

nats bench



- [Prerequisites](#)
 - [Start the NATS server with monitoring enabled](#)
 - [Run a publisher throughput test](#)
 - [Run a publish/subscribe throughput test](#)
 - [Run a 1:N throughput test](#)
 - [Run a N:M throughput test](#)
 - [Run a request-reply latency test](#)
 - [Run JetStream benchmarks](#)
 - [Measure JetStream publication performance](#)
 - [Measure JetStream consumption \(replay\) performance](#)
 - [Measuring publication and consumption together](#)
 - [Measure KV performance](#)
 - [Play around with the knobs](#)
 - [Leave no trace: clean up the resources when you are finished](#)
-

NATS is fast and lightweight, and places a priority on performance. The `nats` CLI tool can, amongst many other things, be used for running benchmarks and measuring performance of your target NATS service infrastructure. In this tutorial, you learn how to benchmark and tune NATS on your systems and environment.

Note: the numbers below are just examples and were obtained using a MacBook Pro M4 (November 2024) running version 2.12.1 of `nats-server`:

```
Model Name: MacBook Pro
Model Identifier: Mac16,1
Model Number: MW2U3LL/A
Chip: Apple M4
Total Number of Cores: 10 (4 performance and 6 efficiency)
Memory: 16 GB
System Firmware Version: 13822.1.2
```

Prerequisites

- [Install the NATS CLI Tool](#)
- [Install the NATS server](#)

Start the NATS server with monitoring enabled

```
nats-server -m 8222 -js
```

Verify that the NATS server starts successfully, as well as the HTTP monitor:

```
[[2932] 2025/10/28 12:29:02.879297 [INF] Starting nats-server
[2932] 2025/10/28 12:29:02.879658 [INF]   Version:  2.12.1
[2932] 2025/10/28 12:29:02.879661 [INF]   Git:      [fab5f99]
[2932] 2025/10/28 12:29:02.879664 [INF]   Name:
NBIYCV5UNYPP2ZBZJZNGQ7UJNJILSQZCD6MK2CPWU6UY7PHYPKW0YYS4
[2932] 2025/10/28 12:29:02.879667 [INF]   Node:      YNleYaHo
[2932] 2025/10/28 12:29:02.879668 [INF]   ID:
NBIYCV5UNYPP2ZBZJZNGQ7UJNJILSQZCD6MK2CPWU6UY7PHYPKW0YYS4
[2932] 2025/10/28 12:29:02.880586 [INF] Starting http monitor on 0.0.0.0:8222
[2932] 2025/10/28 12:29:02.880696 [INF] Starting JetStream
[2932] 2025/10/28 12:29:02.880755 [WRN] Temporary storage directory used, data could be lost
on system reboot
[2932] 2025/10/28 12:29:02.881014 [INF]
[2932] 2025/10/28 12:29:02.881018 [INF]
[2932] 2025/10/28 12:29:02.881019 [INF]
[2932] 2025/10/28 12:29:02.881020 [INF]
[2932] 2025/10/28 12:29:02.881020 [INF]
[2932] 2025/10/28 12:29:02.881021 [INF] https://docs.nats.io/jetstream
[2932] 2025/10/28 12:29:02.881022 [INF]
[2932] 2025/10/28 12:29:02.881022 [INF] ----- JETSTREAM -----
```

```
[2932] 2025/10/28 12:29:02.881023 [INF] Strict: true
[2932] 2025/10/28 12:29:02.881026 [INF] Max Memory: 12.00 GB
[2932] 2025/10/28 12:29:02.881027 [INF] Max Storage: 233.86 GB
[2932] 2025/10/28 12:29:02.881027 [INF] Store Directory:
"/var/folders/cx/x13pjm0n3ds6w4q_4xhr_c0r0000gn/T/nats/jetstream"
[2932] 2025/10/28 12:29:02.881029 [INF] API Level: 2
[2932] 2025/10/28 12:29:02.881030 [INF] -----
[2932] 2025/10/28 12:29:02.881335 [INF] Listening for client connections on 0.0.0.0:4222
[2932] 2025/10/28 12:29:02.881434 [INF] Server is ready
```

