



Java message service

🕒 Created	@March 2, 2022 11:06 AM
☑ Reviewed	<input type="checkbox"/>

Table of contents

[Introduction](#)

[Why We Need JMS?](#)

[Message Header](#)

[NSQ \(New simple queue](#)

[nsqd](#)

[nsqlookupd](#)

[Topology](#)

[nsqadmin](#)

[Purpose of ZMQ](#)

[Implementation](#)

[Explanation link](#)

[Key differences to conventional sockets](#)

[Bind vs Connect](#)

[Send and it's flags](#)

[Receive](#)

[Linger](#)

[High water level and socket types](#)

[Request reply pattern](#)

[Official doc](#)

Java Message Service

▼ Introduction

Java Message Service is an API that supports the formal communication called **messaging between computers** on a network. JMS provides a common interface for standard message protocols and message services in support of the Java programs.

JMS provides the **facility to create, send and read messages**. The JMS API reduces the concepts that a programmer must learn to use the messaging services/products and also provides the features that support the messaging applications.

JMS helps in building the **communication between two or more applications in a loosely coupled manner**. It means that the **applications which have to communicate are not connected directly they are connected through a common destination**.



Loose coupling means that the degree of dependency between two components is very low. Tight coupling means that the degree of dependency between two components is very high.

<https://www.youtube.com/watch?v=AMgU1wlMEEg>

<https://www.youtube.com/watch?v=IsAyTeUUXHk>

Starting activeMQ

OPEN → /home/khush/Downloads/apache-activemq-5.16.4-bin/apache-activemq-5.16.4

OPEN TERMINAL FROM SAME FOLDER → ./activemq start

OPEN CHROME → <http://localhost:8161/index.html>

It's alternative: Kafka

Kafka and JMS both are messaging system. Java message service is an api which are provided by Java. It is used for implementing messaging system in your application. JMS supports queue and publisher /subscriber(topic) messaging system. With queues, when first consumer consumes a message, message gets deleted from the queue and others cannot take it anymore. With topics, multiple consumers receive each message but it is much harder to scale.

Kafka is a generalization of these two concepts - it allows scaling between members of the same consumer group, but it also allows broadcasting the same message between many different consumer groups. Kafka also provides automatic rebalancing when new consumer join or left the consumer group.

Sr. No.	Key	Apache Kafka	JMS
1	Basic	Apache Kafka is a distributed publish-subscribe messaging system that receives data from disparate source systems and makes the data available to target systems in real time.	Java message service is an api which are provided by Java. It is used for implementing messaging system in your application.
2	Pull /Push Mechanism	It used pull mechanism, client need to poll for the message every time	It used push based model, message can be broadcast to all consumers
3	Message Retention Policy	It is policy based	Acknowledgment based
4	Auto Rebalancing	It provides autobalancing when new consumer add or remove from consumer group	It doesn't provide autorebalancing
5	Order of Messages	Kafka ensures that the messages are received in the order in which they were sent at the partition level	JMS does not support ordering message.

▼ Why We Need JMS?

▼ Message Header

The JMS message header contains the number of predefined fields which contain those values which are used by the clients and providers to identify and send messages. The predefined headers are:

– JMSDestination– JMSDeliveryMode– JMSMessageID– JMSTimestamp– JMSCorrelationID– JMSReplyTo– JMSRedelivered– JMSType– JMSExpiration– JMSPriority

▼ Message Properties

In message properties we can create and set properties for messages. The message properties are custom name value pairs which are set or read by applications. The message properties are useful for supporting filtering messages. The JMS API provides some predefined property that a provider can support. The message property is optional.

▼ Message Body

In message bodies the JMS API defines five message body formats which are also called as message types which allow us to send and receive data in many different forms and also provides compatibility with the existing messaging formats. It basically consists of the actual message sent from JMS sender to receiver.

The different message types are:

- **Text:** Represented by *javax.jms.TextMessage*. It is used to represent a block of text.
- **Object:** Represented by *javax.jms.ObjectMessage*. It is used to represent a java object.
- **Bytes:** Represented by *javax.jms.BytesMessage*. It is used to represent the binary data.
- **Stream:** Represented by *javax.jms.StreamMessage*. It is used to represent a list of java primitive values.
- **Map:** Represented by *javax.jms.MapMessage*. It is used to represent a set of keyword or value pairs

New Simple Queue

▼ NSQ (New simple queue)

- NSQ → set path. In ~/.bashrc inserting export "PATH=\$PATH:\$path_to_nsqd_folder/bin" and check using \$PATH and command nsqlookupd.
- Make maven project. Make .java file in java folder.
- Add dependency (brainlag nsq-client in the pom file)
- NSQ file are in https://nsq.io/clients/client_libraries.html. (javansqclient)
- Start nsqlookupd, nsqd, nsqadmin https://nsq.io/overview/quick_start.html
- Run the program

PPT: <https://speakerdeck.com/snakes/nsq-nyc-golang-meetup?slide=61>

▼ NSQ ports

- nsqlookupd
 - port **4160** tcp (used for communication)
Ports are numbered and used as global standards **to identify specific processes or types of network services**. Much like before shipping something to a foreign country, you'd agree where you'd be shipping out of and where you'd have it arriving, TCP ports allow for standardized communication between devices.
 - port 4161 http (HTML data)
It is the port from which a computer sends and receives Web client-based communication and messages from a Web server and is used to send and receive HTML pages or data.
- nsqd: connected to 4160
 - port **4150** tcp {producer is created on this port}
 - port 4151 http
- nsqadmin: connected to 4161
 - port 4171 http

▼ ActiveMQ vs NSQ

Developers describe **ActiveMQ** as "A message broker written in Java together with a full JMS client". Apache ActiveMQ is fast, supports many Cross Language Clients and Protocols, comes with easy to use Enterprise Integration Patterns and many advanced features while fully supporting JMS 1.1 and J2EE 1.4. Apache ActiveMQ is released under the Apache 2.0 License.

