NIP-01

Basic protocol flow description

`draft` `mandatory` `author:fiatjaf` `author:distbit` `author:scsibug` `author:kukks` `author:jb55` `author:semisol`

This NIP defines the basic protocol that should be implemented by everybody. New NIPs may add new optional (or mandatory) fields and messages and features to the structures and flows described here.

Events and signatures

Each user has a keypair. Signatures, public key, and encodings are done according to the Schnorr signatures standard for the curve `secp256k1`.

The only object type that exists is the `event`, which has the following format on the wire:

```
{
  "id": <32-bytes lowercase hex-encoded sha256 of the serialized event data>
  "pubkey": <32-bytes lowercase hex-encoded public key of the event creator>,
  "created_at": <unix timestamp in seconds>,
  "kind": <integer>,
  "tags": [
    ["e", <32-bytes hex of the id of another event>, <recommended relay URL>],
    ["p", <32-bytes hex of a pubkey>, <recommended relay URL>],
    ... // other kinds of tags may be included later
  ],
  "content": <arbitrary string>,
  "sig": <64-bytes hex of the signature of the sha256 hash of the serialized event data, which is the same as the "id" field>
}
```

To obtain the `event.id`, we `sha256` the serialized event. The serialization is done over the UTF-8 JSON-serialized string (with no white space or line breaks) of the following structure:

```
[
  0,
  <pubkey, as a (lowercase) hex string>,
  <created_at, as a number>,
  <kind, as a number>,
  <tags, as an array of arrays of non-null strings>,
  <content, as a string>
]
```

Communication between clients and relays

Relays expose a websocket endpoint to which clients can connect.

From client to relay: sending events and creating subscriptions

Clients can send 3 types of messages, which must be JSON arrays, according to the following patterns:

- `["EVENT", <event JSON as defined above>]`, used to publish events.
- `["REQ", <subscription_id>, <filters JSON>...]`, used to request events and subscribe to new updates.
- `["CLOSE", <subscription_id>]`, used to stop previous subscriptions.

`<subscription_id>` is an arbitrary, non-empty string of max length 64 chars, that should be used to represent a subscription.

`<filters>` is a JSON object that determines what events will be sent in that subscription, it can have the following attributes:

```
{
  "ids": <a list of event ids or prefixes>,
  "authors": <a list of pubkeys or prefixes, the pubkey of an event must be one of these>,
  "kinds": <a list of a kind numbers>,
  "#e": <a list of event ids that are referenced in an "e" tag>,
  "#p": <a list of pubkeys that are referenced in a "p" tag>,
  "since": <an integer unix timestamp, events must be newer than this to pass>,
  "until": <an integer unix timestamp, events must be older than this to pass>,
  "limit": <maximum number of events to be returned in the initial query>
}
```

Upon receiving a `REQ` message, the relay SHOULD query its internal database and return events that match the filter, then store that filter and send again all future events it receives to that same websocket until the websocket is closed. The `CLOSE` event is received with the same `<subscription_id>` or a new `REQ` is sent using the same `<subscription_id>`, in which case it should overwrite the previous subscription.

Filter attributes containing lists (such as `ids`, `kinds`, or `#e`) are JSON arrays with one or more values. At least one of the array's values must match the relevant field in an event for the condition itself to be considered a match. For scalar event attributes such as `kind`, the attribute from the event must be contained in the filter list. For tag attributes such as `#e`, where an event may have multiple values, the event and filter condition values must have at least one item in common.

The `ids` and `authors` lists contain lowercase hexadecimal strings, which may either be an exact 64-character match, or a prefix of the event value. A prefix match is when the filter string is an exact string prefix of the event value. The use of prefixes allows for more compact filters where a large number of values are queried, and can provide some privacy for clients that may not want to disclose the exact authors or events they are searching for.

All conditions of a filter that are specified must match for an event for it to pass the filter, i.e., multiple conditions are interpreted as `&&` conditions.

A `REQ` message may contain multiple filters. In this case, events that match any of the filters are to be returned, i.e., multiple filters are to be interpreted as `||` conditions.

The `limit` property of a filter is only valid for the initial query and can be ignored afterward. When `limit: n` is present it is assumed that the events returned in the initial query will be the latest `n` events. It is safe to return less events than `limit` specifies, but it is expected that relays do not return (much) more events than requested so clients don't get unnecessarily overwhelmed by data.

From relay to client: sending events and notices

Relays can send 3 types of messages, which must also be JSON arrays, according to the following patterns:

- `["EVENT", <subscription_id>, <event JSON as defined above>]`, used to send events requested by clients.
- `["EOSE", <subscription_id>]`, used to indicate the *end of stored events* and the beginning of events newly received in real-time.
- `["NOTICE", <message>]`, used to send human-readable error messages or other things to clients.

This NIP defines no rules for how `NOTICE` messages should be sent or treated.

`EVENT` messages MUST be sent only with a subscription ID related to a subscription previously initiated by the client (using the `REQ` message above).

Basic Event Kinds

- `0`: `set_metadata`: the `content` is set to a stringified JSON object `{name: <username>, about: <string>, picture: <url, string>}` describing the user who created the event. A relay may delete past `set_metadata` events once it gets a new one for the same pubkey.
- `1`: `text_note`: the `content` is set to the plaintext content of a note (anything the user wants to say). Do not use Markdown! Clients should not have to guess how to interpret content like `[]()`. Use different event kinds for parsable content.
- `2`: `recommend_server`: the `content` is set to the URL (e.g., `wss://somerelay.com`) of a relay the event creator wants to recommend to its followers.

A relay may choose to treat different message kinds differently, and it may or may not choose to have a default way to handle kinds it doesn't know about.

Other Notes:

- Clients should not open more than one websocket to each relay. One channel can support an unlimited number of subscriptions, so clients should do that.
- The `tags` array can store a tag identifier as the first element of each subarray, plus arbitrary information afterward (always as strings). This NIP defines `"p"` — meaning "pubkey", which points to a pubkey of someone that is referred to in the event —, and `"e"` — meaning "event", which points to the id of an event this event is quoting, replying to or referring to somehow. See NIP-10 for a detailed description of "e" and "p" tags.
- The `<recommended relay URL>` item present on the `"e"` and `"p"` tags is an optional (could be set to `""`) URL of a relay the client could attempt to connect to fetch the tagged event or other events from a tagged profile. It MAY be ignored, but it exists to increase censorship resistance and make the spread of relay addresses more seamless across clients.
- Clients should use the created_at field to judge the age of a metadata event and completely replace older metadata events with newer metadata events regardless of the order in which they arrive. Clients should not merge any filled fields within older metadata events into empty fields of newer metadata events.

NIP-02

Contact List and Petnames

`final` `optional` `author:fiatjaf` `author:arcbtc`

A special event with kind `3`, meaning "contact list" is defined as having a list of `p` tags, one for each of the followed/known profiles one is following.

Each tag entry should contain the key for the profile, a relay URL where events from that key can be found (can be set to an empty string if not needed), and a local name (or "petname") for that profile (can also be set to an empty string or not provided), i.e., `["p", <32-bytes hex key>, <main relay URL>, <petname>]`. The `content` can be anything and should be ignored.

For example:

```
{
  "kind": 3,
  "tags": [
    ["p", "91cf9..4e5ca", "wss://alicerelay.com/", "alice"],
    ["p", "14aeb..8dad4", "wss://bobrelay.com/nostr", "bob"],
    ["p", "612ae..e610f", "ws://carolrelay.com/ws", "carol"]
  ],
  "content": "",
  ...other fields
}
```

Every new contact list that gets published overwrites the past ones, so it should contain all entries. Relays and clients SHOULD delete past contact lists as soon as they receive a new one.

Uses

Contact list backup

If one believes a relay will store their events for sufficient time, they can use this kind-3 event to backup their following list and recover on a different device.

Profile discovery and context augmentation

A client may rely on the kind-3 event to display a list of followed people by profiles one is browsing; make lists of suggestions on who to follow based on the contact lists of other people one might be following or browsing; or show the data in other contexts.

Relay sharing

A client may publish a full list of contacts with good relays for each of their contacts so other clients may use these to update their internal relay lists if needed, increasing censorship-resistance.

Petname scheme

The data from these contact lists can be used by clients to construct local "petname" tables derived from other people's contact lists. This alleviates the need for global human-readable names. For example:

A user has an internal contact list that says

```
[
  ["p", "21df6d143fb96c2ec9d63726bf9edc71", "", "erin"]
]
```

And receives two contact lists, one from `21df6d143fb96c2ec9d63726bf9edc71` that says

```
[
  ["p", "a8bb3d884d5d90b413d9891fe4c4e46d", "", "david"]
]
```

and another from `a8bb3d884d5d90b413d9891fe4c4e46d` that says

```
[
  ["p", "f57f54057d2a7af0efecc8b0b66f5708", "", "frank"]
]
```

When the user sees `21df6d143fb96c2ec9d63726bf9edc71` the client can show *erin* instead; When the user sees `a8bb3d884d5d90b413d9891fe4c4e46d` the client can show *david.erin* instead; When the user sees `f57f54057d2a7af0efecc8b0b66f5708` the client can show *frank.david.erin* instead.

NIP-03

OpenTimestamps Attestations for Events

`draft` `optional` `author:fiatjaf`

When there is an OTS available it MAY be included in the existing event body under the `ots` key:

```
{
  "id": ...,
  "kind": ...,
  ...,
  ...,
  "ots": <base64-encoded OTS file data>
}
```

The *event id* MUST be used as the raw hash to be included in the OpenTimestamps merkle tree.

The attestation can be either provided by relays automatically (and the OTS binary contents just appended to the events it receives) or by clients themselves when they first upload the event to relays — and used by clients to show that an event is really "at least as old as [OTS date]".

NIP-04

Encrypted Direct Message

`final` `optional` `author:arcbtc`

A special event with kind `4`, meaning "encrypted direct message". It is supposed to have the following attributes:

`content` MUST be equal to the base64-encoded, aes-256-cbc encrypted string of anything a user wants to write, encrypted using a shared cipher generated by combining the recipient's public-key with the sender's private-key; this appended by the base64-encoded initialization vector as if it was a querystring parameter named "iv". The format is the following: `"content": "<encrypted_text>?iv=<initialization_vector>"`.

`tags` MUST contain an entry identifying the receiver of the message (such that relays may naturally forward this event to them), in the form `["p", "<pubkey, as a hex string>"]`.

`tags` MAY contain an entry identifying the previous message in a conversation or a message we are explicitly replying to (such that contextual, more organized conversations may happen), in the form `["e", "<event_id>"]`.

Note: By default in the libsecp256k1 ECDH implementation, the secret is the SHA256 hash of the shared point (both X and Y coordinates). In Nostr, only the X coordinate of the shared point is used as the secret and it is NOT hashed. If using libsecp256k1, a custom function that copies the X coordinate must be passed as the `hashfp` argument in `secp256k1_ecdh`. See here.

Code sample for generating such an event in JavaScript:

```
import crypto from 'crypto'
import * as secp from '@noble/secp256k1'

let sharedPoint = secp.getSharedSecret(ourPrivateKey, '02' + theirPublicKey)
let sharedX = sharedPoint.slice(1, 33)

let iv = crypto.randomFillSync(new Uint8Array(16))
var cipher = crypto.createCipheriv(
  'aes-256-cbc',
  Buffer.from(sharedX),
  iv
)
let encryptedMessage = cipher.update(text, 'utf8', 'base64')
encryptedMessage += cipher.final('base64')
let ivBase64 = Buffer.from(iv.buffer).toString('base64')

let event = {
  pubkey: ourPubKey,
  created_at: Math.floor(Date.now() / 1000),
  kind: 4,
  tags: [['p', theirPublicKey]],
  content: encryptedMessage + '?iv=' + ivBase64
}
```

Security Warning

This standard does not go anywhere near what is considered the state-of-the-art in encrypted communication between peers, and it leaks metadata in the events, therefore it must not be used for anything you really need to keep secret, and only with relays that use `AUTH` to restrict who can fetch your `kind:4` events.

Client Implementation Warning

Clients *should not* search and replace public key or note references from the `.content`. If processed like a regular text note (where `@npub...` is replaced with `#[0]` with a `["p", "..."]` tag) the tags are leaked and the mentioned user will receive the message in their inbox.

NIP-05

Mapping Nostr keys to DNS-based internet identifiers

`final` `optional` `author:fiatjaf` `author:mikedilger`

On events of kind `0` (`set_metadata`) one can specify the key `"nip05"` with an internet identifier (an email-like address) as the value. Although there is a link to a very liberal "internet identifier" specification above, NIP-05 assumes the `<local-part>` part will be restricted to the characters `a-z0-9-_.`, case-insensitive.

Upon seeing that, the client splits the identifier into `<local-part>` and `<domain>` and use these values to make a GET request to `https://<domain>/.well-known/nostr.json?name=<local-part>`.

The result should be a JSON document object with a key `"names"` that should then be a mapping of names to hex formatted public keys. If the public key for the given `<name>` matches the `pubkey` from the

`set_metadata` event, the client then concludes that the given pubkey can indeed be referenced by its identifier.

Example

If a client sees an event like this:

```
{
  "pubkey": "b0635d6a9851d3aed0cd6c495b282167acf761729078d975fc341b22650b07b9",
  "kind": 0,
  "content": "{\"name\": \"bob\", \"nip05\": \"bob@example.com\"}"
  ...
}
```

It will make a GET request to `https://example.com/.well-known/nostr.json?name=bob` and get back a response that will look like

```
{
  "names": {
    "bob": "b0635d6a9851d3aed0cd6c495b282167acf761729078d975fc341b22650b07b9"
  }
}
```

or with the **optional** `"relays"` attribute:

```
{
  "names": {
    "bob": "b0635d6a9851d3aed0cd6c495b282167acf761729078d975fc341b22650b07b9"
  },
  "relays": {
    "b0635d6a9851d3aed0cd6c495b282167acf761729078d975fc341b22650b07b9": [ "wss://relay.example.com", "wss://relay2.example.com" ]
  }
}
```

If the pubkey matches the one given in `"names"` (as in the example above) that means the association is right and the `"nip05"` identifier is valid and can be displayed.

The optional `"relays"` attribute may contain an object with public keys as properties and arrays of relay URLs as values. When present, that can be used to help clients learn in which relays the specific user may be found. Web servers which serve `/.well-known/nostr.json` files dynamically based on the query string SHOULD also serve the relays data for any name they serve in the same reply when that is available.

Finding users from their NIP-05 identifier

A client may implement support for finding users' public keys from *internet identifiers*, the flow is the same as above, but reversed: first the client fetches the *well-known* URL and from there it gets the public key of the user, then it tries to fetch the kind `0` event for that user and check if it has a matching `"nip05"`.

Notes

Clients must always follow public keys, not NIP-05 addresses

For example, if after finding that `bob@bob.com` has the public key `abc...def`, the user clicks a button to follow that profile, the client must keep a primary reference to `abc...def`, not `bob@bob.com`. If, for any reason, the address `https://bob.com/.well-known/nostr.json?name=bob` starts returning the public key `1d2...e3f` at any time in the future, the client must not replace `abc...def` in his list of followed profiles for the user (but it should stop displaying "bob@bob.com" for that user, as that will have become an invalid `"nip05"` property).

Public keys must be in hex format

Keys must be returned in hex format. Keys in NIP-19 `npub` format are only meant to be used for display in client UIs, not in this NIP.

User Discovery implementation suggestion

A client can also use this to allow users to search other profiles. If a client has a search box or something like that, a user may be able to type "bob@example.com" there and the client would recognize that and do the proper queries to obtain a pubkey and suggest that to the user.

Showing just the domain as an identifier

Clients may treat the identifier `_@domain` as the "root" identifier, and choose to display it as just the `<domain>`. For example, if Bob owns `bob.com`, he may not want an identifier like `bob@bob.com` as that is redundant. Instead, Bob can use the identifier `_@bob.com` and expect Nostr clients to show and treat that as just `bob.com` for all purposes.

