NIP-01

Basic protocol flow description

fiatjaf and the contributors to the nips repository

github.com/nostr-protocol/nips · 01.md

01

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 between 0 and 65535>,
  "tags": [
    [<arbitrary string>...],
    // ...
  ],
  "content": <arbitrary string>,
  "sig": <64-bytes lowercase hex of the signature of the sha256 hash of the seri-
   alized 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 (which is described below) 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>
]
```

To prevent implementation differences from creating a different event ID for the same event, the following rules MUST be followed while serializing:

- UTF-8 should be used for encoding.
- Whitespace, line breaks or other unnecessary formatting should not be included in the output JSON.
- The following characters in the content field must be escaped as shown, and all other characters must be included verbatim:
  - A line break (`0x0A`), use `\n`
  - A double quote (`0x22`), use `\"`
  - A backslash (`0x5C`), use `\\`
  - A carriage return (`0x0D`), use `\r`
  - A tab character (`0x09`), use `\t`
  - A backspace, (`0x08`), use `\b`
  - A form feed, (`0x0C`), use `\f`

Tags

Each tag is an array of one or more strings, with some conventions around them. Take a look at the example below:

```
{
  "tags": [
    ["e", "5c83da77af1dec6d7289834998ad7aafbd9e2191396d75ec3cc27f5a77226f36",
   "wss://nostr.example.com"],
    ["p", "f7234bd4c1394dda46d09f35bd384dd30cc552ad5541990f98844fb06676e9ca"],
    ["a",
   "30023:f7234bd4c1394dda46d09f35bd384dd30cc552ad5541990f98844fb06676e9ca:abcd",
   "wss://nostr.example.com"],
    ["alt", "reply"],
    // ...
  ],
  // ...
}
```

The first element of the tag array is referred to as the tag *name* or *key* and the second as the tag *value*. So we can safely say that the event above has an `e` tag set to `"5c83da77af1dec6d7289834998ad7aafbd9e2191396d75ec3cc27f5a77226f36"`, an `alt` tag set to `"reply"` and so on. All elements after the second do not have a conventional name.

This NIP defines 3 standard tags that can be used across all event kinds with the same meaning. They are as follows:

- The `e` tag, used to refer to an event: `["e", <32-bytes lowercase hex of the id of another event>, <recommended relay URL, optional>, <32-bytes lowercase hex of the author's pubkey, optional>]`
- The `p` tag, used to refer to another user: `["p", <32-bytes lowercase hex of a pubkey>, <recommended relay URL, optional>]`
- The `a` tag, used to refer to an addressable or replaceable event
  - for an addressable event: `["a", "<kind integer>:<32-bytes lowercase hex of a pubkey>:<d tag value>", <recommended relay URL, optional>]`
  - for a normal replaceable event: `["a", "<kind integer>:<32-bytes lowercase hex of a pubkey>:", <recommended relay URL, optional>]` (note: include the trailing colon)

As a convention, all single-letter (only english alphabet letters: a-z, A-Z) key tags are expected to be indexed by relays, such that it is possible, for example, to query or subscribe to events that reference the event `"5c83da77af1dec6d7289834998ad7aafbd9e2191396d75ec3cc27f5a77226f36"` by using the `{"#e": ["5c83da77af1dec6d7289834998ad7aafbd9e2191396d75ec3cc27f5a77226f36"]}` filter. Only the first value in any given tag is indexed.

Kinds

Kinds specify how clients should interpret the meaning of each event and the other fields of each event (e.g. an `"r"` tag may have a meaning in an event of kind 1 and an entirely different meaning in an event of kind 10002). Each NIP may define the meaning of a set of kinds that weren't defined elsewhere. NIP-10, for instance, specifies the `kind:1` text note for social media applications.

This NIP defines one basic kind:

- `0`: **user metadata**: the `content` is set to a stringified JSON object `{name: <nickname or full name>, about: <short bio>, picture: <url of the image>}` describing the user who created the event. Extra metadata fields may be set. A relay may delete older events once it gets a new one for the same pubkey.

And also a convention for kind ranges that allow for easier experimentation and flexibility of relay implementation:

- for kind `n` such that `1000 <= n < 10000 || 4 <= n < 45 || n == 1 || n == 2`, events are **regular**, which means they're all expected to be stored by relays.
- for kind `n` such that `10000 <= n < 20000 || n == 0 || n == 3`, events are **replaceable**, which means that, for each combination of `pubkey` and `kind`, only the latest event MUST be stored by relays, older versions MAY be discarded.
- for kind `n` such that `20000 <= n < 30000`, events are **ephemeral**, which means they are not expected to be stored by relays.
- for kind `n` such that `30000 <= n < 40000`, events are **addressable** by their `kind`, `pubkey` and `d` tag value -- which means that, for each combination of `kind`, `pubkey` and the `d` tag value, only the latest event MUST be stored by relays, older versions MAY be discarded.

In case of replaceable events with the same timestamp, the event with the lowest id (first in lexical order) should be retained, and the other discarded.

When answering to `REQ` messages for replaceable events such as `{"kinds":[0],"authors":[<hex-key>]}`, even if the relay has more than one version stored, it SHOULD return just the latest one.

These are just conventions and relay implementations may differ.

Communication between clients and relays

Relays expose a websocket endpoint to which clients can connect. Clients SHOULD open a single websocket connection to each relay and use it for all their subscriptions. Relays MAY limit number of connections from specific IP/client/etc.

Relays MUST only accept connections to a single endpoint when additional path segments do not influence its behavior.

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>, <filters1>, <filters2>, ...]`, 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. It represents a subscription per connection. Relays MUST manage `<subscription_id>`s independently for each WebSocket connection. `<subscription_id>`s are not guaranteed to be globally unique.

`<filtersX>` 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>,
  "authors": <a list of lowercase pubkeys, the pubkey of an event must be one of
   these>,
  "kinds": <a list of a kind numbers>,
  "#<single-letter (a-zA-Z)>": <a list of tag values, for #e — a list of event
   ids, for #p — a list of pubkeys, etc.>,
  "since": <an integer unix timestamp in seconds. Events must have a created_at >=
   to this to pass>,
  "until": <an integer unix timestamp in seconds. Events must have a created_at <=
   to this to pass>,
  "limit": <maximum number of events relays SHOULD return in the initial query>
}
```

