lupinfort

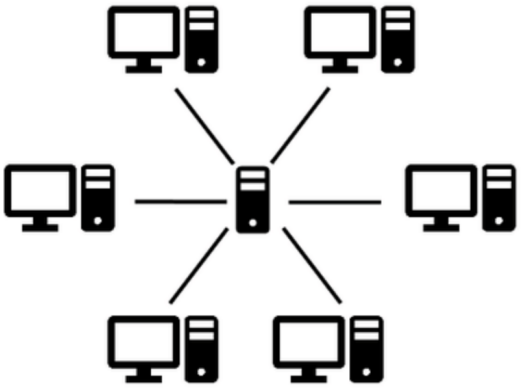
a protocol for a decentralised, peer-to-peer social media platform

Conall Keane // keanec10
04/05/21

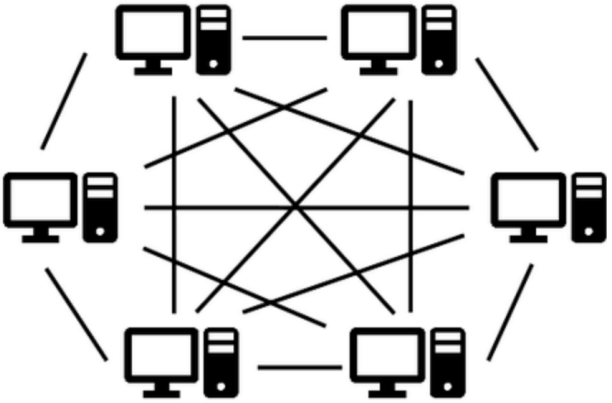
Table of contents

Introduction

The way current social media platforms operate is that users of the platform all interact with a central server to store and request data (on right).



Since all interactions on the platform pass through this central server, it allows the company managing the platform to monitor the activity of its users, and exploit this in order to show them specific advertisements or push them towards particular content in an attempt to keep them on the platform.



For this reason, what we want to do is change it so that instead of relying on a central server, the platform can run itself with just the users, with no company or private interest involved (on left).

To do this, we need to find enough storage space and network bandwidth to replace the functionality of the central server, so where can we find this?

Assuming the platform sees some adoption, then across all the users' devices there should be more than enough storage space and network bandwidth, so we can leverage this to fulfil our goal of removing the need for a central server.

In essence, then, the problem is very simple to outline; we just need all users on the platform to pitch in a small portion of their device storage and network bandwidth to store the platform's data and serve requests.

This is what the **lupinfort** protocol sets out to coordinate, so that users can join a privacy-respecting, transparent, not-for-profit, peer-to-peer social media platform, with a nearly identical user experience and functionality to existing centralised social media platforms.

Prerequisite knowledge

Cryptographic Tools

Cryptography is the crux of every trustless system, and we leverage a number of cryptographic techniques in order to achieve suitable coordination across a **lupinfort** network.

Hash functions

A hash function provides a deterministic mapping from arbitrary input data of any size to a seemingly random output of a specified fixed size, called a "hash".

Typically the number of possible hashes is so great that the probability of two data inputs producing the same hash is miniscule, which allows for a data hash to serve as a proxy for the data in many protocol mechanisms, often referred to as a "data fingerprint".

Hashes are so abstracted from their corresponding inputs that it's impossible to determine what the original data was, and this allows for private, yet deterministic, content handling based on hashes of data.

Message signing and verifying

By utilising the mathematical properties of elliptic curves, we can derive a pair of numbers that are the sole inverse of one another when "applied" to a value via a special operation.

The size of these numbers and the complexity of this operation is such that, given only one of the numbers, there is no known algorithm to recover the other and it's infeasible to discover it by brute force methods.

These properties enable us to compose our own "digital signature" by generating a pair of such numbers- we keep one of our numbers secret, which we call our "private key", and we publicise our other number for everyone to find, which we call our "public key.

We "sign" messages by applying our private key to them, and any skeptic who wishes to verify our signature simply applies our public key to it and checks if it reverts back to the original message.

Since each public-key-private-key pair is uniquely connected, the only way we could produce a signature that could be reversed by our public key is if we possess the corresponding private key, thus validating our "ownership" of the public key.

Any malicious actor who tries to impersonate someone will fail such a check, because whatever private key they've generated won't match up with the public key they're targeting.