Reasoning for the `/.well-known/nostr.json?name=<local-part>` format

By adding the `<local-part>` as a query string instead of as part of the path, the protocol can support both dynamic servers that can generate JSON on-demand and static servers with a JSON file in it that may contain multiple names.

Allowing access from JavaScript apps

JavaScript Nostr apps may be restricted by browser CORS policies that prevent them from accessing `/.well-known/nostr.json` on the user's domain. When CORS prevents JS from loading a resource, the JS program sees it as a network failure identical to the resource not existing, so it is not possible for a pure-JS app to tell the user for certain that the failure was caused by a CORS issue. JS Nostr apps that see network failures requesting `/.well-known/nostr.json` files may want to recommend to users that they check the CORS policy of their servers, e.g.:

```
$ curl -sI https://example.com/.well-known/nostr.json?name=bob | grep -i ^Access-Control
Access-Control-Allow-Origin: *
```

Users should ensure that their `/.well-known/nostr.json` is served with the HTTP header `Access-Control-Allow-Origin: *` to ensure it can be validated by pure JS apps running in modern browsers.

Security Constraints

The `/.well-known/nostr.json` endpoint MUST NOT return any HTTP redirects.

Fetchers MUST ignore any HTTP redirects given by the `/.well-known/nostr.json` endpoint.

NIP-06

Basic key derivation from mnemonic seed phrase

`draft` `optional` `author:fiatjaf`

BIP39 is used to generate mnemonic seed words and derive a binary seed from them.

BIP32 is used to derive the path `m/44'/1237'/<account>'/0/0` (according to the Nostr entry on SLIP44).

A basic client can simply use an `account` of `0` to derive a single key. For more advanced use-cases you can increment `account`, allowing generation of practically infinite keys from the 5-level path with hardened derivation.

Other types of clients can still get fancy and use other derivation paths for their own other purposes.

NIP-07

`window.nostr` capability for web browsers

`draft` `optional` `author:fiatjaf`

The `window.nostr` object may be made available by web browsers or extensions and websites or web-apps may make use of it after checking its availability.

That object must define the following methods:

```
async window.nostr.getPublicKey(): string // returns a public key as hex
async window.nostr.signEvent(event: Event): Event // takes an event object, adds `id`, `pubkey` and `sig` and returns it
```

Aside from these two basic above, the following functions can also be implemented optionally:

```
async window.nostr.getRelays(): { [url: string]: {read: boolean, write: boolean} } // returns a basic map of relay urls to relay policies
async window.nostr.nip04.encrypt(pubkey, plaintext): string // returns ciphertext and iv as specified in nip-04
async window.nostr.nip04.decrypt(pubkey, ciphertext): string // takes ciphertext and iv as specified in nip-04
```

Implementation

- horse (Chrome and derivatives)
- nos2x (Chrome and derivatives)
- Alby (Chrome and derivatives, Firefox, Safari)
- Blockcore (Chrome and derivatives)
- nos2x-fox (Firefox)
- Flamingo (Chrome and derivatives)
- AKA Profiles (Chrome, stores multiple keys)

> **Warning** `unrecommended`: deprecated in favor of NIP-27

NIP-08

Handling Mentions

`final` `unrecommended` `optional` `author:fiatjaf` `author:scsibug`

This document standardizes the treatment given by clients of inline mentions of other events and pubkeys inside the content of `text_note`s.

Clients that want to allow tagged mentions they MUST show an autocomplete component or something analogous to that whenever the user starts typing a special key (for example, "@") or presses some button to include a mention etc -- or these clients can come up with other ways to unambiguously differentiate between mentions and normal text.

Once a mention is identified, for example, the pubkey `27866e9d854c78ae625b867eefdfa9580434bc3e675be08d2acb526610d96fbe`, the client MUST add that pubkey to the `.tags` with the tag `p`, then replace its textual reference (inside `.content`) with the notation `#[index]` in which "index" is equal to the 0-based index of the related tag in the tags array.

The same process applies for mentioning event IDs.

A client that receives a `text_note` event with such `#[index]` mentions in its `.content` CAN do a search-and-replace using the actual contents from the `.tags` array with the actual pubkey or event ID that is mentioned, doing any desired context augmentation (for example, linking to the pubkey or showing a preview of the mentioned event contents) it wants in the process.

Where `#[index]` has an `index` that is outside the range of the tags array or points to a tag that is not an `e` or `p` tag or a tag otherwise declared to support this notation, the client MUST NOT perform such replacement or augmentation, but instead display it as normal text.

NIP-09

Event Deletion

`draft` `optional` `author:scsibug`

A special event with kind `5`, meaning "deletion" is defined as having a list of one or more `e` tags, each referencing an event the author is requesting to be deleted.

Each tag entry must contain an "e" event id intended for deletion.

The event's `content` field MAY contain a text note describing the reason for the deletion.

For example:

```
{
  "kind": 5,
  "pubkey": <32-bytes hex-encoded public key of the event creator>,
  "tags": [
    ["e", "dcd59..464a2"],
    ["e", "968c5..ad7a4"],
  ],
  "content": "these posts were published by accident",
  ...other fields
}
```

Relays SHOULD delete or stop publishing any referenced events that have an identical `pubkey` as the deletion request. Clients SHOULD hide or otherwise indicate a deletion status for referenced events.

Relays SHOULD continue to publish/share the deletion events indefinitely, as clients may already have the event that's intended to be deleted. Additionally, clients SHOULD broadcast deletion events to other relays which

don't have it.

Client Usage

Clients MAY choose to fully hide any events that are referenced by valid deletion events. This includes text notes, direct messages, or other yet-to-be defined event kinds. Alternatively, they MAY show the event along with an icon or other indication that the author has "disowned" the event. The `content` field MAY also be used to replace the deleted events' own content, although a user interface should clearly indicate that this is a deletion reason, not the original content.

A client MUST validate that each event `pubkey` referenced in the `e` tag of the deletion request is identical to the deletion request `pubkey`, before hiding or deleting any event. Relays can not, in general, perform this validation and should not be treated as authoritative.

Clients display the deletion event itself in any way they choose, e.g., not at all, or with a prominent notice.

Relay Usage

Relays MAY validate that a deletion event only references events that have the same `pubkey` as the deletion itself, however this is not required since relays may not have knowledge of all referenced events.

Deleting a Deletion

Publishing a deletion event against a deletion has no effect. Clients and relays are not obliged to support "undelete" functionality.

NIP-10

On "e" and "p" tags in Text Events (kind 1).

`draft` `optional` `author:unclebobmartin`

Abstract

This NIP describes how to use "e" and "p" tags in text events, especially those that are replies to other text events. It helps clients thread the replies into a tree rooted at the original event.

Positional "e" tags (DEPRECATED)

> This scheme is in common use; but should be considered deprecated.

`["e", <event-id>, <relay-url>]` as per NIP-01.

Where:

- `<event-id>` is the id of the event being referenced.
- `<relay-url>` is the URL of a recommended relay associated with the reference. Many clients treat this field as optional.

The positions of the "e" tags within the event denote specific meanings as follows:

- No "e" tag:
  This event is not a reply to, nor does it refer to, any other event.
- One "e" tag:
  `["e", <id>]` : The id of the event to which this event is a reply.
- Two "e" tags: `["e", <root-id>]`, `["e", <reply-id>]`
  `<root-id>` is the id of the event at the root of the reply chain. `<reply-id>` is the id of the article to which this event is a reply.
- Many "e" tags: `["e", <root-id>] ["e", <mention-id>]`, ..., `["e", <reply-id>]`
  There may be any number of `<mention-ids>`. These are the ids of events which may, or may not be in the reply chain.
  They are citings from this event. `root-id` and `reply-id` are as above.

> This scheme is deprecated because it creates ambiguities that are difficult, or impossible to resolve when an event references another but is not a reply.

Marked "e" tags (PREFERRED)

`["e", <event-id>, <relay-url>, <marker>]`

Where:

- `<event-id>` is the id of the event being referenced.
- `<relay-url>` is the URL of a recommended relay associated with the reference. Clients SHOULD add a valid `<relay-URL>` field, but may instead leave it as `""`.
- `<marker>` is optional and if present is one of `"reply"`, `"root"`, or `"mention"`.

The order of marked "e" tags is not relevant. Those marked with `"reply"` denote the id of the reply event being responded to. Those marked with `"root"` denote the root id of the reply thread being responded to. For top level replies (those replying directly to the root event), only the `"root"` marker should be used. Those marked with `"mention"` denote a quoted or reposted event id.

A direct reply to the root of a thread should have a single marked "e" tag of type "root".

> This scheme is preferred because it allows events to mention others without confusing them with `<reply-id>` or `<root-id>`.

The "p" tag

Used in a text event contains a list of pubkeys used to record who is involved in a reply thread.

When replying to a text event E the reply event's "p" tags should contain all of E's "p" tags as well as the `"pubkey"` of the event being replied to.

Example: Given a text event authored by `a1` with "p" tags [`p1`, `p2`, `p3`] then the "p" tags of the reply should be [`a1`, `p1`, `p2`, `p3`] in no particular order.

NIP-11

Relay Information Document

`draft` `optional` `author:scsibug` `author:doc-hex` `author:cameri`

Relays may provide server metadata to clients to inform them of capabilities, administrative contacts, and various server attributes. This is made available as a JSON document over HTTP, on the same URI as the relay's websocket.

When a relay receives an HTTP(s) request with an `Accept` header of `application/nostr+json` to a URI supporting WebSocket upgrades, they SHOULD return a document with the following structure.

```
{
  "name": <string identifying relay>,
  "description": <string with detailed information>,
  "pubkey": <administrative contact pubkey>,
  "contact": <administrative alternate contact>,
  "supported_nips": <a list of NIP numbers supported by the relay>,
  "software": <string identifying relay software URL>,
  "version": <string version identifier>
}
```

Any field may be omitted, and clients MUST ignore any additional fields they do not understand. Relays MUST accept CORS requests by sending `Access-Control-Allow-Origin`, `Access-Control-Allow-Headers`, and `Access-Control-Allow-Methods` headers.

Field Descriptions

Name

A relay may select a `name` for use in client software. This is a string, and SHOULD be less than 30 characters to avoid client truncation.

Description

Detailed plain-text information about the relay may be contained in the `description` string. It is recommended that this contain no markup, formatting or line breaks for word wrapping, and simply use double newline characters to separate paragraphs. There are no limitations on length.

Pubkey

An administrative contact may be listed with a `pubkey`, in the same format as Nostr events (32-byte hex for a `secp256k1` public key). If a contact is listed, this provides clients with a recommended address to send encrypted direct messages (See `NIP-04`) to a system administrator. Expected uses of this address are to report abuse or illegal content, file bug reports, or request other technical assistance.

Relay operators have no obligation to respond to direct messages.

Contact

An alternative contact may be listed under the `contact` field as well, with the same purpose as `pubkey`. Use of a Nostr public key and direct message SHOULD be preferred over this. Contents of this field SHOULD be a URI, using schemes such as `mailto` or `https` to provide users with a means of contact.

Supported NIPs

As the Nostr protocol evolves, some functionality may only be available by relays that implement a specific `NIP`. This field is an array of the integer identifiers of `NIP`s that are implemented in the relay. Examples would include `1`, for `"NIP-01"` and `9`, for `"NIP-09"`. Client-side `NIPs` SHOULD NOT be advertised, and can be ignored by clients.

Software

The relay server implementation MAY be provided in the `software` attribute. If present, this MUST be a URL to the project's homepage.

Version

The relay MAY choose to publish its software version as a string attribute. The string format is defined by the relay implementation. It is recommended this be a version number or commit identifier.

Extra Fields

Server Limitations

These are limitations imposed by the relay on clients. Your client should expect that requests which exceed these *practical* limitations are rejected or fail immediately.

```
{
...
  "limitation": {
        "max_message_length": 16384,
        "max_subscriptions": 20,
        "max_filters": 100,
        "max_limit": 5000,
        "max_subid_length": 100,
        "min_prefix": 4,
        "max_event_tags": 100,
        "max_content_length": 8196,
        "min_pow_difficulty": 30,
        "auth_required": true,
        "payment_required": true,
  }
...
}
```

- `max_message_length`: this is the maximum number of bytes for incoming JSON that the relay will attempt to decode and act upon. When you send large subscriptions, you will be limited by this value. It also effectively limits the maximum size of any event. Value is calculated from `[` to `]` and is after UTF-8 serialization (so some unicode characters will cost 2-3 bytes). It is equal to the maximum size of the WebSocket message frame.
- `max_subscriptions`: total number of subscriptions that may be active on a single websocket connection to this relay. It's possible that authenticated clients with a (paid) relationship to the relay may have higher limits.
- `max_filters`: maximum number of filter values in each subscription. Must be one or higher.
- `max_subid_length`: maximum length of subscription id as a string.
- `min_prefix`: for `authors` and `ids` filters which are to match against a hex prefix, you must provide at least this many hex digits in the prefix.
- `max_limit`: the relay server will clamp each filter's `limit` value to this number. This means the client won't be able to get more than this number of events from a single subscription filter. This clamping is typically done silently by the relay, but with this number, you can know that there are additional results if you narrowed your filter's time range or other parameters.
- `max_event_tags`: in any event, this is the maximum number of elements in the `tags` list.
- `max_content_length`: maximum number of characters in the `content` field of any event. This is a count of unicode characters. After serializing into JSON it may be larger (in bytes), and is still subject to the

`max_message_length`, if defined.

- `min_pow_difficulty`: new events will require at least this difficulty of PoW, based on NIP-13, or they will be rejected by this server.
- `auth_required`: this relay requires NIP-42 authentication to happen before a new connection may perform any other action. Even if set to False, authentication may be required for specific actions.
- `payment_required`: this relay requires payment before a new connection may perform any action.

Event Retention

There may be a cost associated with storing data forever, so relays may wish to state retention times. The values stated here are defaults for unauthenticated users and visitors. Paid users would likely have other policies.

Retention times are given in seconds, with `null` indicating infinity. If zero is provided, this means the event will not be stored at all, and preferably an error will be provided when those are received.

```
{
...
  "retention": [
    { "kinds": [0, 1, [5, 7], [40, 49]], "time": 3600 },
    { "kinds": [[40000, 49999]], "time": 100 },
    { "kinds": [[30000, 39999]], "count": 1000 },
    { "time": 3600, "count": 10000 }
  ]
...
}
```

`retention` is a list of specifications: each will apply to either all kinds, or a subset of kinds. Ranges may be specified for the kind field as a tuple of inclusive start and end values. Events of indicated kind (or all) are then limited to a `count` and/or time period.

It is possible to effectively blacklist Nostr-based protocols that rely on a specific `kind` number, by giving a retention time of zero for those `kind` values. While that is unfortunate, it does allow clients to discover servers that will support their protocol quickly via a single HTTP fetch.

There is no need to specify retention times for *ephemeral events* as defined in NIP-16 since they are not retained.

Content Limitations

Some relays may be governed by the arbitrary laws of a nation state. This may limit what content can be stored in cleartext on those relays. All clients are encouraged to use encryption to work around this limitation.

It is not possible to describe the limitations of each country's laws and policies which themselves are typically vague and constantly shifting.

Therefore, this field allows the relay operator to indicate which countries' laws might end up being enforced on them, and then indirectly on their users' content.

Users should be able to avoid relays in countries they don't like, and/or select relays in more favourable zones. Exposing this flexibility is up to the client software.

```
{
...
  "relay_countries": [ "CA", "US" ],
...
}
```

- `relay_countries`: a list of two-level ISO country codes (ISO 3166-1 alpha-2) whose laws and policies may affect this relay. `EU` may be used for European Union countries.

Remember that a relay may be hosted in a country which is not the country of the legal entities who own the relay, so it's very likely a number of countries are involved.

