

Client Protocol



- [Client Protocol](#)
- [Protocol conventions](#)
- [Protocol messages](#)
- [INFO](#)
 - [Description](#)
 - [Syntax](#)
 - [connect_urls](#)
 - [Example](#)
- [CONNECT](#)
 - [Description](#)
 - [Syntax](#)
 - [Example](#)
- [PUB](#)
 - [Description](#)
 - [Syntax](#)
 - [Example](#)
- [HPUB](#)
 - [Description](#)
 - [Syntax](#)
 - [Example](#)
- [SUB](#)
 - [Description](#)
 - [Syntax](#)
 - [Example](#)
- [UNSUB](#)
 - [Description](#)
 - [Syntax](#)
 - [Example](#)

- [MSG](#)
 - [Description](#)
 - [Syntax](#)
 - [Example](#)
 - [HMSG](#)
 - [Description](#)
 - [Syntax](#)
 - [Example](#)
 - [PING/PONG](#)
 - [Description](#)
 - [Syntax](#)
 - [Example](#)
 - [+OK/ERR](#)
 - [Description](#)
 - [Syntax](#)
-

Client Protocol

The wire protocol used to communicate between the NATS server and clients is a simple, text-based publish/subscribe style protocol. Clients connect to and communicate with `nats-server` (the NATS server) through a regular TCP/IP socket using a small set of protocol operations that are terminated by a new line.

Unlike traditional messaging systems that use a binary message format that require an API to consume, the text-based NATS protocol makes it easy to implement clients in a wide variety of programming and scripting languages. In fact, refer to the topic [NATS Protocol Demo](#) to play with the NATS protocol for yourself using telnet.

The NATS server implements a [zero allocation byte parser](#) that is fast and efficient.

Protocol conventions

Control Line with Optional Content: Each interaction between the client and server consists of a control, or protocol, line of text followed, optionally by message content. Most of the protocol

messages don't require content, only `PUB`, `MSG`, `HPUB`, and `HMSG` include payloads.

Field Delimiters: The fields of NATS protocol messages are delimited by whitespace characters (space) or (tab). Multiple whitespace characters will be treated as a single field delimiter.

Newlines: NATS uses followed by (, `0x0D0A`) to terminate protocol messages. This newline sequence is also used to mark the end of the message payload in `PUB`, `MSG`, `HPUB`, and `HMSG` protocol messages.

Subject names: Subject names, including reply subject names, are case-sensitive and must be non-empty alphanumeric strings with no embedded whitespace. All UTF-8 characters except spaces/tabs and separators which are `.` and `>` are allowed. Subject names can be optionally token-delimited using the dot character (`.`), e.g.:

`FOO`, `BAR`, `foo.bar`, `foo.BAR`, `FOO.BAR` and `FOO.BAR.BAZ` are all valid subject names

`FOO. BAR`, `foo. .bar` and `foo..bar` are *not* valid subject names

A subject is comprised of 1 or more tokens. Tokens are separated by `.` and can be any non whitespace UTF-8 character. The full wildcard token `>` is only valid as the last token and matches all tokens past that point. A token wildcard, `*` matches any token in the position it was listed. Wildcard tokens should only be used in a wildcard capacity and not part of a literal token.

Character Encoding: Subject names should be UTF-8 compatible.

Wildcards: NATS supports the use of wildcards in subject subscriptions.

- The asterisk character (`*`) matches a single token at any level of the subject.
- The greater than symbol (`>`), also known as the *full wildcard*, matches one or more tokens at the tail of a subject, and must be the last token. The wildcarded subject `foo.>` will match `foo.bar` or `foo.bar.baz.1`, but not `foo`.
- Wildcards must be a separate token (`foo*.baz` or `foo.>` are syntactically valid; `foo*.bar`, `f*o.b*r` and `foo>` are not)

For example, the wildcard subscriptions `foo*.quux` and `foo.>` both match `foo.bar.quux`, but only the latter matches `foo.bar.baz`. With the full wildcard, it is also possible to express interest in every subject that may exist in NATS: `sub > 1`, limited of course by authorization settings.

Protocol messages

The following table briefly describes the NATS protocol messages. NATS protocol operation names are case insensitive, thus `SUB foo 1` and `sub foo 1` are equivalent.