On the other hand, **NSQ** is detailed as "*A realtime distributed messaging platform*". NSQ is a realtime distributed messaging platform designed to operate at scale, handling billions of messages per day. It promotes distributed and decentralized topologies without single points of failure, enabling fault tolerance and high availability coupled with a reliable message delivery guarantee. See features & guarantees.

ActiveMQ and NSQ can be primarily classified as "**Message Queue**" tools.

ActiveMQ and NSQ are both open source tools.

▼ Purpose

It is designed to:

- Producer and consumer **need not to be on the same server**. They are connected through a common destination.
- support topologies that enable high-availability and eliminate SPOFs (single point of failure)
- address the need for stronger message delivery guarantees
- bound the memory footprint of a single process (by persisting some messages to disk)
- greatly simplify configuration requirements for producers and consumers
- provide a straightforward upgrade path
- improve efficiency

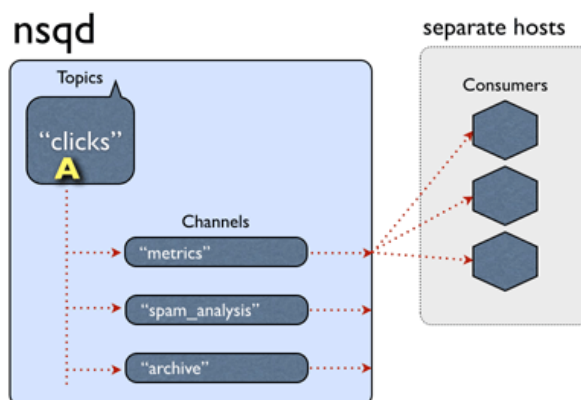
▼ Simplifying Configuration and Administration

A single `nsqd` instance is designed to handle multiple streams of data at once. Streams are called "topics" and a topic has 1 or more "channels". Each channel receives a *copy* of all the messages for a topic. In practice, a channel maps to a downstream service consuming a topic.

Topics and channels are *not* configured a priori. **Topics are created** on first use by publishing to the named topic or by subscribing to a channel on the named topic. Channels are created on first use by subscribing to the named channel.

Topics and channels all buffer data independently of each other, preventing a slow consumer from causing a backlog for other channels (the same applies at the topic level).

A channel can, and generally does, have multiple clients connected. Assuming all connected clients are in a state where they are ready to receive messages, each message will be delivered to a random client. For example:



To summarize, messages are multicast from topic -> channel (every channel receives a copy of all messages for that topic) but evenly distributed from channel -> consumers (each consumer receives a portion of the messages for that channel).

1 producer - 1 consumer: Sequentially

1 producer - 1 consumer opened after producing all msg: Not sequentially

1 producer - Multiple clients: Messages passed turn by turn randomly to **all**.

NSQ also includes a helper application, `nsqlookupd`, which provides a directory service where consumers can lookup the addresses of `nsqd` instances that provide the topics they are interested in subscribing to. In terms of configuration, this decouples the consumers from the producers (they both individually only need to know where to contact common instances of `nsqlookupd`, never each other), reducing complexity and maintenance.

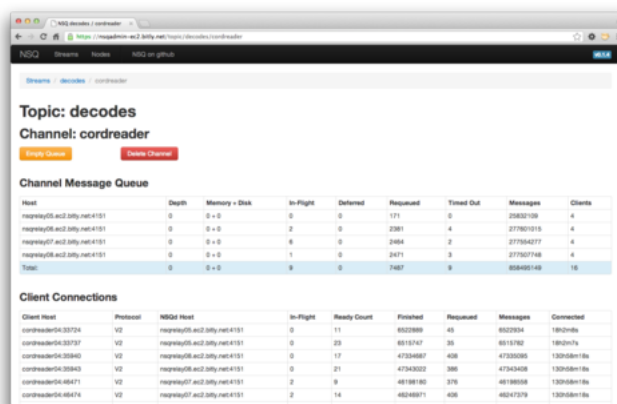
At a lower level each `nsqd` has a long-lived TCP connection to `nsqlookupd` over which it periodically pushes its state. This data is used to inform which `nsqd` addresses `nsqlookupd` will give to consumers. For consumers, an HTTP `/lookup` endpoint is exposed for polling.

To introduce a new distinct consumer of a topic, simply start up an **NSQ** client configured with the addresses of your `nsqlookupd` instances. There are no configuration changes needed to add either new consumers or new publishers, greatly reducing overhead and complexity.

NOTE: in future versions, the heuristic `nsqlookupd` uses to return addresses could be based on depth, number of connected clients, or other “intelligent” strategies. The current implementation is simply *all*. Ultimately, the goal is to ensure that all producers are being read from such that depth stays near zero.

It is important to note that the `nsqd` and `nsqlookupd` daemons are designed to operate independently, without communication or coordination between siblings.

We also think that it’s really important to have a way to view, introspect, and manage the cluster in aggregate. We built `nsqadmin` to do this. It provides a web UI to browse the hierarchy of topics/channels/consumers and inspect depth and other key statistics for each layer. Additionally it supports a few administrative commands such as removing and emptying a channel (which is a useful tool when messages in a channel can be safely thrown away in order to bring depth back to 0).



The screenshot shows the NSQ Admin UI for the topic 'decodes' and channel 'cordreader'. It includes a 'Channel Message Queue' table and a 'Client Connections' table.

