BUILDING NOSTR

A book about Nostr protocol design

By @hodlbod

Table of Contents

Introduction: Nostr is for Builders **6**

1. Complex Problems, Simple Solutions **7**

The Economics of Censorship 7

A Path Forward 9

A Simple Protocol 10

2. Events and Kinds **12**

A Single Data Type 13

Numbers, not Names 14

More Kinds, Less Ambiguity 15

Time is Hard 17

Content and Tags 18

Behavior and Data 19

Behavior and Data Part Deux 21

Deleting Events 23

Filtering Events 23

A Light Touch 24

Backwards Compatibility 26

3. Signatures and Identity **28**

Information Wants to Be Free 28

Publicity Technology 29

Dis-intermediating Data 30

Your Very Own Number 31

Agency and Identity 32
Identity Webs 33
Holding Keys 35
Naming Keys 37

4. Relays are Repositories 39
Why WebSockets 39
Multi-Master 41
Functional Relays 42
Replication and Routing 44
The Outbox Model And Friends 45
Bootstrapping 49
Relay Hints 51
Content Migration 52
Relays as Transport 54
Relay as Design Pattern 55
Relay as Proxies 56
Wrapping Up 57

5. Radically Open 58
Politics and Protocol 60
Thinking in Public 63
Backwards Compatibility 64
Hackability 66
Application Ecosystems 69

6. Value for Value 72
Money is Time 73
Customers, Patrons, and Participants 74
Paying It Forward 76

Zaps and Nutzaps 78

7. Communities 81

Communities are for People 81

Digital Architecture 82

Social Clusters 83

Group Chats 84

Discussion Forums 85

Owned Communities 88

Commons 89

Roles and Spaces 90

Managing Complexity 92

Interface Complexity 93

Appendix A: Alternatives to Nostr 95

Matrix and ActivityPub 95

Scuttlebutt 96

Pubky 97

Bluesky 97

What Makes Nostr Different 98

Starting Over 99

Appendix B: Resources 101

NIPs 101

Protocol development 101

App lists 101

Introduction: Nostr is for Builders

Nostr stands for "Notes and Other Stuff Transmitted by Relays." It was created as an answer to social media censorship by big tech companies, but has since become much more. Chances are, if you're reading this book you're already interested in Nostr development. Maybe it's the cryptographic identity that hooked you, or the network effect that application interoperability makes possible. Maybe it's the social media use cases, or the "other stuff" that brings you here.

Nostr is many things to many people — the architecture is simple, but a diverse application ecosystem has grown from it.

Nostr is an open protocol, which means anyone can build on it and anyone can extend it. It relies on digital signatures for data authentication and self-hosted or otherwise user-aligned infrastructure for hosting.

The simplest way to think about it is as a Twitter alternative, but one where it's very difficult for any single actor to de-platform users. But Nostr is far broader than just "social media". Other applications from media hosting to decentralized marketplaces, reviews, chat rooms, calendar events, and even virtual pets have found ways to leverage Nostr in their own domain.

This book is for builders — for people who are excited about decentralization and have ideas for a product or service that would benefit from interoperability and user-oriented software. This book presents a philosophical overview of Nostr with an emphasis on *the responsibility developers have to keep it decentralized*.

This book will cover some of the basics in passing, but primarily as a stepping stone to explaining a paradigm for how to use the protocol effectively. For a more technical overview of the protocol itself, take a look at NostrBook.

Like Nostr itself, this book is not a comprehensive guide for how Nostr works (or ought to work). It is only a set of opinions, ideas, and guidelines for people who wish to create software for themselves or others — software that enriches its users rather than exploiting them.

1. Complex Problems, Simple Solutions

To begin with, I want to present a summary of how the profit motive of internet businesses creates a system of incentives which drives censorship and surveillance capitalism, hurting users and undermining corporations' own value propositions. Next, I'll describe what sort of solution Nostr actually is, and end with quick survey of how Nostr works at a high level — just so we're all on the same page.

The Economics of Censorship

Nostr was originally designed to be "censorship-resistant". But this is only one part of a bigger vision. Nostr is designed to *promote the digital freedoms of users* — things like privacy, access to information, anonymity, freedom of association, and credible exit. Because of how cryptography was adopted (primarily by institutions) in the early days, the internet is no longer aligned with the interests of its users, but has largely been captured by "platforms".

As a result, we have a state of rent-seeking in which the users who contribute all the value that the internet has to offer are exploited, undermined, manipulated, deceived, and sold by the custodians of that value. This is a far cry from the humanitarian dream of the early internet. In the words of Tim Berners-Lee,

We create the Web, by designing computer protocols and software; this process is completely under our control. We choose what properties we want it to have and not have[...] The goal of the Web is to serve humanity. We build it now so that those who come to it later will be able to create things that we cannot ourselves imagine.

In the fifteen years since this was written, the internet has only become more predatory, and proprietary. Everywhere you go, you're being tracked — maybe for the purposes of understanding your behavior to serve you ads, maybe for reasons more convoluted, but always in the interest of the ones doing the tracking.

This isn't to say the internet isn't a net positive to those who use it. The internet is the single most successful technology in human history for promoting access to information, alternative community, and trade. But it is not what it should be. Reforming the internet requires active investment from those who stand the gain or lose the most from how it evolves — its users. No one can do this for us; *we* have to hold companies and platforms accountable ourselves. And in order to do that, we need to understand why censorship happens.

Censorship is not the product of mere arbitrary exercise of power, but of incentives. In authoritarian regimes, censorship is implemented to protect the regime's power. In business, censorship is implemented to protect the business' revenue.

Traditionally, products are sold directly to customers, who then get access to the good or service they've purchased. This business model can transfer to some extent to the realm of information-based products, particularly in a business-to-business setting, where information can be justified in terms of greater productivity.

But consumer software is an entirely different story. If your users are only there to improve their quality of life through access to entertainment or communication, and the marginal cost of production trends toward zero for digital goods, then the price will also trend toward zero because competitors can always offer the same product at a lower price.

Combine that with the imperative for platforms to aggressively acquire users, and you get a strong downward pressure on the upfront pricing of software products. "Network effect" is the idea that a network's value does not grow linearly with the number of users, but quadratically with the number of *connections between users*.

Building a large network requires significant resources in order to fund growth or stay ahead of competitors. These resources in turn must be provided by people with capital. As a result, businesses that rely on scaling network effects are typically funded by investors and oriented at a return on investment. This includes social media platforms and similar businesses, such as search engines, marketplaces, and entertainment platforms.

Capitalists want to see the businesses they've invested in make a profit. The best way to make a profit in a market, according to Peter Thiel's *Zero to One*, is to establish a monopoly. Because user retention depends on network effects, market saturation is always the goal, and switching costs must be high — companies must "capture" their users.

The art of creating this moat seems to have been perfected. Facebook has been around for a long time, as has Google. In contrast to early internet companies like MySpace, Yahoo, and AOL, these second-generation internet giants have learned how to simultaneously retain and monetize their users.

Very little of these companies' revenue comes from direct monetization. Instead, these big tech companies monetize their users by making them the product. Users have two things that platform owners can sell: their attention and their data. Platforms can only sell these things because they have access to them by virtue of being intermediaries. Platforms subsist by inserting themselves into private or public relationships in order to siphon off these intangible goods.

This works because the majority of internet users either don't believe their attention or data are worth anything ("why do I need privacy? I have nothing to hide"), or feel helpless to opt out of the attention economy. Seeing a billboard in your peripheral vision or having cameras record your license plate seem hardly comparable to paying a monthly fee for road use. Whether you care about being recorded or not, you're still going to use the roads.

It's true that data and attention are not particularly valuable on an individual basis. But they're massively powerful in the aggregate. "Big data" is not mere information, but *the ability to predict patterns of behavior*, which can then be used at scale to manipulate people — to buy, believe, and act.

Advertising is not just about informing willing buyers about products and services, it's about conditioning people to act in predictable ways. Social engineering (like all engineering) is progressive and totalizing — unless restrained. The cost of giving away our attention and data to these internet intermediaries is, ultimately, the loss of our free will.

This is particularly true because users are progressively getting less and worse service in exchange for their time and attention. Because big internet companies have access to economies of scale that individuals don't, they have the ability to asymmetrically impose complexity costs on users who fall outside of the streamlined "ideal customer profile" they've built their business on. Quoting from this blog post:

Bigger players like corporations and governments gained a big systemic advantage over individuals: they make the rules but they don’t bear the costs of the rules breaking. They increase the complexity of everyday life and instead of being punished for it, they are rewarded for it.

A Path Forward

This slide isn't inevitable. But it's also human nature — everyone wants something for nothing, and so internet users have voluntarily given up their free will to gain connectivity, entertainment, and efficiency. Simply pushing back is not enough — the problem is structural, and time has allowed its beneficiaries to entrench their position.

This problem needs more than a technical solution — politics, culture, community, and capital all have roles to play here. Nostr's role is to offer tools that individuals can use according to their own values, and for their own interests. Nostr makes it possible for internet users to preserve their own digital sovereignty and hold tech platforms accountable.

My personal hope for Nostr is that it will aid in the restoration and cultivation of human flourishing and agency in a world increasingly mediated by digital technology. Nostr does not solve every problem. The ones it does solve, it sometimes solves poorly. Nostr is not a panacea, it is a toolset that can be applied to our benefit or to our detriment, but which has certain intrinsic qualities that the internet desperately needs right now.

I want to emphasize that it is only *through people* that any of the internet's ills can be cured. Nostr is not a comprehensive system for restraining digital evil (such a thing would be itself totalitarian), but a way of allowing individual and community agency to be exercised in solving problems germane to individuals' and communities' own particular circumstances.

A Simple Protocol

Nostr is in many ways a "dumb" solution to a complex problem. Incentives, structures of power, and technological ecosystems are irreducibly complex, and impossible to understand in detail. But there are key architectural patterns that the modern internet is built on which enable and promote its particular dysfunctions — most importantly, the centrality of servers for both data storage and authentication.

Instead of using server-based authentication, Nostr leverages cryptography in order reduce the role of servers from powerful hubs to disposable, user-aligned repositories. To join the Nostr network, all a user has to do is unilaterally generate a `secp256k1` cryptographic key pair.

Any number of these cryptographic identities can be used in parallel for whatever purpose — whether to partition user data, create pseudonyms, or to encrypt data. Most importantly, key pairs can be used to create digital signatures of content, which are then embedded in data structures called "events" alongside the user's public key, the event's hash, and some other information.

Once created, events are then transmitted to other people, primarily via "relays," which are simple WebSocket servers, although they can also be sent over Bluetooth, emailed as attachments, or even written down and sent by carrier pigeon. The relay protocol is very simple, dealing almost exclusively with sending, receiving, rejecting, and requesting events. This makes it possible to create multiple interoperable implementations.

The content types (or "kinds") the protocol describes are very loosely coupled, allowing a wide range of applications (or "clients") to be built that can talk to each other. Not every client needs to implement the entire protocol though — Nostr is ideal for building small apps that do one thing, and which don't have switching costs.

Here's an example of what this architecture makes possible: Imagine following someone on your microblogging platform of choice, then opening your blog reader and all of their posts are already there. You highlight some text and type a comment. Later, you go back to your microblogging platform and read replies to the comment you made — written by people who follow you in dozens of other apps.

In many ways, Nostr has the potential to be an upgrade to the entire internet. By decoupling data storage (relays), identity (keys), and functionality (clients), it becomes much easier to compose whatever pieces you might need in new and interesting ways. That's a big claim, and there are definitely some things that Nostr *can't* do well.

In terms of the CAP theorem, Nostr has no consistency guarantees, but plenty of availability and partition tolerance. Nostr is essentially a distributed database which allows every node to operate independently without any coordination. By copying the same (signed) data to multiple relays, it will always be available even if some nodes go offline or ban the author.

To recap: Nostr combines cryptographic identities, interoperable data storage, and an open ecosystem of content types to create a system where neither user identity nor user data can be captured by a single entity. This subverts the central role that servers have in the modern internet, placing users at the center, rather than corporations.

2. Events and Kinds

The term "NIP" stands for "Nostr Implementation Possibility", a term the community uses to refer to Nostr protocol specifications. These specs might describe content types, behavior for relays, expectations for clients, or sub-protocols for other related services like signers, wallets, and "data vending machines" (services which perform arbitrary computation and expose an event-based interface).

When writing Nostr specifications, keep in mind that a key part of the Nostr design ethic is: *trust users, trust developers, and keep things simple*.

Protocol documents are often bloated with clarifications for every edge case, resulting in standards no one actually reads. Nostr specifications should be brief and to the point so that regular people (rather than spec wonks) can understand them and hack on the protocol. Again, in the words of Dave Winer:

I write for people who have brains, like to think, are educated, care about interop. I understand that people reading specs are not computers.

Because nostr is open, anyone can write a spec and publish it anywhere they want. Unfortunately, this also means that there's no comprehensive list of specs anywhere.

While Nostr is about as anarchic a protocol as you can get, there remains an irreducible political component to protocol design. Software cannot be interoperable if communication doesn't exist. Now, this communication might simply take the form of reverse-engineering other people's events and code, but communication is (obviously) improved through the use of language.

Right now, the best place to collaborate is on the github repository. This is a carefully curated collection of Nostr specs, and the place the most Nostr developers participate (open pull requests are a great place to get feedback and signal support for emerging specifications). There has been some work towards creating a solution that exists purely on Nostr where we can get permissionless participation, a comprehensive list of specs, and a system for consensus, but so far a complete solution hasn't really emerged.

With that in mind, let's get into events — the "what" of Nostr.

A Single Data Type

All data on Nostr (with very few exceptions) is contained in "events". Here's an example of an event:

```
{
  "id":
"b13ef435ad614ee37b74f6d6e8055c303a79b95354dcb65b075fb3964a6731e0",
  "pubkey":
"3bf0c63fcb93463407af97a5e5ee64fa883d107ef9e558472c4eb9aaaefa459d",
  "created_at": 1682377261,
  "kind": 1,
  "tags": [
    ["p",
"73c7f6d5bb599bb7d7cee84c72e89dbd549df53da522ed6c7611055cc0db64bc"],
    ["a",
"31337:73c7f6d5bb599bb7d7cee84c72e89dbd549df53da522ed6c7611055cc0db64bc:
5jqxarp0z1kr7yem"]
  ],
  "content": "is this Lana Del Rey?",
  "sig":
"91d7ae3d5a7ef72e917b52829f0dd76538be2b1720f3c09d0bf71131cffd81f18b89956
ef2bab9ea44e656edd96529d0d3fc71c29e305a5378fb4087bfb08f7a"
}
```

Here's what those fields mean:

- `id` is a hash of the event (serialization is defined in NIP 01)
- `pubkey` is the public key of the event author
- `created_at` is the second-granularity unix timestamp for when the event was created
- `kind` is the content type of the event, which determines how it should be handled
- `tags` is a list of lists of strings, which provide metadata for the event
- `content` is a (usually) human-readable representation of the event for display
- `sig` is a `secp256k1` cryptographic signature of the event `id`

The core of the event is the cryptographic properties: `id`, `pubkey`, and `sig`. These three properties make it possible to verify the event author, and refer to the event by content hash rather than location. We'll talk more about how to work with cryptographic identities later, but it's hard to understate just how important these properties are. For now though, I want to get into a different topic: data modeling.

Numbers, not Names

Every event has a "kind," which is a 16-bit integer. An event's `kind` determines what that event means (it's "content type") and how it should be interpreted. Using numbers instead of names is an unconventional choice, but it has certain benefits.

In a distributed system built on signed data, names can't change. Once you choose a particular term, you're stuck with it forever. The nice thing about integers is that they're meaningless, which allows for any number of subjective interpretations of a single standard, while keeping its meaning intact.

Names have a weakness, in that they import meaning with them into any new context. This meaning might vary from person to person, or the person who chose the name might have made a poor choice. This results in confusion about the purpose and semantics of a given data type.

Numbers carry no meaning (except in a memetic sense, like how Nostr uses `kind 1984` for content reporting). This frees up every implementer to assign their own term to an event based on their understanding of it. If their understanding of the content type changes, they can change the term they're using for it without affecting its actual meaning. This allows linguistic conventions to develop independently of the technical characteristics of a thing.

Here's how Dave Winer explains this approach to naming things:

I once led a standards discussion beginning with this rule: We always had to come up with the worst possible name for every element. That way when someone said "I think foo is better" (and they did) we could all laugh and say that's exactly why we won't use it.

It totally doesn't matter what we call it. We can learn to use anything. There are more important things to spend time on.

Think of people whose first language isn't English. To them the names we choose are symbols, they don't connote anything.

An alternative approach to creating collision-proof names is that of "reverse domain notation." This is a great way to create an identifier that does not collide with anyone else's, but (apart from having the same semantic difficulties as unqualified names) it also implies some level of ownership of a given content type.

But Nostr is a permissionless protocol. Any event that is signed and sent is valid simply by virtue of existing (at least to the extent that it is accepted as valid by a meaningful proportion of the network — more on that in chapter 5). Because there is such a diversity of content types, and because the protocol is open for extension, it's impossible to validate every event kind based on data structure, which naturally limits validation to hash and signature validation (and a few other things).

This is very different from how we usually think about protocols. Normally, protocols use natural language to describe data structures or processes that must be followed. On Nostr, kinds are their own vocabulary. Because they are numeric, they have no metaphorical referent, which means that usage determines meaning.

This is the same way that natural language works. Usage evolves over time, resulting in a mess of neologisms, malapropisms, idioms, and ambiguities. Dictionaries were invented only as a way of understanding language as it already exists. The Oxford English Dictionary includes not only definitions, but also the history and etymology of words, defined by actual use.

The same word might vary from place to place. You might have the same word with different meanings in different places. Language is organic and very difficult to systematize. Nostr embraces this ambiguity — which is admittedly a risky choice, since we don't know if software-mediated language can evolve in the same way as human-to-human language.

But because of this design choice, Nostr is a very humble protocol - it is little more than an empty shell which users can then fill with content types that solve their use cases.

More Kinds, Less Ambiguity

This isn't to say that there aren't conventions that should be followed when designing new content types.

In Structure and Interpretation of Computer programs, Alan Perlis famously claimed:

It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures.

Because nostr is a social protocol, it depends on network effects, not just of users, but of standards. The more implementations that use a single kind, the more users are in turn able to interact. So when creating a new standard, check to see if there is an existing spec and follow it if at all possible. Dave Winer again:

If you want to add a feature to a format, first carefully study the existing format and namespaces to be sure what you're doing hasn't already been done. If it has, use the original version. This is how you maximize interop.

Nostr abides by this rule to the extent that implementations prioritize interoperability over functionality. As the NIPs repo stipulates, "there should be no more than one way of doing the same thing". Fewer data formats means less complexity, and greater interoperability.

This is a corollary to this, which at first glance might seem contradictory, but which reinforces the ability to deal unambiguously with a given data structure: don't pretend different data structures are the same thing. We see this mistake in JavaScript, where Arrays are actually a special type of Object. This is not what Perlis meant.

To apply this to Nostr, if there are two approaches to the same problem that are equally valid, use the one that has the most adoption. At the same time, if there are two use cases that are very similar but differ in minor ways, it's almost always a better choice to split it into multiple kinds. For example, NIP 52 uses separate kinds for time-based and date-based events. This not only reduces ambiguity, but also allows clients to work more easily with one or the other content type.

Some data might make sense to group together in a single event, for example kind profile metadata. But the more data is attached to a single event, the more contention there will be over writing to that event, and the more conflicts become possible. If I had to redesign profile metadata today, I would create separate events for user name, profile picture, bio, lightning address, nostr address, and every other field on the event. This would be marginally more data for clients to download, but it would prevent data loss in the event of incomplete synchronization.

When choosing whether to create a new kind or adopt an existing one, it's important to understand that the "one way" rule is only an *ideal*, and can still be violated if design goals vary, or an existing spec is broken in some crucial way. Just be prepared to defend your divergence from convention if you want other people to migrate to the new way of doing things.

Time is Hard

There are an incredible number of falsehoods programmers believe about time. Dealing with time is inherently difficult, especially in a distributed system which has no single time-stamping authority.

