampering. When activating a connection, BitSwap nodes exchange their ledger information. If it does not match exactly, the ledger is reinitialized from scratch, losing the accrued credit or debt. It is possible for malicious nodes to purposefully "lose" the Ledger, hoping to erase debts. It is unlikely that nodes will have accrued enough debt to warrant also losing the accrued trust; however the partner node is free to count it as misconduct, and refuse to trade.

type Ledger struct {		
owner	Nodeld	
partner	Nodeld	
bytes_sent	int	
byte_recv	int	
timestamp	Timestamp	
}		

Nodes are free to keep the ledger history, though it is not necessary for correct operation. Only the current ledger entries are useful. Nodes are also free to garbage collect ledgers as necessary, starting with the less useful ledgers: the old (peers may not exist anymore) and small.

3.4.4 BitSwap Specification

```
BitSwap nodes follow a simple protocol
```

// Additional state kept
type BitSwap struct {
 ledgers map[NodeId]Ledger
 //Ledgers known to this node, inc inactive

active map[NodeId]Peer // currently open connections to other nodes

need_list []Multihash // checksums of blocks this node needs

have_list []Multihash
// checksums of blocks this node has
}

```
type Peer struct {
```

nodeid Nodeld ledger Ledger // Ledger between the node and this peer

last_seen Timestamp
// timestamp of last received message

want_list []Multihash
// checksums of all blocks wanted by peer
// includes blocks wanted by peer's peers

}

```
// Protocol interface:
```

```
interface Peer {
```

open (nodeid :Nodeld, ledger :Ledger); send_want_list (want_list :WantList); send_block (block :Block) -> (complete : Bool); close (final :Bool);

}

Sketch of the lifetime of a peer connection:

1. Open: peers send ledgers until they agree.

- 2. Sending: peers exchange want_lists and blocks.
- 3. Close: peers deactivate a connection.

4. Ignored: (special) a peer is ignored (for the duration of a timeout) if a node's strategy avoids sending

Peer.open(Nodeld, Ledger).

When connecting, a node initializes a connection with a Ledger, either stored from a connection in the past or a new one zeroed out. Then, sends an Open message with the Ledger to the peer.

Upon receiving an Open message, a peer chooses whether to activate the connection. If - according to the receiver's Ledger - the sender is not a trusted agent (transmission below zero, or large outstanding debt) the receiver may opt to ignore the request. This should be done probabilistically with an ignore_cooldown timeout, as to allow errors to be corrected and attackers to be thwarted.

If activating the connection, the receiver initializes a Peer object with the local version of the Ledger and sets the last_seen timestamp. Then, it compares the received Ledger with its own. If they match exactly, the connections have opened. If they do not match, the peer creates a new zeroed out Ledger and sends it.

Peer.send_want_list(WantList).

While the connection is open, nodes advertise their want_list to all connected peers. This is done (a) upon opening the connection, (b) after a randomized periodic timeout, (c) after a change in the want_list and (d) after receiving a new block.

Upon receiving a want_list, a node stores it. Then, it checks whether it has any of the wanted blocks. If so, it sends them according to the BitSwap Strategy above.

Peer.send_block(Block).

Sending a block straightforward. The node simply transmits the block of data. Upon receiving all the data, the receiver computes the Multihash checksum to verify it matches the expected one, and returns confirmation.

Upon finalizing the correct transmission of a block, the receiver moves the block from need_list to have_list, and both the receiver and sender update their ledgers to reflect the additional bytes transmitted.

If a transmission verification fails, the sender is either malfunctioning or attacking the receiver. The receiver is free to refuse further trades. Note that BitSwap expects to operate on a reliable transmission channel, so transmission errors - which could lead to incorrect penalization of an honest sender - are expected to be caught before the data is given to BitSwap.

Peer.close(Bool).

The final parameter to close signals whether the intention to tear down the connection is the sender's or not. If false, the receiver may opt to re-open the connection immediately. This avoids premature closes.

A peer connection should be closed under two conditions:

- a silence_wait timeout has expired without receiving any messages from the peer (default BitSwap uses 30 seconds). The node issues Peer.close(false).
- the node is exiting and BitSwap is being shut down. In this case, the node issues Peer.close(true).