Host	Depth	Memory + Disk	In-Flight	Deferred	Required	Timed Out	Messages	Clients
nsqlookupd.ac2.dilly.net:4151	0	0 + 0	0	0	171	0	2083108	4
nsqlookupd.ac2.dilly.net:4151	0	0 + 0	2	0	2581	4	277601815	4
nsqlookupd.ac2.dilly.net:4151	0	0 + 0	6	0	2484	2	277354277	4
nsqlookupd.ac2.dilly.net:4151	0	0 + 0	1	0	2471	3	277627746	4
Total	0	0 + 0	9	0	7487	9	858461148	16

Client Host	Protocol	NSQD Host	In-Flight	Pending Count	Prioritised	Required	Messages	Connected
cordreader04.33754	V2	nsqlookupd.ac2.dilly.net:4151	0	11	602088	45	602084	100mb/s
cordreader04.33757	V2	nsqlookupd.ac2.dilly.net:4151	0	23	6015147	35	6015182	100mb/s
cordreader04.33840	V2	nsqlookupd.ac2.dilly.net:4151	0	17	47354687	408	47355085	130mb/s
cordreader04.33843	V2	nsqlookupd.ac2.dilly.net:4151	0	21	47343022	385	47343458	130mb/s
cordreader04.48471	V2	nsqlookupd.ac2.dilly.net:4151	2	9	46188186	376	46188561	130mb/s
cordreader04.48474	V2	nsqlookupd.ac2.dilly.net:4151	2	14	46246871	406	46247376	130mb/s

▼ Straightforward Upgrade Path

This was one of our **highest** priorities. Our production systems handle a large volume of traffic, all built upon our existing messaging tools, so we needed a way to slowly and methodically upgrade specific parts of our infrastructure with little to no impact.

First, on the message *producer* side we built `nsqd` to match `simplequeue`. Specifically, `nsqd` exposes an HTTP `/pub` endpoint, just like `simplequeue`, to POST binary data (with the one caveat that the endpoint takes an additional query parameter specifying the “topic”). Services that wanted to switch to start publishing to `nsqd` only have to make minor code changes.

Second, we built libraries in both Python and Go that matched the functionality and idioms we had been accustomed to in our existing libraries. This eased the transition on the message *consumer* side by limiting the code changes to bootstrapping. All business logic remained the same.

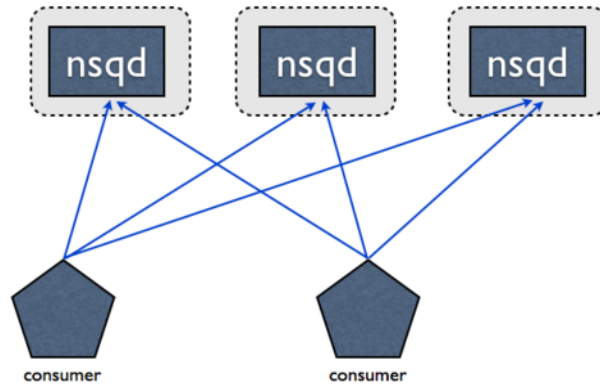
Finally, we built utilities to glue old and new components together. These are all available in the `examples` directory in the repository:

- `nsq_to_file` - durably write all messages for a given topic to a file

- `nsq_to_http` - perform HTTP requests for all messages in a topic to (multiple) endpoints

▼ Eliminating SPOFs

NSQ is designed to be used in a distributed fashion. `nsqd` clients are connected (over TCP) to **all** instances providing the specified topic. There are no middle-men, no message brokers, and no SPOFs:



This topology eliminates the need to chain single, aggregated, feeds. Instead you consume directly from **all** producers. *Technically*, it doesn't matter which client connects to which **NSQ**, as long as there are enough clients connected to all producers to satisfy the volume of messages, you're guaranteed that all will eventually be processed.

For `nsqlookupd`, high availability is achieved by running multiple instances. They don't communicate directly to each other and data is considered eventually consistent. Consumers poll *all* of their configured `nsqlookupd` instances and union the responses. Stale, inaccessible, or otherwise faulty nodes don't grind the system to a halt.

▼ Message Delivery Guarantees

NSQ guarantees that a message will be delivered **at least once**, **though duplicate messages are possible**. Consumers should expect this and de-dupe or perform idempotent operations.

This guarantee is enforced as part of the protocol and works as follows (assume the client has successfully connected and subscribed to a topic):

1. client indicates they are ready to receive messages
2. **NSQ** sends a message and temporarily stores the data locally (in the event of re-queue or timeout)
3. client replies FIN (finish) or REQ (re-queue) indicating success or failure respectively. If client does not reply **NSQ** will timeout after a configurable duration and automatically re-queue the message)

This ensures that the only edge case that would result in message loss is an unclean shutdown of an `nsqd` process. In that case, any messages that were in memory (or any buffered writes not flushed to disk) would be lost.

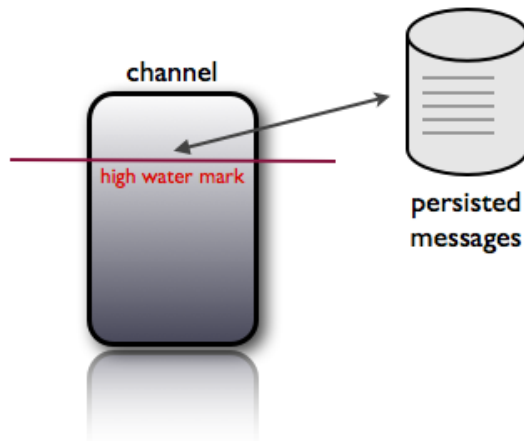
If preventing message loss is of the utmost importance, even this edge case can be mitigated. **One solution is to stand up redundant `nsqd` pairs (on separate hosts) that receive copies of the same portion of messages**. Because you've written your consumers to be idempotent, doing double-time on these messages has no downstream impact and allows the system to endure any single node failure without losing messages.

The takeaway is that **NSQ** provides the building blocks to support a variety of production use cases and configurable degrees of durability.

▼ Bounded Memory Footprint

`nsqd` provides a configuration option `--mem-queue-size` that will determine the number of messages that are kept *in memory* for a given queue. If the depth of a queue exceeds this threshold messages are **transparently written to disk**. This bounds the memory footprint of a given `nsqd` process to `mem-queue-size * #_of_channels_and_topics`:

Default value: `10000`



Also, an astute observer might have identified that this is a convenient way to gain an even higher guarantee of delivery by setting this value to something low (like 1 or even 0). The disk-backed queue is designed to survive unclean restarts (although messages might be delivered twice).

Also, related to message delivery guarantees, *clean* shutdowns (by sending a `nsqd` process the TERM signal) safely persist the messages currently in memory, in-flight, deferred, and in various internal buffers.