For this reason, Nostr's core protocol entirely punts on the problem — the `created_at` timestamp on events is supplied unilaterally by the event author, who can falsify the timestamp with no real constraints.

There are certain mitigations for this, for example NIP 03 allows for the use of Open Timestamp Attestations to prove an event was published after a certain point in time. In general though, Nostr chooses to solve this problem by moving it from the technical to the social layer. In other words, if you don't trust someone to be honest about their timestamps, why are you reading their content anyway? Reputation is key in social media, and provides a natural solution to many technical problems.

Nostr leans into this further by only supporting second-granularity timestamps. This is consistent with the very simple, human-oriented approach to social media that Nostr aims for. On a human time scale, sub-second granularity rarely matters. The temptation with millisecond granularity is that you might expect to be able to reliably sort events, but in a distributed system, that's not actually possible without centralized coordination. Clock skew across different computers is commonly more than one second, so messages may show up out of order even with millisecond-granularity timestamps.

A possible solution to this is to use something like vector clocks where every piece of data increments a conflict-free counter, or to build linked-list style data structures where every event refers to a previous piece of data in the chain. By default, Nostr does not specify a mechanism for this, which allows for natural solutions to emerge for each individual use case. Some content types form a natural tree structure, others explicitly provide affordances for enforcing order.

Dealing with time in a distributed system is frequently use-case dependent. By providing a very weak mechanism for working with time, Nostr leaves the door open to use-case-specific strategies for working with time.

Content and Tags

Finally, we have `content`, and `tags`. An event's content is generally a human-readable payload, while tags are structured data.

In reality, content isn't always human-readable. There are several event kinds that encode JSON or encrypted data into the content field. That pattern should be seen as an exception — this precedent was set early in the protocol when conventions for designing events weren't well established. Data should generally go in the tags array, and human-readable information should go in the content field.

You might be wondering why tags is a list of lists, rather than an associative data structure. The advantage of a nested lists data structure is 1. you can include the same key twice, and 2. tags preserve order. You see similar data types with ordered dictionaries in Python, and URL query parameters. This allows for more flexible data modeling than simple dictionaries would provide.

Tags are always an array of strings, not numbers or composite data structures. This keeps parsers simple (at the cost of sometimes doing dumb things like encoding numbers as strings or nesting JSON-encoded strings in JSON data structures). Usually there are at least two entries, conventionally (though not necessarily) being treated as key/value pairs. Some specifications include four or more positional items in a tag, while other include only one.

For example, the NIP 70 protected tag `['-']` has no values. The presence of the tag merely indicates that relays should only accept such event if they're published directly by their author.

Another example is the NIP 18 `q` tag, which is used to reference a quoted event. This tag contains an event ID, a relay URL where the event might be found, and the pubkey of the event's author, for example: `["q", "<event id>", "wss://relay.example.com/", "<pubkey>"]`.

In general, tags should be as short as is reasonable. Two to three entries is all you really need; if you have more than that, you're probably trying to pack more data into a single tag than really belongs. A better approach is to create other tags that form an index that can be used to look up additional metadata. This makes it easier for consumers to parse the tags array instead of having brittle positional arguments.

For example (this isn't how Nostr actually works), the above `q` tag could be split into `q`, `relay_hint`, and `pubkey_hint` tags:

```
[
  ["q",
"29916f5671eb3f93ee1d3f655ef14109591f6905ff1328a76fccc27bf6b449a6"],
  ["relay_hint",
"29916f5671eb3f93ee1d3f655ef14109591f6905ff1328a76fccc27bf6b449a6",
"wss://relay.example.com/"],
  ["pubkey_hint",
"29916f5671eb3f93ee1d3f655ef14109591f6905ff1328a76fccc27bf6b449a6",
"12fec3c26b3efdaa521db0bd02056cca90d401aafff0233635f78da2f4cb8d44"]
]
```

The benefit here is that new information about the event ID can be added associatively. It also allows for adding multiple values for a single key (like if you wanted to include multiple relay hints). The downside is that this pattern makes events more verbose.

This is the same trade-off as exists in programming languages when choosing named vs positional arguments (for example python's `*` construct). Named arguments make it a lot easier to maintain backwards compatibility as a function's signature evolves.

Behavior and Data

Tags can fall into one of three categories: data, filters, and behavior. These three things are often intermingled and hard to differentiate. This terminology is my own, and not reflected in any specifications or event structure, but I think it's useful for understanding how different design concerns interact.

"Data" tags are primarily useful for handling or displaying the event. "Filter" tags might also be data tags, but are especially useful for filtering and retrieval. "Behavior" tags introduce additional context that determines how implementations should handle an event, independent of the specification which defines the event's kind.

An example of a "data" tag is the NIP 92 `imeta` tag, which includes metadata for a URL for display. Because `imeta` is not a single letter, most relay implementations are not going to allow filtering by `#imeta` (which wouldn't make sense anyway). `imeta` is only useful for enhancing the rendering and handling of a given event after you've already retrieved it.

In contrast, `t` tags are useful not only as data, but as filters, since they represent "hashtag" or "topic". Since they are a single letter, it's possible to filter events by topic.

Finally, tags like NIP 40's `expiration` tag are entirely orthogonal to the semantics of an event's kind, and can be applied to any event in order to request special handling by relays.

These three types of tag are subtly different; as a result they were conflated when Nostr was originally designed. I believe this was a design mistake which introduces conflicts into the semantics of certain tags.

The difference between "data" and "filter" tags isn't too consequential because filtering tags are often useful as data as well. Plus, the two are somewhat differentiated based on whether the tag key is a single letter or multiple characters (although this unnecessarily couples tag name and purpose — it may make sense at times to index multi-character tags).

Behavior tags, on the other hand, are not defined by an event kind's spec, but by some other NIP, and can be applied to any event. Here are some examples:

- NIP 70 defines a `-` tag, which specifies that relays should only accept directly from its author
- NIP 29 defines the `h` tag, which is used in to publish an event to a "group" or "room"
- NIP 40 defines the `expiration` tag, which is used to request deletion after a certain amount of time
- NIP 89 defines the `client` tag, which can be used to indicate which client published the event

These tags in particular aren't too problematic since they're not re-used in other places, but there are some tags which are defined differently by different NIPs. An example of this is the `e` tag, which is used in the following ways:

- To indicate reply/reaction parent (NIPs 10, 25, 17)
- To indicate which event a wiki article was forked from (NIP 54)
- To request a merge into a wiki article (NIP 54)
- To build a transaction history graph (NIP 60)
- To reference auctions and bids (NIP 15)
- To indicate any number of different event kinds in lists (NIP 51)
- To indicate the object of a report (NIP 56)

- To indicate moderator approval of a post (NIP 72)

That a single tag can sustain so many different meanings is a testament to the effectiveness of the `kind` system.

This would all be fine, if `e` tags didn't *also* indicate the "mention" of an event by ID, which is specified in old versions of NIPs 10 and 18, and 23. This pattern of "mentioning" events by `e` tag is intended to be applicable to any event — but this comes into conflict with all the special meanings enumerated above, which means it's impossible to unambiguously use the "generally applicable" meaning of the `e` tag on any event kind that defines `e` tag semantics.

This problem could have been solved by separating data, filter, and behavior tags into separate fields on an event, rather than coupling a tag's name with its semantics. One `e` tag could be placed in a `filter_tags` field, and a different one in the `data_tags` field. But this is just me imagining alternate futures. In practical terms, this problem is just something to keep in mind when designing tags that can be applied to "anything", especially when they need to be indexable.

Here's a simple heuristic for dealing with overloaded tags: when resolving the meaning of a tag, always first look at the specifications for the event's kind, which has the right (being the most specific spec) to override any general tag behavior. Only secondarily should you look at specifications that define tags in a broad sense.

Behavior and Data Part Deux

Another example of this data/behavior design flaw is the idea of event kind ranges.

The idea of an "event" is that it is a self-contained, irrevocable fact. But for some use cases, like `kind 0` profile metadata, only the most recent event by a given pubkey is really useful. So, in order to avoid a bunch of redundant profile metadata events, it was decided early on that kind `0` should be "replaceable". In other words, every time a user publishes a kind `0` event, the relay should delete all the old kind `0`s from that pubkey.

This freed up room in the database, but it also eliminated the ability to fetch multiple versions of profile metadata, which meant that race conditions that occur as a result of updating user metadata in multiple places at the same time could no longer be resolved except by special purpose relays or services that retain all versions of profile metadata.

Events are amazing because their ID is the hash of their content, which makes them a "referentially transparent" data type. In other words, events are "content addressable" — if you have an event ID, you know the event is never going to change underneath you.

But when replaceability was introduced, a new way to refer to an event even if the content changes became necessary. And so the "address" was born. An address is a string in the format of `kind:pubkey:identifier` which points to all events with the same key, pubkey, and identifier.

This solution was also applied to kind `3` follows, and seemed useful enough to make into a general pattern. As a result, different event "ranges" were invented in order to provide certain variants of this behavior.

Kind `1` through `9999` (except `3`) are "regular" events with no special behavior. Kind `10000` through `19999` are "replaceable" events which have no "identifier" address component. The idea with these is that users can have only one event of each "replaceable" kind. Kind `10002` relay selection lists are an example — when you publish a new event of that kind, it replaces any previous ones.

The next range is kind `20000` to `29999`, which indicate "ephemeral" events which are addressable in the same way but have the additional behavior of not being retained by relays - they're deleted immediately after being broadcasted to other clients that are listening for them.

Finally, we have kind `30000` through `39999`, which are called "parameterized replaceable" events. These act the same way as "replaceable" events, but are partitioned by an identifier contained in the event's `d` tag. This allows the user to create multiple replaceable events for their pubkey. For example, editable kind `30023` blog posts allow you to "update" a post by creating a new event with the same `d` tag as the original.

This all makes sense, but it has a major downside — it couples kind numbers with behavior, regardless of what spec the kind is defined by. You might think this is fine because specs can just choose a kind that is in the correct range. And that's mostly true, except that these retention/replacement policies are really orthogonal to the data type represented by the kind. It's entirely possible to conceive of multiple follow lists, or a replaceable microblogging event.

This behavior really should have belonged in behavior tags, which would allow users to select retention behavior for any event regardless of content type. In fact, we have an `expiration` tag, which is a superset of kind `2XXXX` ephemeral event functionality, making that entire kind range essentially redundant. Similarly, any event could be made replaceable by adding a `d` tag (set to an empty string, if regular replaceable functionality is desired).

But I digress. We might get these extensions in the future, but for now we have to work with the limitations of kind ranges. My point is, when you're defining new event types, you should have a slight bias towards regular, non-replaceable events because they don't break referential transparency.

Deleting Events

One final bit of behavior that's worth mentioning in this context is deletion requests. These are `kind 5`events that can either `e` tag an event's ID or `a` tag an event's address. Relays receiving a `kind 5` event are expected to delete all matching events — provided the author is the same.

It's important that these be considered deletion *requests*, not actual deletions. Relays are not obligated to honor the request (although in practice many do).

This is the thing about signed data. Once it's published, there's no way to un-sign or un-publish the data. Anyone who has a copy can keep it. This is great for keeping public figures accountable for their words and actions, but not great for taking something down that shouldn't have been posted. Care has to be taken when using signed data. This is true of unsigned data as well, which can always be screenshotted or copied, but signatures remove deniability. This is a good trade-off for public broadcast social media, but it is something to keep in mind.

Filtering Events

Another concept that is closely related to events is that of filters, which are a data structure that allows clients to request specific events from relays. A filter is a dictionary with one or more of the following fields:

- `ids` is a list of hex-encoded event ids
- `authors` is a list of hex-encoded pubkeys for matching event authors
- `kinds` is a list of a kind numbers
- `since` and `until` are an second-granularity unix timestamps for filtering based on `created_at`

- `limit` is the maximum number of events relays should return in the initial query

In addition, filters can be applied to event tags by prefixing the tag name with an octothorpe (the "#" character) and adding it to the filter - for example, `#p` can be set to a list of pubkeys in order match events with a matching `p` tag. Relays generally only support filters for single-letter tags in order to reduce the number of database indexes that have to be maintained.

There are some other less common filter extensions:

- Prefix matches allow for matching event ID, pubkey, or tag value by partial value
- An additional `search` property is defined by NIP 50 and implemented by some relays
- Negative matches have been proposed, but rejected because of the impact they would have on relay performance

A Light Touch

Dealing with other people's signed events is always going to require some adversarial thinking. Events can be malformed, either from a buggy implementation or from someone trying to attack software applications. A `p` tag might have an invalid or missing value, or a relay hint that is trying to spy on users. Event `content` might include HTML injection attacks, or illegal content.

Distinguishing between when to discard an event and when to try to handle it anyway is difficult. The most important thing when handling events is to protect the user from attacks. Everything else should be handled in a fault-tolerant way, assuming incompetence rather than hostility.

It's important to note that Postel's "be liberal in what you accept" has two different possible interpretations here. Fault tolerance does *not* mean repairing data that contradicts established conventions. A common example of this is when clients publish events with bech32-encoded data in tags instead of hex-encoded data.

This is of course a difficult judgment call, since as mentioned above the vocabulary of nostr is idiomatic and evolving. The important thing is not to allow a minority, non-standard data format to hijack a well-established content type. This can go some way toward taming the chaos of a decentralized system, as well as protect against organized attacks that would attempt to "embrace, extend, extinguish" the protocol.

Here are a few examples of malformed data and how it should be handled:

- If an event's signature is invalid, discard it — there's no way to prove it isn't forged. There are of course exceptions to this, like with NIP 59 wrapped events, which have no signature, but are still authenticated cryptographically.
- If an event contains code intended to crash, denial-of-service, or hijack the client, the client should protect itself using conventional defensive programming techniques — by sanitizing HTML before displaying it, not using regular expressions with user provided data, and by handling errors using null checks and fallbacks.
- If an event contains an adversarial relay hint or suspicious media URL, clients should have a policy in place, informed by user preferences, which blocks connections to unknown or untrustworthy hosts. This can be accomplished through domain white/black listing, or through trust assessments based on proof-of-work or the reputation of the event author.
- If an event contains data that doesn't adhere to the relevant specification, for example missing `start` or `end` times on a calendar event, clients might choose to show a fallback value, e.g. "not specified", or an error indicator.

One reason to prefer fault-tolerant handling over aggressive validation is that strict validation reduces interoperability unnecessarily by converting minor, recoverable errors into catastrophic exceptions that ultimately result in a poor UX. Care has to be taken to balance user protection, user experience, and long-term stewardship of the protocol when making these decisions.

An additional wrinkle to this problem involves writing rather than reading events. A number of bugs have emerged where data published by one client will be unwittingly deleted by another client. Here are a few examples:

- Some clients at one point started adding JSON-encoded relay selections to the `content` field of kind `3` follower list events. Other clients would drop these selections when updating user follows.
- Some clients started adding muted words to kind `10000` mute lists, in addition to muted pubkeys. Clients that weren't expecting this ended up dropping muted words when updating muted pubkeys.
- Similarly, some clients began adding private mutes to kind `100000` mute lists by JSON-encoding the tags, encrypting the result, and placing it in the event's `content` field. This resulted in mutes being ignored or overwritten on update.

In every case, these bugs are a result of poor design — overloading a single kind for multiple purposes is always a recipe for disaster. But in cases where poor designs become conventional, it can be helpful to handle events in such a way that unanticipated data doesn't get dropped.

One way I've found to do this is to write parsers for an event in order to turn it into a standard data structure that can be used elsewhere in my application with the original event attached. Then, when saving a new version of the event, I update the event directly, keeping tags and content intact as much as possible before re-publishing. Here's an example:

```
readProfile = e => {...decodeJson(e.content), e}
writeProfile = ({e, ...p}) => {kind: 0, content: encodeJson(p), tags:
e.tags}
```

You can see that the original event is passed through to `writeProfile`, allowing us to avoid dropping any `tags` that may have been included.

This adds a certain amount of additional effort to implementations, but is important for avoiding disruption of user experience.

Backwards Compatibility

Backwards incompatibility is one of the big problems of spec design. When breaking backwards compatibility, not only do you break other existing implementations, but in a system where events can't be migrated to the new format you also break all historical data, even if it was published by your own app.

At the same time, breaking backwards compatibility can free you up to improve the protocol in important ways. In many cases, it also won't have much of an impact. Supporting historical data is important for some applications, like archival services, but many applications have a strong recency bias such that they don't necessarily mind throwing away old data.

I think there's a balance to this. I don't think we can be backwards compatibility maximalists, but we also shouldn't be careless about throwing away old formats for no reason. There is a perverse impulse among software developers to refactor for dumb reasons which should absolutely be avoided.

When introducing new backwards-compatible functionality to Nostr, it's usually best to enhance existing data formats as long as it doesn't result in overloading a single kind with multiple use cases.

When breaking backwards compatibility however, it's best to create an entirely new kind and evangelize for migrating to it. This is much harder than modifying the behavior of existing event kinds, but it's far more polite. The downside is it breaks network effects - and can still result in a poor UX for clients that don't adopt the new format if it results in missing content.

Over time, Nostr protocol development has gotten increasingly conservative. As use cases proliferate, it becomes more difficult to get feedback on new data formats. As implementations proliferate, it becomes more difficult to advocate for breaking changes. Nostr specifications are not sacred, but their effectiveness relies almost entirely on interoperability. Maintaining compatibility requires conscientiousness, communication, and contributions to other projects - there's no better way to get people to listen to you than writing code to improve their implementation.

3. Signatures and Identity

Now that we've gotten some of the boring stuff out of the way, we can get into some of the meat of what makes Nostr special. While Nostr makes things marginally more exciting by virtue of its design choices, data modeling is common to pretty much every form of software. What isn't as commonplace is Nostr's use of cryptography.

Nostr is built on the same elliptic curve that Bitcoin is built on: secp256k1. Asymmetric cryptography enables all kinds of neat things from digital signatures, to encryption, to secret sharing, and even distributed signature schemes. This chapter isn't a complete catalog of what we might be able to accomplish with it, but will instead focus on how Nostr uses it and why it matters.

Information Wants to Be Free

Digital signatures are essential to making Nostr work. The goal of Nostr is to break down walled gardens by subverting one of their key value propositions: content authentication. Or, in other words, the ability to know that a particular person said a particular thing.

This is a challenge in the digital world because information can be copied or fabricated at will. Simply saying that someone authored a particular piece of content doesn't make it so. When you go to Twitter and you load up a tweet, you only know that tweet is real because you trust Twitter. And if someone takes a screenshot of that or copies the text and emails it to you, then you have even less assurance that what's been presented to you is authentic.

What this means is that data that is not cryptographically signed is tightly coupled to custody. The only person who can reliably attest to the authenticity of a given piece of information is the person who can trace its provenance from the author, through storage, and to your device. This is very convenient for social media platforms — reliance on unsigned data means that they are needed. There has to be a single trustworthy custodian in order for unsigned data to work. The same is true of search results on Google; you don't know that search results are any good unless Google says they are.

What signed data gives us is the ability to know that something is true without having to trust anyone. If I create a note on Nostr and use my private key to sign it, anyone can verify the signature using the hash of the event and my public key (which is attached to the event). This lets them know that the event was created by the person who has access to my private key, i.e., me.

A Nostr event can thus be sent over an untrusted communication channel without the recipient losing the ability to know that it was me who signed it. As long as they know my public key, I can email a Nostr event, I can send a Nostr event over a peer-to-peer communication or over Bluetooth or over the LAN, or I can print it up and send it by mail. No intermediary can stop me without securing a monopoly on my communication.

Publicity Technology

The business model that fuels today's social media platforms is predicated on the capture of user data for their exclusive monetization. The user has become the product. Our data is used in a focused way to create targeted advertisements, or in the aggregate to understand and anticipate user behavior.

Signed data solves only half of this problem — it actually *worsens* the problem to the extent that data is public and accessible to anyone who wants to analyze it for patterns. Designing digital identity also has an incredible amount of complexity involved, and must be approached with caution. From Philip Sheldrake's essay, Human identity: the number one challenge in computer science:

Put starkly, many millions of people have been excluded, persecuted, and murdered with the assistance of prior identity architectures, and no other facet of information technology smashes into the human condition in quite the same way as digital identity[...] This should give anyone involved in digital identity cause to consider the emergent (i.e. unplanned) consequences of their work.