After a **close** message, both receiver and sender tear down the connection, clearing any state stored. The **Ledger** may be stored for the future, if it is useful to do so.

Notes.

Non-open messages on an inactive connection should be ignored. In case of a send_block message, the receiver may check the block to see if it is needed and correct, and if so, use it. Regardless, all such out-of-order messages trigger a close(false) message from the receiver to force re-initialization of the connection.

3.5 Object Merkle DAG

The DHT and BitSwap allows IPFS to form a massive peer-to-peer system for storing and distributing blocks quickly and robustly. On top of these, IPFS builds a Merkle DAG, a directed acyclic graph where links between objects are cryptographic hashes of the targets embedded in the sources. This is generalization of the Git data structure. Merkle DAGs provide IPFS many useful properties, including:

1. **Content Addressing:** all content is uniquely identified by its multihash checksum, **including links**.

2. **Tamper resistance:** all content is verified with its checksum. If data is tampered with or corrupted, IPFS detects it.

3. **Deduplication:** all object that hold the exact same content are equal, and only stored once. This is particularly useful with index objects, such as git trees and commits, or common portions of data.

The IPFS Object format is:

type IPFSLink struct { Name string // name or alias of this link

> Hash Multihash // cryptogrphic hash of target

Size int // total size of target

}

type IPFSObject struct { links [] IPFSLink // array of links

data []byte // opaue content data

}

The IPFS Merkle DAG is an extremely flexible way to store data. The only requirements are that object references be (a) content addressed, and (b) encoded in the format above. IPFS grants applications complete control over the data field; applications can use any custom data format they chose, which IPFS may not understand. The separate in-object link table allows IPFS to:

• List all object references in an object. Foe example:

> ipfs Is /XLZ1625Jjn7SubMDgEyeaynFu84gin2vzbXLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x 189458 less XLHBNmRQ5sJJrdMPuu48pzeyTtRo39tNDR5 19441 scriptXLF4hwVHsVu778FZK6fozf8Jj9WEURMbCX4 586 template <object multihash> <object size> <link name>

- Resolve string path lookups, such as foo/bar/baz. Given an object, IPFS resolves the first path component to a hash in the object's link table, fetches that second object, and repeats with the next component. Thus, string paths can walk the Merkle DAG no matter what the data formats are.
- Resolve all objects referenced recursively:

> ipfs refs --recursive $\$

/XLZ1625Jjn7SubMDgEyeaynFuR84ginqvzbXLLxhdgJcXzLbtsLRL1twCHA2NrURp4H38sXLYkgq61DYaQ8NhkcqyU7rLcnSa7 dSHQ16xXLBNmRQ5sJJrdMPuu48pzeyTtRo39tNDR5XLWVQDqxo9Km9zLyquoC9gAP8CL1gWnHZ7z...

A raw data field and a common link structure are the necessary components for constructing arbitrary data structures on top of IPFS. While it is easy to see how the Git object model fits on top of this DAG, consider these other potential model fits on top of this DAG, consider these other potential data structures: (a) key-value stores (b) traditional relational databases (c) Linked Data triple stores (d) linked document publishing systems (e) linked communications platforms (f) cryptocurrency blockchains. These can all be modeled on top of the IPFS Merkle DAG, which allows any of these systems to use IPFS as a transport protocol for more complex applications.

3.5.1 Paths

IPFS objects can be traversed with a string path API. Paths work as they do in traditional UNIX filesystems and the Web. The Merkle DAG links make traversing it easy. Note that full paths in IPFS are of the form:

format

/ipfs/<hash-of-object>/<name-path-to-object>

example

/ipfs/XLYkgq61DYaQ8NhkcayU7rLcnSa7dSHQ16x/f00.txt

The / ipfs prefix allows mounting into existing systems at a standard mount point without conflict (mount point names are of course configurable). The second path component (first within IPFS) is the hash of an object. This is always the case, as there is no global root. A root object would have the impossible task of handling consistency of millions of objects in a distributed (and possibly disconnected) environment. Instead, we simulate the root with content addressing. All objects are always accessible via their hash. Note this means that given three objects in path <foo>/bar/baz, the last object is accessible by all:

/ipfs/<hash-of-foo>/bar/baz /ipfs/<hash-of-bar>/baz /ipfs/<hash-of-baz>

3.5.2 Local Objects

IPFS clients require some local storage, an external system on which to store and retrieve local raw data for the objects IPFS manages. The Type of storage depends on the node's use case. In most cases, this is simply a portion of disk space (either managed by the native filesystem, by a key-value store such as leveldb, or directly by the IPFS client). In others, for example non-persistent caches, this storage is just a portion of RAM.

Ultimately, all blocks available in IPFS are in some node's local *storage*. When users request objects, they are found, downloaded, and stored locally, at least temporarily. This provides fast lookup for some configurable amount of time thereafter.

3.5.3 Object Pinning

Nodes who wish to ensure the survival of particular objects can do so by pinning the objects. This ensures the objects are kept in the node's **local storage**. Pinning can be done recursively, to pin down all linked descendant objects as well. All objects pointed to are then stored locally. This is particularly useful to persists files, including references. This also makes IPFS a Web where links are **permanent**, and Objects can ensure the survival of others they point to.

3.5.4 Publishing Objects

IPFS is globally distributed. It is designed to allow the files of millions of users to coexist together. The DHT, with files of millions of users to coexist together. The DHT, with content-hash addressing, allows publishing objects in a fair, secure, and entirely distributed way. Anyone can publish an object by simply adding its key to the DHT, adding themselves as a peer, and giving other users the object's path. Note that Objects are essentially immutable, just like in Git. New versions hash differently, and thus are new objects. Tracking versions is the job of additional versioning objects.

3.5.5 Object-level Cryptography

IPFS is equipped to handle object-level cryptographic operations. An encrypted or signed is wrapped in a special frame that allows encryption or verification of the raw bytes.

```
Type EncryptedObject struct {
Object []bytes
// raw object data encrypted
Tag []bytes
// optional tag for encryption groups
```

}

Type SignedObject struct { Object []bytes // raw object data signed

Signature []bytes // hmac signature

PublicKey []multihash // multiohash identifying key

}

Cryptographic operation change the object's hash, defining a different object. IPFS automatically verifies signatures, and can decrypt data with user-specified keychains. Links of encrypted objects are protected as well, making traversal impossible without a decryption key. It is possible to have a parent object encrypted under one key, and a child under another or not at all. This secures links to shared objects.

3.6 Files

IPFS also defines a set of objects for modeling a versioned filesystem om top of the Merkle DAG. This object is similar to Git's:

- 1. block: a variable-size block of data.
- 2. list: a collection of blocks or other lists.
- 3. tree: a collection of blocks, lists, or other trees.
- 4. commit: a snapshot in the version history of a tree.

I hoped to use Git object formats exactly, but had to depart to introduce certain features useful in a distributed filesystem, namely (a) fast size lookups (aggregate byte size have been added to objects), (b) large file deduplication (adding a list object), and (c) embedding of **commits** into **trees**. However, IPFS File objects are close enough to Git that conversion between the two i possible. Also, a set of Git object can be introduced to convert without losing any information (unix file permissions, etc).

Notation: File object formats below use JSON. Note that this structure is actually binary encoded using protobufs, though ipfs includes import/export to JSON.

3.6.1 File Object: blob

The blob object contains an addressable unit of data, and represents a file. IPFS Blocks are like Git blobs or filesystem data blocks. They store the users' data. Note that IPFS files can be represented by both lists and blobs. Blobs have no links.

```
{
    "data": "some data here",
    // blobs have no links
}
```