Run a publisher throughput test

Let's run a first test to see how fast a single publisher can publish one million 16 byte messages to the NATS server. This should yield very high numbers as there is no subscriber on the subject being used.

```
nats bench pub foo --size 16 --msgs 1000000
```

The output tells you the number of messages and the number of payload bytes that the client was able to publish per second:

```
12:45:18 Starting Core NATS publisher benchmark [clients=1, msg-size=16 B, msgs=1,000,000,
multi-subject=false, multi-subject-max=100,000, sleep=0s, subject=foo]
12:45:18 [1] Starting Core NATS publisher, publishing 1,000,000 messages
Finished      0s [=====] 100%

NATS Core NATS publisher stats: 14,786,683 msgs/sec ~ 226 MiB/sec ~ 0.07us
```

Run a publish/subscribe throughput test

While the measurement above is an interesting data point, it is purely an academic measurement as you will usually have one (or more) subscribers for the messages being published.

Let's look at throughput for a single publisher with a single subscriber. For this, we need to run two instances of `nats bench` at the same time (e.g. in two shell windows), one to subscribe and one to publish.

First start the subscriber (it doesn't start measuring until it receives the first message from the publisher).

```
nats bench sub foo --size 16 --msgs 1000000
```

Then start the publisher.

```
nats bench pub foo --size 16 --msgs 1000000
```

Publisher's output:

```
13:15:53 Starting Core NATS publisher benchmark [clients=1, msg-size=16 B, msgs=1,000,000,
multi-subject=false, multi-subject-max=100,000, sleep=0s, subject=foo]
13:15:53 [1] Starting Core NATS publisher, publishing 1,000,000 messages
Finished      0s [=====] 100%

NATS Core NATS publisher stats: 4,925,767 msgs/sec ~ 75 MiB/sec ~ 0.20us
```

Subscriber's output:

```
13:15:50 Starting Core NATS subscriber benchmark [clients=1, msg-size=16 B, msgs=1,000,000,
multi-subject=false, subject=foo]
13:15:50 [1] Starting Core NATS subscriber, expecting 1,000,000 messages
Finished      0s [=====] 100%

NATS Core NATS subscriber stats: 4,928,153 msgs/sec ~ 75 MiB/sec ~ 0.20us
```

We can also increase the size of the messages using `--size`, for example:

Publisher:

```
nats bench pub foo --size 16kb
```

```
13:20:18 Starting Core NATS publisher benchmark [clients=1, msg-size=16 KiB, msgs=100,000,
multi-subject=false, multi-subject-max=100,000, sleep=0s, subject=foo]
13:20:18 [1] Starting Core NATS publisher, publishing 100,000 messages
Finished      0s [=====] 100%

NATS Core NATS publisher stats: 230,800 msgs/sec ~ 3.5 GiB/sec ~ 4.33us
```

Subscriber:

```
nats bench sub foo --size 16kb
```

```
13:20:15 Starting Core NATS subscriber benchmark [clients=1, msg-size=16 KiB, msgs=100,000,
multi-subject=false, subject=foo]
```

```
13:20:15 [1] Starting Core NATS subscriber, expecting 100,000 messages
```

```
Finished      0s [=====] 100%
```

```
NATS Core NATS subscriber stats: 226,091 msgs/sec ~ 3.4 GiB/sec ~ 4.42us
```

As expected, while the number of messages per second decreases with the larger messages, the throughput, however, increases massively.

Run a 1:N throughput test

You can also measure performance with a message fan-out where multiple subscribers receive a copy of the message. You can do this using the `--client` flag, each client being a Go-routine, making it's own connection to the server and subscribing to the subject.

When specifying multiple clients `nats bench` will also report aggregated statistics.