When designing systems that make use of digital identity, it's important to work from a conception of identity not as *objective*, but as *subjective* — that is, defined not by a set of static attributes, but by the dialectical contexts and relationships the person behind the identity participates in. The former kind of identity allows others to *act upon* the identity; the latter allows the person who own the identity *to act.*

Cryptographic identity doesn't automatically make this distinction, but can be used in either way. If the goal is user empowerment, a system of identity that is crafted to protect the digital freedoms of the user must be carefully designed.

Because identity is intended to be shared in a social setting, Nostr is not really "privacy technology". Rather, Nostr is "publicity technology".

When you create an event and you send it to untrusted custodians (particularly if left unprotected by access controls or encryption) you are advertising something about yourself to the entire world. All the data included in an event and all the metadata that can be harvested by observers and middlemen points back to you.

This is suitable for Twitter-like use cases (although user privacy is a concern even in a broadcast social media context), but always has to be considered when building products on Nostr. For users, it's best to use a VPN and Tor in combination with Nostr if you're concerned about privacy. Even so, in the aggregate signed data can still be collected and used to understand both individual users and entire social clusters.

Dis-intermediating Data

With that in mind, signed data does help reduce the capture of user attention by dis-intermediating content delivery. The current business model of social media platforms is predicated on the attention users give the platform, which is maximized by designs which stimulate "engagement", the creation and consumption of digital content.

The old way of doing this was through centralized content production. A business would create content — for example, movies, magazines, or podcasts — and present it to users for their consumption. Of course, it was a lot easier to directly monetize this content because it was both high quality and protected by intellectual property laws.

On social media, content is not produced by the platform, but by users. This introduces a second side to engagement — users not only consume, but also produce content. This keeps them even more engaged, and provides even more information about them to the platform.

When content is signed, it can no longer be captured by the platform (even if it is still visible to the platform). The result is that platforms lose the ability to enforce their monopoly on user attention. As a result of signed data, user attention can be diverted to other platforms that host a copy of the data. Nostr takes this effect even further by decoupling data storage and user interaction — relays store notes, but clients mediate user interactions.

On Nostr, clients can be more aligned with users, since they can only capture user attention to the extent that their *functionality* is what's valuable to the user, not the *data* they have access to.

The ability users have on an open network to leave a platform without losing all their data or their social graph is called credible exit. This is the opposite of "vendor lock-in", which occurs when platforms make it difficult to leave them. The export features social platforms offer are nearly useless because they break all the links in your social graph. But if all your social data was signed and the social graph was open, it would be quite easy to leave.

Social media companies can still exist in a world of signed data, but they will have to offer a real value proposition to their users in order to retain them. This means that they'll be more likely to serve their users rather than extract as much value as possible from them.

Whether open source software wins out or for-profit companies start building on Nostr, signed data weakens platforms' hold on their users and realigns the interests of social media platforms with those of their users. And while I think there's still room for skepticism about the effects of social media in general on people and communities, removing lock-in fixes a lot of existing perverse incentives in the system.

Your Very Own Number

On almost every digital platform in existence, identities are the property of the platform, not of the user. In most cases, a user identity is not actually the user's email, phone number, or a username, but an ID that's stored in a database. This number is generated by the platform when a user registers, and belongs to the platform, which can change this number, redefine it, give it away, or delete it at any time and with impunity.

The internet of today is entirely occupied by renters. Users move from one website or app to another, asking the owner of that digital estate if they can *please* check their messages, post a meme, express an opinion, or buy something. Users have next to no leverage to force their way in to the vault that holds their data. Content creators are frequently de-platformed. Community groups are banned. Even websites are frequently taken down.

Even worse, when a website integrates social sign-on (to make life easier for the user of course), not only does the identity provider get the ability to track your movements across the internet, but they also get the ability to revoke your access to these third party services! Even if you don't use social sign on, your email provider probably has the same ability to revoke your access from the rest of the internet. This over-reliance on identity providers leaves users extremely dependent and vulnerable.

An "account" is simply an entry in a database that associates some asset with a person or an entity. The user is the entity, and the custodian holds the asset. Whether you're signing in to Netflix, sharing your PPI with your bank teller, or swiping your keycard to get into work, the credentials you present are *issued* to you by the service or an authority they trust (yes, even your birth date, which is only meaningful because it's printed on your ID).

Cryptographic identity turns this on its head, because instead of being granted an identity by a platform, you provide your own user ID. And because you don't have to tell anyone your private key in order to prove you own it (in contrast with, say, social security numbers), *you* have exclusive access to your identity.

Secret keys can also be generated out of thin air, giving you as many identities as you need. This allows you to re-use an identity as much or as little as you need, depending on whether you're looking for privacy or convenience in any given case. For example, using the same key for your social media profile as well as for your wallet makes it easy to pay friends — but might also link your payment history to your public persona.

This of course requires users to be responsible and understand the privacy trade-offs of reusing their keys, just like it requires users to know not to reuse their passwords. But user education is a tractable problem, while digital freedom is impossible without cryptographic identities. Users can be exclusively and finally in control of their identity.

Agency and Identity

Cryptographic identities attack walled gardens from multiple directions at once. Because data is signed and verifiable, platforms are no longer necessary for tracing the provenance of a given message or social media post.

The flip side of this is that platforms are no longer able to act on behalf of their users with impunity. If data is expected to be signed and users hold their own keys, user activity can no longer be forged. This is important in cases where action is required in order to de-platform someone — bank accounts can be frozen, but bitcoin wallets are held in alodium.

Cryptographic identities are harder to steal than claims that are shared with custodians. When you sign in with a username and password, that goes into a database that may be the target of leaks or hacks. If it's valuable, this data almost always ends up on the dark web.

In contrast, when using asymmetric key pairs the only thing that can leak from the third party database is your public key, since your private key never leaves your possession. Of course there's a whole art to keeping your private key safe, which we'll get into later. But even without sophisticated key management, cryptographic identities drastically decrease users' vulnerability to identity theft and phishing attacks.

Identity Webs

Identity *a la carte* is not the end of the story though. Cryptographic identities can free data from custodians, but they can also be referenced *by* that data, making it possible to assign meaning to identities themselves. A simple example of this is a "mention", which allows someone to be notified about a conversation and engage with it. The same goes for direct messages, payments, friends, follows, and more. Since each reference of a public key is also signed, relationships between keys can begin to form. This is known informally as "webs of trust", although the word "trust" might overstate how reliable these connections are. A better term might be "identity webs".

The semantics of connections between identities may be more or less explicit (a social media "follow" is a pretty well-defined concept, while a comment on a blog post carries far less meaning), but every connection has significance. In the aggregate, the same types of analysis that threaten user privacy (as mentioned above) can be leveraged to benefit users who voluntarily produce content for public consumption.

Whether through simplistic analysis of follow graphs or sophisticated analysis that scores transitive relationships or different types of interactions, reputation can be assigned to identities. We often think of "reputation" in objective terms, but in reality it's entirely relative to the values and network of the person assessing reputation.

In concrete terms, if you follow someone you're expressing a non-zero amount of "trust" in that person. It doesn't mean that you endorse their views or that you would trust them to watch your kids, but it does indicate that they're of interest to you in some way. Other people can then borrow this attestation to augment their own ability to assess reputations, expanding their ability to traverse the network. However, with each connection the ability to assess reputation diminishes since trust is increasingly delegated.

For this reason, reputation has to be calculated from a trusted starting point and only via trusted connections, or else it becomes subject to sybil attack. If reputations are not calculated conservatively, it can become trivially easy for sock-puppet accounts to infiltrate a network. In other words, if you calculate reputation starting from the perspective of a scammer, you're going to have scammy scores. Likewise, if you over-weight weak attestations (for example, ambiguous emoji reactions), it's unlikely you're going to get good results. Garbage in, garbage out.

This conservatism naturally results in network partitioning, in which someone from one social cluster might find it difficult to establish the reputation of someone in a different social cluster. There are "global" algorithms like PageRank (named after Google's Larry Page) which attempt to circumvent this constraint, but are inherently vulnerable to sybil attack.

One solution to this problem that is frequently proposed is to supplement implicit or weak attestations with more explicit, annotated attestations. Instead of using a single number which "scores" the subject, multiple scores might be calculated along any number of axes. For example, instead of "I follow this person", you might want to indicate "I trust this person to recommend movies."

This approach can work in certain contexts like reviews for products or services. But in most cases, getting users to publish such explicit attestations is difficult to incentivize. A good example of this is PGP which, despite the enthusiasm nerds have for building trust graphs, has failed to be widely adopted by non-technical users. The reason for this is that the creation of these attestations has no immediate utility — they are entirely instrumental, requiring up-front investment in order to achieve their purpose. In contrast, follows, likes, and replies all have their own immediate, intuitive incentive.

For this reason, webs of trust will always be more or less implicit, and rely on careful and qualified data analysis. Just because someone is good at recommending recipies does not mean you should trust their political opinions. This is especially true when differentiating between different trust contexts. Trust derived from follow graphs may be very useful for identifying spam or impersonation, but might not be useful for assessing whether someone is safe to meet in a dark alley.

Digital representation of social trust is inherently weaker than authentic social trust, because it is predicated only on the subset of information that can be collected. Major factors in assessing trust in real life (such as physical appearance, credit score, or criminal record) might be entirely absent in a digital setting. This is especially true for new users who don't yet have any attestations from other users about the authenticity or value of their activity.

However, there are additional affordances that Nostr provides which can supplement the trust graph. Synthetic attestations can be generated on behalf of new users by services with an established reputation by reviewing government-issued identification (or by asking for a picture of the person with a shoe on their head). Another method is "proof-of-work", originally invented to frustrate email spam. NIP 13 specifies how a given event can demonstrate that a certain amount of computational work went into its creation. Both of these methods have their limitations (identity theft, AI-generated imagery, and GPU farms can all bypass these measures if the incentive is valuable enough), but they can factor into a more robust trust profile.

Holding Keys

Cryptographic identity promises a lot, but all of its benefits are predicated on users being able to securely and conveniently hold their private key, which is a big assumption. Over a decade of research has gone into building key storage solutions for Bitcoin wallets. Many of the the complexities and trade-offs involved in those design decisions apply to Nostr as well.

The stakes are slightly different though. If you lose your Bitcoin keys you can lose real money; if you lose your Nostr keys you can lose your social identity. Which one is worse depends entirely on who you are, but both have the potential to be catastrophic.

Nostr also has some disadvantages in comparison with Bitcoin. With Bitcoin you can rotate keys by simply sending funds to a new address and throwing away the old address (although if you lose your key entirely you're still out of luck). On Nostr, your identity is your public key, and therefore much harder to move away from. While nothing prevents copying and re-signing all your social media data with your new key, all references to your cryptographic identity would be broken — and along with them your entire reputation.

Key rotation is something that we have to figure out if Nostr is going to succeed in the long term. There have been some proposals, but none of them have gained enough traction to be implemented. Key rotation is an inherently difficult problem, because you need to borrow the reputation of the old key in order to validate the new key in such a way that an attacker wouldn't be able to do the same thing. Decentralized identifiers (DIDs) are not really a solution because they result either in a circular system of keys or dependence upon a trusted name registrar. Hierarchical keys have the same problem; the root key still has to be competently secured.

There are some things we can do though to make key storage safer and more convenient.

A number of different signer protocols exist as alternatives to pasting your private key into every Nostr client you try, including browser extensions, Android intent-based signers, and remote "bunker" signers.

The last of these is, in my opinion, the most promising. Remote signers can run on any device connected to the internet, and used in any type of application. Browser extension or Android signers can be useful as progressive enhancement, but should always be considered secondary to remote signers.

The difficulty with any of these solutions is that they require new users to learn about and set up an additional component when getting started with Nostr, which is an awful UX from an onboarding perspective. One possible solution to this is for every client to include a remote signer implementation so that users can log into other clients remotely. But this is a lot of complexity to impose on client implementations, and is very difficult to execute successfully.

The most promising solution to the onboarding problem is "multisig" bunkers, which allow for multiple unrelated custodians to coordinate when signing events by breaking the user's private key up into "shards". A scheme like "2 of 3" would require that two custodians collude in order to act on behalf of the user.

But even better in the context of onboarding would be to use a "2 of 2" signature scheme, where half of the key is held by a custodian, and half of the key is held by the client onboarding the new user. This means that unless the client and the custodian collude, the user's key remains under their control. Given the likelihood of non-technical users failing to secure their keys, this can be a viable alternative to self custody for certain people.

This approach can be improved further by sending an email to the new user with a version of their private key encrypted with a password as a backup. This helps defer the learning process for key management until the user has developed a reputation on the network, making the key valuable. When they're ready to take custody of their key, they'll have it sitting in their email inbox.

One other common pattern worth mentioning is seed words. Seed words are great for securing Bitcoin in cold storage because cold storage ideally requires manual entry onto a physical medium or air-gapped device. A 64-character hex key is going to be a lot harder to write down correctly than 12 or 24 words, especially if you're engraving the characters on steel.

But on Nostr keys are almost always stored on an internet-connected signing device, making them vulnerable in ways bitcoin keys aren't. Seed words only useful for physically backing up Nostr keys that exist only as distributed shards. This can make sense for advanced or paranoid users, but is not really relevant to onboarding new users. In most cases, putting your key in a password manager (optionally encrypted) is enough.

Naming Keys

Using a public key as your identity is a huge improvement over the status quo, but it does come with the important trade-off of how names are associated with keys. Zooko's Triangle asserts the difficulty of creating a name that is human-meaningful, secure, and decentralized.

Nostr public keys are secure and decentralized, but not human-meaningful. A complete solution to this problem is possible, but prohibitively expensive — for example, names could be associated with keys via a blockchain, but that would introduce a hard dependency of the naming system.

Nostr opts, instead to combine several more lightweight systems — none of which fully solve the naming problem, but which together make *cryptographic identities as a whole* meaningful to humans.

First, users can simply name themselves by publishing a `kind 0` profile. This name isn't unique to that user, it's only a self-designation. On its own, this allows attackers to impersonate anyone simply by replicating their profile data (and, if they want to go all-out, their social media posts, follows, list, etc).

For this reason, when displaying information user identities, it's imperative to qualify that information by validating the user's claims. Simply displaying the user's public key, or a color derived from it are *not* sufficient, because that data is not meaningful to humans (especially if they have never seen it before) — opaque data does not help qualify human-readable data.

The best solution to this is, again, identity webs. Identities can only be validated as authentic by sybil-resistant attestations not under the identity's own control. Follow graphs, emanating from the user's own identity are a great way to handle this (at least, in principle — this can still be attacked if a well-known user's key is stolen and their followers are co-opted to validate an impersonator instead).

One mistake that is worth mentioning here is the use of NIP 05 addresses for verifying users, because addresses themselves can be verified as belonging to a given user. But just like domain names, addresses are not self-validating. Is google.org owned by Google? You don't know. In the same way, is chris@nostrplebs.com or chris@primal.net the correct address? Addresses are only useful for *finding* users, not for validating their authenticity.

The basis of identity in a decentralized system has to be cryptographic identity and the relationships established between them.

4. Relays are Repositories

Thus far we've focused mostly on "Nostr as data." But data modeling, signatures, and identity are really only half of the story. Nostr is much more than a data format — what is much more interesting is its network architecture.

Why WebSockets

Many cutting-edge decentralized protocols are "peer-to-peer," which means they attempt to repurpose the architecture of the internet to facilitate direct communication between peers. Peer-to-peer technology is really neat, but it's also hard to get right because it works against the grain of the internet as it has been used for the last 30 years.

That's not to say P2P technologies are of no use. Mesh network systems and P2P communication techniques can be great when used as fallbacks from more traditional infrastructure and networking technologies. But just like it would be very hard or impossible to completely replace the hub-and-spoke architecture of the internet with a mesh network, P2P systems are not a complete replacement for the client-server model.

Nostr attempts to get many of the benefits of P2P-style architectures without a lot of the hassle by using a pretty boring piece of web technology: WebSocket servers. WebSocket servers have all the same problems that regular servers have in terms of being gate-kept by internet service providers, since normally they rely on TLS root certificates, ISPs, and DNS registrars. Domain name resolution can be bypassed, but then users have to deal with IP addresses. TLS can be bypassed, but then implementations have to run things through Tor, which has its own trade-offs.

Most nostr relays use TLS with conventional domain names. And in true Nostr fashion, this is good enough™. Because anyone can buy a domain name and spin up a server, there are hundreds of relays out there that can be used for storing signed data. If relay operators or users want to up the ante (for example if the country they live in man-in-the-middle attacks TLS), they can always resort to more advanced techniques. And if a particular relay goes offline, it's only one of many redundant nodes.

In "Simple Made Easy," Rich Hickey defines the distinction between easy and simple.

"Easy" means something is "close at hand". Something that's easy may not be the best solution, but it's accessible. "Simple" means that something does one thing well without being complicated by different concerns. In technical terms, relays are not really simple. Relays are easy. Relays use WebSockets which are a layer on top of HTTP which is a layer on top of TCP/IP, which is itself a fairly complicated protocol.

The trade-off here is that the technology, again, is not ideal, but good enough. WebSocket servers also allow for duplex communication rather than request-response, which allows servers to push new information to the client as soon as they receive it.

But using WebSockets doesn't preclude the use of other transport protocols to send events between peers.

As an aside, I'm not talking about Nostr over HTTP. HTTP, while simple and more familiar for developers who are used to working with web technologies, is strictly worse than using WebSockets because HTTP doesn't allow duplex communication, increases latency through polling (or requiring support for server-sent-events), and makes clients and relays less efficient. If you're going to rely on the web stack, just use WebSockets.

What I mean rather is alternative transport protocols that have a different simplicity/ease trade-off balance from WebSockets. In theory you could come up with a binary encoding of Nostr events and send them over QUIC, which would eliminate a lot of the overhead associated with WebSockets. This might be useful for high performance applications and servers.

Alternatively, you might put a bunch of Nostr events in a JSONL file and torrent it. These events would then be replicated on a different decentralized network and could be downloaded by a client supporting that transport, improving censorship resistance (assuming the dataset is valuable enough for people to seed it).

Another example would be to use P2P technologies as progressive enhancement. In other words, you could enhance the connectivity of Nostr applications in certain scenarios without breaking the base protocol. This might be a useful when designing a mobile application for people in rural areas with little network coverage. If the network isn't available, devices would fall back to direct connection over Bluetooth or WiFi Direct.

This technique was pioneered by Secure Scuttlebutt (SSB), which started with P2P technologies and then added pubs as a way of bridging devices that weren't directly connected. Nostr simply reverses this model and makes pubs the primary means of communication and optionally adds P2P when required. Of course, these alternative transport protocols have by and large not been specified or implemented, but there's nothing to keep us from creating them.

In terms of transport protocol, Nostr opted for "easy" by default. But this dimension of the relay network is secondary to the network architecture itself, which adheres much more closely to Hickey's idea of "simplicity". So, setting aside the transport questions for now, let's get into what relays actually do.

Multi-Master

A key aspect of Nostr's architecture is its reliance on more than one relay for storing data. As we mentioned in the previous chapter, relays can't falsify anything because your data is signed. They don't have the ability to lock users in or create a data moat. What they do have, however, is the ability to censor or delete user data.

If you use only a single relay, that relay has complete control over which of your signed events get rebroadcast to other people. It may even selectively censor data depending on who's asking for it or what you're publishing. Adding an additional relay decreases your vulnerability to censorship in a straight-forward way. If, say, you have a 10% chance, all things being equal, of being de-platformed by a single relay, publishing to two relays gives you a 1% chance; three relays gives you a 0.1% chance. Beyond that, if you're particularly sensitive to the risk of de-platforming, your best option is probably to self-host or select relays you can explicitly trust to host your content.

The fallacy that "more is better" is evident with relay operators — many people run a relay just so they can "run a node". But (to borrow a term from Bitcoin), you have to have "an economical node". In other words, people have to use it — and you have to keep it running.