Community Preferences

For public text notes at least, a relay may try to foster a local community. This would encourage users to follow the global feed on that relay, in addition to their usual individual follows. To support this goal, relays MAY specify some of the following values.

```
{
...
  "language_tags": [ "en", "en-419" ],
  "tags": [ "sfw-only", "bitcoin-only", "anime" ],
  "posting_policy": "https://example.com/posting-policy.html",
...
}
```

- `language_tags` is an ordered list of IETF language tags indicating the major languages spoken on the relay.
- `tags` is a list of limitations on the topics to be discussed. For example `sfw-only` indicates that only "Safe For Work" content is encouraged on this relay. This relies on assumptions of what the "work" "community" feels "safe" talking about. In time, a common set of tags may emerge that allow users to find relays that suit their needs, and client software will be able to parse these tags easily. The `bitcoin-only` tag indicates that any *altcoin*, *"crypto"* or *blockchain* comments will be ridiculed without mercy.
- `posting_policy` is a link to a human-readable page which specifies the community policies for the relay. In cases where `sfw-only` is True, it's important to link to a page which gets into the specifics of your posting policy.

The `description` field should be used to describe your community goals and values, in brief. The `posting_policy` is for additional detail and legal terms. Use the `tags` field to signify limitations on content, or topics to be discussed, which could be machine processed by appropriate client software.

Pay-To-Relay

Relays that require payments may want to expose their fee schedules.

```
{
...
  "payments_url": "https://my-relay/payments",
  "fees": {
    "admission": [{ "amount": 1000000, "unit": "msats" }],
    "subscription": [{ "amount": 5000000, "unit": "msats", "period": 2592000 }],
    "publication": [{ "kinds": [4], "amount": 100, "unit": "msats" }],
  },
...
}
```

Examples

As of 2 May 2023 the following `curl` command provided these results.

```
 >curl -H "Accept: application/nostr+json" https://eden.nostr.land

{"name":"eden.nostr.land",
"description":"Eden Nostr Land - Toronto 1-01",
"pubkey":"00000000827ffaa94bfea288c3dfce4422c794fbb96625b6b31e9049f729d700",
"contact":"me@ricardocabral.io",
"supported_nips":[1,2,4,9,11,12,15,16,20,22,26,28,33,40],
"supported_nip_extensions":["11a"],
"software":"git+https://github.com/Cameri/nostream.git",
"version":"1.22.6",
"limitation":{"max_message_length":1048576,
              "max_subscriptions":10,
              "max_filters":2500,
              "max_limit":5000,
              "max_subid_length":256,
              "min_prefix":4,
              "max_event_tags":2500,
              "max_content_length":65536,
              "min_pow_difficulty":0,
              "auth_required":false,
              "payment_required":true},
"payments_url":"https://eden.nostr.land/invoices",
"fees":{"admission":[{"amount":5000000,"unit":"msats"}],
        "publication":[]}}
```

NIP-12

Generic Tag Queries

`draft` `optional` `author:scsibug` `author:fiatjaf`

Relays may support subscriptions over arbitrary tags. `NIP-01` requires relays to respond to queries for `e` and `p` tags. This NIP allows any single-letter tag present in an event to be queried.

The `<filters>` object described in `NIP-01` is expanded to contain arbitrary keys with a `#` prefix. Any single-letter key in a filter beginning with `#` is a tag query, and MUST have a value of an array of strings. The filter condition matches if the event has a tag with the same name, and there is at least one tag value in common with the filter and event. The tag name is the letter without the `#`, and the tag value is the second element. Subsequent elements are ignored for the purposes of tag queries.

Example Subscription Filter

The following provides an example of a filter that matches events of kind `1` with an `r` tag set to either `foo` or `bar`.

```
{
  "kinds": [1],
  "#r": ["foo", "bar"]
}
```

Client Behavior

Clients SHOULD use the `supported_nips` field to learn if a relay supports generic tag queries. Clients MAY send generic tag queries to any relay, if they are prepared to filter out extraneous responses from relays that do not support this NIP.

Rationale

The decision to reserve only single-letter tags to be usable in queries allow applications to make use of tags for all sorts of metadata, as it is their main purpose, without worrying that they might be bloating relay indexes. That also makes relays more lightweight, of course. And if some application or user is abusing single-letter tags with the intention of bloating relays that becomes easier to detect as single-letter tags will hardly be confused with some actually meaningful metadata some application really wanted to attach to the event with no spammy intentions.

Suggested Use Cases

Motivating examples for generic tag queries are provided below. This NIP does not promote or standardize the use of any specific tag for any purpose.

- Decentralized Commenting System: clients can comment on arbitrary web pages, and easily search for other comments, by using a `r` ("reference", in this case an URL) tag and value.
- Location-specific Posts: clients can use a `g` ("geohash") tag to associate a post with a physical location. Clients can search for a set of geohashes of varying precisions near them to find local content.
- Hashtags: clients can use simple `t` ("hashtag") tags to associate an event with an easily searchable topic name. Since Nostr events themselves are not searchable through the protocol, this provides a mechanism for user-driven search.

NIP-13

Proof of Work

`draft` `optional` `author:jb55` `author:cameri`

This NIP defines a way to generate and interpret Proof of Work for nostr notes. Proof of Work (PoW) is a way to add a proof of computational work to a note. This is a bearer proof that all relays and clients can universally validate with a small amount of code. This proof can be used as a means of spam deterrence.

`difficulty` is defined to be the number of leading zero bits in the `NIP-01` id. For example, an id of `000000000e9d97a1ab09fc381030b346cdd7a142ad57e6df0b46dc9bef6c7e2d` has a difficulty of `36` with `36` leading 0 bits.

`002f...` is `0000 0000 0010 1111...` in binary, which has 10 leading zeroes. Do not forget to count leading zeroes for hex digits `<= 7`.

Mining

To generate PoW for a `NIP-01` note, a `nonce` tag is used:

```
{"content": "It's just me mining my own business", "tags": [["nonce", "1", "21"]]}
```

When mining, the second entry to the nonce tag is updated, and then the id is recalculated (see NIP-01). If the id has the desired number of leading zero bits, the note has been mined. It is recommended to update the `created_at` as well during this process.

The third entry to the nonce tag `SHOULD` contain the target difficulty. This allows clients to protect against situations where bulk spammers targeting a lower difficulty get lucky and match a higher difficulty. For example, if you require 40 bits to reply to your thread and see a committed target of 30, you can safely reject it even if the note has 40 bits difficulty. Without a committed target difficulty you could not reject it. Committing to a target difficulty is something all honest miners should be ok with, and clients `MAY` reject a note matching a target difficulty if it is missing a difficulty commitment.

Example mined note

```
{
  "id": "000006d8c378af1779d2feebc7603a125d99eca0ccf1085959b307f64e5dd358",
  "pubkey": "a48380f4cfcc1ad5378294fcac36439770f9c878dd880ffa94bb74ea54a6f243",
  "created_at": 1651794653,
  "kind": 1,
  "tags": [
    [
      "nonce",
      "776797",
      "21"
    ]
  ],
  "content": "It's just me mining my own business",
  "sig": "284622fc0a3f4f1303455d5175f7ba962a3300d136085b9566801bc2e0699de0c7e31e44c81fb40ad9049173742e904713c3594a1da0fc5d2382a25c11aba977"
}
```

Validating

Here is some reference C code for calculating the difficulty (aka number of leading zero bits) in a nostr event id:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int countLeadingZeroes(const char *hex) {
    int count = 0;

    for (int i = 0; i < strlen(hex); i++) {
        int nibble = (int)strtol((char[]){hex[i], '\0'}, NULL, 16);
        if (nibble == 0) {
            count += 4;
        } else {
            count += __builtin_clz(nibble) - 28;
            break;
        }
    }

    return count;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <hex_string>\n", argv[0]);
        return 1;
    }

    const char *hex_string = argv[1];
    int result = countLeadingZeroes(hex_string);
    printf("Leading zeroes in hex string %s: %d\n", hex_string, result);

    return 0;
}
```

Here is some JavaScript code for doing the same thing:

```
// hex should be a hexadecimal string (with no 0x prefix)
function countLeadingZeroes(hex) {
  let count = 0;

  for (let i = 0; i < hex.length; i++) {
    const nibble = parseInt(hex[i], 16);
    if (nibble === 0) {
      count += 4;
    } else {
      count += Math.clz32(nibble) - 28;
      break;
    }
  }

  return count;
}
```

Querying relays for PoW notes

Since relays allow searching on prefixes, you can use this as a way to filter notes of a certain difficulty:

```
$ echo '["REQ", "subid", {"ids": ["000000000"]}]'  | websocat wss://some-relay.com | jq -c '.[2]'
{"id":"000000000121637feeb68a06c8fa7abd25774bdedfa9b6ef648386fb3b70c387", ...}
```

Delegated Proof of Work

Since the `NIP-01` note id does not commit to any signature, PoW can be outsourced to PoW providers, perhaps for a fee. This provides a way for clients to get their messages out to PoW-restricted relays without having to do any work themselves, which is useful for energy-constrained devices like mobile phones.

NIP-14

Subject tag in Text events.

`draft` `optional` `author:unclebobmartin`

This NIP defines the use of the "subject" tag in text (kind: 1) events.
(implemented in more-speech)

`["subject": <string>]`

Browsers often display threaded lists of messages. The contents of the subject tag can be used in such lists, instead of the more ad hoc approach of using the first few words of the message. This is very similar to the way email browsers display lists of incoming emails by subject rather than by contents.

When replying to a message with a subject, clients SHOULD replicate the subject tag. Clients MAY adorn the subject to denote that it is a reply. e.g. by prepending "Re:".

Subjects should generally be shorter than 80 chars. Long subjects will likely be trimmed by clients.

NIP-15

Nostr Marketplace (for resilient marketplaces)

`draft` `optional` `author:fiatjaf` `author:benarc` `author:motorina0` `author:talvasconcelos`

> Based on https://github.com/lnbits/Diagon-Alley

> Implemented here https://github.com/lnbits/nostrmarket

Terms

- `merchant` - seller of products with NOSTR key-pair
- `customer` - buyer of products with NOSTR key-pair
- `product` - item for sale by the `merchant`
- `stall` - list of products controlled by `merchant` (a `merchant` can have multiple stalls)
- `marketplace` - clientside software for searching `stalls` and purchasing `products`

Nostr Marketplace Clients

Merchant admin

Where the `merchant` creates, updates and deletes `stalls` and `products`, as well as where they manage sales, payments and communication with `customers`.

The `merchant` admin software can be purely clientside, but for `convenience` and uptime, implementations will likely have a server client listening for NOSTR events.

Marketplace

`Marketplace` software should be entirely clientside, either as a stand-alone app, or as a purely frontend webpage. A `customer` subscribes to different merchant NOSTR public keys, and those `merchants` `stalls` and `products` become listed and searchable. The marketplace client is like any other ecommerce site, with basket and checkout. `Marketplaces` may also wish to include a `customer` support area for direct message communication with `merchants`.

`Merchant` publishing/updating products (event)

A merchant can publish these events:

Kind		Description	NIP
`0`	`set_meta`	The merchant description (similar with any `nostr` public key).	NIP01
`30017`	`set_stall`	Create or update a stall.	NIP33 (Parameterized Replaceable Event)
`30018`	`set_product`	Create or update a product.	NIP33 (Parameterized Replaceable Event)
`4`	`direct_message`	Communicate with the customer. The messages can be plain-text or JSON.	NIP04
`5`	`delete`	Delete a product or a stall.	NIP09

Event `30017`: Create or update a stall.

Event Content:

```
{
   "id": <String, UUID generated by the merchant. Sequential IDs (`0`, `1`, `2`...) are discouraged>,
   "name": <String, stall name>,
   "description": <String (optional), stall description>,
   "currency": <String, currency used>,
   "shipping": [
       {
           "id": <String, UUID of the shipping zone, generated by the merchant>,
           "name": <String (optional), zone name>,
           "cost": <float, cost for shipping. The currency is defined at the stall level>,
           "countries": [<String, countries included in this zone>],
       }
   ]
}
```

Fields that are not self-explanatory:

- `shipping`:
  - an array with possible shipping zones for this stall. The customer MUST choose exactly one shipping zone.
  - shipping to different zones can have different costs. For some goods (digital for example) the cost can be zero.
  - the `id` is an internal value used by the merchant. This value must be sent back as the customer selection.

Event Tags:

```
  "tags": [["d", <String, id of stall]]
```

- the `d` tag is required by NIP33. Its value MUST be the same as the stall `id`.

Event `30018`: Create or update a product

Event Content:

```
{
   "id": <String, UUID generated by the merchant.Sequential IDs (`0`, `1`, `2`...) are discouraged>,
   "stall_id": <String, UUID of the stall to which this product belong to>,
   "name": <String, product name>,
   "description": <String (optional), product description>,
   "images": <[String], array of image URLs, optional>,
   "currency": <String, currency used>,
   "price": <float, cost of product>,
   "quantity": <int, available items>,
   "specs": [
     [ <String, spec key>, <String, spec value>]
    ]
}
```

Fields that are not self-explanatory:

- `specs`:
  - an array of key pair values. It allows for the Customer UI to present present product specifications in a structure mode. It also allows comparison between products
  - eg: `[["operating_system", "Android 12.0"], ["screen_size", "6.4 inches"], ["connector_type", "USB Type C"]]`

Open: better to move `spec` in the `tags` section of the event?

Event Tags:

```
  "tags": [
      ["d", <String, id of product],
      ["t", <String (optional), product category],
      ["t", <String (optional), product category],
      ...
   ]
```

- the `d` tag is required by NIP33. Its value MUST be the same as the product `id`.
- the `t` tag is as searchable tag (NIP12). It represents different categories that the product can be part of (`food`, `fruits`). Multiple `t` tags can be present.

Checkout events

All checkout events are sent as JSON strings using (NIP04).

The `merchant` and the `customer` can exchange JSON messages that represent different actions. Each `JSON` message `MUST` have a `type` field indicating the what the JSON represents. Possible types:

Message Type	Sent By	Description
0	Customer	New Order
1	Merchant	Payment Request
2	Merchant	Order Status Update

Step 1: `customer` order (event)

The below json goes in content of NIP04.

```
{
    "id": <String, UUID generated by the customer>,
    "type": 0,
    "name": <String (optional), ???>,
    "address": <String (optional), for physical goods an address should be provided>
    "message": "<String (optional), message for merchant>,
    "contact": {
        "nostr": <32-bytes hex of a pubkey>,
        "phone": <String (optional), if the customer wants to be contacted by phone>,
        "email": <String (optional), if the customer wants to be contacted by email>,
    },
    "items": [
        {
            "product_id": <String, UUID of the product>,
            "quantity": <int, how many products the customer is ordering>
        }
    ],
    "shipping_id": <String, UUID of the shipping zone>
}
```

Open: is `contact.nostr` required?

Step 2: `merchant` request payment (event)

Sent back from the merchant for payment. Any payment option is valid that the merchant can check.

The below json goes in `content` of NIP04.

`payment_options` / `type` include:

- `url` URL to a payment page, stripe, paypal, btcpayserver, etc
- `btc` onchain bitcoin address
- `ln` bitcoin lightning invoice
- `lnurl` bitcoin lnurl-pay

```
{
    "id": <String, UUID of the order>,
    "type": 1,
    "message": <String, message to customer, optional>,
    "payment_options": [
        {
            "type": <String, option type>,
            "link": <String, url, btc address, ln invoice, etc>
        },
        {
            "type": <String, option type>,
            "link": <String, url, btc address, ln invoice, etc>
        },
                {
            "type": <String, option type>,
            "link": <String, url, btc address, ln invoice, etc>
        }
    ]
}
```

Step 3: `merchant` verify payment/shipped (event)

Once payment has been received and processed.

The below json goes in `content` of NIP04.