3.6.2 File Object. list

The list object represents a large or deduplicated file made up of several IPFS blobs concatenated together. Lists contain an ordered sequence of blob or list objects. In a sense, the IPFS list functions like a filesystem file with indirect blocks. Since lists can contain other lists, topologies including linked lists and balanced trees are possible. Directed graphs where the same node appears in multiple places allow in-file deduplication. Of course, cycles are not possible, as enforced by hash addressing.

```
{
  "data" : ["blob", "list", "blob"],
    // lists have an array of object types as data
  "links" : [
    { "hash" :
                "XLYkgq61DYaQ8NhkcqyU7rLcnSa7dSHQ16x",
      "size" :
                189458 }.
    { "hash" :
                "XLHBNmRQ5sJJrdMPuu48pzevTtRo39tNDR5",
      "size" :
                19441},
                "XLWVQDqxo9Km9zLyquoC9qAP8CL1gWnHZ7z",
    { "hash" :
      "size" :
                5286 }
    // lists have no names in links
  ]
}
```

3.6.3 File Object. tree

The tree object in IPFS is similar to Git's: it represents a directory, a map of names to hashes. The hashes reference blobs, lists, other trees or commits. Note that traditional path naming is already implemented by the Merkle DAG.

3.6.4 File Object : commit

The **commit** object in IPFS represents a snapshot in the version history of any object. It is similar to Git's, but can reference any type of object. It also links to author objects.

```
{
    "data": {
        "type": "tree",
        "date": "2021-08-22 16:33:08Z",
        "message": "This is a commit message."
    },
    "links": [
```



```
>ipfs file-cat <ccc111 - hash> --json
 {
   "data" : {
       "type" : "tree",
      "date": "2021-08-22 16:33:08Z",
      "message": "This is a commit message."
   },
   "links" : [
      { "hash" : "<ccc000 - hash>",
          "name" : "parent", "size" : 25309 },
      { "hash" : "<ttt111 -hash>",
          "name" : "object", "size" : 5198 }
      { "hash" : "<aaa111 - hash>",
          "name" : "author", "size" : 109 }
    ]
}
 >ipfs
         file-cat <ttt111 - hash> --json
{
  "data": ["tree", "tree", "blob"],
  "links" : [
      { "hash" : "<ttt222 - hash>",
          "name" : "ttt222 - name", "size" : 1234 },
      { "hash" : "<ttt333 - hash>",
          "name" : "ttt333 - name", "size" :
                                             3456 }.
      { "hash" : "<bbb222 - hash>",
          "name" : "bbb222 - name", "size" : 22 }
   ]
}
>ipfs file-cat <bbb222 - hash> --json
{
 "data": "blob222 data",
 "links" : []
}
        Figure 3: Sample Objects
   { "hash" : "XLa1qMBKiSEEDhojb9FFZ4tEvLf7FEQdhdU",
     "name" : "parent", "size" : 25309 },
   { "hash" : "XLGw74Ay9junbh28x7ccWov9inu1Vo7pnX",
     "name" : "object", "size" : 5198 },
```

3.6.5 Version control

The commit object represents a particular snapshots in the version history of an object. Comparing the objects (and children) of two different commits reveals the differences between two versions of the filesystem. As long as a single commit and all the children objects it references are accessible, all preceding versions are retrievable and the full history of the filesystem changes can be accessed. This falls out of the Merkle DAG object model.

The full power of the Git version control tools is available to IPFS users. The object model is compatible, though not the same. It i possible to (a) build a version of the Git tools modified to use the IPFS object graph, (b) build a mounted FUSE filesystem that mounts an IPFS tree as a Git repo, translating Git filesystem read/writes to the IPFS formats.

3.6.6 Filesystem Paths

As we saw in the Merkle DAG section, IPFS objects can be traversed with a string path API. The IPFS File Objects are designed to make mounting IPFS onto a UNIX filesystem simpler. They restrict **trees** to have no data, in order to represent them as directories. And commits can either be represented as directories or hidden from the filesystem entirely.

3.6.7 Splitting Files into Lists and Blob

One of the main challenges with versioning and distributing large files is finding the right way to split them into independent blocks. Rather than assume it can make the right decision for every type of file, IPFS offers the following alternatives:

- a. Use Rabin Fingerprints as in LBFS to pick suitable block boundaries.
- b. Use the rsync rolling-checksum algorithm, to detect blocks that have changed between versions.
- c. Allow users to specify block-splitting functions highly tuned to specific files.