Click the name to see more detailed information, including syntax:

OP Name	Sent By	Description
<code>INFO</code>	Server	Sent to client after initial TCP/IP connection
<code>CONNECT</code>	Client	Sent to server to specify connection information
<code>PUB</code>	Client	Publish a message to a subject, with optional reply subject
<code>HPUB</code>	Client	Publish a message to a subject including NATS headers, with optional reply subject
<code>SUB</code>	Client	Subscribe to a subject (or subject wildcard)
<code>UNSUB</code>	Client	Unsubscribe (or auto-unsubscribe) from subject
<code>MSG</code>	Server	Delivers a message payload to a subscriber
<code>HMSG</code>	Server	Delivers a message payload to a subscriber with NATS headers
<code>PING</code>	Both	PING keep-alive message
<code>PONG</code>	Both	PONG keep-alive response
<code>+OK</code>	Server	Acknowledges well-formed protocol message in <code>verbose</code> mode
<code>-ERR</code>	Server	Indicates a protocol error. May cause client disconnect.

The following sections explain each protocol message.

INFO

Description

A client will need to start as a plain TCP connection, then when the server accepts a connection from the client, it will send information about itself, the configuration and security requirements necessary for the client to successfully authenticate with the server and exchange messages.

When using the updated client protocol (see `CONNECT` below), `INFO` messages can be sent anytime by the server. This means clients with that protocol level need to be able to asynchronously handle `INFO` messages.

Syntax

```
INFO {"option_name":option_value,...}
```

The valid options are as follows, encoded as JSON:

name	description	type	presence
server_id	The unique identifier of the NATS server.	string	always
server_name	The name of the NATS server.	string	always
version	The version of NATS.	string	always
go	The version of golang the NATS server was built with.	string	always
host	The IP address used to start the NATS server, by default this will be 0.0.0.0 and can be configured with - client_advertise host:port.	string	always
port	The port number the NATS server is configured to listen on.	int	always
headers	Whether the server supports headers.	bool	always
max_payload	Maximum payload size, in bytes, that the server will accept from the client.	int	always
proto	An integer indicating the protocol version of the server. The server version 1.2.0 sets this to 1 to indicate that it supports the "Echo" feature.	int	always
client_id	The internal client identifier in the server. This can be used to filter client connections in monitoring, correlate with error logs, etc...	uint64	optional
auth_required	If this is true, then the client should try to authenticate upon connect.	bool	optional

name	description	type	presence
<code>tls_required</code>	If this is true, then the client must perform the TLS/1.2 handshake. Note, this used to be <code>ssl_required</code> and has been updated along with the protocol from SSL to TLS.	bool	optional
<code>tls_verify</code>	If this is true, the client must provide a valid certificate during the TLS handshake.	bool	optional
<code>tls_available</code>	If this is true, the client can provide a valid certificate during the TLS handshake.	bool	optional
<code>connect_urls</code>	List of server urls that a client can connect to.	[string]	optional
<code>ws_connect_urls</code>	List of server urls that a websocket client can connect to.	[string]	optional
<code>ldm</code>	If the server supports <i>Lame Duck Mode</i> notifications, and the current server has transitioned to lame duck, <code>ldm</code> will be set to <code>true</code> .	bool	optional
<code>git_commit</code>	The git hash at which the NATS server was built.	string	optional
<code>jetstream</code>	Whether the server supports JetStream.	bool	optional
<code>ip</code>	The IP of the server.	string	optional
<code>client_ip</code>	The IP of the client.	string	optional
<code>nonce</code>	The nonce for use in CONNECT.	string	optional
<code>cluster</code>	The name of the cluster.	string	optional
<code>domain</code>	The configured NATS domain of the server.	string	optional

connect_urls

The `connect_urls` field is a list of urls the server may send when a client first connects, and when there are changes to server cluster topology. This field is considered optional, and may be omitted based on server configuration and client protocol level.

When a NATS server cluster expands, an `INFO` message is sent to the client with an updated `connect_urls` list. This cloud-friendly feature asynchronously notifies a client of known servers, allowing it to connect to servers not originally configured.