For the average social media user, three to five popular relays is generally enough. As long as the relays provide a good mix of different admins, jurisdictions, and policies, you're very unlikely to lose all your hosts at once. Even if you do, you still have your key! Just keep a periodic backup of your content and re-broadcast it when you find more trustworthy hosts.

This ignores the problem of which relays should be used to store what content. Solving this problem is the key to making Nostr's multi-master architecture work;

naïve replicas result in either excessive duplication of content across all relays, or persistent failure to locate content. We'll get to that soon, but first I want to define what a relay actually is.

Functional Relays

Relays are simple repositories of events. They hold a bunch of events in some database or other and grant access to those events using the Nostr WebSocket protocol. This protocol involves sending JSON-encoded messages over WebSockets using just a few core commands:

- Clients can send an `EVENT` message in order to publish an event to a relay
- Relays respond to client `EVENT` messages with an `OK` message, which includes whether the event was accepted or rejected and an human-readable message with details
- On the read side, clients can send a `REQ` with a subscription ID and a filter
- Relays can respond by sending one or more `EVENT` messages, each of which contains an event

There are a few other control messages, including `CLOSE`, which allows a client to close a `REQ`; `CLOSED`, which allows a relay to close a `REQ`, and `EOSE`, which allows relays to let clients know when they've sent all events initially matched by a `REQ` (the relay might continue to stream matching events as it receives them).

There are some other details which we'll get to below, but this is pretty much it! A relay is just a bucket that you put events into. Later, you or someone else might take them out again.

Some additional commands have been proposed, and in some cases added to the protocol, but to the extent that they stray outside of this basic repository paradigm they only complicate things for little or no benefit. One example is `COUNT`, which might seem useful at first until you consider that in order to count events in a decentralized network you need a way to reconcile those events between multiple servers. COUNT doesn't do that, making it useful only when working with a single relay.

For the sake of decentralization, it's my opinion that the relay interfaces should be minimal in order to reduce implementation burden and maximize interoperability.

That said, there is one additional responsibility that relays can't really delegate:

access control. Access control on Nostr is implemented by the `AUTH` verb. When a relay wants to know who opened a connection, it issues an `AUTH` message with a challenge string. The client then incorporates this challenge string into a `kind 22242` event, signs it, and sends it back to the relay.

From this point on, the relay can implement any policy it wants on the basis of the user's cryptographic identity by rejecting published events, refusing to serve requests, or selectively filtering data before returning it to the client.

This is useful for two distinct things: content curation and access control. Many relays have policies about what kind of content is allowed on their relay, whether based on content analysis, social graph analysis, proof of work, or when an event was published. Much of this data is already available on any event that gets published to a relay, but the identity of the person publishing an event can also be an important way to vet content cross-posted from elsewhere on the network, or if the user is paying to store content on the relay.

Access control is similar to, but distinct from, content curation. Relays may or may not care what events in particular they store, but they may want to allow access only to particular users. This can be useful for community relays, relays that handle direct messages, or relays that proxy content on behalf of paid subscribers. Likewise, relays may allow anyone to request events, but be very selective of what content is actually stored.

These policies together make relays less fungible — which is, counter-intuitively, a good thing. Some large hubs accept everything except for obvious spam, and are therefore mostly interchangeable — there's no reason to establish a relationship with one rather than the other. Other relays implement custom policies for protecting community content or direct messages, which make them more appropriate for certain use cases. Another example would be relays which return an algorithmic selection of notes from across the network, making social media "algorithms" possible.

This is an important point — to the extent that relays are treated as commodities they also have to be either subsidized by businesses operating on nostr, or run *pro bono*. Relays that don't have a distinctive value proposition don't have a business model. This presents the danger of leading us back into the surveillance capitalism of the existing internet. Presenting relays to the end user as distinct services providers forces users to consider who they do business with and why, creating affordances for direct monetization and alignment of incentives that wouldn't exist if relays were abstracted away.

One final thing to note in this context is Negentropy, a set-based reconciliation protocol used for efficiently syncing events between two relays (or between a client and a relay). This allows a client to request only events it doesn't yet have, significantly reducing bandwidth requirements.

This does a lot to facilitate content replication across relays without burning through resources, which in turn makes it possible for the network to re-organize itself to align with expectations about where a given event "should" be stored.

Replication and Routing

The Nostr network is highly partition tolerant, unlike (for example) Secure Scuttlebutt, which links all events from a single key together, making it impossible to download a single event without downloading all events that came before it. On Nostr, you can download any dataset you want, because events aren't tied together. The cost of this is that you never know if you have all the events; the benefit is that content can be replicated more selectively across the network.

This is actually how Twitter's architecture works too — in order to scale, they maintain a network of interrelated caching nodes. When the average user posts content it normally gets sent immediately to read caches for all of their followers. But when a very popular account posts content, it's replicated to special "famous person" caches, distributing load without excessive duplication. Their system is also overprovisioned and able to respond to dynamic load, since news events may happen without warning.

This is of course an over-simplified view of Twitter's architecture, but you can see how nostr relays might correspond to cache nodes. If Nostr is to scale, it will naturally adopt a similar architecture. This requires not only the availability of enough nodes to support the network, but (arguably more importantly) heuristics for which relay to ask for a given event.

In the past, naïve content replication has been used to solve the routing problem. Instead of developing methods for relay selection, all content was sent to every relay. One project in particular called Blastr encouraged people to publish their events to special write-proxy relays, which would then re-publish it to hundreds of other relays.

This not only makes these relays a chokepoint for new content, but is also increasingly expensive as the amount of content being published to the network grows. With aggressive content replication, hundreds of duplicates of each event are foisted upon relays that may have no interest in storing them.

This is not horizontal scaling; it's redundant vertical scaling. If every relay has to hold every event, small relays become impossible, centralizing the network in a few mega-relays which can afford to store and serve everything. This isn't really economical at any scale, but is outright prohibitive when databases reach a billion plus events.

Aggressive replication is a brute force solution to the problem. In theory it would allow clients to ask any relay for any event, providing a 100% query hit rate. In reality though, this would actually destroy the network.

The alternative to brute forcing through content replication is intelligent relay selection. In order for decentralization to work, clients have to be responsible to "route" events and requests for events to the correct custodian in a way that is predictable. In this sense, redundancy is only relevant for degenerate cases, where the correct custodian fails to store the events it is responsible for. Routing heuristics are a form of interoperability that doesn't have to do with data modeling, but with network organization.

In order to solve this problem, we must have rules for:

- Where to send a given event
- Where to send a given filter

In the latter case there is less information available for solving the routing problem, which means multiple heuristics might be relevant for a given event, depending on how it might be queried.

The Outbox Model And Friends

This is where the "Outbox Model" comes in. The Outbox model combines cryptographic identity with selective content replication, and was the first heuristic defined for solving the routing problem, but is certainly not the only one — other heuristics have emerged over time as the same problem cropped up in different contexts.

At the end of this section, I'll enumerate a few other relay selection scenarios and their accompanying heuristics. For now, I want to focus in some detail on the Outbox in particular, since understanding it is crucial for keeping Nostr decentralized, and the heuristics it defines are paradigmatic for the rest.

In the early days, Nostr was used primarily for microblogging (and to a large extent still is), and so NIP 65 was created in order to solve this problem *in that specific context*. NIP 65 allows users to publish a `kind 10002` "relay selections" event "to advertise relays where the user generally writes to".

By publishing their relay selections, users are declaring to the rest of the network that events they *create for public consumption* can be found there. This covers things like blog posts, microblogging events, and user profiles.

So, in order to find a given user's blog posts (for example), I first have to look up their `10002` event (we'll cover how this happens in a bit), find the user's "write" relays in that list, then ask those relays for blog posts by the target user.

This is a very simple heuristic on its own, but it's important to be clear that it is not sufficient for every use case. The term "outbox model" is often used as a way to refer to the general idea of heuristics for relay selection, but in fact the outbox model is only one such heuristic, and is relevant only in the particular (though very common) context of author-based retrieval of public social media content.

The outbox heuristic is not sufficient to solve the routing problem in general for two reasons. First, there are many notes that should not be posted to user outboxes - for example, a `kind 22242` auth response, or a chat message posted to a NIP 29 group. Second, any event may be retrieved based on criteria other than event author.

A second heuristic, also defined by NIP 65 is the "inbox model", which is intended to make social media posts discoverable based on users @-mentioned in the event. This heuristic is distinct from, but complementary to, the outbox heuristic.

For example, a note by Alice which mentions Bob should be posted both to Alice's outbox relays, as well as to Bob's inbox relays. To retrieve Alices's replies, we would use the outbox heuristic based on her `kind 10002` event — but if we want to retrieve Bob's mentions, we would use the inbox heuristic based on his `kind 10002` event.

Similarly, when posting to a NIP 72 community, you may or may not choose to also post an event to inbox or outbox relays, depending on how private you want the event to be. But in addition, it's important to post events to the locations defined by the group definition's `relays` tag. The reason for this is simple: when fetching a list of community posts, searching the outboxes of all the members of the community may not be feasible.

Posting to certain relays based on all applicable heuristics can be thought of as equivalent to creating multiple indexes on a database table, each to support a different query scenario. An index connects a query condition which supports a particular use case with where on disk matching records are stored. In the same way, a routing heuristic connects a filter that might be constructed to support a particular use case with the relay where matching events are stored.

Just like database indexes, relay selection heuristics are generally additive, and should all be applied when relevant so that the event in question can be found using the heuristic most appropriate to a given context. The exception to this is when some form of access control is desired — i.e., that the event *not* be discoverable using a particular heuristic.

An example of this is content posted to NIP 29 groups. Because access control is part of the purpose of the NIP 29 spec, it would be a violation of the user's intentions to publish an event posted to a group according to the `outbox` heuristic — doing so would "leak" content which was intended to be protected.

This means, of course, that the usual `inbox` heuristic is not available for notifying users when they are mentioned in a group context. This might be considered a feature or a bug depending on your perspective; this is one of the tradeoffs involved in a system without a single arbiter governing access to content.

There are also situations where events might be available in locations not predicted by any standard heuristic, but instead as a result of a relay's content curation policy. You could make requests against (for example) an archival relay using any filter, and get whatever it returns. This is an ad-hoc heuristic, defined by the relay's policy and based on the relay's identity, not on any standard protocol feature. Relays have the right to override any heuristic, whether using negentropy to synchronize, attracting users to use their relay in a particular way, or by rejecting content that doesn't fit their policies.

Needless to say, this can get very complex. This is a natural result of the openness of the protocol and its permissionless extensibility. For publishers, the list of heuristics available to any given event should be well-defined, either by general-purpose heuristics like `inbox` and `outbox`, or by an additional or overriding heuristic defined for the event's `kind`. On the read side, however, no implementation will have a complete view of every heuristic that is applicable in a given situation, which means that there will inevitably be a certain amount of spontaneity to event discovery, particularly when the heuristic is user- or relay-defined. It's also important to note that applications only need to implement the heuristics relevant to use cases they support. An exhaustive routing policy is neither possible nor necessary.

Because the routing problem is both complex and important, let me give a few more example scenarios.

- NIP 17 direct messages should be sent to the recipient's `kind 10050` direct message inbox relay. Direct messages are encrypted to only one private key, which means each message gets sent multiple times - once to each user.
- The `outbox` relays NIP 47 zap receipts are sent to should be those of the zap *request* author, not the zap *receipt* author, since the wallet is only acting on behalf of the person sending the zap. In some cases though, it might be best to put the zap on an access controlled relay (for example, if you're zapping within a community group).
- Some events, such as NIP 72's `kind 34550` group metadata events, define which relays related events should be posted to using a `relays` tag.

Which relay an event is posted to depends primarily on *how that event is to be read*. Events need to be discoverable, which means readers need to be able to anticipate where the event is stored. However, there are some kinds which don't provide any of this information to someone wishing to request them — for example, `kind 37515` geocache listings, or any event with a `t` tag representing a social media topic.

In both cases, events belong somewhere that isn't currently well-defined by the protocol. Currently, this is solved by asking users to manually select the relays where they want to look for geocache listings or topics. This can be a perfectly valid heuristic on its own, but will necessarily result either in centralization (by clients who hard-code certain relays), or missed notes (since the ability to retrieve matching events depends on something between randomness and brute force).

Additional signaling may be implemented to solve these problems — for example, relays may advertise their support for certain `t` tags, or for geocache listings in a given region based on `g` tag. In order for this to work though, heuristics have to be defined and followed — NIPs that don't define relay selection heuristics for their use cases are inherently flawed, since they ignore a vital part of Nostr's architecture.

Bootstrapping

One event kind that I've thus far avoided introducing a heuristic for is `kind 10002` relay selections. If these determine where the rest of a user's public notes live, what determines where they belong? This is the classic bootstrapping problem of networks, for which you need a heuristic of a different kind.

In other words, you have to start somewhere. It's impossible to access a "network" without accessing particular nodes. When not yet connected, nodes have to be selected by some heuristic external to the network itself.

The most common way to solve this problem is to hard-code "indexer" or "default" relays in implementations, from which point other relays can be discovered using NIP 65, NIP 66, or relay hints. Alternatively, a distributed hash table (DHT) might be used to store events related to relay selection, improving censorship- and sybil-resistance.

In either case, events that form the basis for relay selection heuristics need to be stored in one of these few starting points in order to be discoverable. If they can't be found, none of the events they help locate can be found either (except accidentally).

In practice, hard-coded lists of bootstrapping relays work pretty well for the same reason that a small amount of redundancy in user relay selections can dramatically reduce the risk of censorship. Because hard-coded indexer relays are selected by implementation authors (or by users themselves), they're easy to swap out if any of them become censorious or go offline. This is strictly worse than using a DHT, but the strategies are not mutually exclusive. As time goes on, I expect a DHT-based solution to the bootstrap problem to be introduced as a progressive enhancement.

One difficulty with the bootstrap relay model, however, is that different types of bootstrapping events are needed to support different relay selection heuristics. The outbox/inbox heuristics are defined by `kind 10002`, but topic-, location-, community-, or language-based heuristics aren't, and any relay discovery events that support them will certainly be replicated less aggressively.

One possible solution to this would be to take a step backward, and instead of relying exclusively on `kind 10002` as the entry point to the network, switch to NIP 66 `kind 30166` relay discovery events for bootstrapping. However, since relay discovery events can be published by anyone, and aren't necessarily related to the user's web of trust, another heuristic has to be used to bootstrap trust.

This illustrates the fact that bootstrapping is not just a physical networking problem, but a trust problem as well. `kind 3` follows is not the only way to do this, but when choosing `kind 30166` events, some heuristic has to be used to assess relay monitor trustworthiness — otherwise, malicious monitors could sybil-attack relay recommendations, directing users to censored or otherwise malicious relays.

Currently kind `30166` events are published exclusively by "relay monitors", but there's no reason regular users couldn't also create them. This would make it possible for people within a user's web of trust to recommend relays for the purpose of discovering events by kind and optionally, indexable tags. Relays discovered this way might be used for finding events explicitly associated with relay selection, like `kind 10002` or `kind 10050` that can then be factored into other relay discovery heuristics, or they may directly provide the content themselves (for example, relays that curate `kind 31990` application handler events).

The main problem with this is that regular users are either unlikely to be competent to make these recommendations, or interested in taking the time since there is no immediate incentive for them to do so. NIP 51 lists partially solve this, since they exist primarily for user convenience, but they're neither comprehensive nor generic.

But assuming we can find a way to get users to publish these (either by solving the incentive problem or by having clients publish them on the user's behalf based on user activity), let me give a concrete example to illustrate what I mean.

Suppose you want to find relays that curate `kind 21` video events about cats by anyone on the network. Assuming the client has already loaded all `kind 10002` relays for the user's social graph, the client would first request relay recommendation events from the user's network targeting cat videos:

```
["REQ", "subid", {
  "kinds": [30166],
  "authors": ["<user's follow list>"],
  "#t": ["cats", "cat", "catstr"],
  "#k": ["21"]
}]
["EVENT", "subid", {
  "kind": 30166,
  "created_at": 1722173222,
  "content": "This relay has some really funny cat-related content on
it.",
  "tags": [
    ["r", "wss://relay.example.com/"],
    ["t", "cats"],
    ["k", "1"],
    ["k", "21"]
  ],
  "pubkey": "<pubkey>",
  "sig": "<signature>",
  "id": "<eventid>"
}]
```

Because these events are loaded from the user's social graph, there is a high level of confidence that these recommendations are genuine (if not necessarily current or reliable). The user's client can then request cat videos from these relays and show them to the user. In this way, the user's social graph is leveraged *indirectly* for discovering content not authored specifically by people the user follows.

Relay Hints

Something that frequently gets confused with heuristics for relay selection is relay hints. Relay hints are baked into certain events, particularly where there isn't sufficient information otherwise available for fetching a related event.

An example of this is `kind 1111` comments, which always exist in relation to another event. In general, it e-tags or a-tags the parent event and provides an event ID or address. An address contains the author's pubkey, which allows for looking up the author's outbox. But an event `id` tells you nothing about where to find it.

Relay hints were designed to solve this problem. By adding a relay URL to the tag containing the event `id`, the user now has at least some idea of where to look for it. Unfortunately, for the same reason that link rot happens across the internet, these relay URLs are not 100% reliable.

As a result, pubkey hints have been added as well. A public key is much more durable because as long as you can find the outbox relay selection for that public key, the user's relay selections can change, relays can go offline and come online, and there's still a well-defined path towards discovering the event in question.

Together, relay and pubkey hints comprise an additional heuristic which can be used as a fallback in case more reliable heuristics (like outbox) fail due to buggy implementations or users changing relay selections without migrating their data.

Relay hints have another function though, which is to force relays to federate. If an event including a relay hint is published to a censorious relay, the hint cannot be removed without invalidating the event's signature. As a result, relays are forced to decide either to reject these events entirely, or accept them and thereby advertise someone they would otherwise choose to censor.

This make is possible for a user's events to still be discoverable across the network, even if they are banned from a significant portion of popular relays. Clients can take advantage of this dynamic in order to heal the network, prompting users to update their relay selections to route around censorship.

If this all seems very abstract, take a look at https://how-nostr-works.pages.dev/#/pathological — this page includes several animations that illustrate how relay hints help keep the network healthy.

Properly implementing robust relay selections across the many heuristics, bootstrap mechanisms, and relay hints is obviously complex, and could be made simpler if re-designed from scratch. But solving this problem is essential to making Nostr censorship resistant, and because of the many different use cases that Nostr supports with its open data model, an unlimited number of heuristics may be appropriate.

Content Migration

There's one more wrinkle to relay selections which needs to be addressed before we're done here. What happens if the relays selected by a given heuristic change over time? This most commonly happens when a user changes his outbox relays, but it can occur for any other relay selection heuristic. Normally this problem is ignored or overlooked in implementations, but because nostr relays are not intended to be trusted either now or perpetually, having a story for how to manage this problem is an essential part of any complete implementation.

Luckily, the solution is actually very simple in principle. Because events are signed, they can simply be replicated to the relays that are expected to be storing them withouth having to do anything complex, like establish a chain of custody from one relay to another.

In an open system like Nostr, no one can force anyone else to do the right thing, which means that the smooth functioning of the system depends on incentive structures. In order for an event to be discoverable, heuristics for locating it need to be anticipated by someone. But who exactly is responsible for sending events to the right place? The obvious answer would be the event author, but that's not actually correct. The person responsible for putting events in the correct place is the person *who wants the event to be discoverable in that place*.

This means that the onus is on users (and by extension their clients) to choose good outbox relays and publish their events to them. Group admins are responsible for properly configuring infrastructure to accept the correct events and make them available. And people that curate topical data are responsible either to scrape the network or incentivize publishing to their relays.

Similarly, it is the responsibility of anyone that changes the result of relay selection heuristics to synchronize events to the new relay. If a user changes his inbox relay, he should copy all events that mention him to the new location. If a group switches relays, group messages should be copied by the admin. If a new indexer relay gets stood up, the admin should copy relevant events from the network.