3.6.8 Path Lookup Performance

Path-based access traverses the object graph. Retrieving each object requires looking up its key in the DHT, connecting to peers, and retrieving its block. This is considerable overhead, particularly when looking up paths with many components. This is mitigated by:

- **tree caching**: since all objects are hash-addressed, they can be cached indefinitely. Additionally, **trees** tend to be small in size so IPFS prioritizes caching them over blobs.
- flattened trees: for any given tree, a special flattened tree can be constructed to list all object reachable from the trees. Names in the flattened tree would really be paths parting from the original tree, with slashes.

For example, flattened tree for ttt111 above:

```
{
"data" :
  {"tree"
            "blob" ,
                       "tree",
                                 "list"
                                          "blob"
                                                  "blob"] ,
"links" : [
  { "hash" : "<ttt222-hash>",
                                "size"
                                          1234
     "name: : "ttt222-name" },
    "hash": "<bbb111-hash>", "size"
                                        •
                                            123
  {
     "name: : "ttt222-name/bbb111-name"
                                         },
    "hash": "<ttt333-hash>",
                                "size"
                                          3456
  {
     "name: : "tttt333-name" },
  { "hash" : "<III11-hash>" , "size"
                                       · 587
     "name: : "ttt333-name/lll111-name" },
   { "hash" : "<bbb222-hash>", "size" : 22
     "name: : "ttt333-name/III111-name/bbb222-name" },
   { "hash": "<bbb222-hash>", "size" : 22
     "name: : "bbb222-name" },
] }
```

3.7 IPNS: Naming and Mutable State

So far, the IPFS stack forms a peer-to-peer block exchange constructing a content-addressed DAG of objects. It serves to publish and retrieve immutable objects. It can even track the version history of these objects. However, there is a critical component missing: mutable naming. Without it, all communication of new content must happen off-band, sending IPFS links. What is required is some way to retrieve mutable state at the same path.

It is worth stating why - if mutable data is necessary in the end - we worked hard to build up an immutable Merkle DAG. Consider the properties of IPFS that fall out of the Merkle DAG: objects can be (a) retrieved via their hash, (b) integrity checked, (c) linked to others, and (d) cached indefinitely. In a sense:

Objects are **permanent**

These are the critical properties of a high-performance distributed system, where data is expensive to move across network links. Object content addressing constructs a web with (a) significant bandwidth optimization s, (b) utrusted content serving, (c) permanent links, and (d) the ability to make full permanent backups of any object and its references.

The Merkle DAG, immutable content-addressed objects, and Naming, mutable pointers to the Merkle DAG, instantiate a dichotomy present in many successful distributed systems. These include the Git Version Control System, with the distributed successor to UNIX, with its mutable Fossil and immutable Venti filesystems. LBFS also uses mutable indices and immutable chunks.

3.7.1 Self-Certified Names

Using the naming scheme from SFS gives us a way to construct self-certified names, in a cryptographically assigned global namespace, that are mutable. The IPFS scheme is as follows.

1. Recall that in IPFS:

Nodeld = hash(node . PubKey)

2. We assign every user a mutable namespace at:

/ipns/<NodeID>

3. A user can publish an Object to this path **Signed** by her private key, say at:

/ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/

4. When other users retrieve the object, they can check the signature matches the public key and NodeId. This verifies the authenticity of the Object published by the user, achieving mutable state retrieval.

Note the following details:

- The ipns (InterPlanetary Name Space) separate prefix is to establish an easily recognizable distinction between mutable and immutable paths, for both programs and human readers.
- Because this is not a content-addressed object, publishing it relies on the only mutable state distribution system in IPFS, the Routing system. The process is (1) publish the object as a regular immutable IPFS object, (2) publish its hash on the Routing system as a metadata value:

routing · setValue(NodeId, <ns-object-hash>)

• Any links in the Object published act as sub-names in the namespace:

/ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/ /ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/docs /ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm/docs/ipfs

• It is advised to publish a commit object, or some other object with a version history, so that clients may be able to find old names. This is left as a user option, as it is not always desired.