Note, a topic/channel whose name ends in the string `#ephemeral` will **not be buffered to disk** and will instead drop messages after passing the `mem-queue-size`. This enables **consumers which do not need message guarantees to subscribe to a channel**. These ephemeral channels will also disappear after its last client disconnects. For an ephemeral topic, this implies that at least one channel has been created, consumed, and deleted (typically an ephemeral channel).

▼ Efficiency

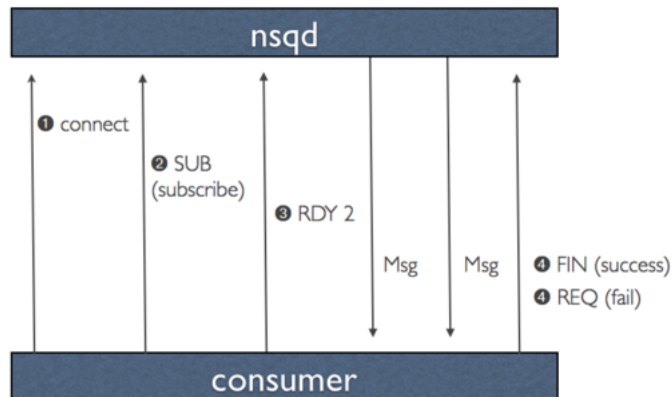
NSQ was designed to communicate over a “memcached-like” command protocol with simple size-prefixed responses. All message data is kept in the core including metadata like number of attempts, timestamps, etc. This eliminates the copying of data back and forth from server to client, an inherent property of the previous toolchain when re-queueing a message. This also simplifies clients as they no longer need to be responsible for maintaining message state.

Also, by reducing configuration complexity, setup and development time is greatly reduced (especially in cases where there are >1 consumers of a topic).

For the data protocol, we made a key design decision that maximizes performance and throughput by pushing data to the client instead of waiting for it to pull. This concept, which we call `RDY` state, is essentially a form of client-side flow control.

When a client connects to `nsqd` and subscribes to a channel it is placed in a `RDY` state of 0. This means that no messages will be sent to the client. When a client is ready to receive messages it sends a command that updates its `RDY` state to some # it is prepared to handle, say 100. Without any additional commands, 100 messages will be pushed to the client as they are available (each time decrementing the server-side RDY count for that client).

Client libraries are designed to send a command to update `RDY` count when it reaches ~25% of the configurable `max-in-flight` setting (and properly account for connections to multiple `nsqd` instances, dividing appropriately).



This is a significant performance knob as some downstream systems are able to more-easily batch process messages and benefit greatly from a higher `max-in-flight`.

Notably, because it is both buffered *and* push based with the ability to satisfy the need for independent copies of streams (channels), we've produced a daemon that behaves like `simplequeue` and `pubsub` combined. This is powerful in terms of simplifying the topology of our systems where we would have traditionally maintained the older toolchain discussed above.

▼ Go

We made a strategic decision early on to build the **NSQ** core in Go. We recently blogged about our use of Go at bitly and alluded to this very project - it might be helpful to browse through that post to get an understanding of our thinking with respect to the language.

Regarding **NSQ**, Go channels (not to be confused with **NSQ** channels) and the language's built in concurrency features are a perfect fit for the internal workings of `nsqd`. We leverage buffered channels to manage our in memory message queues and seamlessly write overflow to disk.

The standard library makes it easy to write the networking layer and client code. The built in memory and cpu profiling hooks highlight opportunities for optimization and require very little effort to integrate. We also found it really easy to test components in isolation, mock types using interfaces, and iteratively build functionality.

▼ nsqd

`nsqd` is the daemon that receives, queues, and delivers messages to clients.

It can be run standalone but is normally configured in a cluster with `nsqllookupd` instance(s) (in which case it will announce topics and channels for discovery).

It listens on **two TCP ports**, one for clients (4150) and another for the HTTP API (4151). It can optionally listen on a third port for HTTPS.

▼ Using cmd: http port of nsqd 4151

Create topic

```
curl -X POST http://127.0.0.1:4151/topic/create?topic=latestTopic
```

Publish message

```
curl -d "hello world" http://127.0.0.1:4151/pub?topic= NewTopic
```

Now open the program Consumer/OneToOne and change the name of topic to receive the message.

Delete topic

```
curl -X POST http://127.0.0.1:4151/topic/delete?topic=latestTopic
```

Return internal statistics


```
curl http://127.0.0.1:4151/stats
```

▼ nsqlookupd

`nsqlookupd` is the daemon that manages topology information. **Clients query `nsqlookupd` to discover `nsqd` producers for a specific topic and `nsqd` nodes broadcasts topic and channel information.**

There are two interfaces: A TCP interface which is used by `nsqd` **for broadcasts** and an **HTTP interface for clients to perform discovery and administrative actions**.

▼ cmd commands on http port: 4161

Get name of all the topics:

```
curl -X GET http://127.0.0.1:4161/topics
```

```
-> {"topics":
```

```
["OneToOne","latestTopic","newTopic","oneToOne","My","neOneToOne","test","OnetoOne","neNewTopic1","myGroup","NewTo
```

▼ Topology

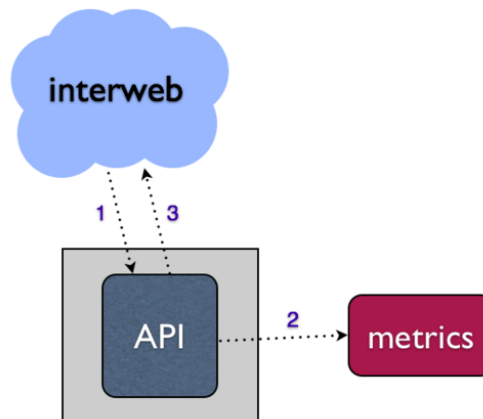
This document describes some NSQ patterns that solve a variety of common problems.

DISCLAIMER: there are *some* obvious technology suggestions but this document generally ignores the deeply personal details of choosing proper tools, getting software installed on production machines, managing what service is running where, service configuration, and managing running processes (`daemontools`, `supervisord`, `init.d`, etc.).