Here's a simple example of what happens if data isn't properly synchronized when relay selections change:

- Alice publishes a `kind 10002` with relay A as her `write` relay
- Alice publishes a `kind 1` to relay A
- Alice updates her `kind 10002` with relay B as her `write` relay
- Bob fetches her `kind 10002`, requests Alice's `kind 1`s from relay B, and finds nothing

Bob correctly followed the outbox heuristic, but failed to find what he was looking for. In essence, Alice lied to Bob — she published a claim that her notes were available on relay B, when in fact they weren't. If she had copied her `kind 1` event from relay A to relay B, however, Bob's query would have been successful.

Less important, but also worthwhile, is event eviction. When relay selection heuristics change, stored events may become undiscoverable, and therefore a waste of space.

This part of content migration is the responsibility of the relay operator, since they are the only person who has any incentive to free up their resources — although users could request eviction as a courtesy. It can also get complex, since a relay won't necessarily have a complete view of every heuristic others are using to locate events on their relay (for example if an event should be availble both via outbox and by inbox, it shouldn't be evicted if only the author's outbox changes). In any case, relays have the prerogative to delete whatever they want, so people storing data should be careful to choose relays that are properly aligned.

Both the protocol and implementations will ultimately need to provide affordances for migration to users.

Currently, the only way to synchronize events to a new relay is to download all involved events and re-publish them manually, which can be fairly expensive in terms of bandwidth. A new primitive for asking a relay to synchronize certain relays from a peer would be a useful optimization. Similarly, there aren't currently any protocol affordances for requesting eviction of events — neither NIP 62 nor NIP 09 quite fit the bill.

Synchronization is also currently absent from most (or all) implementations. In the future, clients might do this in the background on behalf of their users when relay selections change, and community admin tools might include synchronization affordances for community relays. Alternatively, watchtowers could be used to actively synchrnonize data according to certain heuristics. Users might even be able to pay these services to ensure that their content is properly available.

Relays as Transport

Earlier I defined relays as "a repository of events." This definition can be exploited to do some interesting things not originally intended. Specifically, relays can be used to broker transport.

A relay per se is rarely the final destination of a given event, but is rather an intermediary between producers and consumers. Communication via relays may be between humans (as with social media), or relays may be used to allow different software applications to communicate. Both use cases are defined by the event `kind` that is used for payloads, and both are equally valid. This is the basis of some of Nostr's most interesting "other stuff".

An example of this is NIP 46 Nostr Connect, which allows remote signers to monitor a relay for signature (or encryption/decryption) requests. The signer application holds the user's key, and the client asks the signer, via a relay and Nostr events, to do something with the key and return the result. The details of the flow are not important here, but because the relay is publicly addressable the client and the signer are able to communicate.

Other examples are Nostr Wallet Connect, which allows for applications to communicate with the user's bitcoin wallet, and any number of Data Vending Machines (A.K.A. "DVM"s) which do arbitrary computations based on the event kind being used. In many cases, encryption is also employed to create a secure communication channel between parties.

Communication via relay allows anyone to set up a service identified only by a public key rather than an IP address. This is useful for protecting service providers' privacy, and makes running small services extremely convenient. Using this model, service providers can be run in any internet-connected software, without dealing with the complexity involved with NAT traversal. And of course, because relays are interchangeable in terms of protocol, multiple relays can be used at the same time to broker communication.

Relay as Design Pattern

Zooming out even further, relays are useful as a conceptual pattern that can be applied in the context of other protocols. In its simplest form, a relay independent of Nostr is just a server that implements a protocol, and which is advertised for use by the end user on the Nostr network.

For example, Blossom servers are media storage servers that run on a protocol distinct from Nostr, but are referred to using Nostr events. Defined in BUD 03, `kind 10063` events allow Nostr users to advertise which Blossom servers they prefer to use so that other people can find their media even if the URL embedded in a social media post changes.

This allows for the same kind of heuristic used for relay selection to be applied to media servers, piggybacking on Nostr for the indexing and trust layers. And just like events, media can be replicated across multiple Blossom servers, introducing redundancy to media hosting.

Blossom applies the same pattern pioneered by Nostr of interchangeable protocol servers to media storage. Another example is Cashu Mints, which speak the Cashu protocol. This pattern is not unique to the Nostr protocol ecosystem either - Git, ActivityPub, FTP, XMPP, and many other internet protocols also do this, but Nostr is an exceptionally clear example of how useful this pattern is, especially in conjunction with signed data.

Relay as Proxies

There is a different kind of relay on the nostr network, which is worth knowing about: proxies.

In theory, proxies can be helpful — read proxies can fan requests out across the network and de-duplicate the results, preventing users from burning bandwidth. Similarly, write proxies can be used to fan events published by the user out to the rest of the network.

Certain proxies exist which use a wrapper protocol to do this work — which is important, since it allows clients to do relay selection rather than delegating it to the proxy.

However, other types of proxy exist whose purpose is to do relay selection on behalf of the client, without using a wrapper protocol. The idea is that if a clients sends a `REQ` to one of these proxies, it can analyze the filters (as well as the logged-in user) and make requests to particular relays on the client's behalf.

This sounds nice, and it can work in some scenarios — particularly in a public broadcast social media context — but breaks down at the edges.

Part of the core responsibility of relays (as mentioned above) is content curation and access control. In order to know whether a given event should be stored or whether to grant access, the user is frequently highly germane. However, NIP 42 was specifically designed to prevent proxies from proxying `AUTH` (in order to protect users from session hijacking — remember, relays aren't trusted). For this reason, proxies simply cannot access relays on behalf of their users. This problem is becoming increasingly common too, as relays implement more policies to protect against spam and abuse.

Another issue with proxy relays is that relay selection is often based on heuristics only available to clients, and which can't be inferred from relays. A good example is when fetching events from NIP 29 groups — a filter with a particular `h` tag includes no information the proxy can use to infer relay selection. In theory the proxy could download the user's list of group servers, but that may be encrypted, incomplete, or irrelevant to the client's request (for example if the user is viewing a group as a non-member). In this case, clients can always opt-out of using the proxy, but that only proves my point about their limitations.

Proxy relays also tend to have poor support from clients, making the problem even worse — proxies frequently get added to users' inbox/outbox relay selections, which causes problems with delivery, unnecessary traffic, and (if implementers aren't careful) potentially infinite synchronization loops. The location where events are *actually* stored gets obscured by proxies — and there's no way for clients to tell the difference between proxy relays and relays that actually store events.

As a result, it's usually best to avoid using a proxy, although the development of a higher-level wrapper protocol that describes use case or makes room for explicit relay selections could be interesting and useful.

Wrapping Up

This was a long chapter, for which I apologize. But its length is appropriate because relays are a *sine qua non* of Nostr's architecture. Signed data has been around for 50 years, and yet hasn't gained widespread end-user adoption, in part because communication and storage have remained permissioned or proprietary. Relays have the ability to unlock signed data for everyday use.

I'm sure many parts of this chapter left you utterly confused as to what I was talking about. The complexity involved in dealing with relays is in part a natural result of the many different use cases supported by Nostr. But I'll be the first to admit that a fair amount of incidental complexity has also snuck in as the protocol has developed.

It may be that a successor protocol comes along and borrows the good parts of Nostr without any of the bad parts. But for now, it's enough to understand what relays are, and how to use them.

5. Radically Open

All protocols exist on a spectrum between open and closed.

Open protocols invite anyone to participate and build; closed protocols reserve participation to a select few. Closed protocols don't have a published specification, don't have publicly available source code that can be used to reverse engineer the it, and can only be used by authorized developers. Access might be restricted by law, obscurity, complexity, or secrecy. Some closed protocols even lock things down with cryptography using techniques like DRM or VIN locking.

But no matter how much control its owners may try to exert, every protocol is at least somewhat open in the sense that it can be reverse engineered. This idea is proven repeatedly throughout the history of both analog and digital technology. Openness is part of an open system. Both physics and computing, (for example) are "radically" open.

However, just as it's true that no completely closed protocol exists, every "open" protocol is closed to some extent. The closing of open protocols may be intentional (as with the W3C's membership requirements) or it may be accidental due to poor communication — if salient protocol information isn't readily available, developers may have to resort to reverse engineering to make progress.

For this reason, openness is not equivalent to anarchy — governance of some kind has to exist. The conflation of the two can actually cause a protocol to become more obscure through neglect. A similar fallacy is the equivocation of leadership with control. The job of a standards body is to document, not create, protocols.

As the IETF puts it in their mission statement:

[our] mission is to produce high quality, relevant technical and engineering documents that influence the way people design, use, and manage the Internet in such a way as to make the Internet work better. These documents include protocol standards, best current practices, and informational documents of various kinds.

Designs might come from committee participants' research and opinions, or they might come from third-party projects that independently gain adoption. In many cases this works well, particularly at a small scale. But when large commercial entities get involved, standards can end up benefiting the committee members rather than the beneficiaries of the technology. As Cory Doctorow puts it in The Internet Con:

> Patent ambushes are against the rules, but other forms of standards capture are fair game: for example, if the chair, co-chair and secretary all come from a single company (or a duopoly), that's fine, despite the fact that this means that the largest companies are literally setting the agenda.

An example of this is how browser development works. Frequently, Google designs some new API (for example, web components), builds it into Chrome, and because Google owns so much of the market, other implementations are forced to adapt. Unless of course you're Apple, which forces all apps on iOS to use their inferior browser engine, Webkit, in order to make native apps look better in comparison.

The exploitation of protocols is a well-known dynamic, a common pattern being to "Embrace, Extend, Extinguish". This is what Google did with XMPP and Google Talk. First, they found an open protocol that solved a problem and adopted it, immediately benefiting from the network effects that had developed there. They then extended the protocol with incompatible alternatives to standard protocol features for chat archival, file transfers, and more. Initially, Google supported federation but didn't fully implement all standard security features, causing a number of servers to refuse connections to Google's servers. Eventually, Google dropped federation entirely.

Examples abound: Microsoft created incompatible versions of Java, HTML, Kerberos, CIFS, and LDAP; Apple added platform-specific APIs to Safari; Oracle created proprietary PL/SQL extensions; Google extended the web with their Accelerated Mobile Pages.

This approach is a threat to Nostr just like it's a threat to every other open protocol. But what's different about Nostr is that it is *radically* open. The Nostr Protocol is not something that is contained in a spec document that only certain people can write to, but is defined by the actual implementations. Nostr's "kind" system makes the protocol infinitely extensible, broadening its scope to pretty much any content type that can exist in a social setting.

Embrace, Extend, Extinguish works by capturing *users*, not implementations. Therefore, resisting it is possible to the extent that implementers are able to tolerate the loss of users. In a sense, this is impossible to defend against, because users are the source of a network's value. At the same time, Nostr is designed for network partitioning — both by social cluster and by content type. The diversity of use cases on Nostr increases the implementation cost of any whole-protocol attack, while the locality of social clusters means that some users might switch to a fork without affecting the rest of the network.

Politics and Protocol

Open protocols in general are more often collections of patterns found in the wild than top-down specifications that dictate implementation. But Nostr explicitly embraces this paradigm. Nostr, like the internet, is not monolithic. There are some fundamental assumptions (e.g. proper serialization of events), but the vast majority of it is optional and loosely coupled with the rest. As a result, any specification can only be partially comprehensive, and is necessarily relative to the perspective of its author (just like web of trust). In the same way that anyone can write an implementation, anyone can also write a specification describing their work, or even what they think should be the case.

There is an important qualification to this though. One application with no users can create a new event kind with a custom data format, but that doesn't mean anyone else will want to adopt it. On the other hand, a widely used application can create new data formats (or hijack existing ones) pretty much at will, and others will follow suit.

Every protocol feature exists on a spectrum between zero and universal adoption. Because the entire point of a protocol is to standardize something between implementations, this means that the extent to which a given standard is adopted determines how much a part of "the protocol" it is.

Efforts to standardize the Nostr Protocol are more akin to discovery than invention. There is a massive design space for the Nostr Protocol, and it's individual developers who explore that design space to find new solutions to problems. The people that document those solutions are only recording what has been built in the wild. This book is a good example; in it I offer a fair amount of prescriptive advice — but it's up to you to decide whether to follow it.

There's been some criticism of the various "centralized" actors in Nostr, most notably certain funding organizations and the maintainers of the specifications found on the NIPs repository. The idea is that a decentralized protocol should have decentralized funding, decentralized software, and a decentralized specification if it's going to live up to its permissionless ethos. And if a decentralized protocol is successful, then all these aspects of governance should be built on the protocol itself, right?

But technology does not solve problems, people do; protocol development is necessarily dependent on politics.

For some people, "politics" is a trigger word which evokes images of stale bureaucracies and misaligned elected representatives. But I mean it in the Aristotelian sense — a community of people working together to create an environment which they intend to inhabit. This is just another way of saying that a protocol, by definition, has to be developed by people working together in a structured way.

This is very difficult to get right, partly because politics is hard, but also because good protocol development is a result of successful resistance against the natural incentives of the market. Markets are good in that they allow individuals and businesses to optimize for survival, but they also do a poor job of protecting commons, which are ultimately captured or ruined unless the different participants in the system resist the gravity of zero-sum competition.

Right now, the Nostr community is far smaller and less consequential than, for example, the Bitcoin development community. Because fewer corporations are involved, we have fewer malincentives to deal with, which means we have a higher tolerance for political centralization. This has certain advantages with respect to communication efficiency. But the need for decentralization will increase as the monetary value associated with the ability to capture the protocol increases.

Because Nostr developers tend to be a pretty disagreeable bunch, fragmentation of the protocol is already happening. I personally worry that the centrifugal force of this disintegration might have the effect of actually making the protocol *more* closed, not through permissioned politics, but as a result of the complexity inherent in making different parts of the protocol more obscure. In either case, politics becomes dysfunctional, forcing developers to resort to reverse-engineering the protocol in order to interoperate — or go their own way.

That said, protocol decentralization will need to happen sooner or later. The radically decentralized approach is one of Nostr's risky bets, and we may as well fail (or succeed) fast. The sooner we start practicing decentralized protocol development, the sooner we learn how to do it well.

May people already self-publish original NIPs, or even create alternative curations of protocol features. See Appendix B for a non-exhaustive list. The main problem with this is that it makes consensus more difficult and therefore weakens interoperability. If there are 50 different sources for protocol features, it becomes hard to discover them all. Right now, the NIPs repository is the place where the most visibility exists into new protocol proposals.

But the Nostr Protocol could eventually be moved to Nostr. There have been attempts at this, though they haven't been fully developed or adopted, and have some outstanding problems. At a minimum, here's what would be necessary to make such a project viable:

- The ability to fork a spec and view the version history across all forks. Replaceable events don't work for this because they don't include old versions.
- The ability for implementations to signal spec and version support is required to help others decide which to implement in order to be interoperable. Ideally, the reputation of a given opinion would be based on ownership of relevant NIP 89 listings and their attendant user recommendations. This ensures inconsequential implementations don't derail interoperability, and could be set up to avoid over-weighting extremely popular implementations by capping reputation-based weighting.
- A forum for developers to propose new specs before they're implemented in order to get feedback on their design.
- A forum for tracking and resolving interoperability issues (this currently exists on GitHub in the form of the nostrability project).

This kind of thing is hard to organize even in the best case, as the history and relative success of Stack Exchange makes clear. Decentralized consensus is a very high bar. The crypto graveyard is littered with decentralized autonomous organizations (DAOs).

Luckily, some incentives do work in our favor because interoperability is the key value proposition of Nostr. This is why the NIPs repo specifies that any NIP should have at least two client implementations and a relay implementation if relevant. This ensures that nothing makes it into the NIP repo that isn't already an interoperable part of the protocol.

This criteria also helps reduce scope, since a lot of things don't need to be interoperable. For example, mobile push notifications are cryptographically tied to a particular app, which means there's no real point in including them — systems can't talk to each other anyway.

Thinking in Public

One thing that has obscured the actual dynamics of Nostr's protocol development is the use of the NIPs repo for speculative proposals as well as for the documentation of existing work.

In contrast with the actual contents of the repository, the issues and PRs that are hosted on GitHub frequently are requests for comment rather than established protocol features. This goes back to the necessity for political involvement around protocol development. There's a trade-off between implementing the first idea that pops into your head, and enduring a long period of review before calling the idea "official". No review results in garbage specifications getting adopted. Too much review enervates would-be contributors, who lose interest and go elsewhere.

Some protocols, like MLS, have been in development for years without practical application. This is probably a good idea because the entire idea is to have a watertight security and privacy model. But this isn't really how Nostr works.

I think of Nostr as the Javascript of the web protocol world. Javascript, born at Netscape in 1995, was designed and implemented in just ten days by Brendan Eich. Eich had to reconcile his ideas about programming language design, which were borrowed from Scheme, with an appeal to the current popularity of Java for marketing reasons.

The result was Javascript, a language with multiple null values, quirky type coercion, and several arcane features like labels that ultimately fell by the wayside, but which remain supported by implementations. In addition, in the early days, JavaScript implementations were notoriously fragile and incompatible between browsers (partially due to lack of standardization, partly due to browser vendors trying to capture the market).

Much better languages could have been designed for the purpose of scripting in web browsers. But JavaScript hit the ground running, and now it is the single most used programming language in the world.

Nostr is full of half-baked ideas written by application developers (as opposed to protocol developers) who are learning the discipline of protocol development on the fly. Nostr is a mess. Incompatibilities between implementations abound. Data types that are prone to race conditions are a core part of the protocol. Kinds get overloaded. Services go offline.

But this is all by design. And I won't say that there aren't large existential risks associated with this development approach. But given Javascript's success, it's hard to say that it's a much worse model for protocol development than anything else.

In fact, most successful protocols are more or less like this. Protocols (and software) designed in a top-down manner frequently end up failing to live up to their design goals. It's only through implementation that what is actually needed can be understood, and it's only through implementation that multiple implementations can exist to interoperate in the first place. Nostr takes the implementation-first approach to an extreme.

Backwards Compatibility

Prioritizing implementations doesn't mean that the process of design, review, and feedback is not necessary though. In Nostr's openness and "shoot first ask questions later" environment, a lot of discussion and review still happens, just well after protocol features become "official." Just because something is in the NIPs repo doesn't make it sacred.

Migrating away from existing standards has been one of the most divisive issues in the Nostr developer community to date. Maintaining backwards compatibility is a massively important principle in software development, whether you're creating a system that's proprietary or an open protocol.

When deploying an update to a proprietary system, it's important that the old version of the application continue to be able to talk to the database, that the old version of the client continue to be able to talk to the backend, etc. But this requirement is ephemeral because the organization running the infrastructure has the ability to eventually upgrade the entire stack and throw away the old code.

In protocol development, backwards compatibility is a much harder problem. In order to fully implement backwards compatibility, you can never really remove a feature. We see this with HTML, where blink and marquee tags created in the 1990s still work. Programming languages commonly maintain backwards compatibility to make migrations easier — when Python 3 broke compatibility with Python 2, it took almost ten years for users to migrate.

But there is a cost to compatibility: the monotonic increase in implementation complexity. Complexity makes implementation more expensive. This in turn makes the protocol vulnerable to capture by entities with the resources to deal with this complexity.

In a descriptive, lean protocol like Nostr, backwards compatibility remains important (particularly due to the importance of avoiding disruption of network effects), but in a much more practical sense. Backwards compatibility has a purpose, and that purpose is to maintain a good user experience. We don't need to be dogmatic about backwards compatibility. As Ralph Waldo Emerson said, "a foolish consistency is the hobgoblin of little minds."

Backwards incompatible protocol changes aren't as destructive as people frequently make them out to be. This is particularly true because the universal desire for interoperability necessarily encourages conservatism in breaking compatibility.

First of all, the breadth of the nostr protocol makes it possible to change one part of the protocol without affecting the rest. Only the clients which actually implement the feature affected by a change have to care about it. This can be a relatively large number if the feature affected involves follow lists, or microblogging content. But it might be much smaller if it involves some more obscure part of the protocol with only two or three implementations. In contrast, HTML has to be compatible across *billions* of websites.