Note that when users publish this Object, it cannot be published in the same way

3.7.2 Human Friendly Names

While IPNS is indeed a way of assigning and reassigning names, it is not very user friendly, as it exposes long hash values as names, which are notoriously hard to remember. These work for URLs, but not for many kinds of offline transmission. Thus IPFS increases the user-friendliness of IPNS with the following techniques.

Peer Links.

As encouraged by SFS, users can link other users' Objects directly into their own Objects (namespace, home, etc). This has the benefit of also creating a web of trust (and supports the old Certificate Authority model):

Susan links to bob Bob lpfs link / <susan-pk-hash>/friends/bob /<bob-pk-hash>

Doris links to Susan
Ipfs link / <doris-pk-hash>/friends/susan /<susan-pk-hash>

Doris also has access to Bob/ <doris-pk-hash/friends/susan/friends/bob

#access Verisign certified domains
/ <veridign-pk-hash>/foo.com

DNS TXT IPNS Records.

If / ipns /<domain> is a valid domain mane, IPFS looks up key ipns in its DNS TXT records. IPFS interprets the value as either an object hash or another IPNS path:

this DNS TXT record lpfs . benet . ai. TXT "ipfs = XLF2ipQ4jD3U ... "

behaves as symlink In -s / ipns/XLF2ipQ4jD3U / ipns/fs . benet . ai

Proquint Pronounceable Identifiers.

There have always been schemes to encode binary into pronounceable words. IPNS supports Proquint. Thus:

this proquint phrase /ipns/dahih-dolij-sozuk-vosah-luvar-fuluh

will resolve to corresponding /ipns/KhAwNprxYVxKqpDZ

Name Shortening Services.

Services are bound to spring up that will provide name shortening as a service, offering up

their namespaces to users. This is similar to what we see today with DNS and Web URLs:

User can get a link from /ipns/shorten . er/foobar

To her own namespace /ipns/XLF2ipQ4jD3UdeX5xp1KBgeHRhemUtaA8Vm

3.8 Using IPFS

IPFS is designed to be used in a number of different ways. Here are just some of the use cases I will be pursuing:

- 1. As a mounted global filesystem, under / ipfs and /ipns.
- 2. As a mounted personal sync folder that automatically versions, publishes, and backs up any writes.
- 3. As an encrypted file or data sharing system.
- 4. As a versioned package manager for *all software*.
- 5. As the root filesystem of a Virtual Machine.
- 6. As the boot filesystem of a VM (under a hypervisor).
- 7. As a database: applications can write directly to the Merkle DAG data model and get all the versioning, caching, distribution IPFS provides.
- 8. As a linked (and encrypted) communication platform.
- 9. As an integrity checked CDN for large files (without SSL).
- 10. As an encrypted CDN.
- 11. On webpages, as a web CDN.
- 12. As a new Permanent Web where links do not die.

The IPFS implementations target:

- a. An IPFS library to import in your own applications.
- b. Commandline tools to manipulate objects directly.
- c. Mounted file systems, using FUSE or as kernel modules.

4. THE FUTURE

The ideas behind IPFS are the product of decades of successful distributed systems research in academia and open source. IPFS synthesizes many of the best ideas from the most successful systems to date. Aside from BitSwap, which is a novel protocol, the main contribution of IPFS is this coupling of systems and synthesis of designs.

IPFS i an ambitious vision of new decentralized Internet infrastructure, upon which many different kinds applications can be built. At the bare minimum, it can be used as a global, mounted, versioned filesystem and namespace, or as the next generation file sharing system. At its best, it could push the web to new horizons, where publishing valuable information does not impose hosting it on the publisher but upon those interested, where users can trust the content they receive without trusting the peers they receive it from, and where old but important files do not go missing.

IPFS looks forward to bringing us toward the Permanent Web.

5. ACKNOWLEDGEMENTS

IPFS is the synthesis of many great ideas and systems. It would be impossible to dare such ambitious goals without standing on the shoulders of such giants. Personal thanks to all the many contributing such great and brilliant ideas.