This is a vital component of a peer-to-peer network because if nodes can tie themselves to a public key, they have an identity that can persist across sessions, enabling authentic content authorship, "friending", blacklisting, and many other important features.

End-to-end encryption

Possibly the most significant application of cryptography in any online platform, decentralised or not, is enabling end-to-end encryption.

When communication is done over the internet, the raw data signals being exchanged can be quite easily intercepted by malicious third parties who are monitoring physical telecommunications connections.

This is essentially broadcasting all of your online activity for eavesdroppers to listen to, which for any sensitive data would be hugely concerning, especially when it comes to private messages and posts in a social media platform like **lupinfort**.

The way to solve this is for the two devices who are in conversation over the internet to encrypt all of their signals using a shared key that both are aware of.

Since only they possess this key, they can be sure that no third party will be able to listen in on them, since without the ability to decrypt the communications it would appear to be completely gibberish.

This security is known as "end-to-end encryption", since the two endpoints- ie: devices- are encrypting their messages back and forth; this secure communication is built into **lupinfort** by default, helping prevent any chance of exploitation.

Decentralised Networks

Since the **lupinfort** protocol applies to decentralised, peer-to-peer networks, it is important to understand some of the foundational building blocks of such systems.

Bootstrap nodes

When attempting to join a peer-to-peer network, a new node has no knowledge of what addresses the nodes already participating in the network can be found at.

This is in contrast to a centralised network where clients always know what address- or what domain- they're looking for.

However, even in a peer-to-peer network, a new node only needs to learn of a few other nodes' addresses, because from there they can recursively gather the addresses that each new node they encounter are aware of, building up a local index in a bootstrapping process.

The most common method to overcome this hurdle is to designate a number of permanent bootstrap nodes whose addresses are known to all new nodes joining the network, so that there is always an available entry point to the network.

Despite being a necessity, this is undesirable as it involves placing trust in these bootstrap nodes in a system that is intended to be trustless; however, as a peer-to-peer network increases in size, there are more and more potential bootstrap nodes spread across the network, to the point where even the bootstrap process can become suitably decentralised.

Proofs-of-work

A proof-of-work is an operation that is intentionally designed to take a long time to complete, and which is required to be completed before being able to access some part of a system.

This technique is employed to fortify aspects of protocols that can be abused through repetition, in the hopes that it deters people from being able to exploit the protocol since an attack would take so long to carry out.

A common application is in preventing Sybil attacks, an attack in which a user attempts to gain influence over a peer-to-peer network by maintaining ownership of many nodes.

In order to carry out this attack, a malicious user would simply have to generate a huge amount of node IDs- whatever form that might take- so to mitigate this a peer-to-peer network can enforce that a proof-of-work value must be provided as well as a node ID in order to participate in the network.

For honest users this will only be a once-off wait since they will only ever generate a single node ID, but for the malicious user who is trying to generate many node IDs it will slow them down greatly.

An example of how such a proof-of-work value might be enforced is by requesting a value which, when appended to the user's node ID and hashed, results in a number in a particular range.

By the nature of hash functions, there is no quicker way to find such a value than by brute-force searching, and so depending on how large the target range is then the proof-of-work will be more difficult to generate, costing more time.

Kademlia DHT

A Kademlia Distributed-Hash-Table, or Kademlia DHT ***[1]***, is an efficient design structure that can be applied to a peer-to-peer network such that locating nodes in the network can be done very quickly.

In such a network, every node maintains a number of lists of node IDs, with each different list containing node IDs that are increasingly distant from their own ID based on an XOR distance metric.

In the first list will be node IDs that have only their first bit in common with the owner's node ID, in the second will be node IDs that have only their first two bits in common, and so on.

The lists are a fixed maximum size, a common choice being 20, with the least recently seen node being discarded each time in favour of a newly encountered node.

The amazing quality about this structure is that despite each node needing only to maintain knowledge of a relatively tiny portion of the network, once the network is large enough it guarantees retrieval of a node's address in no more than log_2(N) steps, where N is the bit length of the node IDs in the network.

This has a nice intuitive proof; since our node will maintain a list of node IDs at each XOR-distance from our own ID, we will start our search by selecting nodes from whichever list is closest to our target ID- that is, we will be querying nodes whose IDs have the same first K bits as our target ID.