Metrics Collection

Regardless of the type of web service you're building, in most cases you're going to **want to collect some form of metrics** in order to understand your infrastructure, your users, or your business.

For a web service, most often these metrics are produced by events that happen via HTTP requests, like an API. The naive approach would be to structure this synchronously, writing to your metrics system directly in the API request handler.

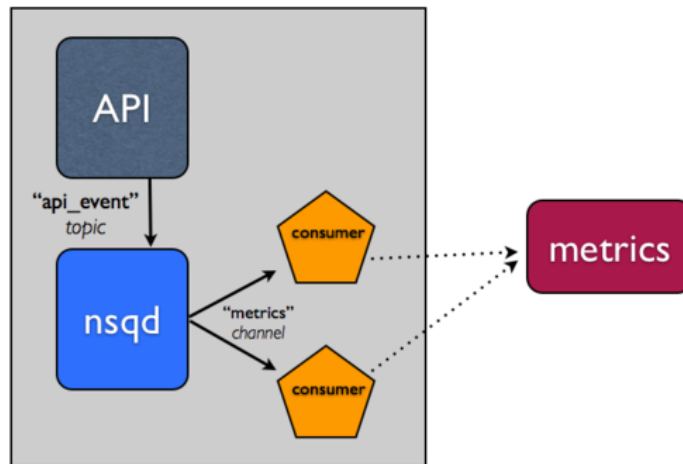


- What happens when your metrics system goes down?
- Do your API requests hang and/or fail?
- How will you handle the scaling challenge of increasing API request volume or breadth of metrics collection?

One way to resolve all of these issues is to somehow perform the work of writing into your metrics system asynchronously - that is, place the data in **some sort of local queue and write into your downstream system via some other process** (consuming that queue). This separation of concerns allows the system to be more robust and fault tolerant. At bitly, we use NSQ to achieve this.

Brief tangent: NSQ has the concept of **topics** and **channels**. Basically, think of a **topic** as a **unique stream of messages** (like our stream of API events above). Think of a **channel** as **a copy of that stream of messages for a given set of consumers**.

Topics and channels are **both independent queues**, too. These properties enable NSQ to support both **multicast** (a topic *copying* each message to N channels) and **distributed** (a channel *equally dividing* its messages among N consumers) message delivery.



Integrating NSQ is straightforward, let's take the simple case:

1. Run an instance of `nsqd` on the same host that runs your API application.
2. Update your API application to write to the local `nsqd` instance to queue events, instead of directly into the metrics system. To be able to easily introspect and manipulate the stream, we generally format this type of data in line-oriented JSON. Writing into `nsqd` can be as simple as performing an HTTP POST request to the `/put` endpoint.
3. Create a consumer in your preferred language using one of our [client libraries](#). This "worker" will subscribe to the stream of data and process the events, writing into your metrics system. **It can also run locally on the host running both your API application and `nsqd`.**

In addition to de-coupling, by using one of our official client libraries, consumers will degrade gracefully when message processing fails. Our libraries have two key features that help with this:

1. **Retries** - when your message **handler indicates failure**, that information is sent to `nsqd` in the form of a `REQ` (re-queue) command. Also, `nsqd` will **automatically time out** (and **re-queue**) a message if it hasn't been responded to in a configurable time window. These two properties are critical to providing a delivery guarantee.
2. **Exponential Backoff** - when message processing fails the reader library will delay the receipt of additional messages for a duration that scales exponentially based on the # of consecutive failures. The opposite sequence happens when a reader is in a backoff state and begins to process successfully, until 0.

In concert, these two features allow the system to respond gracefully to downstream failure, automatically.

Persistence

Ok, great, now you have the ability to withstand a situation where your metrics system is unavailable with no data loss and no degraded API service to other endpoints. You also have the ability to scale the processing of this stream horizontally by adding more worker instances to consume from the same channel.

But, it's kinda hard ahead of time to think of all the types of metrics you might want to collect for a given API event.

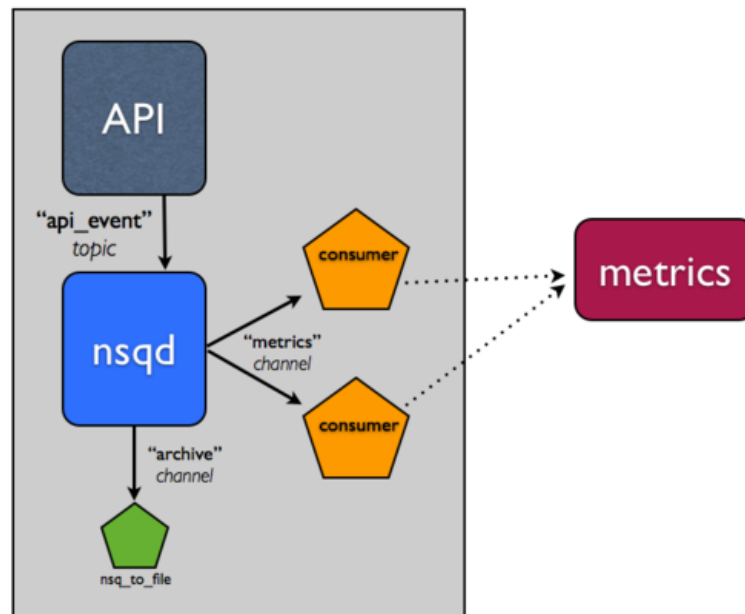
Wouldn't it be nice to have an archived log of this data stream for any future operation to leverage? Logs tend to be relatively easy to redundantly backup, making it a "plan z" of sorts in the event of catastrophic downstream data loss. But, would you want this same consumer to also have the responsibility of archiving the message data? Probably not, because of that whole "separation of concerns" thing.

Archiving an NSQ topic is such a common pattern that we built a utility, `nsq_to_file`, packaged with NSQ, that does exactly what you need.

Remember, in NSQ, each channel of a topic is independent and receives a *copy* of all the messages. You can use this to your advantage when archiving the stream by doing so over a new channel, `archive`. Practically, this means that if your metrics system is having issues and the `metrics` channel gets backed up, it won't effect the separate `archive` channel you'll be using to persist messages to disk.