The `connect_urls` will contain a list of strings with an IP and port, looking like this:

```
"connect_urls":["10.0.0.184:4333","192.168.129.1:4333","192.168.192.1:4333"]
```

Example

Below you can see a sample connection string from a telnet connection to the `demo.nats.io` site.

```
telnet demo.nats.io 4222
```

```
Trying 107.170.221.32...
Connected to demo.nats.io.
Escape character is '^['.
INFO
{"server_id":"Zk0GQ3JBsrg3oyxCRRlE09","version":"1.2.0","proto":1,"go":"go1.10.3","host":"0.0.0.0","port":4222,"max_payload":1048576,"client_id":2392}
```

CONNECT

Description

The `CONNECT` message is the client version of the `INFO` message. Once the client has established a TCP/IP socket connection with the NATS server, and an `INFO` message has been received from the server, the client may send a `CONNECT` message to the NATS server to provide more information about the current connection as well as security information.

Syntax

```
CONNECT {"option_name":option_value,...}[]
```

The valid options are as follows, encoded as JSON:

name	description	type	required
<code>verbose</code>	Turns on <code>+OK</code> protocol acknowledgements.	bool	true

name	description	type	required
pedantic	Turns on additional strict format checking, e.g. for properly formed subjects.	bool	true
tls_required	Indicates whether the client requires an SSL connection.	bool	true
auth_token	Client authorization token.	string	if auth_required is true
user	Connection username.	string	if auth_required is true
pass	Connection password.	string	if auth_required is true
name	Client name.	string	false
lang	The implementation language of the client.	string	true
version	The version of the client.	string	true
protocol	Sending 0 (or absent) indicates client supports original protocol. Sending 1 indicates that the client supports dynamic reconfiguration of cluster topology changes by asynchronously receiving INFO messages with known servers it can reconnect to.	int	false
echo	If set to false, the server (version 1.2.0+) will not send originating messages from this connection to its own subscriptions. Clients should set this to false only for server supporting this feature, which is when proto in the INFO protocol is set to at least 1.	bool	false
sig	In case the server has responded with a nonce on INFO, then a NATS client must use this field to reply with the signed nonce.	string	if nonce received
jwt	The JWT that identifies a user permissions and account.	string	false
no_responders	Enable quick replies for cases where a request is sent to a topic with no responders .	bool	false

name	description	type	required
headers	Whether the client supports headers.	bool	false
nkey	The public NKey to authenticate the client. This will be used to verify the signature (<code>sig</code>) against the <code>nonce</code> provided in the <code>INFO</code> message.	string	false

Example

Here is an example from the default string of the Go client:

```
CONNECT
{"verbose":false,"pedantic":false,"tls_required":false,"name":"","lang":"go","version":"1.2.2",
"protocol":1}
```

Most clients set `verbose` to `false` by default. This means that the server should not confirm each message it receives on this connection with a `+OK` back to the client.

PUB

Description

The `PUB` message publishes the message payload to the given subject name, optionally supplying a reply subject. If a reply subject is supplied, it will be delivered to eligible subscribers along with the supplied payload. Note that the payload itself is optional. To omit the payload, set the payload size to 0, but the second CRLF is still required.

Syntax

```
PUB <subject> [reply-to] <#bytes>[payload]
```

where:

name	description	type	required
subject	The destination subject to publish to.	string	true

name	description	type	required
reply-to	The reply subject that subscribers can use to send a response back to the publisher/requestor.	string	false
#bytes	The payload size in bytes.	int	true
payload	The message payload data.	string	false

Example

To publish the ASCII string message payload "Hello NATS!" to subject FOO:

```
PUB FOO 11Hello NATS!
```

To publish a request message "Knock Knock" to subject FRONT.DOOR with reply subject JOKE.22:

```
PUB FRONT.DOOR JOKE.22 11Knock Knock
```

To publish an empty message to subject NOTIFY:

```
PUB NOTIFY 0
```

HPUB

Description

The `HPUB` message is the same as `PUB` but extends the message payload to include NATS headers. Note that the payload itself is optional. To omit the payload, set the total message size equal to the size of the headers. Note that the trailing CR+LF is still required.

NATS headers are similar, in structure and semantics, to HTTP headers as `name: value` pairs including supporting multi-value headers. Headers can be mixed case and NATS will preserve case between message publisher and message receiver(s). See also [ADR-4 NATS Message Headers](#).

