

Slow Consumers



- [Slow consumers identified in the client](#)
 - [Slow consumers identified by the server](#)
 - [Handling slow consumers](#)
 - [Server Configuration](#)
 - [Client Configuration](#)
-

To support resiliency and high availability, NATS provides built-in mechanisms to automatically prune the registered listener interest graph that is used to keep track of subscribers, including slow consumers and lazy listeners. NATS automatically handles a slow consumer. If a client is not processing messages quick enough, the NATS server cuts it off. To support scaling, NATS provides for auto-pruning of client connections. If a subscriber does not respond to ping requests from the server within the [ping-pong interval](#), the client is cut off (disconnected). The client will need to have reconnect logic to reconnect with the server.

In core NATS, consumers that cannot keep up are handled differently from many other messaging systems: NATS favors the approach of protecting the system as a whole over accommodating a particular consumer to ensure message delivery.

What is a slow consumer?

A slow consumer is a subscriber that cannot keep up with the message flow delivered from the NATS server. This is a common case in distributed systems because it is often easier to generate data than it is to process it. When consumers cannot process data fast enough, back pressure is applied to the rest of the system. NATS has mechanisms to reduce this back pressure.

NATS identifies slow consumers in the client or the server, providing notification through registered callbacks, log messages, and statistics in the server's monitoring endpoints.

What happens to slow consumers?

When detected at the client, the application is notified and messages are dropped to allow the consumer to continue and reduce potential back pressure. When detected in the server, the server will disconnect the connection with the slow consumer to protect itself and the integrity of the messaging system.

Slow consumers identified in the client

A [client can detect it is a slow consumer](#) on a local connection and notify the application through use of the asynchronous error callback. It is better to catch a slow consumer locally in the client rather than to allow the server to detect this condition. This example demonstrates how to define and register an asynchronous error handler that will handle slow consumer errors.

```
func natsErrorHandler(nc *nats.Conn, sub *nats.Subscription, natsErr error) {
    fmt.Printf("error: %v\n", natsErr)
    if natsErr == nats.ErrSlowConsumer {
        pendingMsgs, _, err := sub.Pending()
        if err != nil {
            fmt.Printf("couldn't get pending messages: %v", err)
            return
        }
        fmt.Printf("Falling behind with %d pending messages on subject %q.\n",
            pendingMsgs, sub.Subject)
        // Log error, notify operations...
    }
    // check for other errors
}

// Set the error handler when creating a connection.
nc, err := nats.Connect("nats://localhost:4222",
    nats.ErrorHandler(natsErrorHandler))
```

With this example code and default settings, a slow consumer error would generate output something like this:

```
error: nats: slow consumer, messages dropped
Falling behind with 65536 pending messages on subject "foo".
```

Note that if you are using a synchronous subscriber, `Subscription.NextMsg(timeout time.Duration)` will also return an error indicating there was a slow consumer and messages have been dropped.

Slow consumers identified by the server

When a client does not process messages fast enough, the server will buffer messages in the outbound connection to the client. When this happens and the server cannot write data fast enough to the client, in order to protect itself, it will designate a subscriber as a "slow consumer" and may drop the associated connection.

When the server initiates a slow consumer error, you'll see the following in the server output:

```
[54083] 2017/09/28 14:45:18.001357 [INF] ::1:63283 - cid:7 - Slow Consumer Detected
```

The server will also keep count of the number of slow consumer errors encountered, available through the monitoring `varz` endpoint in the `slow_consumers` field.

Handling slow consumers

Apart from using [JetStream](#) or optimizing your consuming application, there are a few options available: scale, meter, or tune NATS to your environment.

Scaling with queue subscribers

This is ideal if you do not rely on message order. Ensure your NATS subscription belongs to a [queue group](#), then scale as required by creating more instances of your service or application. This is a great approach for microservices - each instance of your microservice will receive a portion of the messages to process, and simply add more instances of your service to scale. No code changes, configuration changes, or downtime whatsoever.

Create a subject namespace that can scale

You can distribute work further through the subject namespace, with some forethought in design. This approach is useful if you need to preserve message order. The general idea is to publish to a deep subject namespace, and consume with wildcard subscriptions while giving yourself room to expand and distribute work in the future.

For a simple example, if you have a service that receives telemetry data from IoT devices located throughout a city, you can publish to a subject namespace like `Sensors.North`, `Sensors.South`, `Sensors.East` and `Sensors.West`. Initially, you'll subscribe to `Sensors.>` to process everything in one consumer. As your enterprise grows and data rates exceed what one consumer can handle, you

can replace your single consumer with four consuming applications to subscribe to each subject representing a smaller segment of your data. Note that your publishing applications remain untouched.

Meter the publisher

A less favorable option may be to meter the publisher. There are several ways to do this varying from simply slowing down your publisher to a more complex approach periodically issuing a blocking request-reply to match subscriber rates.

Tune NATS through configuration

The NATS server can be tuned to determine how much data can be buffered before a consumer is considered slow, and some officially supported clients allow buffer sizes to be adjusted. Decreasing buffer sizes will let you identify slow consumers more quickly. Increasing buffer sizes is not typically recommended unless you are handling temporary bursts of data. Often, increasing buffer capacity will only *postpone* slow consumer problems.

Server Configuration

The NATS server has a write deadline it uses to write to a connection. When this write deadline is exceeded, a client is considered to have a slow consumer. If you are encountering slow consumer errors in the server, you can increase the write deadline to buffer more data.

The `write_deadline` configuration option in the NATS server configuration file will tune this:

```
write_deadline: 2s
```

Tuning this parameter is ideal when you have bursts of data to accommodate. ***Be sure you are not just postponing a slow consumer error.***

Client Configuration

Most officially supported clients have an internal buffer of pending messages and will notify your application through an asynchronous error callback if a local subscription is not catching up. Receiving an error locally does not necessarily mean that the server will have identified a subscription as a slow consumer.

This buffer can be configured through setting the pending limits after a subscription has been created:

```
if err := sub.SetPendingLimits(1024*500, 1024*5000); err != nil {  
    log.Fatalf("Unable to set pending limits: %v", err)  
}
```

```
}
```

The default subscriber pending message limit is `65536`, and the default subscriber pending byte limit is `65536*1024`

If the client reaches this internal limit, it will drop messages and continue to process new messages. This is aligned with NATS at most once delivery. It is up to your application to detect the missing messages and recover from this condition.

❏ #3

❏ cocobook-sync ❏ 2026-10-28 11:14:48 UTC

❏ cocobook-sync ❏ 2026-10-29 01:42:40 UTC