Building a distributed message processing system in Go using NSQ

Greg Bray (@GBrayUT on Twitter)
Edge Platform Operations @WalmartLabs

Slides at bit.ly/nsqslides

Some content from nsq.io

Goal: Process logs in a complex environment

Logs and other messages are produced by:

- Many servers across many different locations
- For different purposes:
  - Web logs
  - System logs
  - Security logs
  - Telemetry data
  - Etc …
- From different sources
  - Direct from various services
  - Log files (stdout / stderror)
  - syslog

Need a way to aggregate and process logs efficiently, with flexibility to meet fluid requirements

Destinations:

- Long term storage (kafka to hdfs)
- Search backend (ELK stack)
- Other pipelines (nsq, syslog, etc)
- Monitoring system (Prometheus metrics)
- Adhoc troubleshooting
- /dev/null (emergency overflow)

Some destinations need filtering or prioritization of data streams.

Some data streams have other requirements like encryption, low latency, etc...

NSQ Docs 1.0.0-compat

Secure | https://nsq.io

NSQ v1.0.0-compat

Documentation Community GitHub

NSQ

A realtime distributed messaging platform

Star 12,524 Watch 697

Get Started

simplereach heavy water LYTICS Rakuten MEDIAFORGE socialradar WISTIA

Distributed

NSQ 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.

Scalable

NSQ scales horizontally, without any centralized brokers. Built-in discovery simplifies the addition of nodes to the cluster. Supports both pub-sub and load-balanced message delivery. It's fast, too.

Ops Friendly

NSQ is easy to configure and deploy and comes bundled with an admin UI. Binaries have no runtime dependencies and we provide pre-compiled releases for linux, darwin, freebsd and windows as well as an official Docker image.

Integrated

Official Go and Python libraries are available as well as many community supported libraries for most major languages (see client libraries). If you're interested in building your own, there's a protocol spec.

	RabbitMQ	ZeroMQ	NSQ
	Message Queue	Message Queue	Message Queue
	See RabbitMQ alternatives	See ZeroMQ alternatives	See NSQ alternatives
Favorites	107	16	19
Stacks	1.99K	101	50
	I Use This	I Use This	I Use This
Fans	1.31K	87	79
Votes	385	46	123
Jobs	1.12K	39	97
Hacker News, Reddit, Stack Overflow Stats	402 / 320 / 8.14K	708 / 797 / 2.46K	286 / 15 / 38
GitHub Stats	4.14K / 1.08K / 2 days ago	4.33K / 1.31K / 1 day ago	12.5K / 1.64K / 1 day ago

https://stackshare.io/stackups/nsq-vs-rabbitmq-vs-zeromq

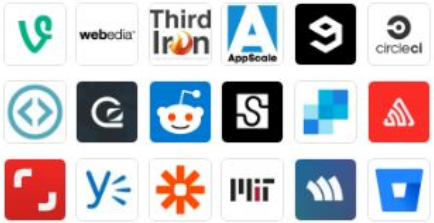
Pros

- 23 It's in golang
- 18 Distributed
- 17 Lightweight
- 16 Easy setup
- 14 High throughput
- 9 Publish-Subscribe
- 6 Save data if no subscribers are found
- 6 Scalable
- 5 Open source
- 4 Temporarily kept on disk

Cons

- 3 Needs Erlang runtime. Need ops good with Erlang runtime
- 1 Too complicated cluster/HA config and management
- 1 Configuration must be done first, not by your code

Companies using RabbitMQ



See more stacks

Cons

- 1 No message durability
- 1 Not a very reliable system - message delivery wise
- 1 M x N problem with M producers and N consumers

