Dazaar

This document describes how Dazaar (https://dazaar.com) works in detail. Dazaar is a network and marketplace for sharing, selling, and buying various datasets through a peer to peer network. Dazaar is agnostic to the payment method, supporting any blockchain or fiat payment processor. Dazaar supports any data format that can be represented using an append-only log data structure.

High-level Overview

Dazaar works as a protocol extension to the Hypercore Protocol, https://hypercore-protocol.org.

It can be seen as 3 interacting components; verifiable, random-access storage, an encrypted communications protocol and a pluggable payment gateway, where the first two are provided by the Hypercore protocol and the latter by the Dazaar extension.

Each component is briefly introduced here:

- **Storage**: Hypercore (https://hypercore-protocol.org) is a append-only log structure, with each entry (block) verified by a Merkle tree. Root nodes of the Merkle tree are signed using an asymmetric key pair. Due to the Merkle tree, efficient random access is possible and combined with a signature the whole tree is always proven to be authentic from the original writer.
- **Protocol**: When Hypercore communicates over the wire it performs a Noise handshake, where the client (initiator) proves their identity to the server (responder) and vice versa. An encrypted transport channel is then used for a series of negotiation messages to decide the correct data feed, offset, payment details etc.
- **Payment Gateway**: Customer Noise keys are stored locally (or in a database) for resuming previous sessions. Due payment can be checked against any blockchain or traditional payment processor, through a pluggable API. Initial support includes Lightning and EOS

In-depth Details

Storage

Dazaar's core primitive is a distributed append-only log data structure named Hypercore, https://github.com/hypercore-protocol/hypercore, which serves as the main format for storing and distributing data.

Hypercore is similar to a single-writer ledger, which does not need any proof of work, stake or authority, since only the creator of the ledger is able to append to it. Each entry appended to a Hypercore is addressed by a sequence number, at which it is inserted, similar to an array. To guarantee data integrity, Hypercore uses a cryptographically signed Merkle tree. Only the content creator holds the secret key for signing new data. Using a Merkle tree also gives efficient updates and random access. When a new item is appended to the Hypercore, the Merkle tree is updated and the root(s) are signed.

```
# A Hypercore with 3 items appended looks like this where the first item
# is addressed by sequence number 0, then 1, then 2, and so fourth.

0: hello
1: distributed
2: world
```

The Merkle tree "spans" the data in the following way

```
  1 <-------+
 / \        |--- Merkle tree roots
0   2  4 <--+

h   d  w
e   i  o
l   s  r
l   t  l
o   .  d
    .
```

Appending the value "another" to the Hypercore, the Merkle tree grows into the following structure

```
      3 <------- Merkle tree root
   /     \
  1       5
 / \     / \
0   2   4   6

h   d   w   a
e   i   o   n
l   s   r   o
l   t   l   t
o   .   d   .
    .       .
```

In the case of multiple Merkle roots (ie. `count(data) !== 2 ^ n`), all roots are hashed together, creating a tree hash, which is then signed to construct a single Merkle proof. These loose roots are sometimes called "peaks" or "shoulders".

Signing the root of the Merkle Tree verifies the origin of the data, since a replicating peer will use the corresponding public key to verify the origin of the data, as well as the integrity. The public key acts as a distributed identifier for the Hypercore, often referred to as the "Hypercore key".

As previously mentioned, the Merkle Tree allows for efficient random access. This feature allows peers to securely download only the parts of the log they are interested in. This property makes Hypercore ideal for time series data with live updates or sparsely replicating very large datasets.

The random access features also permits powerful data structures to be implemented on top, which opens for much more rich applications.

Examples include:

- Random access key-value store: https://github.com/hypercore-protocol/hypertrie
- Distributed file system: https://github.com/hypercore-protocol/hyperdrive

Hypercores also served as the foundation of the Dat protocol, and further technical details can be found in the Dat white paper: https://github.com/datprotocol/whitepaper/blob/master/dat-paper.pdf and on the Hypercore protocol website, https://hypercore-protocol.org

Distribution and access control

Since Hypercores are an append-only, tamper-proof data structure, any peer can relay data to other peers, without talking to the original author.