Now recall that each of these nodes we contact will be maintaining lists similar to ours, except that their lists are of course based on XOR-distance from their own ID.

This means that as long as they have at least one node in each of their lists, they will be able to provide us with a node that is at worst one bit closer to the node ID we're looking for.

It's guaranteed to be closer because we initiated our search by querying nodes who have their first K bits in common with our target ID, and now each of these nodes will have one of their lists consisting of node IDs with their first (K + 1) bits in common with our target ID.

They will send us back nodes from that list that have (K + 1) bits in common with our target ID and we can then query those nodes and get node IDs that have (K + 2) bits in common with our target ID, and so on.

Note that this is a worst-case scenario- the nodes we contact may just happen to have nodes who are more than one bits closer to our target ID, but at the very least they will be one bit closer.

This is the power of using XOR as a distance metric; at each successive step of our search we have a baseline number of bits that are guaranteed to be in common from the start of the ID, and it's impossible for us to go backwards from that.

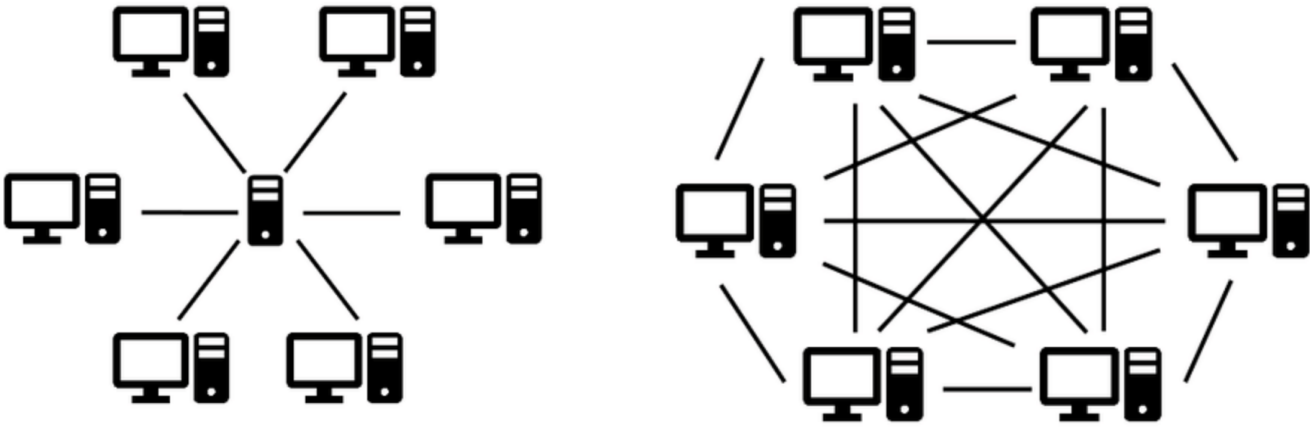
Design of the lupinfort protocol

In this section, we will outline the core elements of the protocol and the reasoning for their consideration.

The problem

Recall that our problem is ensuring that all users on our social media platform pitch in a small portion of their device storage and network bandwidth to store the platform's data and serve requests.

This would allow us to change from a centralised network structure (left) to a decentralised, peer-to-peer network structure (right):



To do this, we need to examine the properties of current centralised networks and replicate their functionality in our decentralised, peer-to-peer network.

Characteristics of a social media platform

Since nearly all current social media platforms are centralised, we will examine the structure of a centralised social media platform as being synonymous with a social media platform in general.

Fundamentally, all a centralised social media platform provides is a central repository where users can store their posts, comments, and messages.

This makes the networking extremely simple; users only need to connect to "xyz.com" and they can fully navigate the content on the platform.

Interacting with the platform can be boiled down to one of two operations:

- **storing data in the repository:** *making posts or comments, and sending messages*
- **retrieving data from the repository:** *viewing posts or comments, and receiving messages*

Data storage is by far the more important of the two operations, since data retrieval simply follows from re-tracing the steps from the storage procedure.