Companies using ZeroMQ

See more stacks

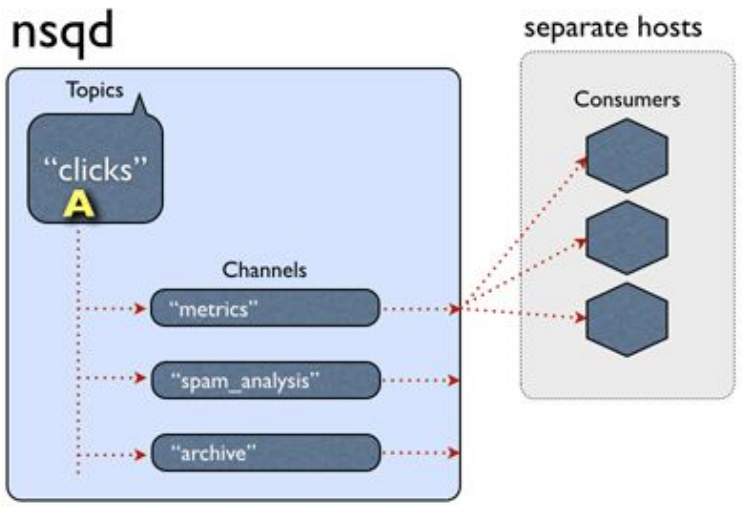
Cons

No Cons submitted yet for NSQ

Companies using NSQ

Topics, Channels, and Consumers

- Top level: **Topics** are streams of data
- Split topics into **Channels**
  - Each channel gets a copy of all messages
- Channels can have one or more **Consumers**
  - Consumers pull messages from a channel and must FIN (finish) or REQ (re-queue) each message it takes
  - Configurable timeout for automatic re-queueing
  - Scale out: more consumers for more throughput
  - Can be local or remote (basic discovery via lookupd)
  - A consumer may just filter messages and publish them into another Topic on a local or remote system
- Topics and Channels are created at runtime, just start publishing/subscribing (Auto cleanup if #Ephemeral)



Good example of a high performance system written in Go. See internals at https://nsq.io/overview/internals.html

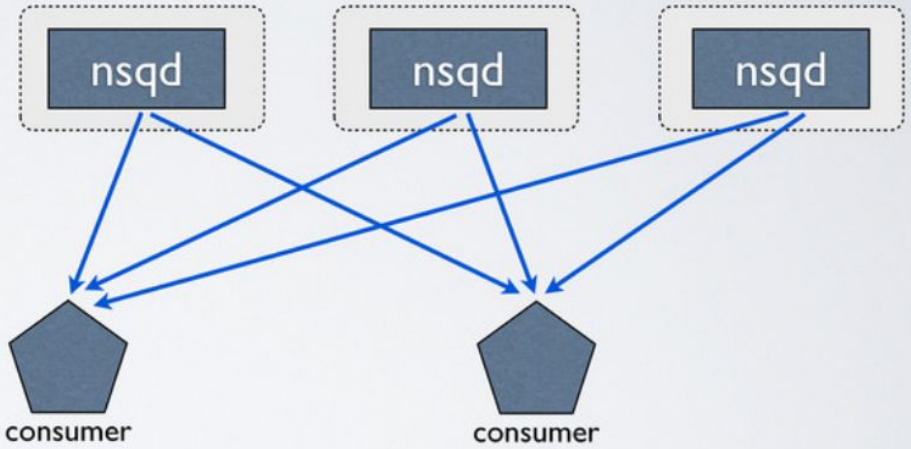
Lookupd: nsqd registers topics and channels

Lookupd: Consumers query for nodes/topics

Redundancy... eliminate Single Point of Failure

ELIMINATE ALL THE SPOF

- easily enable *distributed* and *decentralized* topologies
- no brokers
- consumers connect to all producers
- messages are *pushed* to consumers
- **nsqlookupd** instances are *independent* and require no coordination (run a few for HA)

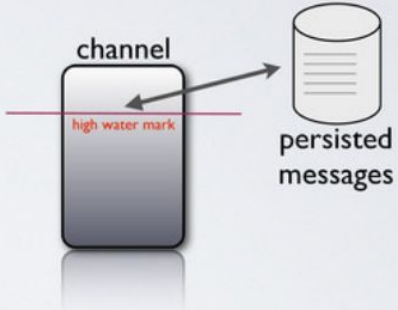


Messages usually stored in memory (but overflow to disk)

