LBRY: A Decentralized Digital Content Marketplace

Alex Grintsvayg (grin@lbry.io), Jeremy Kauffman (jeremy@lbry.io)

Introduction

LBRY is a protocol for accessing and publishing digital content in a global, decentralized marketplace. LBRY uses a public blockchain to provide a single shared index of published content, as well as content discovery and payment.

Clients can use LBRY to publish, host, find, download, and pay for content — books, movies, music, or anything else that can be represented as a stream of bits. The protocol is permissionless and censorship-resistant, which means that participation is open to everyone and no one can unilaterally block or remove content.

LBRY is a step forward from previous generations of decentralized networks, which provide no discovery or payment mechanisms. For creators, LBRY is unparalleled in trust and earning potential. For consumers, LBRY is the first system that provides end-to-end digital content consumption that does not require trusting a third-party. For the world, LBRY is designed to engender the most complete catalog of information to ever exist, and to be controlled by the only party that could be trusted with such monumental responsibility: no one.

Assumptions

This document assumes that the reader is familiar with distributed hash tables (DHTs), the BitTorrent protocol, Bitcoin, and blockchain technology in general. It does not attempt to document these technologies or explain how they work. The Bitcoin developer reference <https://bitcoin.org/en/developer-reference> and BitTorrent protocol specification <http://www.bittorrent.org/beps/bep_0003.html> are recommended for anyone wishing to understand the technical details.

Overview

This document defines the LBRY protocol, its components, and how they fit together. LBRY consists of several discrete components that are used together in order to provide the end-to-end capabilities of the protocol. There are two distributed data stores (blockchain and DHT), a peer-to-peer protocol for exchanging data, and specifications for data structure, encoding, and retrieval.

Conventions and Terminology

blob	The unit of data transmission on the data network. A published file is split into many blobs.
stream	A set of blobs that can be reassembled into a file. Every stream has one or more content blobs which contain the published file, and a manifest blob which contains a list of the content blob hashes.
blob hash	The cryptographic hash of a blob. Hashes are used to uniquely identify blobs and to verify that the contents of the blob are correct. Unless otherwise specified, LBRY uses SHA-384 <https://en.wikipedia.org/wiki/SHA-2> as the hash function.
metadata	Information about the contents of a stream (e.g. creator, description, stream hash, etc). Metadata is stored in the blockchain.
name	A human-readable UTF8 string that is associated with a claim.
stake	An entry in the blockchain that sets aside some credits and associates them with a name.
claim	A stake that contains metadata about a stream or channel.
support	A stake that lends its credits to bolster a claim.
channel	The unit of pseudonymous publisher identity. Claims may be part of a channel.
URL	A memorable reference to a claim.

Blockchain

The LBRY blockchain is a public, proof-of-work blockchain. The design is based on the Bitcoin <https://bitcoin.org/bitcoin.pdf> blockchain, with substantial modifications. This document does not cover or specify any aspects of LBRY that are identical to Bitcoin, and instead focuses on the differences.

The blockchain serves three key purposes:

1. An index of the content available on the network
2. A payment system and record of purchases for priced content
3. A source of cryptographic publisher identities

Stakes

A *stake* is a a single entry in the blockchain that commits credits toward a name. The two types of stakes are *claims* and *supports*.

All stakes have these properties:

id	A 20-byte hash, unique among all stakes. See Stake Identifier Generation.
amount	A quantity of tokens used to back the stake. See Controlling.

Claims

A *claim* is a stake that stores metadata. There are two types of claims:

stream claim	Declares the availability, access method, and publisher of a stream.
channel claim	Creates a pseudonym that can be declared as the publisher of stream claims.

Claim Properties

In addition to the properties that all stakes have, claims have two more properties:

name	A UTF-8 string of up to 255 bytes used to address the claim. See URLs.
value	Metadata about a stream or a channel. See Metadata.

Example Claim

Here is an example stream claim:

```
{
  "claimID": "6e56325c5351ceda2dd0795a30e864492910ccbf",
  "amount": 1.0,
  "name": "lbry",
  "value": {
    "stream": {
      "title": "What is LBRY?",
      "author": "Samuel Bryan",
      "description": "What is LBRY? An introduction with Alex Tabarrok",
      "language": "en",
      "license": "Public Domain",
      "thumbnail": "https://s3.amazonaws.com/files.lbry.io/logo.png",
      "mediaType": "video/mp4",
      "streamHash": "232068af6d51325c4821ac897d13d7837265812164021ec832cb7f18b9caf6c77c23016b31bac
9747e7d5d9be7f4b752",
    },
  }
}
```


Note: the blockchain treats the `value` as an opaque byte string and does not impose any structure on it. Structure is applied and validated higher in the stack. The value is shown here for demonstration purposes only.

Claim Operations

There are three claim operations: *create*, *update*, and *abandon*.

create	Makes a new claim.
update	Changes the value, amount, or channel of an existing claim. Does not change the claim's ID.
abandon	Withdraws a claim, freeing the associated credits to be used for other purposes.

Supports

A *support* is a stake that lends its amount to bolster an existing claim.