So, add an instance of `nsq_to_file` to the same host and use a command line like the following:

```
/usr/local/bin/nsq_to_file --nsqd-tcp-address=127.0.0.1:4150 --topic=api_requests --channel=archive
```

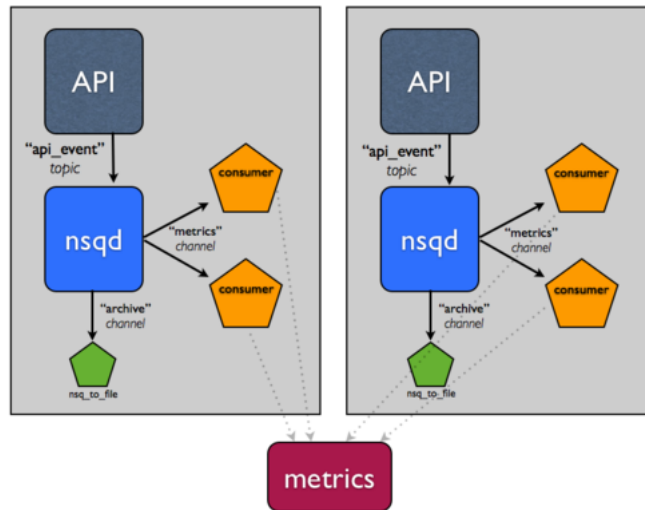


Distributed Systems

You'll notice that the system has not yet evolved beyond a single production host, which is a glaring single point of failure.

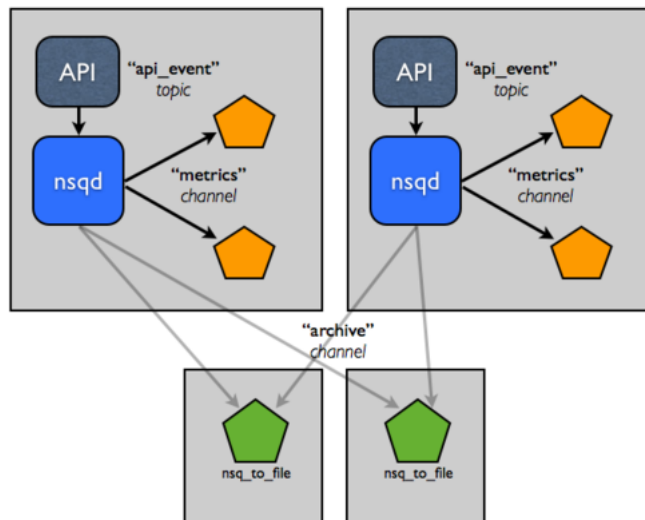
Unfortunately, building a distributed system is hard. Fortunately, NSQ can help. The following changes demonstrate how NSQ alleviates some of the pain points of building distributed systems as well as how its design helps achieve high availability and fault tolerance.

Let's assume for a second that this event stream is *really* important. You want to be able to tolerate host failures and continue to ensure that messages are *at least* archived, so you add another host.



Assuming you have some sort of load balancer in front of these two hosts you can now tolerate any single host failure.

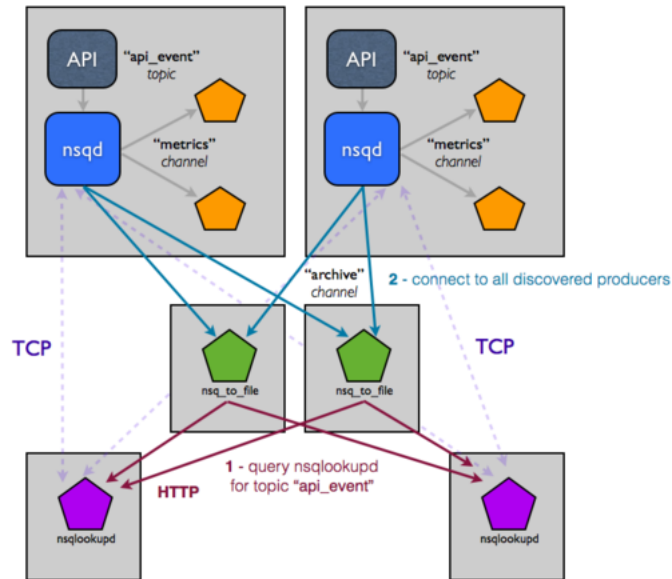
Now, let's say the process of persisting, compressing, and transferring these logs is affecting performance. How about splitting that responsibility off to a tier of hosts that have higher IO capacity?



This topology and configuration can easily scale to double-digit hosts, but you're still managing configuration of these services manually, which does not scale. Specifically, in each consumer, this setup is hard-coding the address of where `nsqd` instances live, which is a pain. What you really want is for the configuration to evolve and be accessed at runtime based on the state of the NSQ cluster. This is *exactly* what we built `nsqlookupd` to address.

`nsqlookupd` is a daemon that records and disseminates the state of an NSQ cluster at runtime. `nsqd` instances maintain persistent TCP connections to `nsqlookupd` and push state changes across the wire. Specifically, an `nsqd` registers itself as a producer for a given topic as well as all channels it knows about. This allows consumers to query an `nsqlookupd` to determine who the producers are for a topic of interest, rather than hard-coding that configuration. Over time, they will *learn* about the existence of new producers and be able to route around failures.

The only changes you need to make are to point your existing `nsqd` and consumer instances at `nsqlookupd` (everyone explicitly knows where `nsqlookupd` instances are but consumers don't explicitly know where producers are, and vice versa). The topology now looks like this:



At first glance this may look *more* complicated. It's deceptive though, as the effect this has on a growing infrastructure is hard to communicate visually. You've effectively decoupled producers from consumers because `nsqlookupd` is now acting as a directory service in between. Adding additional downstream services that depend on a given stream is trivial, **just specify the topic you're interested in (producers will be discovered by querying `nsqlookupd`)**.