6. VIBE

VIBE will be the first in the world providing a open source to movie/drama/short reels uploaders. This decentralized applications adopts the full use of IPFS. Verisign Uploaders decide on their own the cost of sharing viewing <u>"Pay per viewlinks"</u>. links in peer-to-peer viewed will be rewarded upon receipt from uploaders.

-END-





Blockflix_精彩每一天

2022年1月1日Blockflix盛大起航,期待與你分享 精彩生活,讓每一位分享者收穫MNC!





一天

INC!

期待與你分享

精彩世界

發布視頻



我的影厅

社區



幫助中心





#亞洲男子一己之力復仇





#吸血鬼再現人間人類何去何從

#宇宙最強神龍翻身

#神龍降世,再塑神話





#亞洲男子一己之力復仇





#亞洲男子一己之力復仇

主人公強仔憑一己之力對抗強大和社會組織,看猛男如何為父報仇...

支付地址

Ac5c8aDac0A957c63d099AcEAC0b4c8aA7cF1CdA3B

這馨提示

 1.您支付至上述地址後,需要整個網絡確認,12次網絡確認 後可成為片商
 2.請確認支付金額,支付成功後不可退回

100.00 MNC











100 MNC 2021.08.28 14:56:23

交易Hash

Ac5c8aDac0A957c63d099AcEAC0b4c8aA7cF1CdA3B



www.youtube.laksjdlk.com





#亞洲男子一己之力復仇



Blockflix_精彩每一天

2022年1月1日Blockflix盛大起航,期待與你分享精彩生活, 讓每一位分享者收穫MNC!



每一位觀影者都會獲得20%來至下一位觀影者的獎勵

一位觀影者貢獻的票房會有18%放入獎池 每-

每一位觀影者都有機會成為幸運者並獲得高額獎金

Blockflix-分享即收穫

分享生活,分享收益

規則介紹

成為片商即可分享生活中的10個精彩時刻

分享精彩的同時會收到來著觀眾打賞的60%收益

每日收益即可到賬,分分鐘成為創作達人





Blockflix_精彩每一天

片商权益

1.最高可上傳10部影片

2.可享受每部影片票房的60%收益

支付地址

Ac5c8aDac0A957c63d099AcEAC0b4c8aA7cF1CdA3B

這馨提示

 1.您支付至上述地址後,需要整個網絡確認,12次網絡確認 後可成為片商
 2.請確認支付金額,支付成功後不可退回

10000.00 MNC



Blockflix-分享即收穫

八百开洋 八百册兴



Blockflix_精彩每—天



支付成功!

100 MNC 2021.08.28 14:56:23

交易Hash

Ac5c8aDac0A957c63d099AcEAC0b4c8aA7cF1CdA3B



Blockflix-分享即收穫

八百开洋 八百时关



视频标题

請輸入

视频简介

請輸入



20-40分鐘

40-60分鐘

> 60分鐘

標籤







1.為了更好的觀看體驗,請上傳優質觀影地址 2.視頻上傳後需要一些時間發布,請耐心等待





这是一句广告词

2022年1月1日Blockflix盛大起航,期待與你分享精彩生活, 讓每一位分享者收穫MNC!

觀影規則

1.每一位觀影者都會獲得20%來至下一位觀影者的獎勵

2.每一位觀影者貢獻的票房會有18%放入獎池

3.每一位觀影者都有機會成為幸運者並獲得高額獎金

Blockflix-分享即收穫

分享生活,分享收益

規則介紹

1.成為片商即可分享生活中的10個精彩時刻

2.分享精彩的同時會收到來著觀眾打賞的60%收益

3.每日收益即可到賬,分分鐘成為創作達人





獲費 收益(MNC) 銷售(MNC) 5000W 10000



#亞洲男子一己之力復仇



#宇宙最强大龙,亚洲那时的...



數據概括



#亞洲男子一己之力復仇

主人公強仔憑一己之力對抗強大和社會組織,看猛男如何為父報仇...



發佈時間: 2021.08.21 14:53:25

1000 收益(MNC)

> **1000** 累計銷售

200 今日收益(MNC)

> **200** 今日銷售



1.數據更新可能會有延遲,請耐心等待



你就是下一位導演

做自己生活的導演,與我合拍你的精彩人生.