Support Properties

Supports have one extra property in addition to the stake properties:

claimID	The ID of the claim that this support is bolstering.

Example Support

Here is an example support for the above claim:

```
{
  "supportID": "fbcc019294468e03a5970dd2adec1535c52365e6",
  "amount": 45.12,
  "claimID": "6e56325c5351ceda2dd0795a30e864492910ccbf",
}
```

Support Operations

Supports are created and abandoned just like claims (see Claim Operations). Supports cannot be updated or themselves supported.

Claimtrie

A *claimtrie* is a data structure used to store the set of all claims and prove the correctness of URL resolution.

The claimtrie is implemented as a Merkle tree <https://en.wikipedia.org/wiki/Merkle_tree> that maps names to claims. Claims are stored as leaf nodes in the tree. Names are stored as the normalized path from the root node to the leaf node.

The *root hash* is the hash of the root node. It is stored in the header of each block in the blockchain. Nodes use the root hash to efficiently and securely validate the state of the claimtrie.

Multiple claims can exist for the same name. They are all stored in the leaf node for that name. See Claim Ordering

For more details on the specific claimtrie implementation, see the source code <https://github.com/lbryio/lbrycrd/blob/master/src/claimtrie.cpp>.

Statuses

Stakes can have one or more of the following statuses at a given block.

Accepted

An *accepted* stake is one that has been entered into the blockchain. This happens when the transaction containing it is included in a block.

Accepted stakes do not affect the intra-leaf claim order until they are active.

The sum of the amount of a claim stake and all of its accepted supports is called its *total amount*.

Abandoned

An *abandoned* stake is one that was withdrawn by its owner. Spending a transaction that contains a stake will cause that stake to become abandoned. Abandoned stakes are removed from the claimtrie.

While data related to abandoned stakes still resides in the blockchain, it is considered invalid and should not be used to resolve URLs or fetch the associated content. Active claim stakes signed by abandoned identities are also considered invalid.

Active

An *active* stake is an accepted and non-abandoned stake that has been in the blockchain for an algorithmically determined number of blocks. This length of time required is called the *activation delay*.

If the stake is an update to an active claim, is the only accepted non-abandoned claim for a name, or does not cause a change in which claim is controlling the name, the activation delay is 0 (i.e. the stake becomes active immediately).

Otherwise, the activation delay is determined by a formula covered in Activation Delay. The formula's inputs are the height of the current block, the height at which the stake was accepted, and the height at which the controlling claim for that name last changed.

The sum of the amount of an active claim and all of its active supports is called its *effective amount*. The effective amount affects the sort order of claims in a leaf node, and which claim is controlling for that name. Claims that are not active have an effective amount of 0.

Controlling (claims only)

A *controlling* claim is the active claim that is first in the sort order of a leaf node. That is, it has the highest effective amount of all claims with the same name.

Only one claim can be controlling for a given name at a given block.

Activation Delay

If a stake does not become active immediately, it becomes active at the block height determined by the following formula:

```
ActivationHeight = AcceptedHeight + min(4032, floor( (AcceptedHeight-TakeoverHeight)/32 ))
```

Where:

- `AcceptedHeight` is the height when the stake was accepted
- `TakeoverHeight` is the most recent height at which the controlling claim for the name changed

In written form, the delay before a stake becomes active is equal to the height at which the stake was accepted minus height of the last takeover, divided by 32. This delay is capped at a maximum of 4032 blocks, which is 7 days of blocks at 2.5 minutes per block (the target block time). It takes approximately 224 days without a takeover to reach the max delay.

The purpose of this delay is to give long-standing claimants time to respond to changes, while still keeping takeover times reasonable and allowing recent or contentious claims to change state quickly.

Claim Ordering

To determine the order of claims in a leaf node, the following algorithm is used:

1. For each claim, recalculate the effective amount.
2. Sort the claims by effective amount in descending order. Claims tied for the same amount are ordered by block height (lowest first), then by transaction order within the block.
3. If the controlling claim from the previous block is still first in the order, then the ordering is finished.
4. Otherwise, a takeover is occurring. Set the takeover height for this name to the current height, recalculate which stakes are now active, and redo steps 1 and 2.
5. At this point, the claim with the greatest effective amount is the controlling claim at this block.

The purpose of 4 is to handle the case when multiple competing claims are made on the same name in different blocks, and one of those claims becomes active but another still-inactive claim has the greatest effective amount. Step 4 will cause the greater claim to also activate and become the controlling claim.

See the example in the appendix for more information.

Normalization

Names in the claimtrie are normalized when performing any comparisons. This is necessary to avoid confusion due to Unicode equivalence or casing. When names are being compared, they are first converted using Unicode Normalization Form D <http://unicode.org/reports/tr15/#Norm_Forms> (NFD), then lowercased using the en_US locale. This means names are effectively case-insensitive. Since claims competing for the same name are stored in the same node in the claimtrie, names are also normalized to determine the claimtrie path to the node.