QUEUES

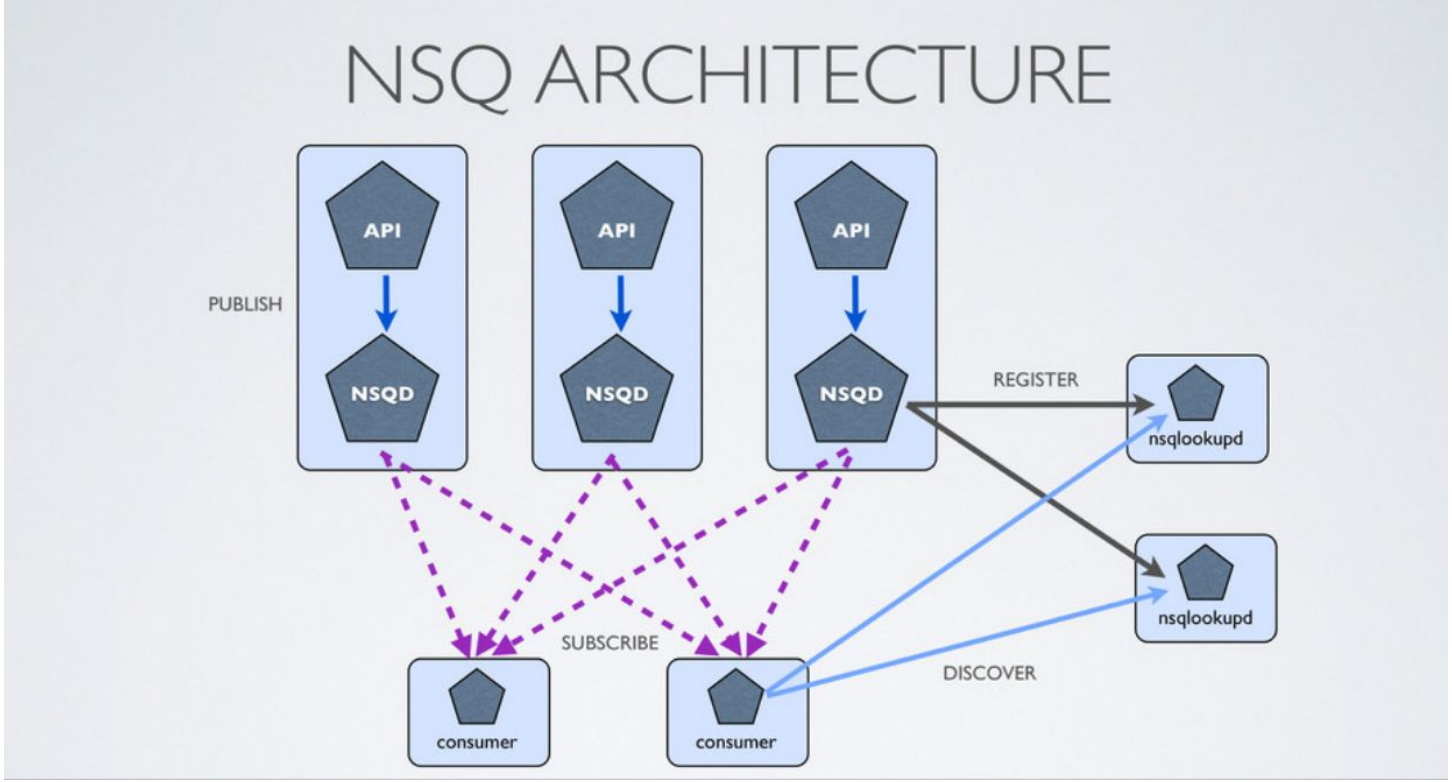
buffer *this*

- **topics** and **channels** are *independent* queues
- queues have arbitrary high water marks (after which messages transparently read/write to disk, bounding memory footprint)
- supports channel-independent degradation and recovery
- 10 lines of Go




```go
for msg := range c.incomingMsgChan {
    select {
    case c.memoryMsgChan <- msg:
    default:
        err := WriteMessageToBackend(&msgBuf, msg, c)
        if err != nil {
            // log whatever
        }
    }
}
```

Example layout. More at https://nsq.io/deployment/topology_patterns.html



```
someuser@servername:~$ curl -s --cacert ~/ca.crt --key ~/nsqd.key --cert ~/nsqd.crt 'https://localhost:4152/stats'
nsqd v0.3.8 (built w/go1.6.2)
start_time 2018-05-09T16:20:59Z
uptime 780h16m54.793013562s

Health: OK

 [topicA  ] depth: 0     be-depth: 0     msgs: 512579562 e2e%:
     [channel1           ] depth: 0     be-depth: 0     inflt: 8    def: 0    re-q: 7256  timeout: 7256  msgs: 301088800 e2e%:
       [V2 ip:49012] state: 3 inflt: 8    rdy: 500  fin: 9630606  re-q: 0        msgs: 9630614  connected: 102h4m56s
       [V2 ip:36945] state: 3 inflt: 0    rdy: 500  fin: 9700390  re-q: 0        msgs: 9700390  connected: 102h18m26s
     [channel2           ] depth: 0     be-depth: 0     inflt: 1    def: 0    re-q: 285   timeout: 285   msgs: 211701180 e2e%:
       [V2 ip:40949] state: 3 inflt: 1    rdy: 5    fin: 43436806 re-q: 0        msgs: 43436807 connected: 166h34m29s
 [topicB  ] depth: 0     be-depth: 0     msgs: 852191855 e2e%:
     [channelX           ] depth: 0     be-depth: 0     inflt: 1    def: 0    re-q: 105   timeout: 105   msgs: 252191855 e2e%:
       [V2 ip:56569] state: 3 inflt: 1    rdy: 5    fin: 24983170 re-q: 0        msgs: 24983171 connected: 93h40m15s
     [channelY           ] depth: 114531 be-depth: 104533 inflt: 5    def: 0    re-q: 375   timeout: 375   msgs: 211670578 e2e%:
       [V2 ip:17175] state: 3 inflt: 5    rdy: 5    fin: 23079733 re-q: 0        msgs: 23079763 connected: 93h20m20s
     [channelZ           ] depth: 52426 be-depth: 42428 inflt: 5    def: 0    re-q: 1014  timeout: 1014  msgs: 512579561 e2e%:
       [V2 ip:63824] state: 3 inflt: 5    rdy: 5    fin: 40935902 re-q: 0        msgs: 40935954 connected: 166h34m31s
 [topicC  ] depth: 0     be-depth: 0     msgs: 512579562 e2e%:
     [channelAbandoned] depth: 46985 be-depth: 36987 inflt: 5    def: 0    re-q: 930   timeout: 930   msgs: 512579561 e2e%:
     [channelNew      ] depth: 0 be-depth: 0 inflt: 5    def: 0    re-q: 936   timeout: 936   msgs: 512579561 e2e%:
       [V2 ip38737] state: 3 inflt: 5    rdy: 5    fin: 40938584 re-q: 0        msgs: 40938639 connected: 166h34m31s
```