For example for a fan-out of 4:

```
nats bench sub foo --clients 4
```

and

```
nats bench pub foo
```

Publisher's output:

```
13:34:26 Starting Core NATS publisher benchmark [clients=1, msg-size=128 B, msgs=100,000,
multi-subject=false, multi-subject-max=100,000, sleep=0s, subject=foo]
```

```
13:34:26 [1] Starting Core NATS publisher, publishing 100,000 messages
```

```
Finished      0s [=====] 100%
```

```
NATS Core NATS publisher stats: 1,012,200 msgs/sec ~ 124 MiB/sec ~ 0.99us
```

Subscribers' output:

```
13:34:24 Starting Core NATS subscriber benchmark [clients=4, msg-size=128 B, msgs=100,000, multi-subject=false, subject=foo]
```

```
13:34:24 [1] Starting Core NATS subscriber, expecting 100,000 messages
```

```
13:34:24 [2] Starting Core NATS subscriber, expecting 100,000 messages
```

```
13:34:24 [3] Starting Core NATS subscriber, expecting 100,000 messages
```

```
13:34:24 [4] Starting Core NATS subscriber, expecting 100,000 messages
```

```
Finished      0s [=====] 100%
```

```
Finished      0s [=====] 100%
```

```
Finished      0s [=====] 100%
```

```
Finished      0s [=====] 100%
```

```
[1] 1,013,938 msgs/sec ~ 124 MiB/sec ~ 0.99us (100,000 msgs)
```

```
[2] 1,014,120 msgs/sec ~ 124 MiB/sec ~ 0.99us (100,000 msgs)
```

```
[3] 1,007,242 msgs/sec ~ 123 MiB/sec ~ 0.99us (100,000 msgs)
```

```
[4] 1,004,311 msgs/sec ~ 123 MiB/sec ~ 1.00us (100,000 msgs)
```

```
NATS Core NATS subscriber aggregated stats: 4,015,923 msgs/sec ~ 490 MiB/sec
```

```
message rates min 1,004,311 | avg 1,009,902 | max 1,014,120 | stddev 4,254 msgs
```

```
avg latencies min 0.99us | avg 0.99us | max 1.00us | stddev 0.00us
```

Run a N:M throughput test

When more than 1 publisher client is specified, `nats bench` evenly distributes the total number of messages (`--msgs`) across the number of publishers (`--clients`).

So let's increase the number of publishers and also increase the number of messages so the benchmark run lasts a little bit longer:

Subscriber:

```
nats bench sub foo --clients 4 --msgs 1000000
```

Publisher:

```
nats bench pub foo --clients 4 --msgs 1000000
```

Publisher's output

```
13:40:24 Starting Core NATS publisher benchmark [clients=4, msg-size=128 B, msgs=1,000,000,
multi-subject=false, multi-subject-max=100,000, sleep=0s, subject=foo]
13:40:24 [1] Starting Core NATS publisher, publishing 250,000 messages
13:40:24 [2] Starting Core NATS publisher, publishing 250,000 messages
13:40:24 [3] Starting Core NATS publisher, publishing 250,000 messages
13:40:24 [4] Starting Core NATS publisher, publishing 250,000 messages
Finished      0s [=====] 100%
Finished      0s [=====] 100%
Finished      0s [=====] 100%
Finished      0s [=====] 100%

[1] 272,785 msgs/sec ~ 33 MiB/sec ~ 3.67us (250,000 msgs)
[2] 271,251 msgs/sec ~ 33 MiB/sec ~ 3.69us (250,000 msgs)
[3] 270,340 msgs/sec ~ 33 MiB/sec ~ 3.70us (250,000 msgs)
[4] 270,040 msgs/sec ~ 33 MiB/sec ~ 3.70us (250,000 msgs)

NATS Core NATS publisher aggregated stats: 1,080,144 msgs/sec ~ 132 MiB/sec
message rates min 270,040 | avg 271,104 | max 272,785 | stddev 1,068 msgs
avg latencies min 3.67us | avg 3.69us | max 3.70us | stddev 0.01us
```