合拍規則

每位觀眾都可參與合拍,與他人共同合拍分享收益.

影片上映可与合拍者共享票房的50%收益



發起合拍

發揮你的才能,發起一個合拍項目.

發起合拍可收穫票房的48%收益.







你就是下一位導演

發起規則

1.合拍總額將會在票房達到1000000MNC時解凍至賬戶.
 2.合拍影片的50%將會分享給參與合拍的觀眾
 48%歸發起人,2%用作社區運營.

支付地址

Ac5c8aDac0A957c63d099AcEAC0b4c8aA7cF1CdA3B

這馨提示

1.您支付至上述地址後,需要整個網絡確認,12次網絡確認 後可成為发起人

2.請確認支付50000MNC,支付成功後不可退回

500000.00 MNC



你就是下一位導演

做白口片洋的道:空的半个边边的蛙剑 / 片



你就是下—位道演



支付成功!

500000 MNC 2021.08.28 14:56:23

交易Hash

Ac5c8aDac0A957c63d099AcEAC0b4c8aA7cF1CdA3B



你就是下一位導演

做白口片洋的道:空阳北合协你的蛙蚁人片



視頻標題

請輸入

視頻簡介

請輸入



20-40分鐘

40-60分鐘

> 60分鐘

標籤





1.每次只能發布一個合拍,發布成功不可修改

2. 達到目標金額後180天需將觀影地址公佈出來





視頻標題



發起成功!

500000 MNC 2021.08.28 14:56:23





#自製 #偶像 #戰爭 #軍旅 #懸疑







#亞洲男子一己之力復仇

主人公強仔憑一己之力對抗強大和社會組織,看猛男如何為父報仇...

#自製 #偶像 #戰爭 #軍旅 #懸疑



100 上映價格(MNC)

發起時間: 2021.08.21 14:53:25



已籌到: 250500 MNC

這馨提示

1.每次只能發布一個合拍,發布成功不可修改

2. 達到目標金額後180天需將觀影地址公佈出來





視頻標題

#亞洲男子一己之力復仇

視頻簡介

主人公強仔憑一己之力對抗強大和社會組織,看猛男如何為父 報仇…



20-40分鐘 40-60分鐘

>60分鐘

標籤

#自製 #偶像 #戰爭 #軍旅 #懸疑

價格 (MNC)



請輸入



1.為了更好的觀看體驗,請上傳優質觀影地址 2.視頻上傳後需要一些時間發布,請耐心等待







#亞洲男子一己之力復仇





#神龍降世,再塑神話



#吸血鬼再現人間人類何去何從

#宇宙最強神龍翻身



#亞洲男子一己之力復仇



#亞洲男子一己之力復仇





#亞洲男子一己之力復仇

主人公強仔憑一己之力對抗強大和社會組織,看猛男如何為父報仇...

#自製 #偶像 #戰爭 #軍旅 #懸疑

20-40分钟 時長

100 上映價格(MNC)

500000

發起時間: 2021.08.21 14:53:25



已籌到: 250500 MNC

參與金額 (MNC)











#亞洲男子一己之力復仇

主人公強仔憑一己之力對抗強大和社會組織,看猛男如何為父報仇...

支付地址

Ac5c8aDac0A957c63d099AcEAC0b4c8aA7cF1CdA3B

① 温馨提示

1.您支付至上述地址後,需要整個網絡確認,12次網絡確認 後可成為发起人 2.請確認支付50000MNC,支付成功後不可退回

500000.00 MNC



JUUC









成功參與

500000 MNC 2021.08.28 14:56:23

交易Hash

Ac5c8aDac0A957c63d099AcEAC0b4c8aA7cF1CdA3B



2022.02.28前,請持續關注!





觀影收益(MNC) 合拍分紅(MNC) 中獎金額(MNC) 200 500

觀影記錄



2021.08.15



2021.08.15







2021.08.15

參與合拍



500000 MNC

2021.08.15



500000 MNC 2021.08.15



500000 MNC 2021.08.15



500000 MN(2021.08.15

中獎記錄

中獎金額	時間
3500 MNC	2021.08.15