In addition, there can really be no changes by fiat (even if, as sometimes happens, backwards-incompatible changes make it into a specification document without community buy-in). Nostr is an adversarial environment, and relies on developers to actually implement changes they want to see. To the extent that an implementation has active users, it can choose to go along with the fork, or hold back.

In this way, implementations that want to sponsor either a new protocol feature or an incompatible change can exert pressure on the rest of the network by threatening more or less incompatibility. Likewise, implementations that wish to keep the protocol as is, or not implement a new feature, can do the same. This can be done by shaming competitors in public while maintaining interoperability, or by actually breaking compatibility.

This dynamic is currently playing out with Nostr direct messages. Nostr's original direct messaging implementation (NIP 04) leaks a lot of metadata, which severely undermines user privacy. This problem was severe enough that it led to a new DM standard (NIP 17) which leaks signficantly less metadata. As of the time of this writing, NIP 17 is adopted in the majority of popular Nostr clients, with two significant exceptions — both of which are focused on high quality UX, and so put a premium on DM delivery and product focus.

In a sense, the creation and adoption of NIP 17 DMs can be seen as an attack on these clients because breaking DMs undermines their primary value proposition. The intention behind this "attack" is to protect user privacy but regardless, the introduction of incompatibility forces an eventual resolution to the issue. Eventually, one faction or other will lose users to the other.

One thing to keep in mind, however, is that this kind of adversarial action can cause the protocol itself to lose users, harming all sides. For this reason, backwards incompatibility should be introduced with care. Forks always come with the risk of alienating users, or being on the losing end of network effects, which is a natural restraint to reckless behavior.

Forks are a fact of life in the context of an adversarial system. Any actor can choose to fork at any time, for any reason. Even if some forks are well-intentioned and constructive, others may not be. For this reason, implementers must always be prepared to defend their version of the protocol both technically (through defensive coding) and politically (by educating users about the fork and persuading other developers of their view). Core protocol features will gain momentum over time, contributing to stability, but there will always be a certain amount of chaos at the edges. This chaos is a necessary consequence of a radically open protocol.

Hackability

I've been talking a lot about what interoperability is and how to support it, but I haven't really mentioned what exactly interoperability is for. What good is it for multiple implementations to be able to talk to each other?

This goes back to the problem that Nostr is trying to solve, which is Big Tech's capture of social media platforms and by proxy, their users. In legacy social media, users don't have credible exit — which is to say that they can't leave a platform and retain their identity or their data.

In order for this to work, not only does data have to be open and unconstrained by custodians (which is what relays and signatures give us) there also have to be multiple implementations that can *interpret* this data. A user has to be able to have a choice; if there's only one implementation because of protocol complexity or propriety, then users don't have the ability to exit because they can't enter something else. In order for users to move, there need to be off-ramps and on-ramps.

Openness makes it possible for multiple implementations to interoperate. But that's a fairly low bar. This is true of (for example) the SQL standard, which defines certain data types, syntax, access control, and a framework for transaction isolation levels, making it possible to access the same database from multiple different software applications. But there are limits to the interoperability this provides; application code remains proprietary. While it's possible to look at a database schema and get a vague sense of what the data is, it's much harder to reverse engineer how it's used without looking at the accompanying application.

In the same way, Nostr protocol specifications are only one resource for helping developers create interoperable implementations. This is particularly true because popular features may not necessarily be documented in every case. If a popular implementation isn't open source, it becomes much harder to reverse engineer these protocol features in order to allow others to adopt them. Open-source software functions as reference implementations which can be read, copied, and adapted to support new implementations of the same use case. This indirectly benefits users by providing more options to choose from.

But it also benefits users directly. AI coding assistants are bad at a lot of things, but one thing they can do is allow non-technical users to take off-the-shelf software and adapt it to their needs without having to understand what's going on under the hood. Software fails its users more often because of cosmetic problems than because of flaws in the implementation of their problem domain.

A broken click handler can render tens of thousands of lines of well-tested code useless. In the same way, adding a new button to an existing piece of software is often a trivial operation, but requires the expertise of the end user to determine what the button should do. Certain affordances may also only make sense to a single person, and so would never get implemented due to the maintenance cost outweighing the benefit rendered to the single user.

AI coding assistants, in contrast, enable users to convert rigid, opaque software platforms into environments full of affordances they can take advantage of in the same way an experienced programmer can customize his editor or terminal by writing plugins.

Giving end users the ability to create tools for working with the protocol more directly only increases the utility the protocol offers those users. Even if something goes wrong and the AI hallucinates and creates a fork in the protocol, the damage is limited by virtue of the implementation being a one-off. Diversity of implementations also improves the health of the network itself by distributing control over implementations, making the protocol harder to attack.

This dynamic is maybe more important than it sounds at first. I think this is one of the most important things about open protocols and one of the most exciting things about AI-assisted coding. Used properly, AI-assisted coding has the ability to tear down walled gardens at a scale that we can't even imagine.

If AI has the effect of amplifying user agency in the digital world, users become more capable of shaping the digital world based on their values and goals, which are invariably at odds with the software platforms that we're trapped inside in the current iteration of the internet.

Technology doesn't solve problems, people do. Technology is an extension of human will into the world. And yes, there can be unintended consequences. But when individuals use what Ivan Illich calls "convivial tools", their agency is multiplied and the negative externalities are limited to the scale at which that individual operates.

This creates a diverse ecosystem of self-interested actors who can not only do what they want in the digital world, but protect themselves from other people who would take advantage of them, either through massive structures of control or through exploitation of centralized systems by outsiders.

This attitude of creative use of software in order to overcome its limitations or repurpose it for the user's own agenda is encapsulated in the term "hacker". Hackers look at existing software not as complete or normative, but as an opportunity to do something new and subversive. The hacker mindset is the exertion of human agency over technological systems, and is desperately needed in our day of technological passivity.

It could very well be that AI assistance, both in coding and in research and many other ways, is far more important than the existence of an open protocol. It may even obviate the need for protocol documentation to some extent because LLMs can reverse engineer whatever software is already in use. The role of the protocol could shift from a set of rules for how to implement enumerated functionality to a trellis on which organically synthesized human/machine language can grow.

It's also possible that human nature and our willingness to put up with incredible levels of inconvenience will leave this opportunity untapped. The relative lack of adoption of even simple things like browser extensions is evidence of our natural inclination to be slaves to our technology. But it's possible AI agents may lead to the spread of something akin to the "hacker mentality" even among non-technical users

— not because they make hacking "easy", but by unlocking the universal language of computing for use by non-specialists.

Application Ecosystems

Given the radical openness of the Nostr protocol, some additional mechanisms are necessary in order to cope with the certainty of running into event kinds that implementations don't recognize. It's categorically impossible for clients to implement support for every Nostr data format because new formats are created constantly. Implementations have to have a strategy for dealing with events they don't understand.

Suppose a user embeds an event to a `kind 30023` long form article in a `kind 1` microblogging post and your client doesn't want to add render support for long form articles. How can you handle that event without knowing what it is?

A few mechanisms have been proposed for dealing with this. They could be more robust, but a good foundation has been laid.

First, we have NIP 31, which proposes the inclusion of an `alt` tag on events which describes in human readable language what the event is. Following this, a Nostr client could look at a long form article, see an `alt` tag with the title of the article and an explanation of what that event actually is, and display that to the user.

This is something, but it's not very compelling. This is really a dead-end solution, since all the user knows is that they aren't seeing content that they would like to see. There's no call to action for how to actually view the content (unless a link is included in the alt tag — which would be a brittle and centralizing way to handle the problem). Even a simple error message would probably be an improvement over alt messages because at least it would demonstrate to the user that something is indeed missing or broken.

Luckily, we have a better option. NIP 89 defines some mechanisms that can be used by clients for presenting a call to action when faced with unknown event kinds using a technique commonly known in programming as "reflection", which allows a program to inspect its own constructs at runtime and adapt accordingly.

In NIP 89, An "application handler" is a `kind 31990` event which includes tags describing a given application and its purpose, as well as affordances for opening a given event or profile in that application. Application handlers also include a `k` tag, which can be used to filter handler events.

So in our scenario above, the microblogging client might ask the network for `kind 31990` handler events with a `k` tag matching `30023`. This would return a bunch of application handlers with `web`, `android`, `ios`, or other platform tags. The client can then present options for viewing the long form article to the user.

This allows implementations to reduce their scope in order to avoid implementing features they don't care about supporting, while at the same time improving the state of the network by advertising alternatives or complementary applications to their users.

This isn't something that a walled garden would ever want to do. Their impulse is always to vertically integrate either through acquisition or new product development. On Nostr, because implementations can enjoy the network effect of the entire protocol without implementing all of it, developers can afford to recommend alternatives. In fact, doing so makes their value proposition event stronger because users now have a myriad of alternatives to pick from based on their interests, not those of the platform.

NIP 89 further enhances this ability to search for handlers with `kind 31989` "handler recommendation" events. Because anyone can publish a handler event, it's not a good idea to present them to users without some level of vetting. Otherwise it would be trivial to impersonate an existing client in order execute a phishing attack. Even if application handlers are legitimate, knowing whether a given handler is "good" or not is important signal.

`kind 31989` recommendation events allow anyone to advertise clients that they use to handle a particular kind. In combination with web-of-trust analysis, these recommendations can be used by clients to validate a given application handler. Maybe only one person that the user follows uses an application handler, or maybe a hundred people do.

The more popular app is, the more likely it is to provide the user with a good experience. At the same time, the handler with only one recommendation might still be able provide an interesting alternative to the popular app, eroding the hold that popular applications have on the network. This can help mitigate the concentration of power in the hands of a few popular implementations.

Application handlers are not the only kind of recommendation event on Nostr by any stretch — recommendation events also exist for relays, Cashu mints, DVMs, and more.

In a forest, trees trade sugars to fungi in exchange for minerals extracted from the soil. These wildly different life forms have a mutually beneficial relationship that results from the exchange of these resources. Likewise, Nostr clients and Nostr users benefit from the structured exchange of signed events and functionality.

The existence of recommendations within a web of cryptographic identities enables the development of a social media *ecosystem* incomparable to what has gone before. An "ecosystem" is a complex environment which life forms inhabit. A "media ecosystem" is a complex digital environment that humans inhabit.

In this chapter, I have attempted to highlight the benefits of "radical openness" in protocol development. Top-down design reduces essential complexity, making a protocol easier to understand and implement. In contrast, an open protocol multiplies complexity by virtue of the many participants and their relatively free ability to interact with other users and modify implementations.

An increase in complexity makes an environment harder to understand as a whole, but does not necessarily mean a loss of agency. If the appropriate affordances are provided to users to help them manage the complexity they encounter, their agency can be increased rather than diminished. This in turn allows them to determine their own role in the ongoing emergent complexity of the system, and adapt it to their own purposes.

This is a truer form of "social media" than the sanitized, corporate marketing machines which seek to quantify relationships and extract value. If we "embrace the chaos" of the Nostr protocol, it can become much harder to quantify, let alone control — all to the benefit of its users.

6. Value for Value

The native currency of the internet is not money but attention.

Sure, there are plenty of places where money is paid for goods or services on the internet. But in large part, these payments are made not between users (who have very little money), but between businesses who have a lot of it, as well as the incentive to leverage that capital for increased revenue or market share growth.

The main drivers of trade on the internet are businesses: companies with a product they wish to sell, and platforms that capture user attention and data in order to sell it to advertisers or data brokers.

When regular people like you and me pay for something on the internet, it's very rarely an isolated occurrence - there are almost always complex systems operating behind the scenes which are intended to leverage that transaction in strategic ways.

If you buy a product on Amazon, they upsell you their subscription offering. And once you have a subscription, you have a sunk cost that encourages you to buy more. When you go to buy something, other product recommendations are offered to you, which are often paid for by the companies attempting to sell the product. Once you've bought something, Amazon asks you to leave a review to encourage others to buy. Every transaction is intended to lead to subsequent transactions.

Amazon wants as many transactions as possible to be processed through their platform, not because they delight in brokering connections between buyers and sellers, but because they love to extract their cut from every transaction.

The same is true in a digital goods context, for example, on streaming services. Sure, you can pay for a subscription, but it doesn't mean the platform won't show you ads. And even if you choose to upgrade to the ad-free "premium" offering, you're still caught in a system optimized not to satsify you and let you go, but for the progressive capture of your attention - whether to reduce churn, grow by network effect, or increase stock prices.

We pay for access to the internet with our time and attention, and we do it on their terms, not our own.

Money is Time

In his talk entitled You Wouldn't Zap a Car Crash, Gigi points out this quirk of language: that we *pay* attention and *spend* time. Money is a proxy for the time we have spent accumulating it. But on the internet, that relationship has been reversed. Our time has become a proxy for the money that platforms wish to accumulate.

This is a very clever trick because there are two different ways to view time. Before clocks, time was inhabited either for the purpose of performing work or enjoying leisure, but the two things often coincided. Later, clocks, by allowing us to quantify time spent on a given task, partitioned our time into time with a monetary value (work) and time without monetary value (leisure).

As a result, both work and leisure have been devalued. At work, we're tempted to punch the clock, failing to make the most of our time on the job. At the same time, we often aren't sufficiently jealous of our leisure time because it has no nominal "value". Instead of working to enrich ourselves during our leisure time (whether by doing a hobby, going on a date, or enjoying nature), we fall into the trap of inertia, set for us by a consumer culture.

That's not to say that media and products have no cultural or personal value, but to the extent that the outsourcing of our activity leads to the monetization of our time and attention, our leisure is converted into work — quantified, measured, and capitalized on by third parties.

The time that we spend scrolling, browsing, binge watching, and swiping is time that we feel to be well spent in relaxation, "self-care", or entertainment. But in reality, because these media are designed to be addictive, we spend far more time engaging with them than is healthy. This over-consumption in turn tends to *deprive* us of rest, making us more anxious, distracted, and unproductive.

The simple solution to this predicament is simply to "touch grass". Choosing to opt out is a powerful way to re-assert control over our own time. But the perversion of our digital environment does not obviate its benefits. While many of the things the internet offers are intended to appeal to our baser impulses and vices, the original vision of the internet — the cultivation of communication, trade, freedom, and community — can be recovered.

Gigi posits a world in which the content that is most readily monetized is not the content that appeals to our baser instincts, but to our conscious, intentional allocation of stored value — in the form of actual money. This restores the role of money as a proxy for our time, as it should be.

He uses the evocative phrase "you wouldn't zap a car crash" to illustrate this. You might watch a car crash against your own will, or against your better judgment. But sending your hard-earned money to the person who distracted you from whatever you were doing would require an irrational, un-self-interested decision.

If the internet were powered by microtransactions rather than attention parasitism, only the content that its inhabitants genuinely appreciate, find helpful, enjoyable, interesting, or useful would accumulate value to itself.

Nostr makes this vision possible (at least on a technical level) through Bitcoin micropayments encoded into Nostr events known as "zaps".

Customers, Patrons, and Participants

The idea of "value-for-value" has been around for quite a while, particularly in the podcasting 2.0 space, where listeners can stream satoshis to the podcast they're listening to in order to show appreciation for the content they're consuming. The idea is that because the payment is voluntary, it demonstrates that people are willing to compensate producers based on value received. But this is an inadequate understanding of what is actually going with this model.

In trade, the buyer sends a payment and in return the seller releases the content to them. We see this model at work in a content creation context with Patreon, members-only Twitch streams, Substacks, etc. Trade works really well because it aligns with how value is exchanged. Value is subjective, and price is determined by the balance between supply and demand.

However, as Eric Hughes notes in A Cypherpunk's Manifesto, information "longs to be free". While digital content does have a fixed cost of production, it has an extremely low marginal cost. This means that the supply of new content is limited, but the supply of existing content is infinite. For digital content, price is determined by purchasers' expectation of future production of content, not contention over existing units. Price can be inflated by introducing artificial scarcity through copyright and access-controlled publishing, but this is only a trick which allows the producer to have an asymmetric privilege in the transaction — while most content producers make very little money, some strike it rich simply by replicating the same content and selling it many times.

For this reason, monetizing digital content through conventional trade is essentially incongruous. In this context, value-for-value's reversal of the transaction makes sense — supporters are directly incentivizing the content producer to continue creating new content. This dynamic is explicitly embraced in crowdfunding, in which consumers pledge in advance a certain amount of money for the development of a product.

Because of this dynamic, value-for-value has more in common with patronage than with trade. Just as Julius II sponsored Michelangelo's work on the Sistine Chapel, your $5/mo donation to your favorite podcast encourages the continuation of the project.

Patronage is not without its ulterior motives, however. Apart from purely altruistic motives, patronage is almost always a means for the patron to increase their influence to some other end, whether political, religious, propagandistic, or egoistic. Patronage is, in essence, a public relations tool. In return for promoting art and culture, patrons increase their influence.

Crowd-sourced patronage doesn't have quite this same dynamic because of the relative insignificance of any individual contributor. But supporters still need to get something out of the relationship — people aren't naturally inclined to give away something for nothing, or to voluntarily shoulder their share of responsibility in maintaining a commons.

Rather than influence or reputation per se, the thing that supporters of content creators primarily receive in exchange for their money is a sense of identity or belonging. By giving money to a content creator (or, more broadly, any "cause"), patrons receive a number of social benefits. In this way, distributed patronage is not so different from its more traditional form; in both cases patronage is a form of self-promotion.

Plus, by donating, patrons get to signal their virtue (ironically) as someone who gives without expecting compensation. This in-group signaling (or, less cynically, community building) is frequently more valuable than the project itself.

By contributing to a project, patrons are also assigning themselves a role in its creation, and may gain certain privileges, for example the opportunity to participate in the direction of the project by voting, or to contribute content in the form of fan art or interviews.

In value-for-value podcasting, it's common for boosts to include a message which the content creator then reads during the show. This allows the sender to borrow a little bit of the platform, maybe for marketing purposes, or simply to have their voice heard.

This can be a very healthy dynamic, not to mention pretty fun.

Not only is it nice to see yourself on a leaderboard and try to out-zap other people, but the very act of crowd-funding is itself a unique communication medium full of idiosyncrasies and inside jokes that members of the community can enjoy, and which further galvanize group identity.

This type of content-centric community building can have other second-order benefits, including the development of relationships and a shared set of values.

In most cases, however, this only really works at a small scale. The social benefits that attend community involvement can only scale so far. Once the space that exists within a community is saturated, members' voices become more noise than signal. The time and energy required to actively participate in a given community is also a limit on community growth; community members will take the time to participate to the extent that they are personally invested in the community.

We see this same limit on scale in the growth patterns of certain types of businesses. A company might create a product which is really tailored to their initial users, who show a high willingness to pay. This can encourage additional investment in the business in order to capture more of the market. But the next cohort — even if acquired — is often far harder to monetize, because early adopters tend to be the kind of people who will pay for a niche product, while subsequent cohorts are less invested.

At the end of the day, it is possible to make a value for value model work — but it's important to understand the dynamics actually in play.

Paying It Forward

Micropayments are not inherently social; there are some use cases that may not be as exciting or obvious, but may be even more important.

Money is most commonly used to pay for goods and services. A more obvious statement could hardly be made, but given that the previous section was all about paying for social status (which is neither a good nor a service), it seemed worth repeating.

In the software world, this generally takes the form of puchasing a software product which the user intends to interact with directly. This creates an asymmetric payoff for products that are user-facing vs software libraries that are invisible to the user. This is a well-known problem, and has resulted in attempts to support open-source developers by asking businesses to pledge a certain amount of support.

This system *could* work, in the same way that patronage works; businesses that do support open source can use such donations as a tool for self-promotion. But given that donations are entirely voluntary, and because large businesses are part of the problem Nostr is attempting to solve, this approach has its limits.

But what if instead of asking the companies that rely on open-source software to support developers, we asked *users* to do it directly? With the integration of wallets into Nostr clients, this becomes possible. A client could simply include a manifest which lists the libraries it relies upon, along with Nostr pubkeys for the developers, and users could *directly* donate to all contributors to the software they're using.

This would involve finding solutions for a number of problems, from falsification of developer pubkeys to UX, and the incentives might not work out anyway.