Thus, in building a social media platform, it's important to store data in such a way that the platform possesses two key properties:

- **structured retrievability:** *we need data to be filterable by user-provided metadata- eg: by profile or by topic- so that users can navigate the platform easily*
- **protection from abuse:** *we need to establish some rules to protect against unfair behaviour from the platform or its users*

The aim of preventing abuse of the platform and the users can also be further broken down into a few key factors:

- **storage limiting:** *we need to put a cap on how much data each user can upload to the platform, to avoid running out of storage space*
- **bandwidth limiting:** *we need to put a cap on how quickly each user can upload data to the platform, to avoid service being denied to other users*
- **content moderation:** *we need the ability to remove content from the platform that is deemed to be inappropriate*
- **data permanence:** *we can't allow appropriate content to be dropped from the platform until the user who owns it decides to remove it*

Centralised platforms

In current centralised platforms, all of the above requirements are satisfied:

- **structured retrievability:** *all data is stored in a central repository, so it can simply be accessed and filtered on the platform servers via internal database queries*
- **protection from abuse:**

- **storage limiting:** *all data is stored in a central repository and all requests must go through the platform servers, so the owners of the platform have perfect knowledge of how much storage space each user is using and have full autonomy to refuse further requests for data storage once a user reaches a certain limit*
- **bandwidth limiting:** *all requests must go through the platform servers, so the owners of the platform have perfect knowledge of each user's rate of requests and have full autonomy to put a user on cooldown if their rate of requests exceeds a certain limit*
- **content moderation:** *all data is stored in a central repository, so the owners of the platform have full autonomy to remove content that they deem to be inappropriate*
- **data permanence:** *all data is stored in a central repository, so the owners of the platform have full autonomy to remove content, however they would lose goodwill if they failed to store user content, so it's in their best interests to preserve it*

In a decentralised, peer-to-peer platform however, it's much more complicated to achieve similar results, mainly because every single one of the above current approaches depends on all the data either being stored in a central repository or being managed by a central entity with full autonomy.

Decentralised, peer-to-peer platforms

Our platform is built on a Kademlia DHT where each user operates a node in the network, so they only have knowledge of the data that is being stored at their node and they only have autonomy over that data.

Nodes also can't take other nodes' word for granted unless there is very strong cryptographic and probabilistic evidence available to support it, making coordination even more difficult.

Despite these challenges, we can still tick off a few of the requirements straight away for our peer-to-peer platform:

- **bandwidth limiting:** *each node handles its own incoming requests, so they have perfect knowledge of each user's rate of requests to their particular node and have full autonomy to put a user on cooldown if their rate of requests to their node exceeds a certain limit- if a user proceeds to exceed this limit across a number of nodes, that accumulation will essentially parallel an overall platform cooldown given by a central entity*

- **content moderation:** *each node has full autonomy to remove content that is stored at their node that they deem to be inappropriate- content is backed up across a number of nodes, so each of their individual decisions combine to form a consensus moderation that parallels a central entity's moderation*
- **data permanence:**
    - *each node in the network maintains a list of signed hashes of the data it is currently storing on behalf of the network*
    - *nodes will ask for this list at every request they receive, and they will pick a handful of hashes at random and ask the node making the request to supply the actual stored data those hashes correspond to*
    - *if the node making the request fails to produce data that matches the hashes, their request is denied*
    - *this gives nodes an incentive to continually store the platform's data*

The remaining two requirements, **structured retrievability** and **storage limiting**, are closely linked and require a shared solution.

Replicating database functionality

Recall that in our peer-to-peer network, retrieving data is no longer as simple as connecting to a single server and querying their database.

Since data is stored across the network, the network itself is now our "database" so we need to "query" it to find the data we're looking for.

Our peer-to-peer "query" will actually entail a mapping from some metadata to a node on the network, since that is our analogous network operation to a database lookup using the metadata as a key.

This mapping will return the node at which the data described by the provided metadata is most likely to be stored- our equivalent of a database row entry.

It may be tempting to just derive a common method among nodes for generating metadata, and store all the data according to a particular mapping based on the metadata.

This would satisfy **structured retrievability** since users could easily find content from inputting their desired metadata- however, this arrangement would violate **storage limiting**.

This is because a mapping based solely on metadata doesn't carry any information about how much data a user has already stored on the network, and since each node only has knowledge of what it itself is storing, a malicious user could get away with taking up precious storage space at every single node on the network without consequence.

Storing content

To fix this then, we need a mapping which somehow encodes the amount of storage space a user has already taken up on the network.

Recall that each user can be uniquely identified by the ID of their node in the Kademlia DHT- this is a piece of information that is fundamental to each user on the network, that can't be falsified, and that is visible to anyone on the network that they interact with.