Subscriber's output:

```
13:40:18 Starting Core NATS subscriber benchmark [clients=4, msg-size=128 B, msgs=1,000,000,
multi-subject=false, subject=foo]
13:40:18 [1] Starting Core NATS subscriber, expecting 1,000,000 messages
13:40:18 [2] Starting Core NATS subscriber, expecting 1,000,000 messages
13:40:18 [3] Starting Core NATS subscriber, expecting 1,000,000 messages
13:40:18 [4] Starting Core NATS subscriber, expecting 1,000,000 messages
Finished      0s [=====] 100%
Finished      0s [=====] 100%
Finished      0s [=====] 100%
Finished      0s [=====] 100%

[1] 1,080,830 msgs/sec ~ 132 MiB/sec ~ 0.93us (1,000,000 msgs)
[2] 1,080,869 msgs/sec ~ 132 MiB/sec ~ 0.93us (1,000,000 msgs)
[3] 1,080,849 msgs/sec ~ 132 MiB/sec ~ 0.93us (1,000,000 msgs)
[4] 1,080,821 msgs/sec ~ 132 MiB/sec ~ 0.93us (1,000,000 msgs)

NATS Core NATS subscriber aggregated stats: 4,323,201 msgs/sec ~ 528 MiB/sec
```

```
message rates min 1,080,821 | avg 1,080,842 | max 1,080,869 | stddev 18 msgs  
avg latencies min 0.93us | avg 0.93us | max 0.93us | stddev 0.00us
```

Run a request-reply latency test

You can also test request/reply performance using `nats bench service`.

In one shell start a nats bench to act as a server and let it run:

```
nats bench service serve foo
```

And in another shell send some requests (each request is sent synchronously, one after the other):

```
nats bench service request foo
```

```
13:46:43 Starting Core NATS service requester benchmark [clients=1, msg-size=128 B,  
msgs=100,000, sleep=0s, subject=foo]  
13:46:43 [1] Starting Core NATS service requester, requesting 100,000 messages  
Finished      5s [=====] 100%  
  
NATS Core NATS service requester stats: 19,659 msgs/sec ~ 2.4 MiB/sec ~ 50.87us
```

In this case, the average latency of request-reply between the two `nats bench` processes over NATS was 50.87 micro-seconds. However, since those requests are made synchronously, we can not measure throughput this way. We need to generate a lot more load by having more than one client making those synchronous requests at the same time, and we will also run more than one service instance (as you would in production) such that the requests are load-balanced between the service instances using the queue group functionality.

Start the service instances and leave running:

```
nats bench service serve foo --size 16 --clients 2
```

Clients making requests (since we are using a lot of clients to generate load, we will not show the progress bar while running the benchmark):

```
nats bench service request foo --size 16 --clients 50 --no-progress
```

```
13:57:56 Starting Core NATS service requester benchmark [clients=50, msg-size=16 B,  
msgs=100,000, sleep=0s, subject=foo]
```

```
13:57:56 [1] Starting Core NATS service requester, requesting 2,000 messages
13:57:56 [2] Starting Core NATS service requester, requesting 2,000 messages
...
13:57:56 [49] Starting Core NATS service requester, requesting 2,000 messages
13:57:56 [50] Starting Core NATS service requester, requesting 2,000 messages
```

```
[1] 2,735 msgs/sec ~ 43 KiB/sec ~ 365.62us (2,000 msgs)
[2] 2,700 msgs/sec ~ 42 KiB/sec ~ 370.24us (2,000 msgs)
...
[49] 2,651 msgs/sec ~ 41 KiB/sec ~ 377.14us (2,000 msgs)
[50] 2,649 msgs/sec ~ 41 KiB/sec ~ 377.48us (2,000 msgs)
```

```
NATS Core NATS service requester aggregated stats: 132,438 msgs/sec ~ 2.0 MiB/sec
message rates min 2,649 | avg 2,673 | max 2,735 | stddev 17 msgs
avg latencies min 365.62us | avg 373.93us | max 377.48us | stddev 2.43us
```

Run JetStream benchmarks

You can measure JetStream performance using the `nats bench js` commands.