But even if the libraries can't get support this way, the services that users interact with can. To the extent that clients transparently mediate a user's relationship with a third party service (like a relay, or a DVM), they can include affordances that allow users to support those services directly, without having to actually handle any of the money.

In fact, some work has already been done in this area; Keychat supports relays by sending ecash tokens in-band, and NIP 90 includes affordances for paying DVMs for their work. Relays could also charge money to publishers as a spam mitigation mechanism, or to gatekeep relay-based communities.

None of these ideas have been widely adopted because Nostr is still in its bootstrapping phase, but if you squint your eyes you can see this aspect of the new internet taking shape — where money is deeply integrated into the very protocols that facilitate communication. To the extent this can be achieved, our attention can finally stop being the currency of the internet.

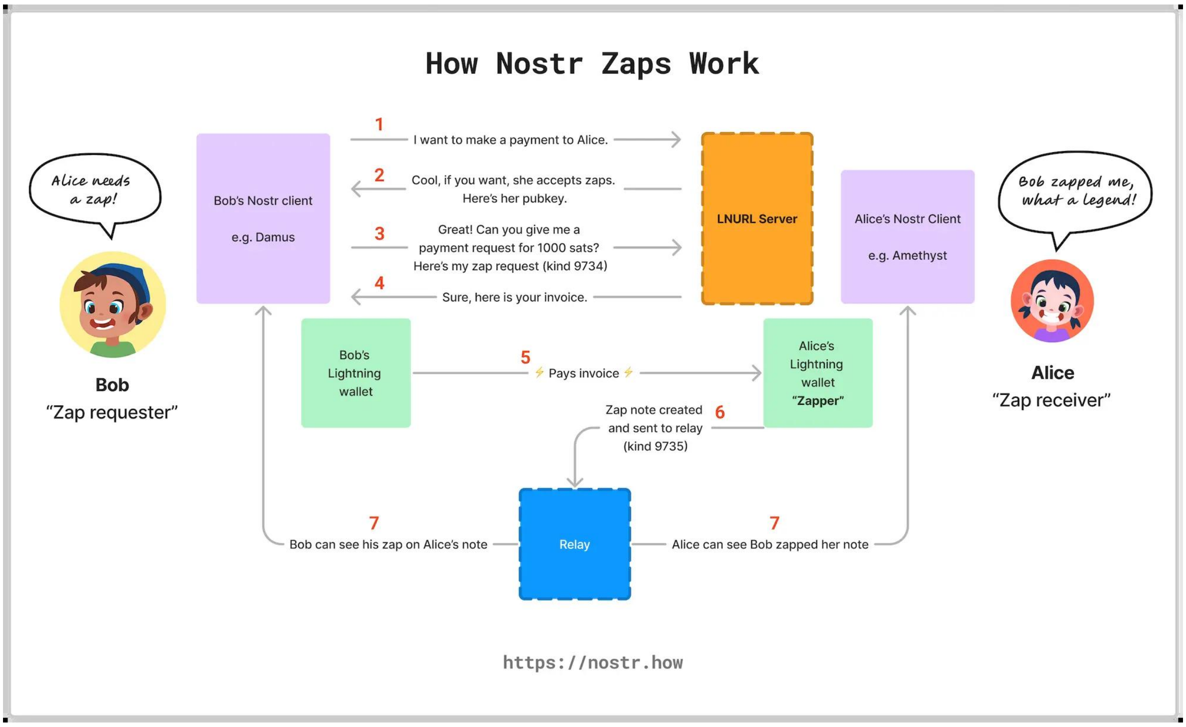
Zaps and Nutzaps

With all that out of the way, let's get into the nuts and bolts of value-for-value micropayments on Nostr.

While other payment mechanisms (e.g., onchain bitcoin, monero, and even fiat) have been proposed, there are currently two standards for micropayments on Nostr, both built on bitcoin.

Lightning Zaps

The first (and original) is NIP 57 zaps, which allow users to couple the payment of a Lightning invoice (requested from the receiver's wallet) to a Nostr event that gets published by the sender's wallet, creating a public artifact demonstrating that they paid the receiver. The flow is fairly complex, so here's a diagram, borrowed from nostr.how:


How Nostr Zaps Work
Alice needs a zap!
Bob
"Zap requester"
Bob's Nostr client
e.g. Damus
1 I want to make a payment to Alice.
2 Cool, if you want, she accepts zaps. Here's her pubkey.
3 Great! Can you give me a payment request for 1000 sats? Here's my zap request (kind 9734)
4 Sure, here is your invoice.
LNURL Server
Alice's Nostr Client
e.g. Amethyst
Bob zapped me, what a legend!
Alice
"Zap receiver"
Bob's Lightning wallet
5 ⚡ Pays invoice ⚡
Alice's Lightning wallet "Zapper"
6 Zap note created and sent to relay (kind 9735)
Relay
7 Bob can see his zap on Alice's note
7 Alice can see Bob zapped her note
https://nostr.how

Zaps support all of the value for value use cases mentioned above and can even be used for paywalled content, since zap receipts are signed by the receiver's wallet key. As long as the receiver trusts their wallet, they can respond to these receipts in any way that they choose.

There are a couple important limitations to zaps. First is that it requires both the sender and the receiver's wallets to be online. This means that wallets either have to be self-hosted, which is a significant hurdle for non-technical users, or they have to be custodial.

The other important flaw is that zap receipts are only useful to the extent that the receiver's wallet can be trusted to competently issue them. This means that the wallet can create as many fake zaps as they want by issuing receipts for invoices that weren't actually paid. Or, zap receipts might be incorrectly constructed, delivered to the wrong relays, or just not issued at all. Because of this, it's fairly common to pay for a zap but fail to receive the social benefit — without any ability to get a refund.

Even if a wallet provider is both competent and honest, anyone can create any number of sock puppet accounts and use them to send zaps to their own wallet. To avoid this problem, implementations have to filter zaps by sender based on web of trust (or another mechanism). For services that provide a "trending" feed based on zaps, this is especially important.

Nutzaps

Next up we have NIP 61 "nutzaps", which are an entirely separate standard powered by pubkey-locked Cashu tokens (hence the pun).

The neat thing about ecash zaps is that they don't rely on any third party to facilitate a transaction — the ecash is *in* the wallet. This makes implementation significantly easier; with no external services to work with (apart from mints, which are interchangeable), a NIP 60 ecash wallet can included in any application simply by calling a library. And because NIP 60 wallets are stored as `kind 17375` Nostr events, your ecash follows you across the network, just like your profile or relay selections.

Because the Cashu tokens are locked to the receiver's pubkey, the receiver can immediately verify that the eCash has not been double spent. This solves the problem of untrustworthy receiver wallets, although it doesn't provide sybil resistance on its own.

Custodian Risk

Both eCash and custodial lightning payments are vulnerable to custodians in different ways.

In both cases, the reputation of a given wallet or mint is largely predicated on how well the service is run in terms of availability, performance, or operator reputation.

The difference between the two is that because of money transmitter laws, custodial lightning wallets are usually more easily identifiable, but also subject to arbitrary enforcement of regulations.

At scale, a single custodian for many wallets becomes a central point of failure, not only in terms of availability, but also privacy, solvency, and censorship. While most wallet providers are unlikely to simply steal funds and disappear, they may be compelled to implement KYC/AML checks or arbitrarily close certain accounts.

At the same time, eCash mint operators (who are often anonymous in order to avoid scrutiny under money transmitter laws) are unable to target individual users due to eCash being a bearer instrument, but can at any time drain the bitcoin backing the eCash, making tokens worthless. This attack would completely burn the provider's reputation (and would likely result in prison time if the operator can be identified), but it also can't really be anticipated by users.

Distributing user funds across multiple mints or wallets does protect the user to some extent because if a service provider rugs, users will lose only part of their balance. This is similar to Nostr relay redundancy, which works because information can be freely copied. Unfortunately, money doesn't work the same way — even if you only lose some of your money, you still really lose it.

One thing that can mitigate this custody risk, in either case, is the "Uncle Jim" model, in which lightning nodes or eCash mints are provided to users within an existing trust structure — for example, within a family, friend group, church, or local community.

7. Communities

In 2020, Twitter surprised everyone by censoring several high-profile accounts. This directly contradicted the ethos they had carefully cultivated of being a place where news breaks and ideas can be discussed. When they started censoring political content to appease advertisers, they undermined their journalistic ethos in the same way that mainstream media had already been doing for decades.

At Twitter's edges, there were some built-in access controls—for example, private accounts and blocking—but the main use case of the platform was to democratize the spread of information and ideas through publicly broadcasted data. Nostr's architecture is well-adapted to this use case, in that it optimizes for censorship resistance and easy distribution of content.

Access controls are similar in many ways to censorship, in that they limit the availability of published data to certain authorized readers. For this reason, an architecture optimized for censorship resistance is necessarily not optimized for access controls or privacy.

I've covered the implications to user privacy of an open network elsewhere, but in this chapter I want to talk about access controls on Nostr from the perspective of digital communities. Communities have a different goal than broadcast social networks, which presents the problem of how (or whether) to adapt Nostr to use cases that require access controls.

Communities are for People

Journalism is about information, facts, and opinions which can be objectified, analyzed, shared and remixed freely. Communities, on the other hand, are about people whose identities are complex and dynamic, who have a personal experience of the world, are self-interested and responsible for their actions, are self-aware and self-reflective, who have beliefs, intentions, preferences, memory, and imagination.

While the design of any artifact or system requires care to ensure that the designer's intention is executed with a minimum of negative externalities, extreme care must be applied if humans are involved — not only because human life and well-being is precious, but also because humans have a unique gift for screwing things up.

The word "community" comes from the latin word "communitatem", which is a contraction of "com", which means "together", and "unitas", which means "unity". So, "community" is "being one, together". This may sound redundant, but it captures the inherent tension of existing in community — the individual is not erased by the community, but is instead uniquely fulfilled in it.

While broadcast social media is monolithic and more or less global, communities are necessarily local and have distinctive subcultures depending on their membership or stated purpose. Communities may form around a topic, around a cause, around a location, or around a system of values. They may have leadership, or be entirely organic. They may be strict about what kinds of activity is in scope, or they may be much more all-encompassing. They may be as small as two people — a single relationship — or as large as millions of people.

Understanding the different types of communities (and how to serve them with software) is irreducibly complex, not just because each one is a unique combination of each of these concerns, but because people are involved — communities inherit all the complexity and dynamism of their members.

Communities also exist in community with one another — to the extent that members of different communities interact (or have membership in multiple communities), communities will be affected by each other. A community is itself a higher-order organism which inhabits an ecosystem comprised of a whole complex of technologies, contexts, and individuals.

Digital Architecture

When writing software for communities, we are creating *digital spaces*. Just as physical spaces must be architected to acommodate the activity they're intended to facilitate, digital spaces must be designed around the type of people and activity that will inhabit them.

Spaces (even digital ones) are not abstract — they necessarily have certain particularities. I think technologists can often fall into the trap of thinking that just because software is infinitely malleable, so are humans. To the extent that a community has certain characteristics, software must be opinionated in supporting those. For the same reason, a given digital space will not be appropriate for every community.

Some design decisions can vary from client to client, but others have to be baked into the protocol layer. A kitchen is a kitchen, and shouldn't be used like a bathroom — in the same way, chat rooms, message boards, or calendars are all distinct media that should be treated differently.

In the remainder of this chapter, I will catalog five different categories of community along with some ideas about how to accommodate each one: social clusters, small group chats, topical discussion forums, owned communities, and commons. This breakdown is in no way normative, but hopefully it is useful when thinking about digital "communties" as a whole.

Social Clusters

In their most organic form, communities exist only as emergent properties of individual relationships within an open network. Another term for this is "social clusters", which can be identified mathematically through graph analysis, or simply by gestalt.

These kinds of communities are open and informally delineated because they are the result of the free association of individuals. And yet they are real and do exist — for example, it's fairly straightforward to assess whether someone is a member of the libertarian community, a particular friend group, or the Christian Church.

None of these communities enforces who can be a member. Individuals are free to form associations, and each of these examples transcends the particular instances of community within them (for example, a particular political party or church). This complex of one-to-one relationships and transitive relationships is what forms large, open social cluster-based communities, and frequently form the basis for the emergence of more formally-defined communities.

In this type of community, topics of discussion (such as current events) may predicate an actual discussion between individuals, but don't usually define a given social cluster on their own.

Social clusters constantly re-form as discussions evolve and relationships form. Clusters also aren't really limited by the Dunbar number, since clustering is transitive, and can grow to billions of members. Member privacy is either protected by virtue of the privacy of individual conversations that feed back into the shared culture of the cluster, or is voluntarily given up in order to attract more attention.

The prominence of various members within a social cluster varies widely as well.

Prominent members serve to solidify the group identity by embodying it in a concrete persona, while the vast majority of members simply follow and either support or register dissent with their "thought leaders", sometimes achieving notoriety themselves, or shifting to an adjacent social cluster. And of course, members may inhabit many social clusters simultaneously, even ones that may not seem adjacent from a network analysis standpoint.

This use case is served well by broadcast social media, but is simultaneously supported by smaller-scale forms of association which allow members to evolve the group identity without being completely public about it.

Group Chats

At the other end of the spectrum are group chats. Group chats are a fairly well-defined type of community because in general their members are also well-defined. Many group chats are comprised of only two people (for example text message conversations or direct messages, but can scale to dozens, or even hundreds, of members.

Group chats are defined by there being a relationship between each member. It may be that multiple people are added to a group chat who don't have a pre-existing relationship, but because every member is necessarily a recipient of every message, every member is implicitly connected.

Stated more formally, if there are `n` members in a group, there are `n * (n - 1) / 2` connections between members. The number of relationships, and therefore the complexity of the social dynamic, increases quadratically as the number of members increases.

This is of course also true of regular in-person social dynamics. When one person is added to a conversation, the dynamic completely changes. If another person then leaves the conversation, the dynamic changes again. A palpable change in tone and topic occurs depending on the conversation's members.

Because of the each member pair necessarily has a unique relationship, the requirement that there be a high level of trust (or willingness to engage with other members) within the group is more strict than in any other type of community. Every conversation involves all members, meaning sidebar conversations are impossible without creating a separate group with a subset of members. This becomes more awkward as the group grows.

To complicate matters, trust is not binary or even one-dimensional. You don't either trust someone or not trust them, you trust them to a greater or lesser extent. Furthermore, you may trust someone in certain areas, but not in others. I might trust a friend to recommend a book, but not to watch my children.

In a conversation with only one other person, what information you share depends exclusively on your relationship with that person. But the moment you introduce someone else to the conversation, what information can be shared is limited by the *least* trustworthy member of the group. And because trust is multi-dimensional, you may be willing to talk about topic A with Alice, and topic B with Bob, but not B with Alice or A with Bob, leaving you very little to say.

The result of this is that as the size of a group grows, chats often stagnate. You invite your five close friends, they invite their friends, and before you know it, the group has 50 members and no one's willing to say anything because they don't want to bother everyone in the group.

For this reason, the medium of group chats simply cannot scale without a way to limit the scope of a given conversation to a subset of the group. In practice, this feature is not usually included in group chat implementations — instead, users are encouraged to create an entirely different group.

For this reason, chat groups tend to be either ephemeral and practical, organized around some short-term need for coordination, or they are predicated on a high level of relational trust shared between all members of the group.

Discussion Forums

Let's bounce back now to the other end of the spectrum. Just this side of "social cluster" type communities are "discussion forums".

Discussion forums are not formless and self-assembling like social clusters are, and neither do they require high levels of trust like small groups do. Instead, discussion forums are predicated upon some *raison d'être* — in other words, a topic of conversation. This topic can vary, from auto repair, to Monty Python fandom, to mathematics.

In every case, the thing that draws members into a community is some object or idea of common interest. Discussion forums may be educational, oriented toward entertainment or professional development, or even places for people to connect over emotional trauma or shared illness.

Discussion forums are not at odds with people, but they are abstracted from them. It would be inappropriate to start a political polemic on a discussion forum centered around gluten intolerance. Doing so would be equivalent to going to your local health food co-op and canvassing for a political candidate.

At the same time, many discussion forums have an "off-topic" room where things unrelated to the topic at hand can be discussed. This is a recognition that when people come together and form a community around a particular topic, they may form relationships that transcend their original context.

Meetups are a real-life analog to discussion forums. You might want to go jogging with other people or learn the game of Go. These activities are valuable in themselves, but can also serve as a starting point for building social relationships that are best served by a different venue.

Discussion groups are similar to social cluster-type communities, in that their membership is largely informal and open (except to people who don't follow the norms of the community). This poses a dilemma. Discussion forums are essentially open in that the conversation surrounding a given topic should not be gatekept, but at the same time moderation is required to keep discussion "on topic".

This results in a dynamic frequently seen on Reddit and ActivityPub, where moderators end up treating a discussion forum as their own private fiefdom, enforcing rules arbitrarily, and banning other members at will.

Discussion forums frequently fragment when they reach a certain size because the topic, which is owned by no one, is coupled with group infrastructure, which is owned by someone. But this breaks down the forum's network effect, which is part of the core value proposition of creating a digital space in the first place — to gather people interested in a topic in *one* place.

Nostr's multi-master architecture can help solve this. If a member of a discussion forum chooses to fork the group to their own relay with different moderation policies, they can do that without sacrificing interoperability. A user who is a member of both can participate in both at once, merging the content in one interface.

This is essentially how NIP 29 works, where a group "belongs" to a relay, but multiple versions of the same group can exist simultaneously. If you're a member of the group on both relays, you get the superset of content across the two versions.

Other versions of this could be imagined that are not necessarily supported by relays, but where moderation is a function of user preferences. Users might choose to see all posts, regardless of moderation, or only posts that have been approved by moderators, or select their own moderators, or use web of trust or proof of work heuristics to filter posts.

All of these are legitimate options in a discussion forum context because missing data is not essential to the conversation. Because there are frequently a large number of members involved in a discussion forum, the conversation is very broad. Filtering the conversation based on the relationship a user might have to various social clusters within the discussion forum is the user's prerogative.

In this sense, missing data isn't a bug. FOMO isn't a problem because the user himself has chosen policies that cause that data to be missing. This is one of the neat things about Nostr: it is partition tolerant. Not everyone needs access to everything.

In practice, Nostr implementations of discussion forums haven't gotten this right. NIP 72 was intended to be "reddit on Nostr", but ended up killing communities by being overprescriptive. Not only did it define how moderators are selected, (which should be out of scope given that Nostr's purpose is for giving the end user control over their experience), but it also prescribed that posts should not be shown unless moderators approve the posts. Which means that in order for a discussion forum to function, you need highly active and invested moderators manually approving every post.

But this is not how discussions grow. They start out informal and unmoderated, adding moderation as problems accumulate as a result of scale. To the extent that the leadership of the discussion forum fails to grapple with the difficulties of managing a group on a social level, the group itself will fail. This is not a failure of technology, but of the people managing the group.

The discussion forum group type is not limited to forums or subreddits, but can be seen anywhere that open access is granted to participate without any prior relationship being established. For example, livestream chats, Discord servers, or blog post comment sections. In any of these cases, the user can simply register, leave their comment, and never come back. Access control is only implemented when a participant breaks the rules or expectations of the discussion forum, whether by posting spam, harassing other users, or going off-topic.

Owned Communities

Next up we have "owned" communities, which are similar to but distinct from discussion forums. Discussion forums are predicated on some general topic that doesn't necessarily belong to any of the members, whereas owned communities are organized by, and for the sake of, some centralized entity.

One example is Patreon members' areas, which attract users by virtue of their interest in the content creator, who has the right to kick anyone out of "their" space at any time. Internal company chats on (say) Slack, are another example. Access to the chat is justified by a person's status as a member of the company and is revoked when that membership ends.

If the company or content creator decides to move the community to a new platform (or delete it entirely), that's their prerogative. Members have neither the right nor the incentive to move discussion to a platform not owned by the topic of discussion, because the whole point is to be close to that person or entity. Doing so would make the group into a discussion forum, because it would then be a group of outsiders, rather than a group of "members".

This kind of community is inherently centralized, and so centralized control is a feature, not a bug. The system administrator and the owner of the community are either the same person or have a relationship of some kind. The admin may be an individual contractor, but more often than not it's a software platform.