So if we use this ID as the basis of our data storage, we can then definitively track each user's storage usage across the network- we do this by agreeing on a mapping whereby users can only store their data at nodes that appear in a sequence of successive hashes of their ID.

For example, if each user was only allowed to have 1000 pieces of content uploaded to the platform at a given time, they would recursively hash their ID 1000 times and be allowed to store a piece of content at each of the nodes corresponding to those 1000 hashes.

If a node is then asked to store a piece of content for another user, they can very easily generate that user's list of hashes and check if their ID appears in the list before agreeing to store the content- if their ID isn't in the list, the node is vindicated in denying to store the content so that the user doesn't exceed their allowed content storage limit:

```
function shouldStore(localID, foreignID) {
     nxtID = hash(foreignID);
     i = 0;
     while (i < 1000) {
          if (localID == nxtID) {
               return true;
          }
          nxtID = hash(nxtID);
          i++;
```

```
    }
    return false;
}
```

With this mapping we've been able to incorporate **storage limiting** into our platform, but we've now lost our **structured retrievability** as a result, since we can't expect users to know the ID of the author of the content they're looking for and that's the only metadata that currently maps to content:

```
// note that "id" and "data" aren't inherently connected
// "id" is derived from the ID of "data"'s author,
// not "data" itself
DHT = Map<id, data> {
    { id: "asdh3671", data: "abc" },
    { id: "3nedbo22", data: "123" },
    { id: "eyuq1jwn", data: "Keane" },
    ...
};
```

Storing content pointers

In order to re-introduce **structured retrievability**, we thus need to establish a second mapping which maps from content-describing metadata to the node on the network where the content itself is stored, as given by our first ID-based mapping.

If we store these content "pointers" across the network, users can then find their desired content based solely on metadata describing that content, without ever having to consider a specific ID:

```
// "id" and "data" still aren't inherently connected
// but "id" for a "pointer" is derived from known metadata
// so we can always deduce "id" for a "pointer" ourselves
DHT = Map<id, data|pointer> {
    { id: "asdh3671", data: "abc" },
    { id: "bbds20j6", pointer: "eyuq1jwn" },
    { id: "3nedbo22", data: "123" },
    { id: "eyuq1jwn", data: "Keane" },
    { id: "54fdx12o", pointer: "yw22sa9u" },
    ...
```

```
};
function mapping2(metadata) {
     // eg: hash the metadata, or similar
      ...
     return id;
};
metadata = "Conall"           // we choose to search this
key = mapping2(metadata)      // = "bbds20j6"
location = DHT.find(key)      // = "eyuq1jwn"
data = DHT.find(location)// = "Keane"
```

This second mapping will vary slightly depending on if the user is pointing to public or private content, because the node being asked to store the pointer won't be able to validate metadata generated from private content since it will be end-to-end encrypted.

Pointing to public content

In the case of public content pointers, there will simply be standard platform-agreed text and image processing techniques with which a number of keywords will be generated from the content.

Each of these keywords will be hashed, and the resulting hashes will give the nodes at which the pointers to this content may be stored:

```
function contentPointerLocations(content) {
     IDs = [];
     textKeywords = processText(content.text);
     imageKeywords = processImages(content.images);
     for (txtK in textKeywords) {
          currentID = hash(txtK);
          IDs.push(currentID);
     }
     for (imgK in imageKeywords) {
          currentID = hash(imgK);
          IDs.push(currentID);
     }
     return IDs;
}
```

These pointer locations can be generated by any user looking for particular content, thus satisfying the property of **structured retrievability** for our platform.

However, note that since this mapping doesn't overlap with our first ID-based mapping, we are again at risk of losing our **storage limiting**.

This is because a malicious user could waste storage space by endlessly storing pointers to pieces of content that aren't present on the platform, simply by using the agreed-upon text and image processing techniques to generate valid keywords from their fake content.

In order to prevent this, we enforce a rule that a user who is looking to store a content pointer must produce a number of signed receipts of the content itself having been stored.

These receipts will be given to the user by the nodes from our initial ID-based mapping who are storing the actual content, following the storage procedure.