URLs

URLs are memorable references to claims. All URLs:

1. contain a name (see Claim Properties), and
2. resolve to a single, specific claim for that name

The ultimate purpose of much of the claim and blockchain design is to provide memorable URLs that can be provably resolved by clients without a full copy of the blockchain (e.g. Simplified Payment Verification <https://bitcoin.org/en/glossary/simplified-payment-verification> wallets).

Components

A URL is a name with one or more modifiers. A bare name on its own resolves to the controlling claim at the latest block height. Here are some common URL structures.

Stream Claim Name

A controlling stream claim.

```
lbry://meet-lbry
```

Channel Claim Name

A controlling channel claim.

```
lbry://@lbry
```

Channel Claim Name and Stream Claim Name

A URL containing both a channel and a stream claim name. URLs containing both are resolved in two steps. First, the channel is resolved to its associated claim. Then the stream claim name is resolved to get the appropriate claim from among the claims in the channel.

```
lbry://@lbry/meet-lbry
```

Claim ID

A claim for this name with this claim ID. Partial prefix matches are allowed (see URL Resolution).

```
lbry://meet-lbry#7a0aa95c5023c21c098
lbry://meet-lbry#7a
lbry://@lbry#3f/meet-lbry
```

Claim Sequence

The *n_th accepted claim for this name. _n* must be a positive number. This can be used to resolve claims in the order in which they were made, rather than by the amount of credits backing a claim.

```
lbry://meet-lbry:1
lbry://@lbry:1/meet-lbry
```

Bid Position

The *n_th claim for this name, ordered by total amount (highest first). _n* must be a positive number. This is useful for resolving non-winning bids in bid order.

```
lbry://meet-lbry$2
lbry://meet-lbry$3
lbry://@lbry$2/meet-lbry
```

Query Params

These parameters have no meaning within the LBRY protocol. They are for use by upstream applications.

```
lbry://meet-lbry?arg=value+arg2=value2
```

Grammar

The full URL grammar is defined using Xquery EBNF notation <https://www.w3.org/TR/2017/REC-xquery-31-20170321/#EBNFNotation>:

```
URL ::= Scheme Path Query?

Scheme ::= 'lbry://'

Path ::=  StreamClaimNameAndModifier | ChannelClaimNameAndModifier ( '/' StreamClaimNameAndModifie
r )?

StreamClaimNameAndModifier ::= StreamClaimName Modifier?
ChannelClaimNameAndModifier ::= ChannelClaimName Modifier?

StreamClaimName ::= NameChar+
ChannelClaimName ::= '@' NameChar+

Modifier ::= ClaimID | ClaimSequence | BidPosition
ClaimID ::= '#' Hex+
ClaimSequence ::= ':' PositiveNumber
BidPosition ::= '$' PositiveNumber

Query ::= '?' QueryParameterList
QueryParameterList ::= QueryParameter ( '&' QueryParameterList )*
QueryParameter ::= QueryParameterName ( '=' QueryParameterValue )?
QueryParameterName ::= NameChar+
QueryParameterValue ::= NameChar+

PositiveDigit ::= [123456789]
Digit ::= '0' | PositiveDigit
PositiveNumber ::= PositiveDigit Digit*

HexAlpha ::= [abcdef]
Hex ::= (Digit | HexAlpha)+
```

```
NameChar ::= Char - [=&#:$@%?/]  /* any character that is not reserved */
Char ::= #x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#xFFFD] | [#x10000-#x10FFFF] /* any Unicode cha
racter, excluding the surrogate blocks, FFFE, and FFFF. */
```

Resolution

URL *resolution* is the process of translating a URL into the associated claim ID and metadata. Several URL components are described below. For more information, see the URL resolution example in the appendix.

No Modifier

Return the controlling claim for the name. Stream claims and channel claims are resolved the same way.

ClaimID

Get all claims for the claim name whose IDs start with the given `ClaimID`. Sort the claims in ascending order by block height and position within the block. Return the first claim.

ClaimSequence

Get all claims for the claim name. Sort the claims in ascending order by block height and position within the block. Return the *n_th claim, where _n* is the given `ClaimSequence` value.

BidPosition

Get all claims for the claim name. Sort the claims in descending order by total effective amount. Return the *n_th claim, where _n* is the given `BidSequence` value.

ChannelClaimName and StreamClaimName

If both a channel name and a stream name are present, resolution happens in two steps. First, remove the `/` and `StreamClaimNameAndModifier` from the path, and resolve the URL as if it only had a `ChannelClaimNameAndModifier`. Then get the list of all claims in that channel. Finally, resolve the `StreamClaimNameAndModifier` as if it was its own URL, but instead of considering all claims, only consider the set of claims in the channel.

If multiple claims for the same name exist inside the same channel, they are resolved via the same resolution rules applied entirely within the sub-scope of the channel.

Design Notes

The most contentious aspect of this design is the choice to resolve names without modifiers (sometimes called *vanity names*) to the claim with the highest effective amount. Before discussing the reasoning behind this decision, it should be noted that only vanity URLs resolve this way. Permanent URLs that are short and memorable (e.g. `lbry://myclaimname#a`) exist and are available for the minimal cost of issuing a transaction.