This is one of the hallmarks of P2P technology as it lowers load and bandwidth requirements for a data producer, by sharing load across the network of peers. This allows for much greater scale in the number of consuming peers (readers).

However, the distributed nature also requires different techniques for implementing access control systems in a P2P system. In this context, what access control means, is methods for controlling who can access data as it is written and methods to revoke future access if a condition is no longer met.

Per default Hypercores ship with a very basic access control system, called a capability system. This capability system ensures, using cryptographic primitives, that only peers that know the public key of a Hypercore are able to replicate data from other peers in the network. The key can be distributed by any means necessary.

This establishes a flow where the data author, can create a new Hypercore, and using a secure channel, for example the Signal messaging app, can share the Hypercore public key with another consumer. The author and the consumer can now establish a P2P network connection between each other, and the capability system built into Hypercore ensures that no eavesdropper will be able to decrypt the data shared in the Hypercore itself, or find peers interested in the Hypercore, unless they have knowledge of the public key.

This basic access control system is however quite limited. It has the power of simplicity, since it is simply sharing a Hypercore key, akin to a URL, but lacks features such as revocation. If the Hypercore had been shared with two different consumers and later one should be revoked, a new Hypercore would have to be created and shared with all peers that should continue to have access.

This is not just cumbersome, but also leads to security difficulties, such as whether the original producer and the new one are in fact the same trusted author.

For something like a distributed marketplace where revocation is continuous, perhaps due to lack of payment, and where there are many consumers, a better solution is needed.

Revokable Data Subscriptions

To support revokable data subscriptions on top of Hypercores, the system needs to support the two phases of a subscription; purchase and access.

1. During the purchase phase, the buyer connects to the seller using a mutually authenticating handshake over an encrypted channel, first providing the public key they want to bind the subscription to and then a proof of purchase. This proof of purchase must be verifiable for the seller, who upon successful validation, will proceed to share data. This verification could for example be querying a blockchain or a payment processor API. A seller may choose to make purchases a one-time fee or an ongoing fee based on, for example, time or data usage.
2. During the access phase, the buyer connects to the seller using the key pair for which they provided the public key during the registration, which the seller then verifies for having an active subscription. An inactive subscription (eg. insufficient payment) will cause a rejection and the access is effectively revoked. Again verification could happen against a blockchain or a local database kept from the purchase phase.

Since connections are mutually authenticated, the buyer will know that it is talking to the correct seller, since the seller must hold the correct secret key, and the seller know they are talking to the correct buyer, since they must hold the secret key to the public key provided during purchase. During access the seller can choose to close the connection in case of insufficient payment, to cut off access for the buyer. In addition the guarantees Hypercore provides reassures the buyer that they are talking to the right seller.

Note that due to the nature of data access, data that has already been seen by the buyer cannot be revoked, however future access to fresh data or further historical data can be restricted.

Protocol extensions

To implement more custom behaviour on top of the Hypercore protocol, an extension protocol is used. With extension messages third parties can add their own message types to Hypercore's wire protocol.

Dazaar utilises this extension protocol to add its marketplace mechanics and content payment facilitation. More precisely it uses the following extension messages:

`dazaar/one-time-feed`

A 32 byte binary message with the feed key of the feed the buyer is buying. The seller sends this message after verifying the buyer and generated a feed for the buyer.

`dazaar/valid`

The seller can choose to send this JSON message to the buyer to indicate how much time/bandwidth the buyer has left. This should only be sent when the seller has validated the buyers' payment as valid and can thereafter be sent periodically.

`dazaar/invalid`

Similarly to the `valid` message, the invalid message can be sent when the buyer runs out of bandwidth/time, to inform the buyer why the seller is no longer uploading data to the buyer.