This means that nodes can safely store pointers while maintaining the platform's **storage limiting**, just as with the storing of the content itself:

```
function shouldStorePointer(localID, content, signatures) {
     for (sig in signatures) {
          if (!isValidSignature(sig, content)) {
               return false;
          }
     }
     for (id in contentPointerLocations(content)) {
          if (localID ≃ id) {
               return true;
          }
     }
     return false;
}
```

Pointing to private content

In the case of private content pointers, a slightly different approach is needed since the content will only be legible to the intended viewers, so metadata generated using the established text and image processing techniques will be useless.

However, unlike with public content pointers, we can assume that users who will be looking to retrieve private content will be aware of the author's ID, which means we can use IDs as the metadata.

This is a certainty since in order to gain access to the content, the users would have to have had a prior interaction to exchange keys for the content's end-to-end encryption, eg: during the "friending" process, when joining a group chat, or in real life.

With this in mind, a productive mapping for private content pointers would be for users to store pointers to private messages at some function of the intended recipient's ID, and store pointers to private posts at some function of their own ID.

This is very convenient since if a user wishes to "check their inbox" for messages, they simply plug their own ID into the private message pointer formula as the intended recipient, and they can check for new messages there.

Equally, if a user wishes to check for new posts made by a friend of theirs, they simply plug their friend's ID into the private post pointer formula as the content author ID, and they can check for new posts there.

Again, we now have a renewed risk of losing our **storage limiting**, especially since we can't simply rely on signed receipts of private content storage as we were able to do with public content.

This is because a node's ability to independently verify the keywords generated from public content was what implicitly tied the content's pointer to its signed receipt, but there's no equivalent here because the content pointer is no longer derived from the content itself, so the signed receipt has no merit on its own.

This means that a malicious user could store a single private message, and waste storage space by "sending" that message to an endless number of users, ie: by re-using those same signed receipts to prove that the message they're pointing to has indeed been stored on the network, which is true.

A node being asked to store this pointer has no idea if the intended recipient is actually a friend of the user or just a random person they derived to waste space, the only objective measure they have are the signed receipts which are actually still perfectly valid in this situation.

The solution that enables us to maintain **storage limiting** is to establish that private content must be stored with a plaintext "tag" specifying where exactly the pointer to the private content will be stored following the storage of the content itself.

An honest user will of course know exactly where the pointer will be stored ahead of time, simply by plugging their intended recipient's ID or their own ID into the corresponding formula for private messages or private posts, so they will have no issues adding the tag to their content.

Since the tag is added before generating the signed receipt, the signed receipt now restricts the private content pointer to a specific node on the network in exactly the same way as with the public content pointers.

Nodes can then ensure **storage limiting** even when storing private content pointers using nearly the exact same verification as before:

```
function shouldStorePointer(localID, content, signatures) {
     for (sig in signatures) {
          if (!isValidSignature(sig, content)) {
               return false;
          }
     }
     if (localID ≃ content.tag) {
          return true;
     } else {
          return false;
     }
}
```

Network periodicity

A subtle final touch we need to incorporate into our protocol design is to make sure that the locations where content and content pointers are stored change over time.

If they weren't to change and it was the same nodes who were being tasked with storing content from the same users or from the same metadata all the time, that could be dangerous for censorship or targeted attacks aiming to silence a particular person or topic.

All we have to do to incorporate this is to establish a network agreed-upon unit of time, keep track of how many such units of time have passed, and append that count to any hashes that are calculated during the network storage process.

Everything will still work in the exact same way, except that users will have to periodically re-upload their content to the network at the new locations- this is actually beneficial to some extent however, because if a user wishes to delete a piece of their content from the network then they simply stop re-uploading it at each network update, instead of trying to catch all the nodes who are storing their content at a time when they happen to be online.

Comparison with existing work

There are a number of existing projects that have some elements in common with **lupinfort**, but none which are wholly identical in purpose.

IPFS

The IPFS *[2]*, or Inter-Planetary FileSystem, is a peer-to-peer distributed file-system that is intended to serve as the foundation of the internet in the future.

Outside of some functional similarities, like the fact that it is structured as a Kademlia DHT and employs a public key infrastructure, the IPFS doesn't have much direct overlap with **lupinfort**.

This is mainly due to the IPFS targeting a much more general application, as opposed to being specifically built as a social media platform- to that end, content in the IPFS doesn't persist by default, it's backed-up only when requested meaning there is no baseline duplication, and the content isn't limited to specific structures so any file can be stored on the network.