LBRY's resolution semantics stem from a dissatisfaction with existing name allocation designs. Most existing public name schemes are first-come, first-serve with a fixed price. This leads to several bad outcomes:

1. Speculation and extortion. Entrepreneurs are incentivized to register common names even if they don't intend to use them, in hopes of selling them to the proper owner in the future for an exorbitant price. While speculation in general can have positive externalities (stable prices and price signals), in this case it is pure value extraction. Speculation also harms the user experience, who will see the vast majority of URLs sitting unused (c.f. Namecoin).

2. Bureaucracy and transaction costs. While a centralized system can allow for an authority to use a process to reassign names based on trademark or other common use reasons, this system is also imperfect. Most importantly, it is a censorship point and an avenue for complete exclusion. Additionally, such processes are often arbitrary, change over time, involve significant transaction costs, and still lead to names being used in ways that are contrary to user expectation (e.g. nissan.com <http://nissan.com>).

3. Inefficiencies from price controls. Any system that does not allow a price to float freely creates inefficiencies. If the set price is too low, there is speculation and rent-seeking. If the price is too high, people are excluded from a good that it would otherwise be beneficial for them to purchase.

Instead, LBRY has an algorithmic design built into consensus that encourage URLs to flow to their highest valued use. Following Coase <https://en.wikipedia.org/wiki/Coase_theorem>, this staking design allows for clearly defined rules, low transaction costs, and no information asymmetry, minimizing inefficiency in URL allocation.

Transactions

The LBRY blockchain includes the following changes to Bitcoin's transaction scripting language.

Operations and Opcodes

To enable interaction with the claimtrie, three new opcodes were added to the scripting language: `OP_CLAIM_NAME`, `OP_UPDATE_CLAIM`, and `OP_SUPPORT_CLAIM`. In Bitcoin they are respectively `OP_NOP6`, `OP_NOP7`, and `OP_NOP8`. The opcodes are used in output scripts to change the state of the claimtrie. Each opcode is followed by one or more parameters. Here's how these opcodes are used:

```
OP_CLAIM_NAME <name> <value> OP_2DROP OP_DROP <outputScript>

OP_UPDATE_CLAIM <name> <claimID> <value> OP_2DROP OP_2DROP <outputScript>

OP_SUPPORT_CLAIM <name> <claimID> OP_2DROP OP_DROP <outputScript>
```

The `<name>` parameter is the name that the claim is associated with. The `<value>` is the protobuf-encoded claim metadata and optional channel signature (see Metadata for more about this value). The `<claimID>` is the claim ID of a previous claim that is being updated or supported.

Each opcode will push a zero on to the execution stack. Those zeros, as well as any additional parameters after the opcodes, are all dropped by `OP_2DROP` and `OP_DROP`. `<outputScript>` can be any valid script, so a script using these opcodes is also a pay-to-pubkey script. This means that claimtrie scripts can be spent just like regular Bitcoin output scripts.

Stake Identifier Generation

Like any standard Bitcoin output script, a claimtrie script is associated with a transaction hash and output index. This combination of transaction hash and index is called an *outpoint*. Each claimtrie script has a unique outpoint. The outpoint is hashed using SHA-256 and RIPEMD-160 to generate the ID for a stake. For the example above, let's say claimtrie script is included in transaction `7560111513bea7ec38e2ce58a58c1880726b1515497515fd3f470d827669ed43` at the output index `1`. Then the ID is `529357c3422c6046d3fec76be2358004ba22e323`. An implementation of this is available here <https://github.com/lbryio/lbry.go/blob/master/lbrycrd/blockchain.go>.

OP_CLAIM_NAME

New claims are created using `OP_CLAIM_NAME`. For example, a claim transaction setting the name `Fruit` to the value `Apple` looks like this:

```
OP_CLAIM_NAME Fruit Apple OP_2DROP OP_DROP OP_DUP OP_HASH160 <address> OP_EQUALVERIFY OP_CHECKSIG
```

OP_UPDATE_CLAIM

`OP_UPDATE_CLAIM` updates a claim by replacing its metadata. An update transaction has an added requirement that it must spend the output for the existing claim that it wishes to update. Otherwise, it is considered invalid and will not make it into the claimtrie. Thus it must have the following redeem script:

```
<signature> <pubKeyForPreviousAddress>
```

The syntax is identical to the standard way of redeeming a pay-to-pubkey script in Bitcoin, with the caveat that `<pubKeyForPreviousAddress>` must be the public key for the address of the output that contains the claim that is being updated.

To change the value of the previous example claim to "Banana", the payout script is

```
OP_UPDATE_CLAIM Fruit 529357c3422c6046d3fec76be2358004ba22e323 Banana OP_2DROP OP_2DROP OP_DUP OP_
HASH160 <address> OP_EQUALVERIFY OP_CHECKSIG
```

The `<address>` in this script may be the same as the address in the original transaction, or it may be a new address.

OP_SUPPORT_CLAIM

A support for the original example claim has the following payout script:

```
OP_SUPPORT_CLAIM Fruit 529357c3422c6046d3fec76be2358004ba22e323 OP_2DROP OP_DROP OP_DUP OP_HASH160
 <address> OP_EQUALVERIFY OP_CHECKSIG
```

The `<address>` in this script may be the same as the address in the original transaction, or it may be a new address.