Syntax

```
HPUB <subject> [reply-to] <#header bytes> <#total bytes>[headers][payload]
```

where:

name	description	type	required
subject	The destination subject to publish to.	string	true
reply-to	The reply subject that subscribers can use to send a response back to the publisher/requestor.	string	false
#header bytes	The size of the headers section in bytes including the <code>----</code> delimiter before the payload.	int	true
#total bytes	The total size of headers and payload sections in bytes.	int	true
headers	Header version <code>NATS/1.0</code> followed by one or more <code>name: value</code> pairs, each separated by <code>.</code>	string	false
payload	The message payload data.	string	false

Example

To publish the ASCII string message payload "Hello NATS!" to subject FOO with one header Bar with value Baz:

```
HPUB F00 22 33NATS/1.0Bar: BazHello NATS!
```

To publish a request message "Knock Knock" to subject FRONT.DOOR with reply subject JOKE.22 and two headers:

```
HPUB FRONT.DOOR JOKE.22 45 56NATS/1.0BREAKFAST: donutLUNCH: burgerKnock Knock
```

To publish an empty message to subject NOTIFY with one header Bar with value Baz:

```
HPUB NOTIFY 22 22NATS/1.0Bar: Baz
```

To publish a message to subject MORNING MENU with one header BREAKFAST having two values and payload "Yum!"

```
HPUB MORNING.MENU 47 51NATS/1.0BREAKFAST: donutBREAKFAST: eggsYum!
```

SUB

Description

`SUB` initiates a subscription to a subject, optionally joining a distributed queue group.

Syntax

```
SUB <subject> [queue group] <sid>[]
```

where:

name	description	type	required
subject	The subject name to subscribe to.	string	true
queue group	If specified, the subscriber will join this queue group.	string	false
sid	A unique alphanumeric subscription ID, generated by the client.	string	true

Example

To subscribe to the subject `F00` with the connection-unique subscription identifier (sid) `1`:

```
SUB F00 1[]
```

To subscribe the current connection to the subject `BAR` as part of distribution queue group `G1` with sid `44`:

```
SUB BAR G1 44[]
```

UNSUB

Description

`UNSUB` unsubscribes the connection from the specified subject, or auto-unsubscribes after the specified number of messages has been received.

Syntax

```
UNSUB <sid> [max_msgs] []
```

where:

name	description	type	required
sid	The unique alphanumeric subscription ID of the subject to unsubscribe from.	string	true
max_msgs	A number of messages to wait for before automatically unsubscribing.	int	false

Example

The following examples concern subject `F00` which has been assigned sid `1`. To unsubscribe from `F00`:

```
UNSUB 1 []
```

To auto-unsubscribe from `F00` after 5 messages have been received:

```
UNSUB 1 5 []
```

MSG

Description

The `MSG` protocol message is used to deliver an application message to the client.

Syntax

```
MSG <subject> <sid> [reply-to] <#bytes> [] [payload] []
```

where:

name	description	type	presence
subject	Subject name this message was received on.	string	always

name	description	type	presence
sid	The unique alphanumeric subscription ID of the subject.	string	always
reply-to	The subject on which the publisher is listening for responses.	string	optional
#bytes	Size of the payload in bytes.	int	always
payload	The message payload data.	string	optional

Example

The following message delivers an application message from subject `FOO.BAR`:

```
MSG FOO.BAR 9 11Hello World
```

To deliver the same message along with a reply subject:

```
MSG FOO.BAR 9 GREETING.34 11Hello World
```

HMSG

Description

The `HMSG` message is the same as `MSG`, but extends the message payload with headers. See also [ADR-4 NATS Message Headers](#).

Syntax

```
HMSG <subject> <sid> [reply-to] <#header bytes> <#total bytes>[headers][payload]
```

where:

name	description	type	presence
subject	Subject name this message was received on.	string	always

name	description	type	presence
<code>sid</code>	The unique alphanumeric subscription ID of the subject.	string	always
<code>reply-to</code>	The subject on which the publisher is listening for responses.	string	optional
<code>#header bytes</code>	The size of the headers section in bytes including the <code>----</code> delimiter before the payload.	int	always
<code>#total bytes</code>	The total size of headers and payload sections in bytes.	int	always
<code>headers</code>	Header version <code>NATS/1.0</code> followed by one or more <code>name: value</code> pairs, each separated by <code>.</code>	string	optional
<code>payload</code>	The message payload data.	string	optional