But what about availability and consistency of the lookup data?

We generally recommend basing your decision on how many to run in congruence with your desired availability requirements. `nsqllookupd` is not resource intensive and can be easily homed with other services. Also, `nsqllookupd` instances do not need to coordinate or otherwise be consistent with each other. Consumers will generally only require one `nsqllookupd` to be available with the information they need (and they will union the responses from all of the `nsqllookupd` instances they know about). Operationally, this makes it easy to migrate to a new set of `nsqllookupd`.

▼ nsqadmin

Metrics

The following metrics are exposed through `nsqadmin` on Topic, Channel and Client Connections.

Message Queues:

- **Depth** : Current sum of messages in memory + on disk (i.e. the “backlog” of messages pending delivery).
- **In-Flight** : Current count of messages delivered but not yet finished (**FIN**), requeued (**REQ**) or timed out. [[Read this to get info of related topic](#)]
- **Deferred** : Current count of messages that were requeued and explicitly deferred which are not yet available for delivery.

If our messages are requeued and again when we try to send the message but when we are not able to send the same again and again then it is set as deferred.

Statistics:

- **Requested** : Total times a message has been added back to the queue due to time outs or explicit requeses.
- **Timed Out** : Total times a message has been requested after not receiving a response from the client before the configured timeout.
- **Messages** : Total count of new messages recieved since node startup.

- **Rate** : The per-second rate of new messages over the previous two statsd intervals (available only when graphite integration is enabled).
- **Connections** : Current number of connected clients.

Client Connections:

- **Client Host** : Client ID (hostname) and on-hover the connection remote-address.
- **Protocol** : NSQ protocol version and client user-agent.
- **Attributes** : TLS and AUTH connection state.
- **NSQd Host** : Address of the nsqd node this client is connected to.
- **In-flight** : Current count of messages awaiting response that have been delivered to this client.
- **Ready Count** : Max number of messages that can be in-flight on this connection. This is controlled by a client's `max_in_flight` setting.
- **Finished** : Total count of messages that this client has finished (**FIN**).
- **Requeued** : Total count of messages that this client has requeued (**REQ**).
- **Messages** : Total count of messages delivered to this client.

ZeroMQ (ZMQ)

▼ Purpose of ZMQ

One, to solve the general problem of "how to connect any code to any code, anywhere".

Two, to wrap that up in the simplest possible building blocks that people could understand and use *easily*.

▼ Implementation

Get started

Get started with zeromq

🔗 <https://zeromq.org/get-started/?language=java&library=jeromq#>



In Python:

A zmq Context creates sockets via its `ctx.socket` method.

Get started

Get started with zeromq

🔗 <https://zeromq.org/get-started/?language=python&library=jeromq#>



▼ Explanation link

Socket API

An open-source universal messaging library

🔗 <https://zeromq.org/socket-api/?language=java&library=jeromq#>



▼ Key differences to conventional sockets

Generally speaking, conventional sockets present a synchronous interface to either connection-oriented reliable byte streams (SOCK_STREAM), or connection-less unreliable datagrams (SOCK_DGRAM). In comparison, ZeroMQ sockets present an

abstraction of an **asynchronous message queue**, with the exact queueing semantics depending on the socket type in use.

Where conventional sockets transfer **streams of bytes** or discrete datagrams, ZeroMQ sockets transfer **discrete messages**.

ZeroMQ sockets being asynchronous means that the timings of the physical connection setup and tear down, reconnect and effective delivery are transparent to the user and organized by ZeroMQ itself. Further, messages may be queued in the event that a peer is unavailable to receive them.

Conventional sockets allow only strict one-to-one (two peers), many-to-one (many clients, one server), or in some cases one-to-many (multicast) relationships. With the exception of PAIR sockets, ZeroMQ sockets **may be connected to multiple endpoints**, while simultaneously accepting incoming connections from multiple endpoints bound to the socket, thus **allowing many-to-many relationships**.

▼ Bind vs Connect

With ZeroMQ sockets it doesn't matter who binds and who connects. In the above you may have noticed that the server used Bind while the client used Connect. Why is this, and what is the difference?

ZeroMQ **creates queues** per underlying connection. If your socket is connected to three peer sockets, then there are three messages queues behind the scenes.

With Bind, you allow peers to connect to you, thus you don't know how many peers there will be in the future and you **cannot create the queues in advance in case of bind**. Instead, **queues are created as individual peers connect to the bound socket**.

With Connect, ZeroMQ knows that there's going to be at least a single peer and thus it can create a single queue immediately. This applies to all socket types except ROUTER, where queues are only created after the peer we connect to has acknowledged our connection.

Consequently, when sending a message to bound socket with no peers, or a ROUTER with no live connections, there's no queue to store the message to.

▼ Send and it's flags

The `zmq_send()` function shall queue the message referenced by the `msg` argument to be sent to the socket referenced by the `socket` argument. The `flags` argument is a combination of the flags defined below:

ZMQ_NOBLOCK: Specifies that the operation should be performed in non-blocking mode. If the message cannot be queued on the `socket`, the `zmq_send()` function shall fail with `errno` set to EAGAIN.

ZMQ_SNDMORE: Specifies that the message being sent is a multi-part message, and that further message parts are to follow. Refer to the section regarding multi-part messages below for a detailed description.

The `zmq_msg_t` structure passed to `zmq_send()` is nullified during the call. If you want to send the same message to multiple sockets you have to copy it using (e.g. using `zmq_msg_copy()`).

A successful invocation of `zmq_send()` does not indicate that the message has been transmitted to the network, only that it has been queued on the `socket` and ØMQ has assumed responsibility for the message.

Multi-part messages

A ØMQ message is composed of 1 or more message parts; each message part is an independent `zmq_msg_t` in its own right. ØMQ ensures atomic delivery of messages; peers shall receive either all *message parts* of a message or none at all.

The total number of message parts is unlimited.

An application wishing to send a multi-part message does so by specifying the `ZMQ_SNDMORE` flag to `zmq_send()`. **The presence of this flag indicates to ØMQ that the message being sent is a multi-part message and that more message parts are to follow.** **When the application wishes to send the final message part it does so by calling `zmq_send()` without the `ZMQ_SNDMORE` flag;** this indicates that no more message parts are to follow.