Proof of Payment

No system can strongly enforce digital intellectual property rights, especially not a decentralized one. Therefore, the protocol must be able to produce evidence that differentiates legitimate and illegitimate use. In LBRY, this is done via blockchain transactions and proofs of payment.

A proof of payment has two components:

1. A transaction on the blockchain that spends credits to the fee address for a claim (the transaction must send a number of credits equal to or greater than the fee amount for the claim).
2. Proof that a client knows the private key of the address that the transaction spends from.

To prove 1, it is sufficient to provide the transaction ID and input index of the spend. Proving 2 requires signing a nonce using the associated private key.

Verifying a proof of payment is done as follows:

1. Look up the fee amount and fee address of the claim that the proof is for.
2. Use the transaction ID from the proof to find the transaction. Verify that it spends the correct amount to the correct address.
3. Use the public key from the transaction output to verify the signed nonce.

The protocol is likely to be extended in the future to enable stricter proofs of payment.

Consensus

In addition to the stake-related changes described above, LBRY makes changes to the following blockchain consensus rules.

Block Timing

The target block time was lowered from 10 minutes to 2.5 minutes to facilitate faster transaction confirmation.

Difficulty Adjustment

The proof-of-work target is adjusted every block to better adapt to sudden changes in hash rate. The exact adjustment algorithm can be seen here <https://github.com/lbryio/lbrycrd/blob/master/src/lbry.cpp>.

Block Hash Algorithm

LBRY uses a combination of SHA-256, SHA-512, and RIPEMD-160. The exact hashing algorithm can be seen here <https://github.com/lbryio/lbrycrd/blob/master/src/hash.cpp#L18>.

Block Rewards

The block reward schedule was adjusted to provide an initial testing period, a quick ramp-up to max block rewards, then a logarithmic decay to 0. The source for the algorithm is here <https://github.com/lbryio/lbrycrd/blob/master/src/main.cpp#L1594>.

Addresses

The address version byte is set to `0x55` for standard (pay-to-public-key-hash) addresses and `0x7a` for multisig (pay-to-script-hash) addresses. P2PKH addresses start with the letter `b`, and P2SH addresses start with `r`.

All the chain parameters are defined here <https://github.com/lbryio/lbrycrd/blob/master/src/chainparams.cpp>.

Metadata

Metadata is structured information about a stream or channel separate from the content itself (e.g. the title, language, media type, etc.). It is stored in the blockchain as the value property of a claim.

Metadata is stored in a serialized binary format using Protocol Buffers <https://developers.google.com/protocol-buffers/>. This allows for metadata to be:

- **Extensibile**. Metadata can encompass thousands of fields for dozens of types of content. It must be efficient to both modify the structure and maintain backward compatibility.
- **Compact**. Blockchain space is expensive. Data must be stored as compactly as possible.
- **Interoperabile**. Metadata will be used by many projects written in different languages.

The serialized metadata may be cryptographically signed to indicate membership in a channel. See Channels for more info.

Specification

The metadata specification is designed to grow and change frequently. The full specification is not detailed here. The types <https://github.com/lbryio/types> repository is considered the precise specification.

Instead, let's look at an example and some key fields.

Example

Here's some example metadata:

```
{
  "stream": {
    "title": "What is LBRY?",
    "author": "Samuel Bryan",
    "description": "What is LBRY? An introduction with Alex Tabarrok",
    "language": "en",
    "license": "Public Domain",
    "thumbnail": "https://s3.amazonaws.com/files.lbry.io/logo.png",
    "mediaType": "video/mp4",
    "streamHash": "232068af6d51325c4821ac897d13d7837265812164021ec832cb7f18b9caf6c77c23016b31bac97
47e7d5d9be7f4b752"
  }
}
```

Note: Some fields are omitted.

Key Fields

Some important metadata fields are highlighted below.

Stream Hash

A unique identifier that is used to locate and fetch the content from the data network. More in Data.

Fee

Information on how to pay for the content. It includes the address that will receive the payment (the *fee address*), the amount to be paid, and the currency.

Example fee:

```
"fee": {
  "address":"bNz8Va7xMyK9eHA5APzLph6cCTjBtGgmDN",
  "amount":"99.95",
  "currency":"LBC"
}
```

Title, Author, Description

Basic information about the stream.

Language

The ISO 639-1 <https://www.iso.org/iso-639-language-codes.html> two-letter code for the language of the stream.

Thumbnail

A URL to be used to display an image associated with the content.

Media Type

The media type of the item as defined <https://www.iana.org/assignments/media-types/media-types.xhtml> by the IANA.

Channels (Identities)

Channels are the unit of identity. A channel is a claim for a name beginning with `@` that contains a metadata structure for identity rather than content. Included in the metadata is the channel's public key. Here's an example:

```
"claimID": "6e56325c5351ceda2dd0795a30e864492910ccbf",
"name": "@lbry",
"amount": 6.26,
"value": {
  "channel": {
```

```
    "keyType": "SECP256k1",
    "publicKey": "3056301006072a8648ce3d020106052b8104000a03420004180488ffcb3d1825af538b0b952f0eba
6933faa6d8229609ac0aeadfdbcf49C59363aa5d77ff2b7ff06cddc07116b335a4a0849b1b524a4a69d908d69f1bcebb"
  }
}
```

Claims published to a channel contain a signature made with the corresponding private key. A valid signature proves channel membership.