Measure JetStream publication performance

You can measure the performance of publishing (storing) messages into a stream using `nats bench js pub`, which offers 3 options:

- `nats bench js pub sync` publishes the messages synchronously one after the other (so while it's good for measuring latency, it's not good to measure throughput).
- `nats bench js pub async` publishes a batch of messages asynchronously, waits for all the publications' acknowledgements and moves on to the next batch (which is a good way to measure throughput).
- `nats bench js pub batch` uses the atomic batch publish (while batching is currently implemented only to provide atomicity, it has the side effect of potentially helping throughout, especially for smaller messages).

• `nats bench js pub` will by default use a stream called `benchstream`, and `--create` will automatically create the stream if it doesn't exist yet. Also you can use `--purge` to clear the stream first. You can specify stream attributes like `--replicas 3` or `--storage memory`, or `--maxbytes` or operate on any existing stream with `--stream`.

For example, test latency of publishing to a memory stream:

```
nats bench js pub sync jsfoo --size 16 --create --storage memory
```

```
18:47:47 Starting JetStream synchronous publisher benchmark [batch=0, clients=1, dedup-
window=2m0s, deduplication=false, max-bytes=1,073,741,824, msg-size=16 B, msgs=100,000, multi-
subject=false, multi-subject-max=100,000, purge=false, replicas=1, sleep=0s, storage=memory,
stream=benchstream, subject=jsfoo]
18:47:47 Using stream: benchstream
18:47:47 [1] Starting JetStream synchronous publisher, publishing 100,000 messages
Publishing    2s [=====] 100%

NATS JetStream synchronous publisher stats: 35,734 msgs/sec ~ 558 KiB/sec ~ 27.98us
```

Test throughput using batch publishing:

```
nats bench js pub batch jsfoo --size 16 --batch 1000 --purge --storage memory
```

```
18:51:27 Starting JetStream batched publisher benchmark [batch=1,000, clients=1, msg-size=16
B, msgs=100,000, multi-subject=false, multi-subject-max=100,000, purge=true, sleep=0s,
stream=benchstream, subject=jsfoo]
18:51:27 Using stream: benchstream
18:51:27 Purging the stream
18:51:27 [1] Starting JetStream batched publisher, publishing 100,000 messages
Finished      0s [=====] 100%

NATS JetStream batched publisher stats: 627,430 msgs/sec ~ 9.6 MiB/sec ~ 1.59us
```

Remove the stream and test to file storage (which is the default)

```
nats stream rm -f benchstream
nats bench js pub async jsfoo --create
```

```
13:09:34 Starting JetStream asynchronous publisher benchmark [batch=500, clients=1, dedup-
window=2m0s, deduplication=false, max-bytes=1,073,741,824, msg-size=128 B, msgs=100,000,
multi-subject=false, multi-subject-max=100,000, purge=false, replicas=1, sleep=0s,
storage=file, stream=benchstream, subject=jsfoo]
13:09:34 Using stream: benchstream
13:09:34 [1] Starting JetStream asynchronous publisher, publishing 100,000 messages
Finished      0s [=====] 100%
```

```
NATS JetStream asynchronous publisher stats: 403,828 msgs/sec ~ 49 MiB/sec ~ 2.48us
```

You can even measure publish performance to an `--replicas 1` stream with asynchronous persistence using `--persistasync` which yields throughput similar to when using memory storage, as by default JetStream flushes disk writes synchronously, meaning that even if the `nats-server` process is killed suddenly no messages will be lost as the OS already has them in it's buffer and will flush them to disk (it can be also configured to not just flush but also sync after every write in which case no message will be lost even if the whole host goes down suddenly, at the expense of latency obviously)).

Measure JetStream consumption (replay) performance

Once you have stored some messages on a stream you can measure the replay performance in multiple ways:

- `nats bench js ordered` uses an ordered *ephemeral* consumer to receive the messages (so each client gets its own copy of the messages).
- `nats bench js consume` uses the `Consume()` (callback) function on a *durable* consumer to receive the messages.
- `nats bench js fetch` uses the `Fetch()` function on a *durable* consumer to receive messages in batches.
- `nats bench js get` gets the messages directly by sequence number (either synchronously one by one or using 'batched gets') *without using a consumer*.

