

Developing a Client



- [NATS Client Development Guide](#)
 - [Client connection options](#)
 - [Client authorization](#)
 - [Verbose mode](#)
 - [Pedantic mode](#)
 - [Ping/pong interval](#)
 - [Parsing the protocol](#)
 - [Deciding on a parsing strategy](#)
 - [Storing and dispatching subscription callbacks](#)
 - [Implementing request/response](#)
 - [Error handling, disconnecting and reconnecting](#)
 - [Cluster support](#)
-

NATS Client Development Guide

This guide provides you with considerations for developing NATS clients, including:

- CONNECT handling
- Authorization
- Verbose (acks)
- Pedantic mode
- Ping/pong interval
- Parsing the protocol
- Deciding on a parsing strategy
- Storing and dispatching subscription callbacks
- Implementing requests/response
- Error handling, disconnecting and reconnecting
- Cluster support

Probably the best way to learn about implementing a client is to look at one of the client's maintained by the Synadia team. These clients are generally full featured, so if you can use them, that is even better, but if you have to write a client these may go beyond your needs while still capturing many of the design considerations discussed here.

- [GoLang](#)
- [Java](#)
- [C# / .NET](#)
- [Rust](#)
- [JavaScript](#)
- [Python](#)
- [C](#)

See [Developing with NATS](#) for more links and community supported languages.

Client connection options

Clients can connect in authenticated or unauthenticated mode, as well as verbose mode which enables acknowledgements. See the [protocol documentation](#) for details.

Client authorization

By default clients can connect to the server in unauthenticated mode. You can configure the NATS server to require password authentication to connect.

For example, using the command line:

```
nats-server -DV -m 8222 -user foo -pass bar
```

The client must then authenticate to connect to the server. For example:

```
nats.Connect("nats://foo:bar@localhost:4222")
```

Verbose mode

When 'verbose' is enabled (via the `CONNECT` message), the NATS server will return `+OK` to acknowledge receipt of a valid protocol message. The NATS server automatically runs in verbose mode. Most client implementations disable verbose mode (set it to `false` in the `CONNECT` message) for performance reasons.

Pedantic mode

A client may also support 'pedantic' mode. Pedantic mode indicates to the server that strict protocol enforcement is required.

Ping/pong interval

NATS implements auto-pruning. When a client connects to the server, the server expects that client to be active. Periodically, the NATS server pings each subscriber, expecting a reply. If there is no reply within the configurable time limit, the server disconnects the client.

Parsing the protocol

NATS provides a text-based message format. The text-based [protocol](#) makes it easy to implement NATS clients. The key consideration is deciding on a parsing strategy.

The NATS server implements a [zero allocation byte parser](#) that is fast and efficient. Off the wire, a NATS message is simply a slice of bytes. Across the wire the message is transported as an immutable string over a TCP connection. It is up to the client to implement logic to parse the message.

The NATS message structure includes the Subject string, an optional Reply string, and an optional Data field that is a byte array. The type `Msg` is a structure used by Subscribers and PublishMsg().

```
type Msg struct {
    Subject string
    Reply   string
    Data    []byte
    Sub     *Subscription
}
```

A NATS publisher publishes the data argument to the given subject. The data argument is left untouched and needs to be correctly interpreted on the receiver. How the client parses a NATS message depends on the programming language.

Deciding on a parsing strategy

Generally, protocol parsing for a NATS client is a string operation. In Python, for example, string operations are faster than regex. The Go and Java clients also use string operations to parse the message. But, if you look at the Ruby client, regex is used to parse the protocol because in Ruby regex is faster than string operations.

In sum, there is no magic formula for parsing—it depends on the programming language. But, you need to take into consideration how you are going to parse the message when you write a client.

Storing and dispatching subscription callbacks

When you make a subscription to the server, you need to store and dispatch callback handlers.

On the client side, you need a hash map for this data structure. The hash map will be storing the callback that maps the subscription ID to the subscription.

The key of the hash map is the subscription ID. The key is used to look up the callback in the hash map. When you process the NATS message off the wire, you pass the parameters subject, reply subject, and the payload to the callback handler, which does its work.

Thus, you must store the mapping of subscription ID to the callback. Inside the subscription you have the callback.

Implementing request/response

When to use pub/sub vs. req/rep depends on your use case. Run the tutorials for each to understand the differences between each style of implementation.

Error handling, disconnecting and reconnecting

Considerations for error handling primarily include handling client disconnections and implementing retry logic.

Cluster support

The NATS client has reconnection logic. So, if you are implementing clustering, you need to implement reconnect callbacks a priori, meaning you cannot modify it during runtime. When you start it, you need to have that information already.

👤 #3

📄 cocobook-sync 👤 2026-10-28 11:14:59 UTC

📄 cocobook-sync 👤 2026-10-29 01:42:41 UTC