```
{
    "id": <String, UUID of the order>,
    "type": 2,
    "message": <String, message to customer>,
    "paid": <Bool, true/false has received payment>,
    "shipped": <Bool, true/false has been shipped>,
}
```

Customer support events

Customer support is handled over whatever communication method was specified. If communicating via nostr, NIP-04 is used https://github.com/nostr-protocol/nips/blob/master/04.md.

Additional

Standard data models can be found here here

NIP-16

Event Treatment

`draft` `optional` `author:Semisol`

Relays may decide to allow replaceable and/or ephemeral events.

Regular Events

A *regular event* is defined as an event with a kind `1000 <= n < 10000`. Upon a regular event being received, the relay SHOULD send it to all clients with a matching filter, and SHOULD store it. New events of the same kind do not affect previous events in any way.

Replaceable Events

A *replaceable event* is defined as an event with a kind `10000 <= n < 20000`. Upon a replaceable event with a newer timestamp than the currently known latest replaceable event with the same kind and author being received, the old event SHOULD be discarded, effectively replacing what gets returned when querying for `author:kind` tuples.

Ephemeral Events

An *ephemeral event* is defined as an event with a kind `20000 <= n < 30000`. Upon an ephemeral event being received, the relay SHOULD send it to all clients with a matching filter, and MUST NOT store it.

Client Behavior

Clients SHOULD use the `supported_nips` field to learn if a relay supports this NIP. Clients SHOULD NOT send ephemeral events to relays that do not support this NIP; they will most likely be persisted. Clients MAY send replaceable events to relays that may not support this NIP, and clients querying SHOULD be prepared for the relay to send multiple events and should use the latest one.

Suggested Use Cases

- States: An application may create a state event that is replaced every time a new state is set (such as statuses)
- Typing indicators: A chat application may use ephemeral events as a typing indicator.
- Messaging: Two pubkeys can message over nostr using ephemeral events.

NIP-18

Reposts

`draft` `optional` `author:jb55` `author:fiatjaf` `author:arthurfranca`

A repost is a `kind 6` note that is used to signal to followers that another event is worth reading.

The `content` of a repost event is empty. Optionally, it MAY contain the stringified JSON of the reposted note event for quick look up.

The repost event MUST include an `e` tag with the `id` of the note that is being reposted. That tag MUST include a relay URL as its third entry to indicate where it can be fetched.

The repost SHOULD include a `p` tag with the `pubkey` of the event being reposted.

Quote Reposts

Quote reposts are `kind 1` events with an embedded `e` tag (see NIP-08 and NIP-27). Because a quote repost includes an `e` tag, it may show up along replies to the reposted note.

NIP-19

bech32-encoded entities

`draft` `optional` `author:jb55` `author:fiatjaf` `author:Semisol`

This NIP standardizes bech32-formatted strings that can be used to display keys, ids and other information in clients. These formats are not meant to be used anywhere in the core protocol, they are only meant for displaying to users, copy-pasting, sharing, rendering QR codes and inputting data.

It is recommended that ids and keys are stored in either hex or binary format, since these formats are closer to what must actually be used the core protocol.

Bare keys and ids

To prevent confusion and mixing between private keys, public keys and event ids, which are all 32 byte strings. bech32-(not-m) encoding with different prefixes can be used for each of these entities.

These are the possible bech32 prefixes:

- `npub`: public keys
- `nsec`: private keys
- `note`: note ids

Example: the hex public key `3bf0c63fcb93463407af97a5e5ee64fa883d107ef9e558472c4eb9aaaefa459d` translates to `npub180cvv07tjdrrgpa0j7j7tmnyl2yr6yr7l8j4s3evf6u64th6gkwsyjh6w6`.

The bech32 encodings of keys and ids are not meant to be used inside the standard NIP-01 event formats or inside the filters, they're meant for human-friendlier display and input only. Clients should still accept keys in both hex and npub format for now, and convert internally.

Shareable identifiers with extra metadata

When sharing a profile or an event, an app may decide to include relay information and other metadata such that other apps can locate and display these entities more easily.

For these events, the contents are a binary-encoded list of `TLV` (type-length-value), with `T` and `L` being 1 byte each (`uint8`, i.e. a number in the range of 0-255), and `V` being a sequence of bytes of the size indicated by `L`.

These are the possible bech32 prefixes with `TLV`:

- `nprofile`: a nostr profile
- `nevent`: a nostr event
- `nrelay`: a nostr relay
- `naddr`: a nostr parameterized replaceable event coordinate (NIP-33)

These possible standardized `TLV` types are indicated here:

- `0`: `special`
  - depends on the bech32 prefix:
    - for `nprofile` it will be the 32 bytes of the profile public key
    - for `nevent` it will be the 32 bytes of the event id
    - for `nrelay`, this is the relay URL
    - for `naddr`, it is the identifier (the `"d"` tag) of the event being referenced
- `1`: `relay`
  - for `nprofile`, `nevent` and `naddr`, *optionally*, a relay in which the entity (profile or event) is more likely to be found, encoded as ascii
  - this may be included multiple times
- `2`: `author`
  - for `naddr`, the 32 bytes of the pubkey of the event
  - for `nevent`, *optionally*, the 32 bytes of the pubkey of the event

- `3`: `kind`
  - for `naddr`, the 32-bit unsigned integer of the kind, big-endian
  - for `nevent`, *optionally*, the 32-bit unsigned integer of the kind, big-endian

Examples

- `npub10elfcs4fr0l0r8af98jlmgdh9c8tcxjvz9qkw038js35mp4dma8qzvjptg` should decode into the public key hex `7e7e9c42a91bfef19fa929e5fda1b72e0ebc1a4c1141673e2794234d86addf4e` and vice-versa
- `nsec1vl029mgpspedva04g90vltkh6fvh240zqtv9k0t9af8935ke9laqsnlfe5` should decode into the private key hex `67dea2ed018072d675f5415ecfaed7d2597555e202d85b3d65ea4e58d2d92ffa` and vice-versa
- `nprofile1qqsrhuxx8l9ex335q7he0f09aej04zpazpl0ne2cgukyawd24mayt8gpp4mhxue69uhhytnc9e3k7mgpz4mhxue69uhkg6nzv9ejuumpv34kytnrdaksjlyr9p` should decode into a profile with the following TLV items:
  - pubkey: `3bf0c63fcb93463407af97a5e5ee64fa883d107ef9e558472c4eb9aaaefa459d`
  - relay: `wss://r.x.com`
  - relay: `wss://djbas.sadkb.com`

Notes

- `npub` keys MUST NOT be used in NIP-01 events or in NIP-05 JSON responses, only the hex format is supported there.
- When decoding a bech32-formatted string, TLVs that are not recognized or supported should be ignored, rather than causing an error.

NIP-20

Command Results

`draft` `optional` `author:jb55`

When submitting events to relays, clients currently have no way to know if an event was successfully committed to the database. This NIP introduces the concept of command results which are like NOTICE's except provide more information about if an event was accepted or rejected.

A command result is a JSON object with the following structure that is returned when an event is successfully saved to the database or rejected:

```
["OK", <event_id>, <true|false>, <message>]
```

Relays MUST return `true` when the event is a duplicate and has already been saved. The `message` SHOULD start with `duplicate:` in this case.

Relays MUST return `false` when the event was rejected and not saved.

The `message` SHOULD provide additional information as to why the command succeeded or failed.

The `message` SHOULD start with `blocked:` if the pubkey or network address has been blocked, banned, or is not on a whitelist.

The `message` SHOULD start with `invalid:` if the event is invalid or doesn't meet some specific criteria (created_at is too far off, id is wrong, signature is wrong, etc)

The `message` SHOULD start with `pow:` if the event doesn't meet some proof-of-work difficulty. The client MAY consult the relay metadata at this point to retrieve the required posting difficulty.

The `message` SHOULD start with `rate-limited:` if the event was rejected due to rate limiting techniques.

The `message` SHOULD start with `error:` if the event failed to save due to a server issue.

Ephemeral events are not acknowledged with OK responses, unless there is a failure.

If the event or `EVENT` command is malformed and could not be parsed, a NOTICE message SHOULD be used instead of a command result. This NIP only applies to non-malformed EVENT commands.

Examples

Event successfully written to the database:

```
["OK", "b1a649ebe8b435ec71d3784793f3bbf4b93e64e17568a741aecd4c7ddeafce30", true, ""]
```

Event successfully written to the database because of a reason:

```
["OK", "b1a649ebe8b435ec71d3784793f3bbf4b93e64e17568a741aecd4c7ddeafce30", true, "pow: difficulty 25>=24"]
```

Event blocked due to ip filter

```
["OK", "b1a649ebe8...", false, "blocked: tor exit nodes not allowed"]
```

Event blocked due to pubkey ban

```
["OK", "b1a649ebe8...", false, "blocked: you are banned from posting here"]
```

Event blocked, pubkey not registered

```
["OK", "b1a649ebe8...", false, "blocked: please register your pubkey at https://my-expensive-relay.example.com"]
```

Event rejected, rate limited

```
["OK", "b1a649ebe8...", false, "rate-limited: slow down there chief"]
```

Event rejected, `created_at` too far off

```
["OK", "b1a649ebe8...", false, "invalid: event creation date is too far off from the current time. Is your system clock in sync?"]
```

Event rejected, insufficient proof-of-work difficulty

```
["OK", "b1a649ebe8...", false, "pow: difficulty 26 is less than 30"]
```

Event failed to save,

```
["OK", "b1a649ebe8...", false, "error: could not connect to the database"]
```

Client Handling

`messages` are meant for humans, with `reason:` prefixes so that clients can be slightly more intelligent with what to do with them. For example, with a `rate-limited:` reason the client may not show anything and simply try again with a longer timeout.

For the `pow:` prefix it may query relay metadata to get the updated difficulty requirement and try again in the background.

For the `invalid:` and `blocked:` prefix the client may wish to show these as styled error popups.

The prefixes include a colon so that the message can be cleanly separated from the prefix by taking everything after `:` and trimming it.

Future Extensions

This proposal SHOULD be extended to support further commands in the future, such as REQ and AUTH. They are left out of this initial version to keep things simpler.

NIP-21

`nostr:` URI scheme

`draft` `optional` `author:fiatjaf`

This NIP standardizes the usage of a common URI scheme for maximum interoperability and openness in the network.

The scheme is `nostr:`.

The identifiers that come after are expected to be the same as those defined in NIP-19 (except `nsec`).

Examples

- `nostr:npub1sn0wdenkukak0d9dfczzeacvhkrgz92ak56egt7vdgzn8pv2wfqqhrjdv9`
- `nostr:nprofile1qqsrhuxx8l9ex335q7he0f09aej04zpazpl0ne2cgukyawd24mayt8gpp4mhxue69uhhytnc9e3k7mgpz4mhxue69uhkg6nzv9ejuumpv34kytnrdaksjlyr9p`
- `nostr:note1fntxtkcy9pjwucqwa9mddn7v03wwwsu9j330jj350nvhpky2tuaspk6nqc`
- `nostr:nevent1qqstna2yrezu5wghjvswqqculvvwxsrcvu7uc0f78gan4xqhvz49d9spr3mhxue69uhkummnw3ez6un9d3shjtn4de6x2argwghx6egpr4mhxue69uhkummnw3ez6ur4vgh8wetvd3hhyer9wghxuet5nxnepm`

NIP-22

Event `created_at` Limits

`draft` `optional` `author:jeffthibault` `author:Giszmo`

Relays may define both upper and lower limits within which they will consider an event's `created_at` to be acceptable. Both the upper and lower limits MUST be unix timestamps in seconds as defined in NIP-01.

If a relay supports this NIP, the relay SHOULD send the client a NIP-20 command result saying the event was not stored for the `created_at` timestamp not being within the permitted limits.

Client Behavior

Clients SHOULD use the NIP-11 `supported_nips` field to learn if a relay uses event `created_at` time limits as defined by this NIP.

Motivation

This NIP formalizes restrictions on event timestamps as accepted by a relay and allows clients to be aware of relays that have these restrictions.

The event `created_at` field is just a unix timestamp and can be set to a time in the past or future. Relays accept and share events dated to 20 years ago or 50,000 years in the future. This NIP aims to define a way for relays that do not want to store events with *any* timestamp to set their own restrictions.

Replaceable events can behave rather unexpectedly if the user wrote them - or tried to write them - with a wrong system clock. Persisting an update with a backdated system now would result in the update not getting persisted without a notification and if they did the last update with a forward dated system, they will again fail to do another update with the now correct time.

A wide adoption of this NIP could create a better user experience as it would decrease the amount of events that appear wildly out of order or even from impossible dates in the distant past or future.

Keep in mind that there is a use case where a user migrates their old posts onto a new relay. If a relay rejects events that were not recently created, it cannot serve this use case.

Python (pseudocode) Example

```
import time

TIME = int(time.time())
LOWER_LIMIT = TIME - (60 * 60 * 24) # Define lower limit as 1 day into the past
UPPER_LIMIT = TIME + (60 * 15)      # Define upper limit as 15 minutes into the future

if event.created_at not in range(LOWER_LIMIT, UPPER_LIMIT):
  ws.send('["OK", event.id, False, "invalid: the event created_at field is out of the acceptable range (-24h, +15min) for this relay"]')
```

Note: These are just example limits, the relay operator can choose whatever limits they want.

NIP-23

Long-form Content

`draft` `optional` `author:fiatjaf`

This NIP defines `kind:30023` (a parameterized replaceable event according to NIP-33) for long-form text content, generally referred to as "articles" or "blog posts".

"Social" clients that deal primarily with `kind:1` notes should not be expected to implement this NIP.

Format

The `.content` of these events should be a string text in Markdown syntax.

Metadata

For the date of the last update the `.created_at` field should be used, for "tags"/"hashtags" (i.e. topics about which the event might be of relevance) the `"t"` event tag should be used, as per NIP-12.

Other metadata fields can be added as tags to the event as necessary. Here we standardize 4 that may be useful, although they remain strictly optional:

- `"title"`, for the article title
- `"image"`, for a URL pointing to an image to be shown along with the title
- `"summary"`, for the article summary
- `"published_at"`, for the timestamp in unix seconds (stringified) of the first time the article was published

Editability

These articles are meant to be editable, so they should make use of the replaceability feature of NIP-33 and include a `"d"` tag with an identifier for the article. Clients should take care to only publish and read these events from relays that implement that. If they don't do that they should also take care to hide old versions of the same article they may receive.

Linking

The article may be linked to using the NIP-19 `naddr` code along with the `"a"` tag (see NIP-33 and NIP-19).

References

References to other Nostr notes, articles or profiles must be made according to NIP-27, i.e. by using NIP-21 `nostr:...` links and optionally adding tags for these (see example below).

Example Event

```
{
  "kind": 30023,
  "created_at": 1675642635,
  "content": "Lorem [ipsum][nostr:nevent1qqst8cujky046negxgwwm5ynqwn53t8aqjr6afd8g59nfqwxpdhylpcpzamhxue69uhhyetvv9ujuetcv9khqmr99e3k7mg8arnc9] dolor sit amet, consectetur adi
  "tags": [
    ["d", "lorem-ipsum"],
    ["title", "Lorem Ipsum"],
    ["published_at", "1296962229"],
    ["t", "placeholder"],
    ["e", "b3e392b11f5d4f28321cedd09303a748acfd0487aea5a7450b3481c60b6e4f87", "wss://relay.example.com"],
    ["a", "30023:a695f6b60119d9521934a691347d9f78e8770b56da16bb255ee286ddf9fda919:ipsum", "wss://relay.nostr.org"]
  ],
  "pubkey": "...",
  "id": "..."
}
```

NIP-25

Reactions

`draft` `optional` `author:jb55`

A reaction is a `kind 7` note that is used to react to other notes.

The generic reaction, represented by the `content` set to a `+` string, SHOULD be interpreted as a "like" or "upvote".

A reaction with `content` set to `-` SHOULD be interpreted as a "dislike" or "downvote". It SHOULD NOT be counted as a "like", and MAY be displayed as a downvote or dislike on a post. A client MAY also choose to tally likes against dislikes in a reddit-like system of upvotes and downvotes, or display them as separate tallies.

The `content` MAY be an emoji, in this case it MAY be interpreted as a "like" or "dislike", or the client MAY display this emoji reaction on the post.

Tags

The reaction event SHOULD include `e` and `p` tags from the note the user is reacting to. This allows users to be notified of reactions to posts they were mentioned in. Including the `e` tags enables clients to pull all the reactions associated with individual posts or all the posts in a thread.

The last `e` tag MUST be the `id` of the note that is being reacted to.

The last `p` tag MUST be the `pubkey` of the event being reacted to.

Example code

```
func make_like_event(pubkey: String, privkey: String, liked: NostrEvent) -> NostrEvent {
    var tags: [[String]] = liked.tags.filter {
        tag in tag.count >= 2 && (tag[0] == "e" || tag[0] == "p")
    }
    tags.append(["e", liked.id])
    tags.append(["p", liked.pubkey])
    let ev = NostrEvent(content: "+", pubkey: pubkey, kind: 7, tags: tags)
    ev.calculate_id()
    ev.sign(privkey: privkey)
    return ev
}

NIP: 26
=======

Delegated Event Signing
-----

`draft` `optional` `author:markharding` `author:minds`

This NIP defines how events can be delegated so that they can be signed by other keypairs.