Another significant functional difference is that the IPFS uses direct content-hashing for addressing, meaning that users can only find content if they know exactly what they're looking for, as opposed to only needing to know some keywords or a particular author.

For all these reasons the IPFS- despite being somewhat similar in its motivation- is a substantially different system to that which **lupinfort** sets out to establish.

Mastodon

Mastodon *[3]* is the largest decentralised social media network that's in operation today.

Unlike **lupinfort**, Mastodon is only decentralised and not peer-to-peer; any user can choose to host their own instance of the platform and act as a server, and the platform consists of a network of all these different instances.

While the idea is great and it has been relatively successful, it still suffers from a few shortcomings; the model of hosting instances still lends itself to issues of

centralisation, the platform is a Twitter-clone so only that style of short post is allowed, and the users who are hosting instances need funding to pay for their server upkeep so it's uncertain if the platform can forever be self-sustaining.

Again, it's still a fantastic example of a decentralised social media platform, but **lupinfort** will be attempting to overcome the issues it faces and become a more permanent solution, so it will differ greatly.

Minds

Minds *[4]* is a very unique social media platform which retains the centralised model, but has aspirations of achieving user empowerment and online equality by backing itself with a cryptocurrency incentive for its users.

Users receive tokens for engaging with the platform and for how much engagement their content receives, which is an interesting way of addressing the issues around data autonomy on social media platforms.

Its content is more similar to **lupinfort** in that it facilitates long-form article-style posts, but its centralised functionality is obviously vastly different to a peer-to-peer structure- Minds' decentralisation comes in the form of its blockchain-based reward system instead of its content network structure.

There have been some concerns recently that Minds, with its priority of user-freedom, is becoming a haven for some of the toxic Parler refugees, but it remains to be seen how much they will be identified with those attitudes, and the platform's divergence from corporate social media is nonetheless very interesting.

Ethical considerations

Given that- as with all peer-to-peer networks- there is no central regulation of **lupinfort**, its introduction raises a number of ethics concerns in addition to the ethical improvements that such a structure brings.

Exposure to harmful content

The downside to the lack of central regulation is that users in a **lupinfort** network are more "exposed" than they would be in a traditional centralised social media network.

This is because in traditional platforms, the central authority running the platform acts as a buffer between the users and the content they consume, and thus is able to shield users from undesirable content like gore, graphic violence, and- in the most extreme cases- from illegal pornographic content.

This filtering is only possible in a centralised platform because its owners have the ability to instantly remove undesirable content from their central servers, as well as having the organisation and resources to employ moderating staff to process reports from users flagging such content.

Conversely, in a **lupinfort** network it is by-design impossible to achieve such central coordination, so instead each user must act as their own moderator for the content they are storing for the platform at their own node.

If left unchecked, not only could this deter users who feel complicit in propagating unethical content, but it could also cause visual trauma for users who attempt to remove unethical content from their node.

With this context, we can appreciate how content moderation is very much a crux of **lupinfort** and see why without recent advancements in machine learning, this platform would be an impossibility.

Thankfully, there now exist a number of fantastic open-source machine learning projects ***[5] [6]*** which could be used in a **lupinfort** software implementation to filter out the vast majority of unsuitable content without requiring human intervention, alleviating much of the above concern.

Possible strategies may include automatically rejecting to store content containing images that exceed a certain probability of being of a sexual nature, or content whose text portion is algorithmically determined to be low-effort, or blurring images by default to allow for safer inspection of stored content.

The characteristics of the **lupinfort** protocol itself also help combat these risks; for example, the structure and size-limit of the content that is allowed on the platform is quite restrictive, which mitigates the potential for uploading abusive or copyrighted material, and the fact that the network's content routing refreshes periodically means that a given user is highly unlikely to be storing a piece of unsuitable content for very long on their node, regardless of how frequently they intervene.

All of these measures combine to offer a range of possible workarounds to the crucial concerns regarding content hosting, but they only pertain to the **lupinfort**-specific model of hosting content publicly- most controversy surrounding peer-to-peer networks in general actually stems from facilitating sharing of illegal and unethical content in private.

This is a separate issue in itself, and one that is still pertinent to **lupinfort** since all private data is end-to-end encrypted by default.

Spread of illegal content

My outlook on this issue, and my justification for **lupinfort**, is predicated on two connected factors- the regrettable prevalence of abuse in existing systems, and a philosophy that just because a new system isn't perfect doesn't mean all of its potential for good should be discarded.