Starting with ordered consumer:

```
nats bench js ordered
```

```
13:33:48 Starting JetStream ordered ephemeral consumer benchmark [clients=1, msg-size=128 B,
msgs=100,000, purge=false, sleep=0s, stream=benchstream]
```

```
13:33:48 [1] Starting JetStream ordered ephemeral consumer, expecting 100,000 messages
```

```
Finished      0s [=====] 100%
```

```
NATS JetStream ordered ephemeral consumer stats: 1,201,540 msgs/sec ~ 147 MiB/sec ~ 0.83us
```

Then using consume to distribute consumption of messages between multiple clients through a durable consumer with explicit acknowledgements:

```
nats bench js consume --clients 4 --no-progress
```

```
13:46:04 Starting JetStream durable consumer (callback) benchmark [acks=explicit, batch=500,
clients=4, consumer=nats-bench, double-acked=false, msg-size=128 B, msgs=100,000, purge=false,
sleep=0s, stream=benchstream]
```

```
13:46:04 [1] Starting JetStream durable consumer (callback), expecting 25,000 messages
```

```
13:46:04 [2] Starting JetStream durable consumer (callback), expecting 25,000 messages
```

```
13:46:04 [3] Starting JetStream durable consumer (callback), expecting 25,000 messages
```

```
13:46:04 [4] Starting JetStream durable consumer (callback), expecting 25,000 messages
```

```
[1] 73,230 msgs/sec ~ 8.9 MiB/sec ~ 13.66us (25,000 msgs)
```

```
[2] 72,921 msgs/sec ~ 8.9 MiB/sec ~ 13.71us (25,000 msgs)
```

```
[3] 72,696 msgs/sec ~ 8.9 MiB/sec ~ 13.76us (25,000 msgs)
```

```
[4] 72,687 msgs/sec ~ 8.9 MiB/sec ~ 13.76us (25,000 msgs)
```

```
NATS JetStream durable consumer (callback) aggregated stats: 290,438 msgs/sec ~ 36 MiB/sec
message rates min 72,687 | avg 72,883 | max 73,230 | stddev 220 msgs
avg latencies min 13.66us | avg 13.72us | max 13.76us | stddev 0.04us
```

Using fetch with two clients to retrieve batches of 400 messages through a durable consumer and without explicit acknowledgements:

```
nats bench js fetch --acks none --clients 2
```

```
14:09:10 Starting JetStream durable consumer (fetch) benchmark [acks=none, batch=500,
clients=2, consumer=nats-bench, double-acked=false, msg-size=128 B, msgs=100,000, purge=false,
sleep=0s, stream=benchstream]
```

```
14:09:10 [1] Starting JetStream durable consumer (fetch), expecting 50,000 messages
```

```
14:09:10 [2] Starting JetStream durable consumer (fetch), expecting 50,000 messages
```

```
Finished      0s [=====] 100%
```

```
Finished      0s [=====] 100%
```

```
[1] 567,330 msgs/sec ~ 69 MiB/sec ~ 1.76us (50,000 msgs)
```

```
[2] 567,067 msgs/sec ~ 69 MiB/sec ~ 1.76us (50,000 msgs)
```

```
NATS JetStream durable consumer (fetch) aggregated stats: 1,128,932 msgs/sec ~ 138 MiB/sec
message rates min 567,067 | avg 567,198 | max 567,330 | stddev 131 msgs
avg latencies min 1.76us | avg 1.76us | max 1.76us | stddev 0.00us
```

Measuring the latency of direct synchronous gets:

```
nats bench js get sync
```

```
14:13:30 Starting JetStream synchronous getter benchmark [clients=1, msg-size=128 B,
msgs=100,000, sleep=0s, stream=benchstream]
14:13:30 [1] Starting JetStream synchronous getter, expecting 100,000 messages
Finished      3s [=====] 100%

NATS JetStream synchronous getter stats: 33,244 msgs/sec ~ 4.1 MiB/sec ~ 30.08us
```