Another application of this proposal is to abstract away the use of the 'root' keypairs when interacting with clients. For example, a user could generate new keypairs for each

#### Introducing the 'delegation' tag

This NIP introduces a new tag: `delegation` which is formatted as follows:

```json
[
  "delegation",
  <pubkey of the delegator>,
  <conditions query string>,
  <delegation token: 64-byte Schnorr signature of the sha256 hash of the delegation string>
]
```

Delegation Token

The **delegation token** should be a 64-byte Schnorr signature of the sha256 hash of the following string:

```
nostr:delegation:<pubkey of publisher (delegatee)>:<conditions query string>
```

Conditions Query String

The following fields and operators are supported in the above query string:

Fields:

1. `kind`
    - *Operators*:
        - `=${KIND_NUMBER}` - delegatee may only sign events of this kind
2. `created_at`
    - *Operators*:
        - `<${TIMESTAMP}` - delegatee may only sign events created ***before*** the specified timestamp
        - `>${TIMESTAMP}` - delegatee may only sign events created ***after*** the specified timestamp

In order to create a single condition, you must use a supported field and operator. Multiple conditions can be used in a single query string, including on the same field. Conditions must be combined with `&`.

For example, the following condition strings are valid:

- `kind=1&created_at<1675721813`
- `kind=0&kind=1&created_at>1675721813`
- `kind=1&created_at>1674777689&created_at<1675721813`

For the vast majority of use-cases, it is advisable that query strings should include a `created_at` ***after*** condition reflecting the current time, to prevent the delegatee from publishing historic notes on the delegator's behalf.

Example

```
 # Delegator:
privkey: ee35e8bb71131c02c1d7e73231daa48e9953d329a4b701f7133c8f46dd21139c
pubkey:  8e0d3d3eb2881ec137a11debe736a9086715a8c8beeeda615780064d68bc25dd

# Delegatee:
privkey: 777e4f60b4aa87937e13acc84f7abcc3c93cc035cb4c1e9f7a9086dd78fffce1
pubkey:  477318cfb5427b9cfc66a9fa376150c1ddbc62115ae27cef72417eb959691396
```

Delegation string to grant note publishing authorization to the delegatee (477318cf) from now, for the next 30 days, given the current timestamp is `1674834236`.

```
nostr:delegation:477318cfb5427b9cfc66a9fa376150c1ddbc62115ae27cef72417eb959691396:kind=1&created_at>1674834236&created_at<1677426236
```

The delegator (8e0d3d3e) then signs a SHA256 hash of the above delegation string, the result of which is the delegation token:

```
6f44d7fe4f1c09f3954640fb58bd12bae8bb8ff4120853c4693106c82e920e2b898f1f9ba9bd65449a987c39c0423426ab7b53910c0c6abfb41b30bc16e5f524
```

The delegatee (477318cf) can now construct an event on behalf of the delegator (8e0d3d3e). The delegatee then signs the event with its own private key and publishes.
```

```
{
  "id": "e93c6095c3db1c31d15ac771f8fc5fb672f6e52cd25505099f62cd055523224f",
  "pubkey": "477318cfb5427b9cfc66a9fa376150c1ddbc62115ae27cef72417eb959691396",
  "created_at": 1677426298,
  "kind": 1,
  "tags": [
    [
      "delegation",
      "8e0d3d3eb2881ec137a11debe736a9086715a8c8beeeda615780064d68bc25dd",
      "kind=1&created_at>1674834236&created_at<1677426236",
      "6f44d7fe4f1c09f3954640fb58bd12bae8bb8ff4120853c4693106c82e920e2b898f1f9ba9bd65449a987c39c0423426ab7b53910c0c6abfb41b30bc16e5f524"
    ]
  ],
  "content": "Hello, world!",
  "sig": "633db60e2e7082c13a47a6b19d663d45b2a2ebdeaf0b4c35ef83be2738030c54fc7fd56d139652937cdca875ee61b51904a1d0d0588a6acd6168d7be2909d693"
}
```


The event should be considered a valid delegation if the conditions are satisfied (`kind=1` , `created_at>1674834236` and `created_at<1677426236` in this example) and, upon validation of the delegation token, are found to be unchanged from the conditions in the original delegation string.

Clients should display the delegated note as if it was published directly by the delegator (8e0d3d3e).

Relay & Client Support

Relays should answer requests such as `["REQ", "", {"authors": ["A"]}]` by querying both the `pubkey` and delegation tags `[1]` value.

Relays SHOULD allow the delegator (8e0d3d3e) to delete the events published by the delegatee (477318cf). NIP-27

Text Note References

`draft` `optional` `author:arthurfranca` `author:hodlbod` `author:fiatjaf`

This document standardizes the treatment given by clients of inline references of other events and profiles inside the `.content` of any event that has readable text in its `.content` (such as kinds 1 and 30023).

When creating an event, clients should include mentions to other profiles and to other events in the middle of the `.content` using NIP-21 codes, such as `nostr:nprofile1qqsw3dy8cpu...6x2argwghx6egsqstvg` .

Including NIP-10-style tags (`["e", <hex-id>, <relay-url>, <marker>]`) for each reference is optional, clients should do it whenever they want the profile being mentioned to be notified of the mention, or when they want the referenced event to recognize their mention as a reply.

A reader client that receives an event with such `nostr:...` mentions in its `.content` can do any desired context augmentation (for example, linking to the profile or showing a preview of the mentioned event contents) it wants in the process. If turning such mentions into links, they could become internal links, NIP-21 links or direct links to web clients that will handle these references.

Example of a profile mention process

Suppose Bob is writing a note in a client that has search-and-autocomplete functionality for users that is triggered when they write the character `@` .

As Bob types `"hello @mat"` the client will prompt him to autocomplete with mattn's profile, showing a picture and name.

Bob presses "enter" and now he sees his typed note as `"hello @mattn"` , `@mattn` is highlighted, indicating that it is a mention. Internally, however, the event looks like this:


```
{
  "content": "hello nostr:nprofile1qqszclxx9f5haga8sfjjrulaxncvkfekj097t6f3pu65f86rvg49ehqj6f9dh",
  "created_at": 1679790774,
  "id": "f39e9b451a73d62abc5016cffdd294b1a904e2f34536a208874fe5e22bbd47cf",
  "kind": 1,
  "pubkey": "79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798",
  "sig": "f8c8bab1b90cc3d2ae1ad999e6af8af449ad8bb4edf64807386493163e29162b5852a796a8f474d6b1001cddbaac0de4392838574f5366f03cc94cf5dfb43f4d",
  "tags": [
    [
      "p",
      "2c7cc62a697ea3a7826521f3fd34f0cb273693cbe5e9310f35449f43622a5cdc"
    ]
  ]
}
```


(Alternatively, the mention could have been a `nostr:npub1...` URL.)

After Bob publishes this event and Carol sees it, her client will initially display the `.content` as it is, but later it will parse the `.content` and see that there is a `nostr:` URL in there, decode it, extract the public key from it (and possibly relay hints), fetch that profile from its internal database or relays, then replace the full URL with the name `@mattn` , with a link to the internal page view for that profile.

Verbose and probably unnecessary considerations

- The example above was very concrete, but it doesn't mean all clients have to implement the same flow. There could be clients that do not support autocomplete at all, so they just allow users to paste raw NIP-19 codes into the body of text, then prefix these with `nostr:` before publishing the event.
- The flow for referencing other events is similar: a user could paste a `note1...` or `nevent1...` code and the client will turn that into a `nostr:note1...` or `nostr:nevent1...` URL. Then upon reading such references the client may show the referenced note in a preview box or something like that -- or nothing at all.
- Other display procedures can be employed: for example, if a client that is designed for dealing with only `kind:1` text notes sees, for example, a `kind:30023` `nostr:naddr1...` URL reference in the `.content` , it can, for example, decide to turn that into a link to some hardcoded webapp capable of displaying such events.
- Clients may give the user the option to include or not include tags for mentioned events or profiles. If someone wants to mention `mattn` without notifying them, but still have a nice augmentable/clickable link to their profile inside their note, they can instruct their client to *not* create a `["p", ...]` tag for that specific mention.
- In the same way, if someone wants to reference another note but their reference is not meant to show up along other replies to that same note, their client can choose to not include a corresponding `["e", ...]` tag for any given `nostr:nevent1...` URL inside `.content` . Clients may decide to expose these advanced functionalities to users or be more opinionated about things.

NIP-28

Public Chat

`draft` `optional` `author:ChristopherDavid` `author:fiatjaf` `author:jb55` `author:Cameri`

This NIP defines new event kinds for public chat channels, channel messages, and basic client-side moderation.

It reserves five event kinds (40-44) for immediate use:

- `40 - channel create`
- `41 - channel metadata`
- `42 - channel message`
- `43 - hide message`
- `44 - mute user`

Client-centric moderation gives client developers discretion over what types of content they want included in their apps, while imposing no additional requirements on relays.

Kind 40: Create channel

Create a public chat channel.

In the channel creation `content` field, Client SHOULD include basic channel metadata (`name`, `about`, `picture` as specified in kind 41).

```
{
    "content": "{\"name\": \"Demo Channel\", \"about\": \"A test channel.\", \"picture\": \"https://placekitten.com/200/200\"}",
    ...
}
```

Kind 41: Set channel metadata

Update a channel's public metadata.

Clients and relays SHOULD handle kind 41 events similar to kind 33 replaceable events, where the information is used to update the metadata, without modifying the event id for the channel. Only the most recent kind 41 is needed to be stored.

Clients SHOULD ignore kind 41s from pubkeys other than the kind 40 pubkey.

Clients SHOULD support basic metadata fields:

- `name` - string - Channel name
- `about` - string - Channel description
- `picture` - string - URL of channel picture

Clients MAY add additional metadata fields.

Clients SHOULD use NIP-10 marked "e" tags to recommend a relay.

```
{
    "content": "{\"name\": \"Updated Demo Channel\", \"about\": \"Updating a test channel.\", \"picture\": \"https://placekitten.com/201/201\"}",
    "tags": [["e", <channel_create_event_id>, <relay-url>]],
    ...
}
```

Kind 42: Create channel message

Send a text message to a channel.

Clients SHOULD use NIP-10 marked "e" tags to recommend a relay and specify whether it is a reply or root message.

Clients SHOULD append NIP-10 "p" tags to replies.

Root message:

```
{
    "content": <string>,
    "tags": [["e", <kind_40_event_id>, <relay-url>, "root"]],
    ...
}
```

Reply to another message:

```
{
    "content": <string>,
    "tags": [
        ["e", <kind_40_event_id>, <relay-url>, "root"],
        ["e", <kind_42_event_id>, <relay-url>, "reply"],
        ["p", <pubkey>, <relay-url>],
        ...
    ],
    ...
}
```

Kind 43: Hide message

User no longer wants to see a certain message.

The `content` may optionally include metadata such as a `reason`.

Clients SHOULD hide event 42s shown to a given user, if there is an event 43 from that user matching the event 42 `id`.

Clients MAY hide event 42s for other users other than the user who sent the event 43.

(For example, if three users 'hide' an event giving a reason that includes the word 'pornography', a Nostr client that is an iOS app may choose to hide that message for all iOS clients.)

```
{
   "content": "{\"reason\": \"Dick pic\"}",
   "tags": [["e", <kind_42_event_id>]],
   ...
}
```

Kind 44: Mute user

User no longer wants to see messages from another user.

The `content` may optionally include metadata such as a `reason`.

Clients SHOULD hide event 42s shown to a given user, if there is an event 44 from that user matching the event 42 `pubkey`.

Clients MAY hide event 42s for users other than the user who sent the event 44.

```
{
   "content": "{\"reason\": \"Posting dick pics\"}",
   "tags": [["p", <pubkey>]],
   ...
}
```

NIP-10 relay recommendations

For NIP-10 relay recommendations, clients generally SHOULD use the relay URL of the original (oldest) kind 40 event.

Clients MAY recommend any relay URL. For example, if a relay hosting the original kind 40 event for a channel goes offline, clients could instead fetch channel data from a backup relay, or a relay that clients trust more than the original relay.

Motivation

If we're solving censorship-resistant communication for social media, we may as well solve it also for Telegram-style messaging.

We can bring the global conversation out from walled gardens into a true public square open to all.

Additional info

- Chat demo PR with fiatjaf+jb55 comments
- Conversation about NIP16

NIP-30

Custom Emoji

`draft` `optional` `author:alexgleason`

Custom emoji may be added to **kind 0** and **kind 1** events by including one or more `"emoji"` tags, in the form:

```
["emoji", <shortcode>, <image-url>]
```

Where:

- `<shortcode>` is a name given for the emoji, which MUST be comprised of only alphanumeric characters and underscores.
- `<image-url>` is a URL to the corresponding image file of the emoji.

For each emoji tag, clients should parse emoji shortcodes (aka "emojify") like `:shortcode:` in the event to display custom emoji.

Clients may allow users to add custom emoji to an event by including `:shortcode:` identifier in the event, and adding the relevant `"emoji"` tags.

Kind 0 events

In kind 0 events, the `name` and `about` fields should be emojified.

```
{
  "kind": 0,
  "content": "{\"name\":\"Alex Gleason :soapbox:\"}",
  "tags": [
    ["emoji", "soapbox", "https://gleasonator.com/emoji/Gleasonator/soapbox.png"]
  ],
  "pubkey": "79c2cae114ea28a981e7559b4fe7854a473521a8d22a66bbab9fa248eb820ff6",
  "created_at": 1682790000
}
```

Kind 1 events

In kind 1 events, the `content` should be emojified.