Return value

The `zmq_send()` function shall return zero if successful. Otherwise it shall return -1 and set `errno` to one of the values defined in the website.

▼ Receive

The `zmq_recv()` function shall receive a message from the socket referenced by the `socket` argument and store it in the message referenced by the `msg` argument. Any content previously stored in `msg` shall be properly deallocated. If there are no messages available on the specified `socket` the `zmq_recv()` function shall block until the request can be satisfied. The `flags` argument is a combination of the flags defined below:

ZMQ_NOBLOCK: Specifies that the operation should be performed in non-blocking mode. If there are no messages available on the specified `socket`, the `zmq_recv()` function shall fail with `errno` set to `EAGAIN`.

▼ Linger

set-linger

`(set-linger socket linger-ms)`

The `linger` option shall set the `linger period` for the specified socket. The `linger period` determines **how long pending messages which have yet to be sent to a peer shall linger in memory after a socket is closed with `close`**, and further affects the termination of the socket's context with `close`. The following outlines the different behaviours:

The **default value of -1** specifies an **infinite linger period**. Pending messages shall not be discarded after a call to `close`; attempting to terminate the socket's context with `close` shall block until all pending messages have been sent to a peer.

The value of 0 specifies no `linger period`. Pending messages shall be discarded immediately when the socket is closed with `close`.

Positive values specify an upper bound for the `linger period in milliseconds`. Pending messages shall not be discarded after a call to `close`; attempting to **terminate the socket's context** with `close` shall block until either all pending messages have been sent to a peer, or the `linger period` expires, **after which any pending messages shall be discarded**.

▼ High water level and socket types

The high water mark is a hard limit on the maximum number of outstanding messages ZeroMQ is queuing in memory for any single peer that the specified socket is communicating with.

If this limit has been reached the socket enters an exceptional state and depending on the socket type, **ZeroMQ will take appropriate action such as blocking or dropping sent messages**. Refer to the individual socket descriptions below for details on the exact action taken for each socket type.

PUB socket

A `PUB` socket is used by a publisher to distribute data. Messages sent are distributed in a fan out fashion to all connected peers. This socket type is not able to receive any messages.

When a `PUB` socket enters the mute state due to having reached the high water mark for a subscriber, then any messages that would be sent to the subscriber in question shall instead be dropped until the mute state ends. The `send` function does never block for this socket type.

PUSH socket

The `PUSH` socket type talks to a set of anonymous `PULL` peers, sending messages using a round-robin algorithm. The `receive` operation is not implemented for this socket type.

When a `PUSH` socket enters the mute state due to having reached the high water mark for all downstream nodes, or if there are no downstream nodes at all, then any `send` operations on the socket will block until the mute state ends or at least one

downstream node becomes available for sending; messages are not discarded.

▼ Request reply pattern

Request-reply pattern

The request-reply pattern is intended for service-oriented architectures of various kinds. It comes in two basic flavors: synchronous (`REQ` and `REP` socket types), and asynchronous socket types (`DEALER` and `ROUTER` socket types), which may be mixed in various ways.

The request-reply pattern is formally defined by RFC [28/REQREP](#).

REQ socket

A `REQ` socket is used by a client to send requests to and receive replies from a service. This socket type allows only an alternating sequence of *sends* and subsequent *receive* calls. A `REQ` socket may be connected to any number of `REP` or `ROUTER` sockets. Each request sent is round-robin among all connected services, and **each reply received is matched with the last issued request. It is designed for simple request-reply models where reliability against failing peers is not an issue.**

If no services are available, then any send operation on the socket will block until at least one service becomes available.

The `REQ` socket will not discard any messages.

Summary of characteristics:

Compatible peer sockets	REP, ROUTER
Direction	Bidirectional
Send/receive pattern	Send, Receive, Send, Receive, ...
Outgoing routing strategy	Round-robin
Incoming routing strategy	Last peer
Action in mute state	Block

REP socket

A `REP` socket is used by a service to receive requests from and send replies to a client. This socket type allows only an alternating sequence of *receive* and subsequent *send* calls. Each request received is fair-queued from among all clients, and each reply sent is routed to the client that issued the last request. If the original requester does not exist any more the reply is silently discarded.

Summary of characteristics:

Compatible peer sockets	REQ, DEALER
Direction	Bidirectional
Send/receive pattern	Receive, Send, Receive, Send ...
Outgoing routing strategy	Fair-robin
Incoming routing strategy	Last peer

▼ Official doc

ZMQ.Socket - `jeromq 0.4.2` javadoc

`czmq-jni` `jeromq` `jzmq` `zeromq-scala-binding_2.10` `zeromq-scala-binding_2.10.0-M5` `zeromq-scala-binding_2.10.0-M6` `zeromq-scala-binding_2.10.0-M7` `zeromq-scala-binding_2.10.0-RC2` `zeromq-scala-binding_2.10.0-RC3` `zeromq-scala-binding_2.10.0-RC5` `zeromq-scala-binding_2.11.0-M3`

 <https://www.javadoc.io/doc/org.zeromq/jeromq/0.4.2/org/zeromq/ZMQ.Socket.html>

Topics and Questions:

- ✓ Why different sequence when we get msg in consumer after we restart consumer. Sending msg from producer when consumer closed

nsqadmin headings meaning

- ✓ memory + disk (how that storage works) in nsqadmin
- ✓ in flight at what stage
- ✓ deferred
- ✓ oneToOne/producer line 26 keep lookup or not → no
- ✓ passing messages from the cmd
- ✓ understanding ndsq and lookup configuration
- ☐ Usage of object mapper
- ✓ member enters first time we can't send message to them
- ✓ one to one → done
- ✓ one to many → done
- ✓ one to many but one client get msg → done
- ✓ ZMQ → java and python → done
- ✓ NSQ memory and disk space → done
- ✓ linger and no blocking difference in zmq
high watermark in zmq