The purpose of channels is to allow content to be clustered under a single pseudonym or identity. This allows publishers to easily list all their content, maintain attribution, and build their brand.

Signing

A claim is considered part of a channel when its metadata is signed by the channel's private key. Here's the structure of a signed metadata value:

field	size	description
Version	1 byte	Format version. See Format Versions.
Channel ID	20 bytes	Claim ID of the channel claim that contains the matching public key. *Skip this field if there is no signature.*
Signature	64 bytes	The signature. *Skip this field if there is no signature.*
Payload	variable	The protobuf-encoded metadata.

Format Versions

The following formats are supported:

format	description
`00000000`	No signature.
`00000001`	Signature using ECDSA SECP256k1 key and SHA-256 hash.

Signing Process

1. Encode the metadata using protobuf.
2. Hash the encoded claim using SHA-256.
3. Sign the hash using the private key associated with the channel.
4. Append all the values (the version, the claim ID of the corresponding channel claim, the signature, and the protobuf-encoded metadata).

Signature Validation

1. Split out the version from the rest of the data.
2. Check the version field. If it indicates that there is no signature, then no validation is necessary.

3. Split out the channel ID and signature from the rest of the data.
4. Look up the channel claim to ensure it exists and contains a public key.
5. Use the public key to verify the signature.

Validation

The blockchain treats metadata as an opaque series of bytes. Clients should not trust the metadata they read from the blockchain. Each client is responsible for correctly encoding and decoding the metadata, and for validating its structure and signatures. This allows evolution of the metadata definition without changes to blockchain consensus rules.

Data

Files published using LBRY are stored in a distributed fashion by the clients participating in the network. Each file is split into many small pieces. Each piece is encrypted and announced to the network. The pieces may also be uploaded to other hosts on the network that specialize in rehosting content.

The purpose of this process is to enable file storage and access without relying on centralized infrastructure, and to create a marketplace for data that allows hosts to be paid for their services. The design is strongly influenced by the BitTorrent protocol <https://en.wikipedia.org/wiki/BitTorrent>.

Encoding

Content on LBRY is encoded to facilitate distribution.

Blobs

The smallest unit of data is called a *blob*. A blob is a chunk of data up to 2MiB in size. Each blob is indexed by its *blob hash*, which is a SHA-384 hash of the blob. Addressing blobs by their hashes protects against naming collisions and ensures that data cannot be accidentally or maliciously modified.

Streams

Multiple blobs are combined into a *stream*. A stream may be a book, a movie, a CAD file, etc. All content on the network is shared as streams. Every stream begins with the *manifest blob*, followed by one or more *content blobs*. The content blobs hold the actual content of the stream. The manifest blob contains information necessary to find the content blobs and decode them into a file. This includes the hashes of the content blobs, their order in the stream, and cryptographic material for decrypting them.

Content blobs are encrypted using AES-256 in CBC mode and PKCS7 padding. In order to keep each encrypted blob at 2MiB max, a blob can hold at most 2097151 bytes (2MiB minus 1 byte) of plaintext data. The source code for the exact algorithm is available here <https://github.com/lbryio/lbry.go/blob/master/stream/blob.go>. The encryption key and the initialization vectors for each blob are stored in the manifest blob.

The blob hash of the manifest blob is called the *stream hash*. It uniquely identifies each stream.

Manifest Contents

A manifest blob's contents are encoded using canonical JSON encoding <http://wiki.laptop.org/go/Canonical_JSON>. The JSON encoding must be canonical to support consistent hashing and validation. Here's an example manifest:


```
{"blobs":[{"blob_hash":"a6daea71be2bb89fab29a2a10face08143411a5245edcaa5efff48c2e459e7ec01ad20edfd
e6da43a932aca45b2cec61","iv":"ef6caef207a207ca5b14c0282d25ce21","length":2097152},{"blob_hash":"bf
2717e2c445052366d35bcd58edb108cbe947af122d8f76b4856db577aeeaa2def5b57dbb80f7b1531296bd3e0256fc","i
```

```
v":"a37b291a37337fc1ff90ae655c244c1d","length":2097152},...,{"blob_hash":"322973617221ddfec6e53bff
4b74b9c21c968cd32ba5a5094d84210e660c4b2ed0882b114a2392a08b06183f19330aaf","iv": "a00f5f458695bdc9d
50d3dbbc7905abc","length":600160}],"filename":"6b706a7977755477704d632e6d7034","key":"94d89c0493c5
76057ac5f32eb0871180","version":1}
```