Example

The following message delivers an application message from subject `F00.BAR` with a header:

```
HMSG F00.BAR 34 45NATS/1.0FoodGroup: vegetableHello World
```

To deliver the same message along with a reply subject:

```
HMSG F00.BAR 9 BAZ.69 34 45NATS/1.0FoodGroup: vegetableHello World
```

PING/PONG

Description

`PING` and `PONG` implement a simple keep-alive mechanism between client and server. Once a client establishes a connection to the NATS server, the server will continuously send `PING` messages to the client at a configurable interval. If the client fails to respond with a `PONG` message within the configured response interval, the server will terminate its connection. If your connection stays idle for too long, it is cut off.

If the server sends a ping request, you can reply with a pong message to notify the server that you are still interested. You can also ping the server and will receive a pong reply. The ping/pong interval is configurable.

The server uses normal traffic as a ping/pong proxy, so a client that has messages flowing may not receive a ping from the server.

Syntax

PING

PONG

Example

The following example shows the demo server pinging the client and finally shutting it down.

```
telnet demo.nats.io 4222

Trying 107.170.221.32...
Connected to demo.nats.io.
Escape character is '^]'.
INFO
{"server_id":"Zk0GQ3JBsrg3oyxCRRlE09","version":"1.2.0","proto":1,"go":"go1.10.3","host":"0.0.0.0","port":4222,"max_payload":1048576,"client_id":2392}
PING
PING
-ERR 'Stale Connection'
Connection closed by foreign host.
```

+OK/ERR

Description

When the `verbose` connection option is set to `true` (the default value), the server acknowledges each well-formed protocol message from the client with a `+OK` message. Most NATS clients set the `verbose` option to `false` using the `CONNECT` message

The `-ERR` message is used by the server indicate a protocol, authorization, or other runtime connection error to the client. Most of these errors result in the server closing the connection.

Handling of these errors usually has to be done asynchronously.

Syntax

`+OK`

`-ERR <error message>`

Some protocol errors result in the server closing the connection. Upon receiving these errors, the connection is no longer valid and the client should clean up relevant resources. These errors include:

- `-ERR 'Unknown Protocol Operation'`: Unknown protocol error
- `-ERR 'Attempted To Connect To Route Port'`: Client attempted to connect to a route port instead of the client port
- `-ERR 'Authorization Violation'`: Client failed to authenticate to the server with credentials specified in the `CONNECT` message
- `-ERR 'Authorization Timeout'`: Client took too long to authenticate to the server after establishing a connection (default 1 second)
- `-ERR 'Invalid Client Protocol'`: Client specified an invalid protocol version in the `CONNECT` message
- `-ERR 'Maximum Control Line Exceeded'`: Message destination subject and reply subject length exceeded the maximum control line value specified by the `max_control_line` server option. The default is 1024 bytes.
- `-ERR 'Parser Error'`: Cannot parse the protocol message sent by the client
- `-ERR 'Secure Connection - TLS Required'`: The server requires TLS and the client does not have TLS enabled.
- `-ERR 'Stale Connection'`: The server hasn't received a message from the client, including a `PONG` in too long.
- `-ERR 'Maximum Connections Exceeded'`: This error is sent by the server when creating a new connection and the server has exceeded the maximum number of connections specified by the `max_connections` server option. The default is 64k.
- `-ERR 'Slow Consumer'`: The server pending data size for the connection has reached the maximum size (default 10MB).
- `-ERR 'Maximum Payload Violation'`: Client attempted to publish a message with a payload size that exceeds the `max_payload` size configured on the server. This value is supplied to the client upon connection in the initial `INFO` message. The client is expected to do proper accounting of byte size to be sent to the server in order to handle this error synchronously.

Protocol error messages where the connection remains open are listed below. The client should not close the connection in these cases.

- -ERR 'Invalid Subject': Client sent a malformed subject (e.g. `sub foo. 90`)
 - -ERR 'Permissions Violation for Subscription to <subject>': The user specified in the `CONNECT` message does not have permission to subscribe to the subject.
 - -ERR 'Permissions Violation for Publish to <subject>': The user specified in the `CONNECT` message does not have permissions to publish to the subject.
-

#3

□ cocobook-sync ### 2026-10-28 11:14:57 UTC

□ cocobook-sync ### 2026-10-29 01:42:41 UTC