```
{
  "kind": 1,
  "content": "Hello :gleasonator: :ablobcatrainbow: :disputed: yolo",
    "tags": [
    ["emoji", "ablobcatrainbow", "https://gleasonator.com/emoji/blobcat/ablobcatrainbow.png"],
    ["emoji", "disputed", "https://gleasonator.com/emoji/Fun/disputed.png"],
    ["emoji", "gleasonator", "https://gleasonator.com/emoji/Gleasonator/gleasonator.png"]
  ],
  "pubkey": "79c2cae114ea28a981e7559b4fe7854a473521a8d22a66bbab9fa248eb820ff6",
  "created_at": 1682630000
}
```

NIP-33

Parameterized Replaceable Events

`draft` `optional` `author:Semisol` `author:Kukks` `author:Cameri` `author:Giszmo`

This NIP adds a new event range that allows for replacement of events that have the same `d` tag and kind unlike NIP-16 which only replaced by kind.

Implementation

The value of a tag is defined as the first parameter of a tag after the tag name.

A *parameterized replaceable event* is defined as an event with a kind `30000 <= n < 40000`. Upon a parameterized replaceable event with a newer timestamp than the currently known latest replaceable event with the same kind, author and first `d` tag value being received, the old event SHOULD be discarded, effectively replacing what gets returned when querying for `author:kind:d-tag` tuples.

A missing or a `d` tag with no value should be interpreted equivalent to a `d` tag with the value as an empty string. Events from the same author with any of the following `tags` replace each other:

- `"tags":[["d",""]]`
- `"tags":[]` : implicit `d` tag with empty value
- `"tags":[["d"]]` : implicit empty value `""`
- `"tags":[["d",""],["d","not empty"]]` : only first `d` tag is considered
- `"tags":[["d"],["d","some value"]]` : only first `d` tag is considered
- `"tags":[["e"]]` : same as no tags
- `"tags":[["d","","1"]]` : only the first value is considered (`""`)

Clients SHOULD NOT use `d` tags with multiple values and SHOULD include the `d` tag even if it has no value to allow querying using the `#d` filter.

Referencing and tagging

Normally (as per NIP-01, NIP-12) the `"p"` tag is used for referencing public keys and the `"e"` tag for referencing event ids and the `note`, `npub`, `nprofile` or `nevent` are their equivalents for event tags (i.e. an `nprofile` is generally translated into a tag `["p", "<event hex id>", "<relay url>"]`).

To support linking to parameterized replaceable events, the `naddr` code is introduced on NIP-19. It includes the public key of the event author and the `d` tag (and relays) such that the referenced combination of public key and `d` tag can be found.

The equivalent in `tags` to the `naddr` code is the tag `"a"`, comprised of `["a", "<kind>:<pubkey>:<d-identifier>", "<relay url>"]` .

Client Behavior

Clients SHOULD use the `supported_nips` field to learn if a relay supports this NIP. Clients MAY send parameterized replaceable events to relays that may not support this NIP, and clients querying SHOULD be prepared for the relay to send multiple events and should use the latest one and are recommended to send a `#d` tag filter. Clients should account for the fact that missing `d` tags or ones with no value are not returned in tag filters, and are recommended to always include a `d` tag with a value.

NIP-36

Sensitive Content / Content Warning

`draft` `optional` `author:fernandolguevara`

The `content-warning` tag enables users to specify if the event's content needs to be approved by readers to be shown. Clients can hide the content until the user acts on it.

Spec

```
tag: content-warning
options:
 - [reason]: optional
```

Example

```
{
    "pubkey": "<pub-key>",
    "created_at": 1000000000,
    "kind": 1,
    "tags": [
      ["t", "hastag"],
      ["content-warning", "reason"] /* reason is optional */
    ],
    "content": "sensitive content with #hastag\n",
    "id": "<event-id>"
}
```

NIP-39

External Identities in Profiles

`draft` `optional` `author:pseudozach` `author:Semisol`

Abstract

Nostr protocol users may have other online identities such as usernames, profile pages, keypairs etc. they control and they may want to include this data in their profile metadata so clients can parse, validate and display this information.

`i` tag on a metadata event

A new optional `i` tag is introduced for `kind 0` metadata event contents in addition to name, about, picture fields as included in NIP-01:

```
{
    "id": <id>,
    "pubkey": <pubkey>,
    ...
    "tags": [
        ["i", "github:semisol", "9721ce4ee4fceb91c9711ca2a6c9a5ab"],
        ["i", "twitter:semisol_public", "1619358434134196225"],
        ["i", "mastodon:bitcoinhackers.org/@semisol", "109775066355589974"]
        ["i", "telegram:1087295469", "nostrdirectory/770"]
    ]
}
```

An `i` tag will have two parameters, which are defined as the following:

1. `platform:identity` : This is the platform name (for example `github`) and the identity on that platform (for example `semisol`) joined together with `:` .
2. `proof` : String or object that points to the proof of owning this identity.

Clients SHOULD process any `i` tags with more than 2 values for future extensibility.
Identity provider names SHOULD only include `a-z` , `0-9` and the characters `._-/` and MUST NOT include `:` .
Identity names SHOULD be normalized if possible by replacing uppercase letters with lowercase letters, and if there are multiple aliases for an entity the primary one should be used.

Claim types

`github`

Identity: A GitHub username.

Proof: A GitHub Gist ID. This Gist should be created by `<identity>` with a single file that has the text `Verifying that I control the following Nostr public key: <npub encoded public key>` .
This can be located at `https://gist.github.com/<identity>/<proof>` .

`twitter`

Identity: A Twitter username.

Proof: A Tweet ID. The tweet should be posted by `<identity>` and have the text `Verifying my account on nostr My Public Key: "<npub encoded public key>"` .
This can be located at `https://twitter.com/<identity>/status/<proof>` .

`mastodon`

Identity: A Mastodon instance and username in the format `<instance>/@<username>` .

Proof: A Mastodon post ID. This post should be published by `<username>@<instance>` and have the text `Verifying that I control the following Nostr public key: "<npub encoded public key>"` . This can be located at `https://<identity>/<proof>` .

`telegram`

Identity: A Telegram user ID.

Proof: A string in the format `<ref>/<id>` which points to a message published in the public channel or group with name `<ref>` and message ID `<id>` . This message should be sent by user ID `<identity>` and have the text `Verifying that I control the following Nostr public key: "<npub encoded public key>"` .
This can be located at `https://t.me/<proof>` .

NIP-40

Expiration Timestamp

`draft` `optional` `author:0xtlt`

The `expiration` tag enables users to specify a unix timestamp at which the message SHOULD be considered expired (by relays and clients) and SHOULD be deleted by relays.

Spec

```
tag: expiration
values:
 - [UNIX timestamp in seconds]: required
```

Example

```
{
    "pubkey": "<pub-key>",
    "created_at": 1000000000,
    "kind": 1,
    "tags": [
      ["expiration", "1600000000"]
    ],
    "content": "This message will expire at the specified timestamp and be deleted by relays.\n",
    "id": "<event-id>"
}
```

Note: The timestamp should be in the same format as the created_at timestamp and should be interpreted as the time at which the message should be deleted by relays.

Client Behavior

Clients SHOULD use the `supported_nips` field to learn if a relay supports this NIP. Clients SHOULD NOT send expiration events to relays that do not support this NIP.

Clients SHOULD ignore events that have expired.

Relay Behavior

Relays MAY NOT delete expired messages immediately on expiration and MAY persist them indefinitely.
Relays SHOULD NOT send expired events to clients, even if they are stored.
Relays SHOULD drop any events that are published to them if they are expired.
An expiration timestamp does not affect storage of ephemeral events.

Suggested Use Cases

- Temporary announcements - This tag can be used to make temporary announcements. For example, an event organizer could use this tag to post announcements about an upcoming event.
- Limited-time offers - This tag can be used by businesses to make limited-time offers that expire after a certain amount of time. For example, a business could use this tag to make a special offer that is only available for a limited time.

Warning

The events could be downloaded by third parties as they are publicly accessible all the time on the relays. So don't consider expiring messages as a security feature for your conversations or other uses.

NIP-42

Authentication of clients to relays

`draft` `optional` `author:Semisol` `author:fiatjaf`

This NIP defines a way for clients to authenticate to relays by signing an ephemeral event.

Motivation

A relay may want to require clients to authenticate to access restricted resources. For example,

- A relay may request payment or other forms of whitelisting to publish events -- this can naïvely be achieved by limiting publication to events signed by the whitelisted key, but with this NIP they may choose to accept any events as long as they are published from an authenticated user;
- A relay may limit access to `kind: 4` DMs to only the parties involved in the chat exchange, and for that it may require authentication before clients can query for that kind.
- A relay may limit subscriptions of any kind to paying users or users whitelisted through any other means, and require authentication.

Definitions

This NIP defines a new message, `AUTH`, which relays can send when they support authentication and clients can send to relays when they want to authenticate. When sent by relays, the message is of the following form:

```
["AUTH", <challenge-string>]
```

And, when sent by clients, of the following form:

```
["AUTH", <signed-event-json>]
```

The signed event is an ephemeral event not meant to be published or queried, it must be of `kind: 22242` and it should have at least two tags, one for the relay URL and one for the challenge string as received from the relay. Relays MUST exclude `kind: 22242` events from being broadcasted to any client. `created_at` should be the current time. Example:

```
{
  "id": "...",
  "pubkey": "...",
  "created_at": 1669695536,
  "kind": 22242,
  "tags": [
    ["relay", "wss://relay.example.com/"],
    ["challenge", "challengestringhere"]
  ],
  "content": "",
  "sig": "..."
}
```

Protocol flow

At any moment the relay may send an `AUTH` message to the client containing a challenge. After receiving that the client may decide to authenticate itself or not. The challenge is expected to be valid for the duration of the connection or until a next challenge is sent by the relay.

The client may send an auth message right before performing an action for which it knows authentication will be required -- for example, right before requesting `kind: 4` chat messages --, or it may do right on connection start or at some other moment it deems best. The authentication is expected to last for the duration of the WebSocket connection.

Upon receiving a message from an unauthenticated user it can't fulfill without authentication, a relay may choose to notify the client. For that it can use a `NOTICE` or `OK` message with a standard prefix `"restricted: "`

that is readable both by humans and machines, for example:

```
["NOTICE", "restricted: we can't serve DMs to unauthenticated users, does your client implement NIP-42?"]
```

or it can return an `OK` message noting the reason an event was not written using the same prefix:

```
["OK", <event-id>, false, "restricted: we do not accept events from unauthenticated users, please sign up at https://example.com/"]
```

Signed Event Verification

To verify `AUTH` messages, relays must ensure:

- that the `kind` is `22242`;
- that the event `created_at` is close (e.g. within ~10 minutes) of the current time;
- that the `"challenge"` tag matches the challenge sent before;
- that the `"relay"` tag matches the relay URL:
  - URL normalization techniques can be applied. For most cases just checking if the domain name is correct should be enough.

NIP-46

Nostr Connect

`draft` `optional` `author:tiero` `author:giowe` `author:vforvalerio87`

Rationale

Private keys should be exposed to as few systems - apps, operating systems, devices - as possible as each system adds to the attack surface.

Entering private keys can also be annoying and requires exposing them to even more systems such as the operating system's clipboard that might be monitored by malicious apps.

Terms

- **App**: Nostr app on any platform that *requires* to act on behalf of a nostr account.
- **Signer**: Nostr app that holds the private key of a nostr account and *can sign* on its behalf.

`TL;DR`

App and **Signer** sends ephemeral encrypted messages to each other using kind `24133`, using a relay of choice.

App prompts the Signer to do things such as fetching the public key or signing events.

The `content` field must be an encrypted JSONRPC-ish **request** or **response**.

Signer Protocol

Messages

Request

```
{
  "id": <random_string>,
  "method": <one_of_the_methods>,
  "params": [<anything>, <else>]
}
```

Response

```
{
  "id": <request_id>,
  "result": <anything>,
  "error": <reason>
}
```

Methods

Mandatory

These are mandatory methods the remote signer app MUST implement:

- **describe**
  - params `[]`
  - result `["describe", "get_public_key", "sign_event", "connect", "disconnect", "delegate", ...]`
- **get_public_key**
  - params `[]`
  - result `pubkey`
- **sign_event**
  - params `[ event ]`
  - result `event_with_signature`

optional

- **connect**
  - params `[ pubkey ]`
- **disconnect**
  - params `[]`
- **delegate**
  - params `[ delegatee, { kind: number, since: number, until: number } ]`
  - result `{ from: string, to: string, cond: string, sig: string }`
- **get_relays**

    - params []
    - result `{ [url: string]: {read: boolean, write: boolean} }`
- **nip04_encrypt**
    - params [`pubkey`, `plaintext`]
    - result `nip4 ciphertext`
- **nip04_decrypt**
    - params [`pubkey`, `nip4 ciphertext`]
    - result [`plaintext`]

NOTICE: `pubkey` and `signature` are hex-encoded strings.

Nostr Connect URI

Signer discovers **App** by scanning a QR code, clicking on a deep link or copy-pasting an URI.

The **App** generates a special URI with prefix `nostrconnect://` and base path the hex-encoded `pubkey` with the following querystring parameters **URL encoded**

- `relay` URL of the relay of choice where the **App** is connected and the **Signer** must send and listen for messages.
- `metadata` metadata JSON of the **App**
    - `name` human-readable name of the **App**
    - `url` (optional) URL of the website requesting the connection
    - `description` (optional) description of the **App**
    - `icons` (optional) array of URLs for icons of the **App**.

JavaScript

```
const uri = `nostrconnect://<pubkey>?relay=${encodeURIComponent("wss://relay.damus.io")}&metadata=${encodeURIComponent(JSON.stringify({"name": "Example"}))}`
```

Example

```
nostrconnect://b889ff5b1513b641e2a139f661a661364979c5beee91842f8f0ef42ab558e9d4?relay=wss%3A%2F%2Frelay.damus.io&metadata=%7B%22name%22%3A%22Example%22%7D
```

Flows

The `content` field contains encrypted message as specified by NIP04. The `kind` chosen is `24133`.

Connect

1. User clicks on **"Connect"** button on a website or scan it with a QR code
2. It will show an URI to open a "nostr connect" enabled **Signer**
3. In the URI there is a pubkey of the **App** ie. `nostrconnect://<pubkey>&relay=<relay>&metadata=<metadata>`
4. The **Signer** will send a message to ACK the `connect` request, along with his public key

Disconnect (from App)

1. User clicks on **"Disconnect"** button on the **App**
2. The **App** will send a message to the **Signer** with a `disconnect` request
3. The **Signer** will send a message to ACK the `disconnect` request

Disconnect (from Signer)

1. User clicks on **"Disconnect"** button on the **Signer**
2. The **Signer** will send a message to the **App** with a `disconnect` request

Get Public Key

1. The **App** will send a message to the **Signer** with a `get_public_key` request
2. The **Signer** will send back a message with the public key as a response to the `get_public_key` request

Sign Event

1. The **App** will send a message to the **Signer** with a `sign_event` request along with the **event** to be signed
2. The **Signer** will show a popup to the user to inspect the event and sign it
3. The **Signer** will send back a message with the event including the `id` and the schnorr `signature` as a response to the `sign_event` request

Delegate

1. The **App** will send a message with metadata to the **Signer** with a `delegate` request along with the **conditions** query string and the **pubkey** of the **App** to be delegated.
2. The **Signer** will show a popup to the user to delegate the **App** to sign on his behalf
3. The **Signer** will send back a message with the signed NIP-26 delegation token or reject it

NIP-47

Nostr Wallet Connect

`draft` `optional` `author:kiwiidb` `author:bumi` `author:semisol` `author:vitorpamplona`

Rationale

This NIP describes a way for clients to access a remote Lightning wallet through a standardized protocol. Custodians may implement this, or the user may run a bridge that bridges their wallet/node and the Nostr Wallet Connect protocol.

Terms

- **client**: Nostr app on any platform that wants to pay Lightning invoices.
- **user**: The person using the **client**, and want's to connect their wallet app to their **client**.
- **wallet service**: Nostr app that typically runs on an always-on computer (eg. in the cloud or on a Raspberry Pi). This app has access to the APIs of the wallets it serves.

Theory of Operation

1. **Users** who which to use this NIP to send lightning payments to other nostr users must first acquire a special "connection" URI from their NIP-47 compliant wallet application. The wallet application may provide this

URI using a QR screen, or a pasteable string, or some other means.

2. The **user** should then copy this URI into their **client(s)** by pasting, or scanning the QR, etc. The **client(s)** should save this URI and use it later whenever the **user** makes a payment. The **client** should then request an `info` (13194) event from the relay(s) specified in the URI. The **wallet service** will have sent that event to those relays earlier, and the relays will hold it as a replaceable event.
3. When the **user** initiates a payment their nostr **client** create a `pay_invoice` request, encrypts it using a token from the URI, and sends it (kind 23194) to the relay(s) specified in the connection URI. The **wallet service** will be listening on those relays and will decrypt the request and then contact the **user's** wallet application to send the payment. The **wallet service** will know how to talk to the wallet application because the connection URI specified relay(s) that have access to the wallet app API.
4. Once the payment is complete the **wallet service** will send an encrypted `response` (kind 23195) to the **user** over the relay(s) in the URI.

Events

There are three event kinds:

- `NIP-47 info event` : 13194
- `NIP-47 request` : 23194
- `NIP-47 response` : 23195

The info event should be a replaceable event that is published by the **wallet service** on the relay to indicate which commands it supports. The content should be a plaintext string with the supported commands, space-seperated, eg. `pay_invoice get_balance` . Only the `pay_invoice` command is described in this NIP, but other commands might be defined in different NIPs.

Both the request and response events SHOULD contain one `p` tag, containing the public key of the **wallet service** if this is a request, and the public key of the **user** if this is a response. The response event SHOULD contain an `e` tag with the id of the request event it is responding to.

The content of requests and responses is encrypted with NIP04, and is a JSON-RPCish object with a semi-fixed structure:

Request:

```
{
    "method": "pay_invoice", // method, string
    "params": { // params, object
        "invoice": "lnbc50n1..." // command-related data
    }
}
```

Response:

```
{
    "result_type": "pay_invoice", //indicates the structure of the result field
    "error": { //object, non-null in case of error
        "code": "UNAUTHORIZED", //string error code, see below
        "message": "human readable error message"
    },
    "result": { // result, object. null in case of error.
        "preimage": "0123456789abcdef..." // command-related data
    }
}
```

The `result_type` field MUST contain the name of the method that this event is responding to. The `error` field MUST contain a `message` field with a human readable error message and a `code` field with the error code if the command was not succesful. If the command was succesful, the `error` field must be null.

Error codes

- `RATE_LIMITED` : The client is sending commands too fast. It should retry in a few seconds.
- `NOT_IMPLEMENTED` : The command is not known or is intentionally not implemented.
- `INSUFFICIENT_BALANCE` : The wallet does not have enough funds to cover a fee reserve or the payment amount.
- `QUOTA_EXCEEDED` : The wallet has exceeded its spending quota.
- `RESTRICTED` : This public key is not allowed to do this operation.
- `UNAUTHORIZED` : This public key has no wallet connected.
- `INTERNAL` : An internal error.
- `OTHER` : Other error.

Nostr Wallet Connect URI

client discovers **wallet service** by scanning a QR code, handling a deeplink or pasting in a URI.

The **wallet service** generates this connection URI with protocol `nostr+walletconnect:` and base path it's hex-encoded `pubkey` with the following query string parameters:

- `relay` Required. URL of the relay where the **wallet service** is connected and will be listening for events. May be more than one.
- `secret` Required. 32-byte randomly generated hex encoded string. The **client** MUST use this to sign events and encrypt payloads when communicating with the **wallet service**.
  - Authorization does not require passing keys back and forth.
  - The user can have different keys for different applications. Keys can be revoked and created at will and have arbitrary constraints (eg. budgets).
  - The key is harder to leak since it is not shown to the user and backed up.
  - It improves privacy because the user's main key would not be linked to their payments.
- `lud16` Recommended. A lightning address that clients can use to automatically setup the `lud16` field on the user's profile if they have none configured.

The **client** should then store this connection and use it when the user wants to perform actions like paying an invoice. Due to this NIP using ephemeral events, it is recommended to pick relays that do not close connections on inactivity to not drop events.

Example connection string

```
nostr+walletconnect:b889ff5b1513b641e2a139f661a661364979c5beee91842f8f0ef42ab558e9d4?relay=wss%3A%2F%2Frelay.damus.io&secret=71a8c14c1407c113601079c4302dab36460f0ccd0ad506f1f2
```

Commands

`pay_invoice`

Description: Requests payment of an invoice.

Request:

```
{
    "method": "pay_invoice",
    "params": {
        "invoice": "lnbc50n1..." // bolt11 invoice
    }
}
```

Response:

```
{
    "result_type": "pay_invoice",
    "result": {
        "preimage": "0123456789abcdef..." // preimage of the payment
    }
}
```

Errors:

- `PAYMENT_FAILED`: The payment failed. This may be due to a timeout, exhausting all routes, insufficient capacity or similar.

Example pay invoice flow

0. The user scans the QR code generated by the **wallet service** with their **client** application, they follow a `nostr+walletconnect:` deeplink or configure the connection details manually.
1. **client** sends an event to the **wallet service** service with kind `23194`. The content is a `pay_invoice` request. The private key is the secret from the connection string above.
2. **wallet service** verifies that the author's key is authorized to perform the payment, decrypts the payload and sends the payment.
3. **wallet service** responds to the event by sending an event with kind `23195` and content being a response either containing an error message or a preimage.

Using a dedicated relay

This NIP does not specify any requirements on the type of relays used. However, if the user is using a custodial service it might make sense to use a relay that is hosted by the custodial service. The relay may then enforce authentication to prevent metadata leaks. Not depending on a 3rd party relay would also improve reliability in this case.

NIP-50

Search Capability

`draft` `optional` `author:brugeman` `author:mikedilger` `author:fiatjaf`

Abstract

Many Nostr use cases require some form of general search feature, in addition to structured queries by tags or ids. Specifics of the search algorithms will differ between event kinds, this NIP only describes a general extensible framework for performing such queries.

`search` filter field

A new `search` field is introduced for `REQ` messages from clients:

```
{
  ...
  "search": <string>
}
```

`search` field is a string describing a query in a human-readable form, i.e. "best nostr apps". Relays SHOULD interpret the query to the best of their ability and return events that match it. Relays SHOULD perform matching against `content` event field, and MAY perform matching against other fields if that makes sense in the context of a specific kind.

A query string may contain `key:value` pairs (two words separated by colon), these are extensions, relays SHOULD ignore extensions they don't support.

Clients may specify several search filters, i.e. `["REQ", "", { "search": "orange" }, { "kinds": [1, 2], "search": "purple" }]`. Clients may include `kinds`, `ids` and other filter field to restrict the search results to particular event kinds.

Clients SHOULD use the supported_nips field to learn if a relay supports `search` filter. Clients MAY send `search` filter queries to any relay, if they are prepared to filter out extraneous responses from relays that do not support this NIP.

Clients SHOULD query several relays supporting this NIP to compensate for potentially different implementation details between relays.

Clients MAY verify that events returned by a relay match the specified query in a way that suits the client's use case, and MAY stop querying relays that have low precision.

Relays SHOULD exclude spam from search results by default if they supports some form of spam filtering.

Extensions

Relay MAY support these extensions:

- `include:spam` - turn off spam filtering, if it was enabled by default

NIP-51

Lists

`draft` `optional` `author:fiatjaf` `author:arcbtc` `author:monlovesmango` `author:eskema` `depends:33`

A "list" event is defined as having a list of public and/or private tags. Public tags will be listed in the event `tags`. Private tags will be encrypted in the event `content`. Encryption for private tags will use NIP-04 - Encrypted Direct Message encryption, using the list author's private and public key for the shared secret. A distinct event kind should be used for each list type created.

If a list type should only be defined once per user (like the 'Mute' list), the list type's events should follow the specification for NIP-16 - Replaceable Events. These lists may be referred to as 'replaceable lists'.

Otherwise, the list type's events should follow the specification for NIP-33 - Parameterized Replaceable Events, where the list name will be used as the 'd' parameter. These lists may be referred to as 'parameterized replaceable lists'.

Replaceable List Event Example

Lets say a user wants to create a 'Mute' list and has keys:

```
priv: fb505c65d4df950f5d28c9e4d285ee12ffaf315deef1fc24e3c7cd1e7e35f2b1
pub: b1a5c93edcc8d586566fde53a20bdb50049a97b15483cb763854e57016e0fa3d
```

The user wants to publicly include these users:

```
["p", "3bf0c63fcb93463407af97a5e5ee64fa883d107ef9e558472c4eb9aaaefa459d"],
["p", "32e1827635450ebb3c5a7d12c1f8e7b2b514439ac10a67eef3d9fd9c5c68e245"]
```

and privately include these users (below is the JSON that would be encrypted and placed in the event content):

```
[
   ["p", "9ec7a778167afb1d30c4833de9322da0c08ba71a69e1911d5578d3144bb56437"],
   ["p", "8c0da4862130283ff9e67d889df264177a508974e2feb96de139804ea66d6168"]
]
```

Then the user would create a 'Mute' list event like below:

```
{
  "kind": 10000,
  "tags": [
    ["p", "3bf0c63fcb93463407af97a5e5ee64fa883d107ef9e558472c4eb9aaaefa459d"],
    ["p", "32e1827635450ebb3c5a7d12c1f8e7b2b514439ac10a67eef3d9fd9c5c68e245"],
  ],
  "content": "VezuSvWak++ASjFMRqBPWS3mK5pZ0vRLL325iuIL4S+r8n9z+DuMau5vMElz1tGC/UqCDmbzE2kwplafaFo/FnIZMdEj4pdxgptyBV1ifZpH3TEF6OMjEtqbYRRqnxgIXsuOSXaerWgpi0pm+raHQPseoELQI/SZ1
  ...other fields
}
```

Parameterized Replaceable List Event Example

Lets say a user wants to create a 'Categorized People' list of `nostr` people and has keys:

```
priv: fb505c65d4df950f5d28c9e4d285ee12ffaf315deef1fc24e3c7cd1e7e35f2b1
pub: b1a5c93edcc8d586566fde53a20bdb50049a97b15483cb763854e57016e0fa3d
```

The user wants to publicly include these users:

```
["p", "3bf0c63fcb93463407af97a5e5ee64fa883d107ef9e558472c4eb9aaaefa459d"],
["p", "32e1827635450ebb3c5a7d12c1f8e7b2b514439ac10a67eef3d9fd9c5c68e245"]
```

and privately include these users (below is the JSON that would be encrypted and placed in the event content):

```
[
   ["p", "9ec7a778167afb1d30c4833de9322da0c08ba71a69e1911d5578d3144bb56437"],
   ["p", "8c0da4862130283ff9e67d889df264177a508974e2feb96de139804ea66d6168"]
]
```

Then the user would create a 'Categorized People' list event like below:

```
{
  "kind": 30000,
  "tags": [
    ["d", "nostr"],
    ["p", "3bf0c63fcb93463407af97a5e5ee64fa883d107ef9e558472c4eb9aaaefa459d"],
    ["p", "32e1827635450ebb3c5a7d12c1f8e7b2b514439ac10a67eef3d9fd9c5c68e245"],
  ],
  "content": "VezuSvWak++ASjFMRqBPWS3mK5pZ0vRLL325iuIL4S+r8n9z+DuMau5vMElz1tGC/UqCDmbzE2kwplafaFo/FnIZMdEj4pdxgptyBV1ifZpH3TEF6OMjEtqbYRRqnxgIXsuOSXaerWgpi0pm+raHQPseoELQI/SZ1
  ...other fields
}
```

List Event Kinds

kind	list type
10000	Mute
10001	Pin
30000	Categorized People
30001	Categorized Bookmarks

Mute List

An event with kind `10000` is defined as a replaceable list event for listing content a user wants to mute. Any standardized tag can be included in a Mute List.

Pin List

An event with kind `10001` is defined as a replaceable list event for listing content a user wants to pin. Any standardized tag can be included in a Pin List.

Categorized People List

An event with kind `30000` is defined as a parameterized replaceable list event for categorizing people. The 'd' parameter for this event holds the category name of the list. The tags included in these lists MUST follow the format of kind 3 events as defined in NIP-02 - Contact List and Petnames.

Categorized Bookmarks List

An event with kind `30001` is defined as a parameterized replaceable list event for categorizing bookmarks. The 'd' parameter for this event holds the category name of the list. Any standardized tag can be included in a Categorized Bookmarks List.

NIP-56

Reporting

`draft` `optional` `author:jb55`

A report is a `kind 1984` note that is used to report other notes for spam, illegal and explicit content.

The content MAY contain additional information submitted by the entity reporting the content.

Tags

The report event MUST include a `p` tag referencing the pubkey of the user you are reporting.

If reporting a note, an `e` tag MUST also be included referencing the note id.

A `report type` string MUST be included as the 3rd entry to the `e` or `p` tag being reported, which consists of the following report types:

- `nudity` - depictions of nudity, porn, etc.
- `profanity` - profanity, hateful speech, etc.
- `illegal` - something which may be illegal in some jurisdiction
- `spam` - spam
- `impersonation` - someone pretending to be someone else

Some report tags only make sense for profile reports, such as `impersonation`

Example events

```
{
  "kind": 1984,
  "tags": [
    [ "p", <pubkey>, "nudity"]
  ],
  "content": "",
  ...
}

{
  "kind": 1984,
  "tags": [
    [ "e", <eventId>, "illegal"],
    [ "p", <pubkey>]
  ],
  "content": "He's insulting the king!",
  ...
}

{
  "kind": 1984,
  "tags": [
    [ "p", <impersonator pubkey>, "impersonation"],
    [ "p", <victim pubkey>]
  ],
  "content": "Profile is imitating #[1]",
  ...
}
```

Client behavior

Clients can use reports from friends to make moderation decisions if they choose to. For instance, if 3+ of your friends report a profile as explicit, clients can have an option to automatically blur photos from said account.

Relay behavior

It is not recommended that relays perform automatic moderation using reports, as they can be easily gamed. Admins could use reports from trusted moderators to takedown illegal or explicit content if the relay does not allow such things.

NIP-57

Lightning Zaps

`draft` `optional` `author:jb55` `author:kieran`

This NIP defines two new event types for recording lightning payments between users. `9734` is a `zap request`, representing a payer's request to a recipient's lightning wallet for an invoice. `9735` is a `zap receipt`, representing the confirmation by the recipient's lightning wallet that the invoice issued in response to a `zap request` has been paid.

Having lightning receipts on nostr allows clients to display lightning payments from entities on the network. These can be used for fun or for spam deterrence.

Protocol flow

1. Client calculates a recipient's lnurl pay request url from the `zap` tag on the event being zapped (see Appendix G), or by decoding their lud06 or lud16 field on their profile according to the lnurl specifications. The client MUST send a GET request to this url and parse the response. If `allowsNostr` exists and it is `true`, and if `nostrPubkey` exists and is a valid BIP 340 public key in hex, the client should associate this information with the user, along with the response's `callback`, `minSendable`, and `maxSendable` values.
2. Clients may choose to display a lightning zap button on each post or on a user's profile. If the user's lnurl pay request endpoint supports nostr, the client SHOULD use this NIP to request a `zap receipt` rather than a normal lnurl invoice.
3. When a user (the "sender") indicates they want to send a zap to another user (the "recipient"), the client should create a `zap request` event as described in Appendix A of this NIP and sign it.
4. Instead of publishing the `zap request`, the `9734` event should instead be sent to the `callback` url received from the lnurl pay endpoint for the recipient using a GET request. See Appendix B for details and an example.
5. The recipient's lnurl server will receive this `zap request` and validate it. See Appendix C for details on how to properly configure an lnurl server to support zaps, and Appendix D for details on how to validate the `nostr` query parameter.
6. If the `zap request` is valid, the server should fetch a description hash invoice where the description is this `zap request` note and this note only. No additional lnurl metadata is included in the description. This will be returned in the response according to LUD06.
7. On receiving the invoice, the client MAY pay it or pass it to an app that can pay the invoice.
8. Once the invoice is paid, the recipient's lnurl server MUST generate a `zap receipt` as described in Appendix E, and publish it to the `relays` specified in the `zap request`.
9. Clients MAY fetch `zap receipt`s on posts and profiles, but MUST authorize their validity as described in Appendix F. If the `zap request` note contains a non-empty `content`, it may display a zap comment. Generally clients should show users the `zap request` note, and use the `zap receipt` to show "zap authorized by ..." but this is optional.

Reference and examples

Appendix A: Zap Request Event

A `zap request` is an event of kind `9734` that is *not* published to relays, but is instead sent to a recipient's lnurl pay `callback` url. This event's `content` MAY be an optional message to send along with the payment. The event MUST include the following tags:

- `relays` is a list of relays the recipient's wallet should publish its `zap receipt` to. Note that relays should not be nested in an additional list, but should be included as shown in the example below.
- `amount` is the amount in *millisats* the sender intends to pay, formatted as a string. This is recommended, but optional.
- `lnurl` is the lnurl pay url of the recipient, encoded using bech32 with the prefix `lnurl`. This is recommended, but optional.
- `p` is the hex-encoded pubkey of the recipient.

In addition, the event MAY include the following tags:

- `e` is an optional hex-encoded event id. Clients MUST include this if zapping an event rather than a person.
- `a` is an optional NIP-33 event coordinate that allows tipping parameterized replaceable events such as NIP-23 long-form notes.

Example:

```
{
  "kind": 9734,
  "content": "Zap!",
  "tags": [
    ["relays", "wss://nostr-pub.wellorder.com"],
    ["amount", "21000"],
    ["lnurl", "lnurl1dp68gurn8ghj7um5v93kketj9ehx2amn9uh8wetvdskkkmn0wahz7mrww4excup0dajx2mrv92x9xp"],
    ["p", "04c915daefee38317fa734444acee390a8269fe5810b2241e5e6dd343dfbecc9"],
    ["e", "9ae37aa68f48645127299e9453eb5d908a0cbb6058ff340d528ed4d37c8994fb"]
  ],
  "pubkey": "97c70a44366a6535c145b333f973ea86dfdc2d7a99da618c40c64705ad98e322",
  "created_at": 1679673265,
  "id": "30efed56a035b2549fcaeec0bf2c1595f9a9b3bb4b1a38abaf8ee9041c4b7d93",
  "sig": "f2cb581a84ed10e4dc84937bd98e27acac71ab057255f6aa8dfa561808c981fe8870f4a03c1e3666784d82a9c802d3704e174371aa13d63e2aeaf24ff5374d9d"
}
```

Appendix B: Zap Request HTTP Request

A signed `zap request` event is not published, but is instead sent using a HTTP GET request to the recipient's `callback` url, which was provided by the recipient's lnurl pay endpoint. This request should have the following query parameters defined:

- `amount` is the amount in *millisats* the sender intends to pay
- `nostr` is the `9734` `zap request` event, JSON encoded then URI encoded
- `lnurl` is the lnurl pay url of the recipient, encoded using bech32 with the prefix `lnurl`

This request should return a JSON response with a `pr` key, which is the invoice the sender must pay to finalize his zap. Here is an example flow:

```
 const senderPubkey // The sender's pubkey
const recipientPubkey = // The recipient's pubkey
const callback = // The callback received from the recipients lnurl pay endpoint
const lnurl = // The recipient's lightning address, encoded as a lnurl
const sats = 21

const amount = sats * 1000
const relays = ['wss://nostr-pub.wellorder.net']
const event = encodeURI(JSON.stringify(await signEvent({
  kind: [9734],
  content: "",
  pubkey: senderPubkey,
  created_at: Math.round(Date.now() / 1000),
  tags: [
    ["relays", ...relays],
    ["amount", amount.toString()],
    ["lnurl", lnurl],
    ["p", recipientPubkey],
  ],
})))

const {pr: invoice} = await fetchJson(`${callback}?amount=${amount}&nostr=${event}&lnurl=${lnurl}`)
```

Appendix C: LNURL Server Configuration

The lnurl server will need some additional pieces of information so that clients can know that zap invoices are supported:

1. Add a `nostrPubkey` to the lnurl-pay static endpoint `/.well-known/lnurlp/<user>`, where `nostrPubkey` is the nostr pubkey your server will use to sign `zap receipt` events. Clients will use this to validate `zap receipt`s.
2. Add an `allowsNostr` field and set it to true.

Appendix D: LNURL Server Zap Request Validation

When a client sends a `zap request` event to a server's lnurl-pay callback URL, there will be a `nostr` query parameter whose value is that event which is URI- and JSON-encoded. If present, the `zap request` event must be validated in the following ways:

1. It MUST have a valid nostr signature
2. It MUST have tags
3. It MUST have only one `p` tag
4. It MUST have 0 or 1 `e` tags
5. There should be a `relays` tag with the relays to send the `zap receipt` to.
6. If there is an `amount` tag, it MUST be equal to the `amount` query parameter.
7. If there is an `a` tag, it MUST be a valid NIP-33 event coordinate

The event MUST then be stored for use later, when the invoice is paid.

Appendix E: Zap Receipt Event

A `zap receipt` is created by a lightning node when an invoice generated by a `zap request` is paid. `Zap receipt`s are only created when the invoice description (committed to the description hash) contains a `zap request` note.

When receiving a payment, the following steps are executed:

1. Get the description for the invoice. This needs to be saved somewhere during the generation of the description hash invoice. It is saved automatically for you with CLN, which is the reference implementation used here.
2. Parse the bolt11 description as a JSON nostr event. This SHOULD be validated based on the requirements in Appendix D, either when it is received, or before the invoice is paid.
3. Create a nostr event of kind `9735` as described below, and publish it to the `relays` declared in the `zap request`.

The following should be true of the `zap receipt` event:

- The content SHOULD be empty.
- The `created_at` date SHOULD be set to the invoice `paid_at` date for idempotency.
- `tags` MUST include the `p` tag AND optional `e` tag from the `zap request`.
- The `zap receipt` MUST have a `bolt11` tag containing the description hash bolt11 invoice.
- The `zap receipt` MUST contain a `description` tag which is the JSON-encoded invoice description.
- `SHA256(description)` MUST match the description hash in the bolt11 invoice.
- The `zap receipt` MAY contain a `preimage` tag to match against the payment hash of the bolt11 invoice. This isn't really a payment proof, there is no real way to prove that the invoice is real or has been paid. You are trusting the author of the `zap receipt` for the legitimacy of the payment.

The `zap receipt` is not a proof of payment, all it proves is that some nostr user fetched an invoice. The existence of the `zap receipt` implies the invoice as paid, but it could be a lie given a rogue implementation.

A reference implementation for a zap-enabled lnurl server can be found here.

Example `zap receipt`:

```
{
   "id": "67b48a14fb66c60c8f9070bdeb37afdfcc3d08ad01989460448e4081eddda446",
   "pubkey": "9630f464cca6a5147aa8a35f0bcdd3ce485324e732fd39e09233b1d848238f31",
   "created_at": 1674164545,
   "kind": 9735,
   "tags": [
     ["p", "32e1827635450ebb3c5a7d12c1f8e7b2b514439ac10a67eef3d9fd9c5c68e245"],
     ["e", "3624762a1274dd9636e0c552b53086d70bc88c165bc4dc0f9e836a1eaf86c3b8"],
     ["bolt11", "lnbc10u1p3unwfusp5t9r3yymhpfqculx78u027lxspgxcr2n2987mx2j55nnfs95nxnzqpp5jmrh92pfld78spqs78v9euf2385t83uvpwk9ldrlvf6ch7tpascqhp5zvkrmemgth3tufcvflmzjzfvjt023
     ["description", "{\"pubkey\":\"32e1827635450ebb3c5a7d12c1f8e7b2b514439ac10a67eef3d9fd9c5c68e245\",\"content\":\"\",\"id\":\"d9cc14d50fcb8c27539aacf776882942c1a11ea4472f8
     ["preimage", "5d006d2cf1e73c7148e7519a4c68adc81642ce0e25a432b2434c99f97344c15f"]
   ],
   "content": "",
   "sig": "b0a3c5c984ceb777ac455b2f659505df51585d5fd97a0ec1fdb5f3347d392080d4b420240434a3afd909207195dac1e2f7e3df26ba862a45afd8bfe101c2b1cc"
 }
```

Appendix F: Validating Zap Receipts

A client can retrieve `zap receipt`s on events and pubkeys using a NIP-01 filter, for example `{"kinds": [9735], "#e": [...]}`. Zaps MUST be validated using the following steps:

- The `zap receipt` event's `pubkey` MUST be the same as the recipient's lnurl provider's `nostrPubkey` (retrieved in step 1 of the protocol flow).
- The `invoiceAmount` contained in the `bolt11` tag of the `zap receipt` MUST equal the `amount` tag of the `zap request` (if present).
- The `lnurl` tag of the `zap request` (if present) SHOULD equal the recipient's `lnurl`.

Appendix G: `zap` tag on zapped event

When an event includes a `zap` tag, clients SHOULD calculate the lnurl pay request based on it's value instead of the profile's field. An optional third argument on the tag specifies the type of value, either `lud06` or `lud16`.

```
{
   "tags": [
       [ "zap", "pablo@f7z.io", "lud16" ]
   ]
}
```

Future Work

Zaps can be extended to be more private by encrypting `zap request` notes to the target user, but for simplicity it has been left out of this initial draft.

NIP-58

Badges

`draft optional author:cameri`

Three special events are used to define, award and display badges in user profiles:

1. A "Badge Definition" event is defined as a parameterized replaceable event with kind `30009` having a `d` tag with a value that uniquely identifies the badge (e.g. `bravery`) published by the badge issuer. Badge definitions can be updated.
2. A "Badge Award" event is a kind `8` event with a single `a` tag referencing a "Define Badge" event and one or more `p` tags, one for each pubkey the badge issuer wishes to award. The value for the `a` tag MUST follow the format defined in NIP-33. Awarded badges are immutable and non-transferrable.

3. A "Profile Badges" event is defined as a parameterized replaceable event with kind `30008` with a `d` tag with the value `profile_badges`. Profile badges contain an ordered list of pairs of `a` and `e` tags referencing a `Badge Definition` and a `Badge Award` for each badge to be displayed.

Badge Definition event

The following tags MUST be present:

- `d` tag with the unique name of the badge.

The following tags MAY be present:

- A `name` tag with a short name for the badge.
- `image` tag whose value is the URL of a high-resolution image representing the badge. The second value optionally specifies the dimensions of the image as `width` x `height` in pixels. Badge recommended dimensions is 1024x1024 pixels.
- A `description` tag whose value MAY contain a textual representation of the image, the meaning behind the badge, or the reason of it's issuance.
- One or more `thumb` tags whose first value is an URL pointing to a thumbnail version of the image referenced in the `image` tag. The second value optionally specifies the dimensions of the thumbnail as `width` x `height` in pixels.

Badge Award event

The following tags MUST be present:

- An `a` tag referencing a kind `30009` Badge Definition event.
- One or more `p` tags referencing each pubkey awarded.

Profile Badges Event

The number of badges a pubkey can be awarded is unbounded. The Profile Badge event allows individual users to accept or reject awarded badges, as well as choose the display order of badges on their profiles.

The following tags MUST be present:

- A `d` tag with the unique identifier `profile_badges`

The following tags MAY be present:

- Zero or more ordered consecutive pairs of `a` and `e` tags referencing a kind `30009` Badge Definition and kind `8` Badge Award, respectively. Clients SHOULD ignore `a` without corresponding `e` tag and viceversa. Badge Awards referenced by the `e` tags should contain the same `a` tag.

Motivation

Users MAY be awarded badges (but not limited to) in recognition, in gratitude, for participation, or in appreciation of a certain goal, task or cause.

Users MAY choose to decorate their profiles with badges for fame, notoriety, recognition, support, etc., from badge issuers they deem reputable.

Recommendations

Badge issuers MAY include some Proof of Work as per NIP-13 when minting Badge Definitions or Badge Awards to embed them with a combined energy cost, arguably making them more special and valuable for users that wish to collect them.

Clients MAY whitelist badge issuers (pubkeys) for the purpose of ensuring they retain a valuable/special factor for their users.

Badge image recommended aspect ratio is 1:1 with a high-res size of 1024x1024 pixels.

Badge thumbnail image recommended dimensions are: 512x512 (xl), 256x256 (l), 64x64 (m), 32x32 (s) and 16x16 (xs).

Clients MAY choose to render less badges than those specified by users in the Profile Badges event or replace the badge image and thumbnails with ones that fits the theme of the client.

Clients SHOULD attempt to render the most appropriate badge thumbnail according to the number of badges chosen by the user and space available. Clients SHOULD attempt render the high-res version on user action (click, tap, hover).

Example of a Badge Definition event

```
{
  "pubkey": "alice",
  "kind": 30009,
  "tags": [
    ["d", "bravery"],
    ["name", "Medal of Bravery"],
    ["description", "Awarded to users demonstrating bravery"],
    ["image", "https://nostr.academy/awards/bravery.png", "1024x1024"],
    ["thumb", "https://nostr.academy/awards/bravery_256x256.png", "256x256"],
  ],
  ...
}
```

Example of Badge Award event

```
{
  "id": "<badge award event id>",
  "kind": 8,
  "pubkey": "alice",
  "tags": [
    ["a", "30009:alice:bravery"],
    ["p", "bob", "wss://relay"],
    ["p", "charlie", "wss://relay"],
  ],
  ...
}
```

Example of a Profile Badges event

Honorable Bob The Brave:

```
{
  "kind": 30008,
  "pubkey": "bob",
  "tags": [
    ["d", "profile_badges"],
    ["a", "30009:alice:bravery"],
    ["e", "<bravery badge award event id>", "wss://nostr.academy"],
    ["a", "30009:alice:honor"],
    ["e", "<honor badge award event id>", "wss://nostr.academy"],
  ],
  ...
}
```

NIP-65

Relay List Metadata

`draft` `optional` `author:mikedilger`

A special replaceable event meaning "Relay List Metadata" is defined as an event with kind `10002` having a list of `r` tags, one for each relay the author uses to either read or write to.

The primary purpose of this relay list is to advertise to others, not for configuring one's client.

The content is not used and SHOULD be an empty string.

The `r` tags can have a second parameter as either `read` or `write`. If it is omitted, it means the author uses the relay for both purposes.

Clients SHOULD, as with all replaceable events, use only the most recent kind-10002 event they can find.

The meaning of read and write

Write relays are for events that are intended for anybody (e.g. your followers). Read relays are for events that address a particular person.

Clients SHOULD write feed-related events created by their user to their user's write relays.

Clients SHOULD read feed-related events created by another from at least some of that other person's write relays. Explicitly, they SHOULD NOT expect them to be available at their user's read relays. It SHOULD NOT be presumed that the user's read relays coincide with the write relays of the people the user follows.

Clients SHOULD read events that tag their user from their user's read relays.

Clients SHOULD write events that tag a person to at least some of that person's read relays. Explicitly, they SHOULD NOT expect that person will pick them up from their user's write relays. It SHOULD NOT be presumed that the user's write relays coincide with the read relays of the person being tagged.

Clients SHOULD presume that if their user has a pubkey in their ContactList (kind 3) that it is because they wish to see that author's feed-related events. But clients MAY presume otherwise.

Motivation

There is a common nostr use case where users wish to follow the content produced by other users. This is evidenced by the implicit meaning of the Contact List in NIP-02

Because users don't often share the same sets of relays, ad-hoc solutions have arisen to get that content, but these solutions negatively impact scalability and decentralization:

- Most people are sending their posts to the same most popular relays in order to be more widely seen
- Many people are pulling from a large number of relays (including many duplicate events) in order to get more data
- Events are being copied between relays, oftentimes to many different relays

Purposes

The purpose of this NIP is to help clients find the events of the people they follow, to help tagged events get to the people tagged, and to help nostr scale better.

Suggestions

It is suggested that people spread their kind `10002` events to many relays, but write their normal feed-related events to a much smaller number of relays (between 2 to 6 relays). It is suggested that clients offer a way for users to spread their kind `10002` events to many more relays than they normally post to.

Authors may post events outside of the feed that they wish their followers to follow by posting them to relays outside of those listed in their "Relay List Metadata". For example, an author may want to reply to someone without all of their followers watching.

It is suggested that relays allow any user to write their own kind `10002` event (optionally with AUTH to verify it is their own) even if they are not otherwise subscribed to the relay because

- finding where someone posts is rather important
- these events do not have content that needs management
- relays only need to store one replaceable event per pubkey to offer this service

Why not in kind `0` Metadata

Even though this is user related metadata, it is a separate event from kind `0` in order to keep it small (as it should be widely spread) and to not have content that may require moderation by relay operators so that it is more acceptable to relays.

Example

```
{
  "kind": 10002,
  "tags": [
    ["r", "wss://alicerelay.example.com"],
    ["r", "wss://brando-relay.com"],
    ["r", "wss://expensive-relay.example2.com", "write"],
    ["r", "wss://nostr-relay.example.com", "read"],
  ],
  "content": "",
  ...other fields
```