Here's the same manifest, with whitespace added for readability:


```
{
  "blobs":[
    {
      "blobHash":"a6daea71be2bb89fab29a2a10face08143411a5245edcaa5efff48c2e459e7ec01ad20edfde6da43
a932aca45b2cec61",
      "iv":"ef6caef207a207ca5b14c0282d25ce21",
      "length":2097152
    },
    {
      "blobHash":"bf2717e2c445052366d35bcd58edb108cbe947af122d8f76b4856db577aeeaa2def5b57dbb80f7b1
531296bd3e0256fc",
      "iv":"a37b291a37337fc1ff90ae655c244c1d",
      "length":2097152
    },
    ...,
    {
      "blobHash":"322973617221ddfec6e53bff4b74b9c21c968cd32ba5a5094d84210e660c4b2ed0882b114a2392a0
8b06183f19330aaf",
      "iv": "a00f5f458695bdc9d50d3dbbc7905abc",
      "length": 600160
    }
  ],
  "filename":"6b706a7977755477704d632e6d7034",
  "key":"94d89c0493c576057ac5f32eb0871180",
  "version":1
}
```


The `blobs` field is an ordered list of blobs in the stream. Each item in the list has the blob hash for that blob, the hex-encoded initialization vector used to create the blob, and the length of the encrypted blob (not the original file chunk).

The `filename` is the hex-encoded name of the original file.

The `key` field contains the hex-encoded *stream key*, which is used to decrypt the blobs in the stream. This field is optional. The stream key may instead be stored by a third party and made available to a client when presented with proof that the content was purchased.

The `version` field is always 1. It is intended to signal structure changes in future versions of this protocol.

Every stream must have at least two blobs - the manifest blob and a content blob. Consequently, zero-length streams are not allowed.

Stream Encoding

A file must be encoded into a stream before it can be published. Encoding involves breaking the file into chunks, encrypting the chunks into content blobs, and creating the manifest blob. Here are the steps:

Setup

1. Generate a random 32-byte stream key. This key will be used to encrypt each content blob in the stream.

Content Blobs

1. Break the file into chunks of at most 2097151 bytes.
2. Generate a random 32-byte initialization vector (IV) for each chuck.
3. Pad each chunk using PKCS7 padding.
4. Encrypt each chunk with AES-CBC using the stream key and the IV for that chunk.
5. An encrypted chunk is a blob.

Manifest Blob

1. Fill in the manifest data as described in the Manifest Contents.
2. Encode the data using the canonical JSON encoding.
3. Compute the stream hash.

An implementation of this process is available here <https://github.com/lbryio/lbry.go/tree/master/stream>.

Stream Decoding

Decoding a stream is like encoding in reverse, and with the added step of verifying that the expected blob hashes match the actual data.

1. Verify that the hash of the manifest blob and matches the stream hash.
2. Parse the JSON in manifest blob.
3. Verify the hashes of the content blobs.
4. Decrypt and remove the padding from each content blob using the stream key and IVs in the manifest.
5. Concatenate the decrypted chunks in order.

Announce

After a stream is encoded, it must be *announced* to the network. Announcing is the process of letting other nodes on the network know that a client has content available for download. LBRY tracks announced content using a distributed hash table.

Distributed Hash Table

Distributed hash tables (or DHTs) are an effective way to build a peer-to-peer content network. LBRY's DHT implementation follows the Kademlia <https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf> specification fairly closely, with some modifications.

A distributed hash table is a key-value store that is spread over multiple nodes in a network. Nodes may join or leave the network anytime, with no central coordination necessary. Nodes communicate with each other using a peer-to-peer protocol to advertise what data they have and what they are best positioned to store.

When a host connects to the DHT, it announces the hash for every blob it wishes to share. Downloading a blob from the network requires querying the DHT for a list of hosts that announced that blob's hash (called *peers*), then requesting the blob from the peers directly.

Announcing to the DHT

A host announces a hash to the DHT in two steps. First, the host looks for nodes that are closest to the target hash. Then the host asks those nodes to store the fact that the host has the target hash available for download.

Finding the closest nodes is done via iterative `FindNode` DHT requests. The host starts with the closest nodes it knows about and sends a `FindNode(target_hash)` request to each of them. If any of the requests return nodes that are closer to the target hash, the host sends `FindNode` requests to those nodes to try to get even closer. When the `FindNode` requests no longer return nodes that are closer, the search ends.

Once the search is over, the host sends a `Store(target_hash)` request to the closest several nodes it found. The nodes receiving this request store the fact that the host is a peer for the target hash.

Download

A client wishing to download a stream must first query the DHT to find peers hosting the blobs in that stream, then contact those peers to download the blobs directly.

Querying the DHT

Querying works almost the same way as announcing. A client looking for a target hash starts by sending iterative `FindValue(target_hash)` requests to the nodes it knows that are closest to the target hash. If a node receives a `FindValue` request and knows of any peers for the target hash, it responds with a list of those peers. Otherwise, it responds with the closest nodes to the target hash that it knows about. The client then queries those closer nodes using the same `FindValue` call. This way, each call either finds the client some peers, or brings it closer to finding those peers. If no peers are found and no closer nodes are being returned, the client determines that the target hash is not available and gives up.