Fully authenticated connections

When Hypercore peers connect with one another, they do so by establishing a fully authenticated connection. To facilitate the secure connection Hypercore uses the Noise protocol framework. Noise is a state of the art cryptographic framework for composing handshakes as part of initiation of a secure channel. It avoids many of the pitfalls and constraints of protocols such as TLS, making it more flexible, and hence ideal in a P2P scenario.

To establish the fully authenticated and encrypted connection the XX handshake pattern is used. The XX pattern first handshakes using ephemeral key pairs, which makes the session unique and provides forward secrecy. An eavesdropper will not be able to identify the two parties based on the data sent here alone. After ephemeral keys have been shared, the connection is "upgraded" by sharing the static keys of each peer.

Dazaar utilises these static keys to authenticate both the buying peer (B) and selling peer (S).

The seller will verify that the public key sent by the buyer B, is authorised to connect. If the buyer is not authorised, Dazaar will send an appropriate error message to the buyer with the reason. Note even if one of the two parties sent public keys they did not posses the corresponding secret key for, they will not be able to read any messages. This is due to the nature of Noise, due to the Diffie-Hellman key exchange algorithm.

Periodically the seller should revalidate that the buyers public key still has a valid subscription and if not revoke the data stream and disconnect from the buyer.

For a buyer to revoke a key pair, they should simply stop providing a payment proof. It should also be noted that it is the sellers responsibility to restrict multiple connections using the same key pair. The protocol, however, ensures that each buyer will be assigned a unique hypercore as described in the next section.

Revokable Hypercores

When a buyer needs to be revoked from further replicating the Hypercore, eg. in case of expired proof of payment or violation of the terms of service, some additional tweaks are needed to the standard Hypercore protocol.

As described earlier, Hypercores do not have a per user revocation scheme built in. If a Hypercore is shared with peers A and B, there is nothing stopping peer A from continue to share it with B, even if the author has stopped sharing with B.

For a marketplace this does not suffice. To solve this we introduce the concept of "rekeyed" Hypercores. A rekeyed Hypercore is a Hypercore that shares data and Merkle Tree with another Hypercore, but its Merkle root is signed by a different key pair, which makes it appear like a different dataset on the network.

If we look at the technical drawing for a Hypercore with 4 pieces of data in the beginning of this paper

```
      3   <--- Merkle Tree root
    /    \
   1      5
  / \    / \
 0   2  4   6
```

To rekey a Hypercore, we simply generate a new key pair and re-sign the Merkle Tree root at `3`. In the worst case there will only ever be `log2(count(data))` Merkle Tree roots, making this operation efficient. We don't need to store any of these signatures on disk as they can simply be generated on demand when a peer requests a new signature for an updated Merkle tree. This means that a rekeyed Hypercore requires zero additional storage except that we need to persist the key pair used to generate the signature.

Two rekeyed Hypercores cannot directly swarm with each other.

```
# Non rekeyed replication:
# A is a hypercore and B and C are peers replicating

   A
  /   \
B --- C

# If A stops replicating with C, B can still forward the data and C can still
# verify it.

   A
  /
B --- C

# rekeyed replication:
# A is a hypercore and B is a rekeyed Hypercore based on A
# and C is a rekeyed Hypercore based on A
```

```
    A
  /   \
 B     C

# In this case B and C cannot swarm directly as the two Hypercores
# are not equal (ie uses different key pairs)
```

It should be noted that if B and C in the scenario choose to replicate regardless, their Merkle Trees will be equivalent, but their tree signatures will not. This means that C could in theory get old data from B as long as it receives the signatures from A for the corresponding Merkle Tree root.

To revoke access to a rekeyed Hypercore a seller should simply stop sharing the rekeyed Hypercore. In addition, to prevent the revoked buyer from relaying the rekeyed Hypercore, it can choose to make the key pair public, that was used to sign the Merkle Tree. By publicising it, the key pair can no longer be trusted to only have been used by the seller, effectively burning the key. In this case the buyer can still re-share the data, but would have to sign it with a key pair the buyer generates by themselves, invalidating that the data actually came from the seller.