Upon receiving a `REQ` message, the relay SHOULD return events that match the filter. Any new events it receives SHOULD be sent to that same websocket until the connection is closed, a `CLOSE` event is received with the same `<subscription_id>`, or a new `REQ` is sent using the same `<subscription_id>` (in which case a new subscription is created, replacing the old one).

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

The `ids`, `authors`, `#e` and `#p` filter lists MUST contain exact 64-character lowercase hex values.

The `since` and `until` properties can be used to specify the time range of events returned in the subscription. If a filter includes the `since` property, events with `created_at` greater than or equal to `since` are considered to match the filter. The `until` property is similar except that `created_at` must be less than or equal to `until`. In short, an event matches a filter if `since <= created_at <= until` holds.

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 MUST be ignored afterwards. When `limit: n` is present it is assumed that the events returned in the initial query will be the last `n` events ordered by the `created_at`. Newer events should appear first, and in the case of ties the event with the lowest id (first in lexical order) should be first. Relays SHOULD use the `limit` value to guide how many events are returned in the initial response. Returning fewer events is acceptable, but returning (much) more should be avoided to prevent overwhelming clients.

When `limit` is zero, the relay MUST NOT return stored events for that filter. After the initial queries for all filters are complete, the relay MUST send `EOSE` and MUST keep the subscription active for newly received matching events.

From relay to client: sending events and notices

Relays can send 5 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.
- `["OK", <event_id>, <true|false>, <message>]`, used to indicate acceptance or denial of an `EVENT` message.
- `["EOSE", <subscription_id>]`, used to indicate the *end of stored events* and the beginning of events newly received in real-time.
- `["CLOSED", <subscription_id>, <message>]`, used to indicate that a subscription was ended on the server side.
- `["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).
- `OK` messages MUST be sent in response to `EVENT` messages received from clients, they must have the 3rd parameter set to `true` when an event has been accepted by the relay, `false` otherwise. The 4th parameter MUST always be present, but MAY be an empty string when the 3rd is `true`, otherwise it MUST be a string formed by a machine-readable single-word prefix followed by a `:` and then a human-readable message. Some examples:
  - `["OK", "b1a649ebe8...", true, ""]`
  - `["OK", "b1a649ebe8...", true, "pow: difficulty 25>=24"]`
  - `["OK", "b1a649ebe8...", true, "duplicate: already have this event"]`
  - `["OK", "b1a649ebe8...", false, "blocked: you are banned from posting here"]`
  - `["OK", "b1a649ebe8...", false, "blocked: please register your pubkey at https://my-expensive-relay.example.com"]`
  - `["OK", "b1a649ebe8...", false, "rate-limited: slow down there chief"]`
  - `["OK", "b1a649ebe8...", false, "invalid: event creation date is too far off from the current time"]`
  - `["OK", "b1a649ebe8...", false, "pow: difficulty 26 is less than 30"]`
  - `["OK", "b1a649ebe8...", false, "restricted: not allowed to write."]`
  - `["OK", "b1a649ebe8...", false, "error: could not connect to the database"]`
  - `["OK", "b1a649ebe8...", false, "mute: no one was listening to your ephemeral event and it wasn't handled in any way, it was ignored"]`
- `CLOSED` messages MUST be sent in response to a `REQ` when the relay refuses to fulfill it. It can also be sent when a relay decides to kill a subscription on its side before a client has disconnected or sent a `CLOSE`. This message uses the same pattern of `OK` messages with the machine-readable prefix and human-readable message. Some examples:
  - `["CLOSED", "sub1", "unsupported: filter contains unknown elements"]`
  - `["CLOSED", "sub1", "error: could not connect to the database"]`
  - `["CLOSED", "sub1", "error: shutting down idle subscription"]`
- The standardized machine-readable prefixes for `OK` and `CLOSED` are: `duplicate`, `pow`, `blocked`, `rate-limited`, `invalid`, `restricted`, `mute` and `error` for when none of that fits.

All NIPs are public domain.