And finally measuring throughput using batched gets with a fan out of 2:

```
nats bench js get batch --clients 2
```

```
14:11:09 Starting JetStream batched direct getter benchmark [batch=500, clients=2, filter=>,
msg-size=128 B, msgs=100,000, sleep=0s, stream=benchstream]
14:11:09 [1] Starting JetStream batched direct getter, expecting 100,000 messages
14:11:09 [2] Starting JetStream batched direct getter, expecting 100,000 messages
Finished      0s [=====] 100%
Finished      0s [=====] 100%

[1] 509,387 msgs/sec ~ 62 MiB/sec ~ 1.96us (100,000 msgs)
[2] 500,449 msgs/sec ~ 61 MiB/sec ~ 2.00us (100,000 msgs)

NATS JetStream batched direct getter aggregated stats: 1,000,898 msgs/sec ~ 122 MiB/sec
message rates min 500,449 | avg 504,918 | max 509,387 | stddev 4,469 msgs
avg latencies min 1.96us | avg 1.98us | max 2.00us | stddev 0.02us
```

Measuring publication and consumption together

While measuring publication and consumption to and from a stream separately yields interesting metrics, during normal operations most of the time the consumers are going to be on-line and consuming while the messages are being published to the stream.

First purge the stream and start the consuming instance of `nats bench`, for example using an ordered consumer and 8 clients (so a fan out of 8):

```
nats bench js ordered --purge --clients 8 --no-progress
```

Then start publishing to the stream, for example using 8 clients doing asynchronous publications:

```
nats bench js pub async jsfoo --clients 8 --no-progress
```

```
15:23:08 Starting JetStream asynchronous publisher benchmark [batch=500, clients=8, msg-size=128 B, msgs=100,000, multi-subject=false, multi-subject-max=100,000, purge=false, sleep=0s, stream=benchstream, subject=jsfoo]
```

```
15:23:08 Using stream: benchstream
```

```
15:23:08 [1] Starting JetStream asynchronous publisher, publishing 12,500 messages
```

```
15:23:08 [2] Starting JetStream asynchronous publisher, publishing 12,500 messages
```

```
...
```

```
15:23:08 [7] Starting JetStream asynchronous publisher, publishing 12,500 messages
```

```
15:23:08 [8] Starting JetStream asynchronous publisher, publishing 12,500 messages
```

```
[1] 33,289 msgs/sec ~ 4.1 MiB/sec ~ 30.04us (12,500 msgs)
```

```
[2] 33,242 msgs/sec ~ 4.1 MiB/sec ~ 30.08us (12,500 msgs)
```

```
...
```

```
[7] 31,947 msgs/sec ~ 3.9 MiB/sec ~ 31.30us (12,500 msgs)
```

```
[8] 31,586 msgs/sec ~ 3.9 MiB/sec ~ 31.66us (12,500 msgs)
```

```
NATS JetStream asynchronous publisher aggregated stats: 252,544 msgs/sec ~ 31 MiB/sec
```

```
message rates min 31,586 | avg 32,614 | max 33,289 | stddev 638 msgs
```

```
avg latencies min 30.04us | avg 30.67us | max 31.66us | stddev 0.60us
```