Discovery and connectivity

Similar to existing distributed networks, Dazaar buyer's needs a network discovery mechanism to find the Dazaar seller they are interested in and establish a P2P connection between the two.

Dazaar benefits from the Hyperswarm discovery network that Hypercore uses, https://hypercore-protocol.org/#hyperswarm. The Hyperswarm discovery network is a distributed Kademlia based DHT that also provides P2P hole-punching capabilities, making it possible to run Dazaar peers from home without having to configure firewalls and networks, in most cases.

Normally when Hypercore peers discover each other on the Hyperswarm network they do so by using a known identifier describing their Hypercore, similar to how BitTorrent peers discovery each using their torrent "info hash". In Hypercore this key is called the "Hypercore Discovery Key" which is a hash of the Hypercore's public key.

Since Dazaar relies on buyers finding the seller of a dataset it uses a slightly different model.

Only selling peers announce themselves to the network. As their discovery topic they use their 32 byte Noise public key instead of the "Discovery key". This makes it easier for buyers to quickly find the seller they are looking for. Since Hypercore does the full Noise XX handshake and Dazaar verifies that the seller's static key was indeed the one the buyer was trying to connect to, bad peers will automatically be rejected if they announce to the sellers topic.

Dazaar Card

To easier distribute the payment terms and other metadata for the dataset being offered on Dazaar in a structured way, we introduce the "Dazaar Card", a JSON object describing your dataset and terms.

A Dazaard card looks like this:

```
{
  "name": "Dazaar card example",
  "description": "Highly valuable market data",
  "homepage": "https://example.com",
  "contact": "janedoe@example.com",
  "provider": "Jane Doe",
  "sellerKey": "dead...beef",
  "payment": [{
    "method": "lnd",
    "currency": "LightningSats",
    "unit": "seconds",
    "interval": "600",
    "amount": "1000"
  }]
}
```

The above Dazaar card describes a dataset of "Highly valuable market data", that can be purchased using an Lightning payment of `1000` sats every 600 seconds.

Buying and selling protocol using Dazaar

We use the above data structures and techniques to construct the Dazaar marketplace for data.

Selling data

Assume a seller, S wants to sell a dataset stored in a Hypercore, HC.

If S has not already generated a key pair to be used as their identity, they do so first. This key pair is persisted using any form of secure storage (i.e. encrypted on disk).

S then announces their IP and port on the HyperSwarm DHT under their public key, and publishes their public key on a web site or some distributed table together with a human readable description of the dataset they are selling, along with details of price, how to pay, terms etc, in the form of a Dazaar card (described above).

Buying data

Buyer B, discovers the dataset listing for HC and wants to purchase it.

Like S, B generates and persists a key pair.

Then, based on the terms that S listed, B performs an initial payment as specified in the Dazaar cards.

B then connects to S. As mentioned in the authenticated connections section above, S validates B's public key by checking that B's public key is present in a recent payment to S.

If so, S generates a rekeyed Hypercore, HC' from HC, and forwards the Hypercore key of HC' to B. If B has previously purchased HC from S, then this feed can be reused so that B does not have to re-download the full dataset again.

It sends back the key of HC' using `dazaar/one-time-feed` extension message described above.

In case it rejects B's public key, it can send the `dazaar/invalid` message with a JSON object containing the reason why it was rejected.

After sending the `one-time-feed` message, the existing Hypercore replication stream is used to replicate the rekeyed Hypercore.

Periodically S will verify that B's public still has a valid subscription to S based on the payment terms. If not, S can either send an `dazaar/invalid` message or terminate the connection to B.

Conclusion

Dazaar is new protocol for sharing, selling and buying data using a fully distributed network without middlemen and payment fees involved. Dazaar is payment agnostic, providing support for any blockchain or payment processor.