It's an unfortunate fact that illegal and unethical content is being made available all the time via existing peer-to-peer networks *[7]*- if somebody wants to share such content badly enough, they are perfectly capable of doing so already.

lupinfort isn't providing any new technology that will magically induce people to produce more of this content, and in a properly-designed peer-to-peer network it is completely impossible to prevent this content from being exchanged.

To focus in on this issue as many policy-makers tend to do is therefore extremely unproductive- it overlooks all the important benefits that peer-to-peer networks offer, and abuses an issue that is best left to law enforcement and "real-world" policing as an excuse to stifle technological progress and user-empowerment.

The analogy I would use is to public parks; many illegal activities can take place in a public park, but to suggest that this would justify banning them is absurd- a realistic solution is to employ proper investigative techniques to identify offending individuals within the much broader context of their day-to-day operations and intervene by those means.

Just as public parks provide many valuable services to their genuine users, so too do peer-to-peer networks; **lupinfort** in particular is intended to combat exploitation of users by private companies, to restore data autonomy to users, and to create a perfectly sustainable and fair way of interacting via social media, and thus I would strongly argue that the positive impact of achieving such goals far outweighs the negligible theoretical increase in illegal activity as a result of the protocol being introduced.

Censorship

On a similar note, a closely related ethical consideration of **lupinfort** is its handling of censorship and freedom of speech.

This is an issue which is always the subject of much debate, often because many people with horrifically backwards views claim that they should be allowed to say whatever they like and not have any consequences for it.

This is a mindset that has been dangerously leaking into online platforms in the last few years, with sites like Parler and now Minds to some extent becoming hosts to nasty discussion from people who would claim to be "championing free speech".

In reality, nobody has gotten it right so far- while I would be more on the side of how the likes of Twitter, Youtube, etc. have handled censorship, and while I believe that the extent to which people dramatise these companies' censorship is far overblown, I can still see on paper how giving private companies such power is potentially harmful.

On the other hand, the solution is not to go completely in the opposite direction and eliminate all oversight, as aforementioned sites like Parler have done.

The fairest solution, by definition, is to let everyone decide what's allowed- and that's exactly what **lupinfort** achieves by randomly distributing the responsibility of storing content and giving each node autonomy over censoring their own node's storage.

If majority of nodes that are asked to store a piece of content deem it to be inappropriate, then they all remove it and it becomes essentially impossible to find, "censoring" it.

The distribution of content also reshuffles periodically so that the nodes being asked to store each piece of content will truly represent a random sample of the platform, even over time.

I think this method of censorship can be objectively seen as the most sensible, and completely inarguable; if majority of people dictate that something is right or wrong, then that's how it should be and that has to be accepted, in every facet of society.

Conclusion

I believe that there is room for great improvement and great positive change in the social media space, it's just that the right option hasn't been made available yet.

It's imperative that social media becomes a part of our lives that is owned and controlled by us, free from control of private interests; it's become such a key part of our society that it merits integration with our personal identities, and the only reason why such a proposal wouldn't be sustainable or healthy currently is because that space is in the hands of corporations.

It makes perfect sense to transform social media from something that is a "product" with many negative connotations, into a wholesome, welcome extension of humanity in many of the same ways that our personal interactions can be.

In setting out the **lupinfort** protocol, I think that I have offered a roadmap to achieving such a goal- a social media presence that doesn't belong to anyone but yourself, that is governed by a shared responsibility of everyone across the globe in equal stakes, and which is as true a reflection of society as possible with no predisposition to corruption or malintent.

Sources

[1]
Kademlia DHT:
https://www.researchgate.net/publication/2492563_Kademlia_A_Peer-to-peer_Information_System_Based_on_the_XOR_Metric

[2]
IPFS:
https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.pdf

[3]
Mastodon:
https://docs.joinmastodon.org/

[4]
Minds:
https://cdn-assets.minds.com/front/dist/browser/en/assets/documents/Minds-Whitepaper-v2.pdf

[5]
NSFW.js:
https://github.com/infinitered/nsfwjs

[6]
BERT:
https://github.com/google-research/bert

[7]
Sharing of illegal content in existing p2p networks:
https://pubmed.ncbi.nlm.nih.gov/24252746/