Companies rent Slack (the product) from Slack (the company) because they don't want to bring the administration of a chat solution in-house. This is a reasonable decision, but comes with certain costs—namely the platform users' privacy and security. So, to the extent that an entity is likely to suffer abuse by the third party they've contracted to host their chat platform, they should consider an alternative third party or a self-hosted solution.

Because of the characteristics of owned communities, they don't benefit from a protocol optimized for decentralization per se. Although to the extent that they interoperate with other products using the same protocol, they may be able to provide a superior value proposition in the form of cryptographic identity support, signed data, and protocol-native content types.

Commons

Finally, we have a type of community which I'm calling a "commons". This may not exactly be the best name for it, but the idea is that this type of community is "owned" in common by its members and self-organized through political means. By the way, what I mean by "politics" is not mere bureaucracy, but communication and cooperation for the purpose of creating of an environment that the community inhabits, and which promotes their formation according to certain values. Politics is the recognition that only humans can grapple with the complexity of human life together. Fisher Ames describes it this way:

Politicks is the science of good sense, applied to public affairs, and, as those are forever changing, what is wisdom to-day would be folly and perhaps, ruin to-morrow. Politicks is not a science so properly as a business. It cannot have fixed principles, from which a wise man would never swerve, unless the inconstancy of men's view of interest and the capriciousness of the tempers could be fixed.

Politics involves values, responsibility, individual agency, leadership, and compromise. In other words, it is a social phenomena which seeks to resist the entropy of conflict by structuring reality into something hospitable to its inhabitants. In this way, a digital commons must be organized in the same way as a state, a city, or a home.

Self-organization does not imply a particular mode of organization, like a democratic voting system or a cryptographic DAO. While a decentralized system of voting or staking might be interesting (and appropriate to certain communities), more conventional centralized techniques can still be used to manage a commons-type community insofar as they don't conflict with the community's ability to hold its leaders accountable.

In a commons-type community, the roles that a given member takes on are especially important, because the commons is sustained by its members willingness to participate in its cultivation. A commons-type group is not only an environment in which a community exists, but the product of its members' activity.

Digital spaces that accommodate commons-type communities are therefore a very large, and very difficult, design space. They can be supported by chat applications or by owned group models, or even by digital spaces originally intended for discussion groups or broadcast social media. But the selection or construction of a digital space must be carried out in close partnership with the actual members of the community.

Roles and Spaces

In the vast majority of communities (not just commons-type communities), membership is not binary, but exists on a spectrum. This is obvious in terms of moderation and administration. If not all members are moderators, some members have a role which grants them privileges not accorded to the other members of the group.

But membership has other kinds of gradation in both formal and practical terms. Formally, members may be granted access only to certain parts of the group or certain abilities. Their content may be promoted more or less to other members of the group.

Alternatively, members may be afforded the same nominal status as other members, but participate more or less frequently. Their influence within the group may increase or decrease organically based on the esteem that their peers have for them, and not based on any procedural mechanism.

An example of gradations of membership in the real world is that of catechumens, which in the Eastern Orthodox Church are not permitted to partake in the Eucharist until they are baptized and chrismated. In the past, catechumens were even excluded from the second half of the liturgy. This represents a literal architectural partition which distinguishes between members.

These distinctions are nearly universal. The vast majority of communities benefit from some form of partitioning of the spaces they inhabit. In a digital space, these partitions can be erected along three different axes: identity, topic, and content type.

Splitting a space up by identity means applying a different set of access controls to different members of the group. For example, within a chat application you might have locked rooms that are accessible only to certain members. Access is granted based on the member's role, either in a social sense or in a formal sense — for example, depending on whether they've paid a subscription in order to gain access.

Partitions based on identity may not necessarily be enforced with access control, but might exist for other reasons. Conversations may simply be irrelevant to certain members of the community, rather than necessarily private.

Digital spaces may be partitioned along topical lines as well. These may be longer-lived topical discussions — for example, a "general" topic or an "announcements" room — or they may be shorter-lived topical divisions, for example a conversation related to an upcoming event.

Topical divisions exist for user convenience and to allow users to opt out of messages they're not interested in. This allows the community as a whole to function without cluttering the common area.

Finally, digital spaces may be partitioned by content type. For example, you might have a company calendar which is accessible to all members, and which includes events of any topic or author, but which only includes calendar events — not microblogging posts or chat messages.

Calendars are an obvious example, but partitioning by content type can be used to good effect in other ways as well. Certain conversations are better served by rapid-fire short chat messages, which are often more synchronous than other forms of written media. Other discussions might best be served by long, thoughtful posts embedded in a slower medium like a series of articles. Some content may also be more or less ephemeral, depending on content type. For example, a wiki should last pretty much as long as the community does, whereas chat messages can be allowed to recede into memory and potentially even be deleted after some time.

All three of these partitioning schemes — by membership, topic, or content type — can be combined. Maybe only the company's secretary can create events on the company calendar. Or maybe only the church's pastor can start a live stream. Similarly, certain members might have access only to certain topics, like budgets or facility maintenance.

And as usual, wherever there are interactions between different things, you get combinatorial complexity. Special cases can be nuanced literally forever. A software developer's job is never done because software is essentially the management of emergent organic social complexity with linear, propositional, systematic tools.

Maybe AI will change all of this. Introducing AI agents into a community context can be a great way to empower members to use the community's infrastructure in unexpected ways, whether to do search and analysis, create new software solutions, or simply accomplish one-off tasks. The deep integration of AI into software solutions in general, and communities in particular, is inevitable, but care should be taken to anticipate negative externalities that might result from its use.

If, for example, a community includes an AI search tool in order to ask the community as a whole a question, this can actually result in the slow dissolution of the social bonds that hold a given community together. In other words, even if the question "where can I find raw milk locally?" is answered *ad nauseam*, the question itself serves as a basis for the development of relationships which is indispensable to community formation.

Managing Complexity

In every community scenario, there is significant complexity involved in implementation. Different group protocols on Nostr have attempted to manage this complexity in different ways.

Some, like NIP 72, NIP 48, and Nostr MLS manage complexity within the protocol itself. Others, like NIP 29, offload much of the complexity of access control to the relay by means of server scripting or other customization.

This has an important trade-off in terms of flexibility and applicability to a given community. The more functionality is defined by the protocol, the less able a given community is going to be to adapt the protocol to its own purposes.

This relates to the decision on whether to use encryption or server-side access controls to enforce access control policies.

In an encryption-based digital space implementation, all clients have to be on the same page about what the group policy is, which means administration and consensus has to be baked into the protocol itself. This may be done using a plugin architecture, like what MLS has, but all clients still have to enforce the same set of rules.

In general, lifting complexity into client implementations is the right decision because "relays are repositories". But this is only true with respect to content type support, not necessarily with respect to community organization, which is a form of access control and therefore belongs conventionally in the domain of relays.

By pushing the implementation complexity of access control and membership policies down to the relay, it becomes much easier to build interoperable clients. Because all the client has to do is speak the content-type part of the protocol, the relay can be allowed to manage access control and content curation using any method without increasing the surface area of the interface that it exposes.

Trusted servers can also implement any policy encrypted groups can, even simulating forward secrecy and post compromise security (without any of the guarantees actually provided by a cryptographic implementation). Whether to use servers to enforce group policy rather than a protocol leveraging encryption thus becomes a question of whether the group's need for trustless, cryptographically-enforced security and privacy outweigh its need for highly-customizable functionality. Server-enforced policies are also going to be much cheaper to implement, since no protocol work has to be done.

At the same time, almost anything a server does can also be automated within the scope of an encrypted group. For example, if group access is granted or revoked based on the status of a member's subscription (in the case of a paid group), a program acting on behalf of a group moderator can implement whatever policy is needed. So while encrypted protocols may be more rigid, and require more careful design than server policies, they are able to provide a more robust solution over the longer term.

Interface Complexity

Even in the case of server-enforced policy, clients can't be entirely passive. At the very least, the different partitioning schemes I enumerated earlier has to be supported by the interface that clients expose. However, different clients may provide a different interface to the same groups. An interesting question here is whether a diversity of interfaces makes digital spaces more or less habitable.

Imagine that you could cook a meal with someone else in two separate kitchens simultaneously. You could use your kitchen, which you're familiar with, they could use theirs. And if you have an appliance that they lack, they can hand the food over to you, and you can use the appliance to process it.

Different clients have different affordances. In some ways, this is obviously necessary. An administrator is going to need administrative tools, which are going to be completely inert if presented to a regular, unprivileged user — it's not unreasonable to provide these users with a separate tool tailored to that use case, rather than cram administration into the same app used by regular group members.

But what about the partitioning of content? If one user uses a client which lumps all chat conversations into a single view and another one separates them by topic, will this cause failures of communication?

In many cases, I think, yes. The only real solution to this is to include expected behavior in the specification. For example, NIP C7 is one of the smallest NIPs in the Nostr repository, in that it simply defines a kind that's used for chat messages with almost no other concerns. But in that NIP is the stipulation that replies must not be deeply nested, but should instead be presented linearly. This provision is completely unenforceable, but is vital for providing the same user experience across clients.

In this chapter I've only scratched the surface of what communities are and how to serve them best within a digital medium. The point I most want to get across is, when developing solutions for communities we should not be afraid of the introduction of multiple sub-protocols which cater to different types of community.

Appendix A: Alternatives to Nostr

I hope that over the course of this book I have made it clear that Nostr is a unique and compelling solution to the decentralization of social media. But at the same time, it has numerous flaws and trade-offs that make it difficult for developers to use, and which leave it open to attacks on the protocol.

Nostr is generally optimized not to create an ideal solution, but to create a "good enough" solution. In a sense, Nostr trades technical purity for social utility, and this may not be the right trade-off.

Of late there has been a proliferation of AI-built Nostr applications. In one sense this is great because it allows individual Nostr users to scratch their own itch and to create products for other people. It also means more use cases are served by Nostr's identity and architecture. But at the same time, it saturates the ecosystem with noise, and to the extent that these applications are architecturally flawed, they may actually undermine the decentralization of Nostr.

It's neither easy nor straightforward to implement relay selection heuristics, application handler selection, and web of trust analysis. These things are largely invisible both to developers hoping to accomplish a particular task and to users intending to use their solution. But ignoring them results in spam, missing content, and centralization forces. I hope that the Nostr community can exercise discipline to maintain the integrity of the protocol rather than use it as a way to spin up applications that aren't serious and won't be maintained.

Open source software is not just the publishing of code. It is the commitment on the part of the developer to maintain it, and to work with the community that forms around it. The same is true of Nostr. Tragedies of the commons are the natural end state for any shared project. We have to care, and we have to act to protect the things that we love.

With that said, I want to enumerate some alternatives to Nostr and describe in brief why I don't see them as viable solutions to decentralizing social media.

Matrix and ActivityPub

I'm handling both Matrix and ActivityPub at the same time because their architectures are very similar, and so the same criticisms apply to both.

Both protocols are built around federation between "home servers", which have the ultimate ability both to authenticate user activity and store content. User identity is not cryptographic, which means that if a home server in either system deplatforms a user, they lose their identity. Data is not signed by users and so it can't be re-uploaded to other home servers.

Neither protocol has credible exit — both systems are the application of the centralized platform model to a decentralized platform model. But the administrators of these platforms are no less powerful within their domain, and in many ways are less acountable; large companies have to follow the law, and are highly sensitive to the effect their actions have on their reputation. Mastodon server admins are far less accountabile to their users.

Small servers are also far less stable and might disappear for any number of reasons other than deplatforming. The only way to have sovereign identity on Mastodon is to run your own home server, which is not something most people are willing to undertake. Federated systems make the fatal mistake of orienting themselves around servers, not users.

Scuttlebutt

Scuttlebutt is the spiritual precursor to Nostr in a lot of important ways — in both cases, cryptographic identity puts users are at the center of the network, and servers are no more than dumb relays that allow user devices to communicate. In Scuttlebut, servers (also called "pubs") were an optional addition to the network topology, since applications were originally built to communicate peer-to-peer.

Scuttlebutt has one main flaw, however, which is that each user event is linked to the previous one published on that device. This means that in order to validate the most recent event by a particular key, you have to download all of that user's data. This doesn't scale to a larger social network, where you might only want a single event from thousands of different users.

Another problem with Scuttlebutt is that a single key can't be used for multiple applications at the same time. Because data is linked, unsynchronized use of the same key for signing data would result in a fork in a user's event chain, and one of the messages would be dropped. This could be solved in many ways, but in practice Scuttlebutt has encouraged its users to use a separate identity for each device, which makes it harder to maintain an identity in different contexts and across time

Nostr is inspired by and builds on Scuttlebutt's successes.

Pubky

Pubky is a social protocol created by John Carvalho, which uses cryptographic identity and home servers.

There are two parts to Pubkey. First, cryptographic identities are used to create and maintain entries on the Mainline DHT in order to route requests to the correct server. The Mainline DHT is the most used DHT in existence and is highly Sybil-resistant, which makes it a much better index than Nostr relays can provide.

On Nostr, indexes are simply relays that store a particular kind of event. This works ok, but is an area where Nostr might borrow from Pubky in order to improve the resilience of Nostr bootstrapping. Unfortunately, Mainline requires the use of a different elliptic curve, and so a different DHT will need to be used or Nostr keys will have to be adapted to whatever curve Mainline uses.

The second half of Pubky is home servers, which have the same problems that Mastodon and Matrix home servers do, except that they can't nuke user identities (since identities are cryptographic). However, content is not signed, and is stored only on a single home server. For most people, servers are going to be hosted by a third party, which would then have the ability to censor any content it chooses.

Pubky almost gets user-centric network architecture right, but fails to solve the problem of aligned storage.

Bluesky

Bluesky was originally intended to be something very like Nostr. And on paper it is — the architecture is sophisticated, and certain problems are addressed that Nostr doesn't try to tackle (like a global view of the network).

But in practice, Bluesky has only managed to re-invent centralized social media on a new architecture. Because of a desire for a complete view of the network, Bluesky clients access content through "app views" powered by a "firehose" relay which collects data from "personal data servers" across the network. This firehose becomes a centralized chokepoint which is expensive to run at scale, pricing out smaller administrators and making users dependent on a third-party server.

Because of this, Bluesky's relays and app views are mostly (if not entirely) run by Bluesky itself. This in turn allows them to implement censorship, ban users, and provide premium features like account verification.

Blue Sky seems to be turning out to be the same kind of next generation monolithic social media platform with all the same problems — but with a more sophisticated architecture.

What Makes Nostr Different

In contrast to all these alternatives, Nostr is the only one that provides both cryptographic identity and decentralized access to content (powered by signed data and redundant storage). This combination orients the network around its users, not its platforms.

An important implication of this architecture is Nostr's tolerance for network partitioning. On Nostr, there is no "global". This may at first seem like a flaw, but only because we have so thoroughly accepted the fallacious idea that a global view can exist at all.

Internet platforms invariably simulate a "global" view of the "network". But in the case of social media platforms, this "network" is no more than a database that they host, and which excludes mountains more social data that doesn't reside on the platform. In the case of search engines, the "network" that the internet forms is genuine, but is largely unavailable for indexing by search engines — whether because of access controls, `robots.txt` directives, or darknet content.

In the real world, there is no global view. In real life, a global view would be more a liability than an asset. Even if we had a way to filter out the immense amount of noise coming from other continents, languages, and viewpoints, the ability to omnisciently navigate to one or another viewpoint would only serve to erode the sense of belonging we get by being in a particular place, time, family, community, culture. The vast majority of people, opinions, events, and facts that exist in the world are irrelevant to my life. Network partitioning is *how humans function in a social context*.

Because of its dual reliance on relays and cryptographic identity webs, Nostr is able to align its network topology with the social graph. This means that social connections can span multiple relays, and relays can serve multiple social clusters. But it also means that not all parts of the network need to be connected in order for people to make the particular connections that bring their life meaning. This is the promise of decentralization. Nostr does not flatter us that we can be everywhere all at once. Instead, it encourages *digital localism* — a more humane way to inhabit digital spaces.

Starting Over

Nostr could be improved. It has numerous cosmetic design flaws, relies on aggressive synchronization in order to avoid race conditions, a number of NIPs overload kinds and tags, bootstrapping is a convoluted process, and incompatibilities between clients are common.

Instead of having a single reference implementation that's used by all clients, Nostr has dozens of separate SDKs, which means differences in paradigm and nomenclature go all the way down to the bottom. In some ways, this is a feature rather than a bug (see Chapter 5), but it also makes for a far more chaotic ecosystem.

What if we started from scratch with Nostr? We could have a single reference implementation, built on sophisticated design decisions. We could use binary protocols and CRDTs. Such a project could work.

But Nostr is more than a technical specification — it is a community. The ability to hack on Nostr is part of what makes it special, and the applications that actually exist are what make it real. None of them are perfect, and many of them will disappear over time. But with Nostr, we have something. Whereas with a successor protocol, we have nothing.

That situation could change. But it's possible to make the perfect the enemy of the good. If Nostr is good enough — which is a big if — then abandoning it would reset the clock on several years of development.

People have been developing decentralized protocols for as long as I've been alive. Over and over they get captured. Maybe the destiny of Nostr is to get captured, but at the same time to provide a brief respite for its users from the interference of governments and corporations.

Nostr doesn't have to live forever in order to succeed. And Nostr doesn't have to be used by the whole world in order to provide value. It's already providing value to the tens of thousands of people that use it every day.

I would like to see more adoption, and I would like to see a better technical foundation, but we can't have all that we want. Part of building a commons is compromise — not only with one another, but with reality as we find it.

Nostr is nothing but compromise. For that reason, I want to end this book by encouraging you and every developer who builds on Nostr to embrace compromise. A *polis* is a place where a community of people lives, whose lives in turn build and rebuild the *polis*. Do not withdraw from the difficult work of coordination and collaboration — instead, embrace the *politics* of protocol design.

As Aristotle puts it:

> A state is not a community of living beings only, but a community of equals, aiming at the best life possible. Now, whereas happiness is the highest good, being a realization and perfect practice of virtue[...] different men seek after happiness in different ways and by different means, and so make for themselves different modes of life and forms of government.

We are living beings, and this is our *polis*.

Appendix B: Resources

NIPs

As discussed in chapter 5, many Nostr developers have opted to self-publish specifications for protocol extensions. Below is a list of NIP repositories, including both more or less "canonical" sources alongside third-party extensions.

- NIPs on GitHub — the original NIPs repository
- NIP PRs — proposals that may be anywhere between idea to partial adoption
- BUDs — "Blossom Upgrade Documents"
- NostrHub allows users to publish custom NIPs on nostr using `kind 30817`
- DIPs — "Damus Improvement Proposals"
- NKBIPs — "Nostr Knowledge Base Implementation Possibilities"
- Custom NIPs for 0xChat

Also see this discussion about third-party NIPs. If any sources of specs are missing from this list, please contact me so I can add them.

Protocol development

Many protocol features exist in the wild without documentation. Below are some resources that can help with designing or discussing protocol features.

- NostrBook summarizes protocol details, including NIPs and kinds
- NostrHub surfaces custom NIPs including comment sections and forking.
- Undocumented Nostr Kinds is a registry of nostr event kinds found in the wild without known documentation.
- NostrDesign outlines best practices for designing Nostr applications
- Nostr.how is an introductory guide with a number of related resources
- Nostr.com is a good place to start learning about the protocol

App lists

There are hundreds of nostr apps — so many that it would be a full time job to track and categorize them. Many are defunct or incomplete. See below for some resources for finding high-quality Nostr applications.

- nostrapps.com - a curated list of high-quality nostr apps.
- nostrapp.link - a decentralized directory of nostr applications, built on NIP 89.
- NostrHub lists nostr applications, as well repositories hosted on nostr and DVMs. Built on NIP 89, NIP 34, and NIP 90.
- Noogle allows for browsing and making requests against certain kinds of DVMs. Built on NIP 90.
- Coracle Stuff - a curated list of nostr apps created by Coracle.
- Fiatjaf's Stuff - a curated list of nostr apps created by fiatjaf.