Example of stats output. Also using TLS with client certificates to secure access.

Utilities include in NSQ codebase:

- nsq_pubsub - expose a pubsub like HTTP interface to topics in an NSQ cluster
- **nsq_stat** - polls /stats for all the producers of the specified topic/channel and displays aggregate stats
- **nsq_tail** - consumes the specified topic/channel and writes to stdout (in the spirit of tail)
- nsq_to_file - consumes the specified topic/channel and writes out to a newline delimited file, optionally rolling and/or compressing the file.
- nsq_to_http - consumes the specified topic/channel and performs HTTP requests (GET/POST) to the specified endpoints.
- **nsq_to_nsq** - consumes the specified topic/channel and re-publishes the messages to destination nsqd
- to_nsq - takes a stdin stream and splits on newlines for re-publishing to destination nsqd

More details including command line arguments at https://nsq.io/components/utilities.html

nsq_to_nsq

Consumes the specified topic/channel and re-publishes the messages to destination `nsqd` via TCP.

Command Line Options

```
-channel string
    nsq channel (default "nsq_to_nsq")
-consumer-opt value
    option to passthrough to nsq.Consumer (may be given multiple times, see http://godoc.org/github.com/nsqio/go-nsq#Config)
-destination-nsqd-tcp-address value
    destination nsqd TCP address (may be given multiple times)
-destination-topic string
    destination nsq topic
-lookupd-http-address value
    lookupd HTTP address (may be given multiple times)
-max-in-flight int
    max number of messages to allow in flight (default 200)
-mode string
    the upstream request mode options: round-robin, hostpool (default), epsilon-greedy (default "hostpool")
-nsqd-tcp-address value
    nsqd TCP address (may be given multiple times)
-producer-opt value
    option to passthrough to nsq.Producer (may be given multiple times, see http://godoc.org/github.com/nsqio/go-nsq#Config)
-require-json-field string
    for JSON messages: only pass messages that contain this field
-require-json-value string
    for JSON messages: only pass messages in which the required field has this value
-status-every int
    the # of requests between logging status (per destination), 0 disables (default 250)
-topic string
    nsq topic
-version
    print version string
-whitelist-json-field value
    for JSON messages: pass this field (may be given multiple times)
```

```
ARGS='--cacert ~/ca.crt --key ~/nsqd.key --cert ~/nsqd.crt -lookupd-http-address lookupd01:4161 -lookupd-http-address lookupd02:4161'
nsq_to_nsq $ARGS -destination-nsqd-tcp-address=localhost:4150 -topic topicA -destination-topic topicA-Aggregated
```

Other utilities we've built in Go

- file2nsq - watches files on disk and generates nsq messages
- nsq2kafka - send messages from specific topics to various Kafka brokers
- nsqarchive - similar to nsq_tail but outputs a tar file with one entry per message
- nsq2es - send messages to ElasticSearch (Can replace ELK stack with ENK stack)
- nsqcopy - replaces multiple nsq_to_nsq instances with a single dynamic service using lookupd and a config file for topic source/destinations
- Also tools for decoding encrypted messages, or generating metrics directly from topics/channels or a filtered topic stream

Each of the above is a simple go program, usually a few hundred lines each. They run on two "log transport" servers in each data center to aggregate logs from all local servers.

Haven’t yet found a breaking point for NSQD (other than running out of disk space)

Any questions?

If this sounds like an interesting problem, you should come help us solve it!

Hiring Dev and DevOps that are familiar with Go and interested in “Industrial Grade” Internet / Websites.

Some other interesting systems we work on (all in Go):

- Edge Compute / FaaS platform (Lua and Go plugins)
- High performance HTTP / HTTP2 / Quic Proxies and Load Balancers
- Internal GSLB (Proximity aware DNS based load balancing)
- External GeoIP / Policy based DNS load balancing
- Solving complex problems at large scale (PCI)
- CSS/HTML/JSON/Image optimizations
- Running an international CDN
- Creating metrics / dashboards / tools to help thousands of developers find large and small needles in a very large haystack

Slides at bit.ly/nsqslides