Consumer's output:

```
15:23:02 Starting JetStream ordered ephemeral consumer benchmark [clients=8, msg-size=128 B, msgs=100,000, purge=true, sleep=0s, stream=benchstream]
```

```
15:23:02 [1] Starting JetStream ordered ephemeral consumer, expecting 100,000 messages
```

```
15:23:02 [2] Starting JetStream ordered ephemeral consumer, expecting 100,000 messages
```

```
...
```

```
15:23:02 [7] Starting JetStream ordered ephemeral consumer, expecting 100,000 messages
```

```
15:23:02 [8] Starting JetStream ordered ephemeral consumer, expecting 100,000 messages
```

```
[1] 111,627 msgs/sec ~ 14 MiB/sec ~ 8.96us (100,000 msgs)
```

```
[2] 110,534 msgs/sec ~ 14 MiB/sec ~ 9.05us (100,000 msgs)
```

```
...
```

```
[7] 109,849 msgs/sec ~ 13 MiB/sec ~ 9.10us (100,000 msgs)
```

```
[8] 109,797 msgs/sec ~ 13 MiB/sec ~ 9.11us (100,000 msgs)
```

```
NATS JetStream ordered ephemeral consumer aggregated stats: 878,326 msgs/sec ~ 107 MiB/sec
```

```
message rates min 109,797 | avg 110,306 | max 111,627 | stddev 556 msgs  
avg latencies min 8.96us | avg 9.07us | max 9.11us | stddev 0.05us
```

Measure KV performance

`nats bench kv` can be used to measure Key Value performance using synchronous put and get operations.

First put some data in the KV:

```
nats bench kv put
```

```
14:26:04 Starting JetStream KV putter benchmark [bucket=benchbucket, clients=1, msg-size=128  
B, msgs=100,000, purge=false, sleep=0s]  
14:26:04 [1] Starting JetStream KV putter, publishing 100,000 messages  
Putting          3s [=====] 100%  
  
NATS JetStream KV putter stats: 30,067 msgs/sec ~ 3.7 MiB/sec ~ 33.26us
```

Then simulate a bunch of clients doing gets on random keys:

```
nats bench kv get --clients 16 --randomize 100000 --no-progress
```

```
14:28:33 Starting JetStream KV getter benchmark [bucket=benchbucket, clients=16, msg-size=128  
B, msgs=100,000, randomize=100,000, sleep=0s]  
14:28:33 [1] Starting JetStream KV getter, trying to get 6,250 messages  
14:28:33 [2] Starting JetStream KV getter, trying to get 6,250 messages  
...  
14:28:33 [15] Starting JetStream KV getter, trying to get 6,250 messages  
14:28:33 [16] Starting JetStream KV getter, trying to get 6,250 messages  
  
[1] 6,568 msgs/sec ~ 821 KiB/sec ~ 152.23us (6,250 msgs)  
[2] 6,579 msgs/sec ~ 822 KiB/sec ~ 151.98us (6,250 msgs)  
...  
[15] 6,474 msgs/sec ~ 809 KiB/sec ~ 154.45us (6,250 msgs)  
[16] 6,451 msgs/sec ~ 806 KiB/sec ~ 155.01us (6,250 msgs)  
  
NATS JetStream KV getter aggregated stats: 102,844 msgs/sec ~ 13 MiB/sec  
message rates min 6,448 | avg 6,509 | max 6,579 | stddev 40 msgs
```

Play around with the knobs

Don't be afraid to test different JetStream storage and replication options (assuming you have access to a JetStream enabled cluster of servers if you want to go beyond `--replicas 1`), and of course the number of publishing/subscribing clients, and the batch and message sizes.

You can also use `nats bench` as a tool to generate traffic at a steady rate by using the `--sleep` flag to introduce a delay between the publication of each message (or batch of messages). You can also use that same flag to simulate processing time when consuming messages.

Note: If you change the attributes of a stream between runs you will have to delete the stream (e.g. run `nats stream rm benchstream`)

Leave no trace: clean up the resources when you are finished

Once you have finished benchmarking streams, remember that if you have stored many messages in the stream (which is very easy and fast to do) your stream may end up using a certain amount of resources on the nats-server(s) infrastructure (i.e. memory and files) that you may want to reclaim.

You can instruct use the `--purge` bench command flag to tell `nats` to purge the stream of messages before starting its benchmark, or purge the stream manually using `nats stream purge benchstream` or just delete it altogether using `nats stream rm benchstream`.

📄 #3

📄 cocobook-sync 📄 2026-10-28 11:10:45 UTC

📄 cocobook-sync 📄 2026-10-29 01:42:35 UTC