Blob Exchange Protocol

Downloading a blob from a peer is governed by the *Blob Exchange Protocol*. It is used by hosts and clients to exchange blobs and check data pricing and blob availability. The protocol is an RPC protocol using Protocol Buffers and the gRPC framework. It has five types of requests.

PriceCheck

PriceCheck gets the price that the server is charging for data transfer. It returns the prices in deweys per KB.

DownloadCheck

DownloadCheck checks whether the server has certain blobs available for download. For each hash in the request, the server returns a true or false to indicate whether the blob is available.

Download

Download requests the blob for a given hash. The response contains the blob, its hash, and the address where to send payment for the data transfer. If the blob is not available on the server, the response instead contains an error.

UploadCheck

UploadCheck asks the server whether blobs can be uploaded to it. For each hash in the request, the server returns a true or false to indicate whether it would accept a given blob for upload. In addition, if any of the hashes in the request is a stream hash and the server has the manifest blob for that stream but is missing some content blobs, it may include the hashes of those content blobs in the response.

Upload

Upload sends a blob to the server. If uploading many blobs, the client should use the UploadCheck request to check which blobs the server actually needs. This avoids needlessly uploading blobs that the server already has. If a client tries to upload too many blobs that the server does not want, the server may consider it a denial of service attack.

The protocol methods and message types are defined in detail here <https://github.com/lbryio/lbry.go/blob/master/blobex/blobex.proto>.

Reflectors and Data Markets

In order for a client to download content, there must be hosts online that have the content the client wants, when the client wants it. To incentivize the continued hosting of data, the blob exchange protocol supports data upload and payment for data. *Reflectors* are hosts that accept data uploads. They rehost (reflect) the uploaded data and charge for downloads.

Using a reflector is optional, but most publishers will probably choose to use them. Doing so obviates the need for the publisher's server to be online and connectable, which can be especially useful for mobile clients or those behind a firewall.

The current version of the protocol does not support sophisticated price negotiation between clients and hosts. The host simply chooses the price it wants to charge. Clients check this price before downloading, and pay the price after the download is complete. Future protocol versions will include more options for price negotiation, as well as stronger proofs of payment.

Appendix

Claim Activation Example

Here is a step-by-step example to illustrate how competing claims activate and are ordered. All stakes are for the same name.

Block 13: Claim A for 10LBC is accepted. It is the first claim, so it immediately becomes active and controlling.
State: A(10) is controlling

Block 1001: Claim B for 20LBC is accepted. Its activation height is `1001 + min(4032, floor((1001-13) / 32)) = 1001 + 30 = 1031`.
State: A(10) is controlling, B(20) is accepted.

Block 1010: Support X for 14LBC for claim A is accepted. Since it is a support for the controlling claim, it activates immediately.
State: A(10+14) is controlling, B(20) is accepted.

Block 1020: Claim C for 50LBC is accepted. The activation height is `1020 + min(4032, floor((1020-13) / 32)) = 1020 + 31 = 1051`.
State: A(10+14) is controlling, B(20) is accepted, C(50) is accepted.

Block 1031: Claim B activates. It has 20LBC, while claim A has 24LBC (10 original + 14 from support X). There is no takeover, and claim A remains controlling.
State: A(10+14) is controlling, B(20) is active, C(50) is accepted.

Block 1040: Claim D for 300LBC is accepted. The activation height is `1040 + min(4032, floor((1040-13) / 32)) = 1040 + 32 = 1072`.
State: A(10+14) is controlling, B(20) is active, C(50) is accepted, D(300) is accepted.

Block 1051: Claim C activates. It has 50LBC, while claim A has 24LBC, so a takeover is initiated. The takeover height for this name is set to 1051, and therefore the activation delay for all the claims becomes `min(4032, floor((1051-1051) / 32)) = 0`. All the claims become active. The totals for each claim are recalculated, and claim D becomes controlling because it has the highest total.
State: A(10+14) is active, B(20) is active, C(50) is active, D(300) is controlling.

URL Resolution Examples

Suppose the following names were claimed in the following order and no other claims exist.

Channel Name	Stream Name	Claim ID	Amount
–	apple	690eea	1
–	banana	714a3f	2
–	cherry	bfaabb	100
–	apple	690eea	10
@Arthur	–	b7bab5	1
@Bryan	–	0da517	1
@Chris	–	b3f7b1	1
@Chris	banana	fc861c	1
@Arthur	apple	37ee1	20
@Bryan	cherry	a18bca	10
@Chris	–	005a7d	100
@Arthur	cherry	d39aa0	20

Here is how the following URLs resolve:

URL	Claim ID
`lbry://apple`	a37ee1
`lbry://banana`	714a3f
`lbry://@Chris`	005a7d
`lbry://@Chris/banana`	*not found* (the controlling `@Chris` does not have a `banana`)
`lbry://@Chris:1/banana`	fc861c
`lbry://@Chris:#fc8/banana`	fc861c
`lbry://cherry`	bfaabb
`lbry://@Arthur/cherry`	d39aa0
`lbry://@Bryan`	0da517
`lbry://banana$1`	714a3f
`lbry://banana$2`	fc861c
`lbry://banana$3`	*not found*
`lbry://@Arthur:1`	b7bab5
