STREAM PROCESSING
PHILOSOPHY, CONCEPTS, AND TECHNOLOGIES

Dan Frank
df@bit.ly
@danielhfrank

What did I just sign up for?

What did I just sign up for?

- Stream processing as a tool for decomposition and modularity

What did I just sign up for?

- Stream processing as a tool for decomposition and modularity
- Stream processing composition building blocks

What did I just sign up for?

- Stream processing as a tool for decomposition and modularity
- Stream processing composition building blocks
- Stream processing in your distributed web application

What did I just sign up for?

- Stream processing as a tool for decomposition and modularity
- Stream processing composition building blocks
- Stream processing in your distributed web application
- NSQ, Bitly's distributed messaging framework

What did I just sign up for?

- Stream processing as a tool for decomposition and modularity
- Stream processing composition building blocks
- Stream processing in your distributed web application
- NSQ, Bitly's distributed messaging framework
- The future now: stream processing within your programs, and technologies to do it

STREAM PROCESSING?

Let's say:

"Near-realtime processing of sequential messages / events"

A QUICK NOTE ON 

- Hadoop is a dominant framework for doing **batch tasks**: tasks that operate on a fully populated dataset and just need to be done "later". *Offline*
- Stream processing is basically the opposite of this: operating as new data comes in, computation happens *online*. No concept of "complete" dataset
- BUT, using the two as complementary data analysis components is very effective

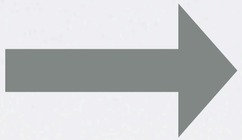
Career Topology

Trendrr®

spotINFLUENCE

Why Stream Processing?

Why Stream Processing?



Why Stream Processing?

There are better reasons!

CASE STUDY:
PROCESSING LINES IN A FILE

NAÏVE "ARCHITECTURE"

```
for line in lines:
  new_line = do_something(line)
  newer_line = do_something_else(new_line)
  # ...
  outputs.append(newest_line)
```

NAÏVE "ARCHITECTURE"

```
for line in lines:
  new_line = do_something(line)
  newer_line = do_something_else(new_line)
  # ...
  outputs.append(newest_line)
```

Composition of our functions is static, built into our program

NAÏVE "ARCHITECTURE"

```
for line in lines:
  new_line = do_something(line)
  newer_line = do_something_else(new_line)
  # ...
  outputs.append(newest_line)
```

Composition of our functions is static, built into our program

Error handling? Uhh

Unix Solution: Pipes

```
< lines do_something | do_something_else | ...
```

Unix Solution: Pipes

```
< lines do_something | do_something_else | ...
```

Composition happens outside the application code

Unix Solution: Pipes

```
< lines do_something | do_something_else | ...
```

Composition happens outside the application code

Errors are printed to stderr, execution continues. It'll do...

ASIDE ON MODULARITY

ASIDE ON MODULARITY

- Modularity in code
  - Logically simpler functions, more easily grokked + tested
  - Smaller functions more easily reused throughout program, DRY

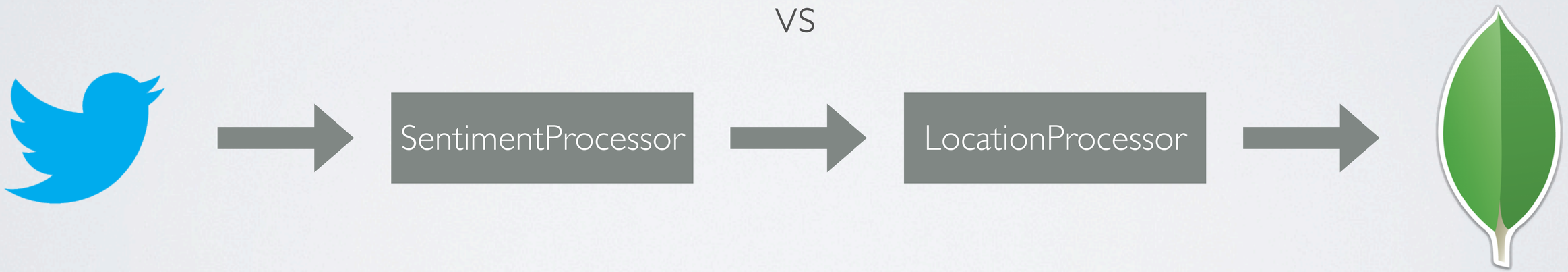
ASIDE ON MODULARITY

- Modularity in code
  - Logically simpler functions, more easily grokked + tested
  - Smaller functions more easily reused throughout program, DRY
- Modularity in architecture
  - Fine grained scaling of individual components
  - Isolate failures
  - All of the above

BIG LEAGUES: TRENDRR STACK VERSION

```
def process_tweet(tweet):
  get_sentiment()
  get_location()
  ...
```

VS



“QUEUEREADER” applications consume messages generated as outlined above

“QUEUEREADER” applications consume messages generated as outlined above

- May modify messages and send further downstream

“QUEUEREADER” applications consume messages generated as outlined above

- May modify messages and send further downstream
- May update some sort of database

"QUEUEREADER" applications consume messages generated as outlined above

- May modify messages and send further downstream
- May update some sort of database
- Probably a good idea to do some archival as well

ARCHIVAL GOODIES

ARCHIVAL GOODIES

- Backfill new systems

ARCHIVAL GOODIES

- Backfill new systems
- Repair busted systems

ARCHIVAL GOODIES

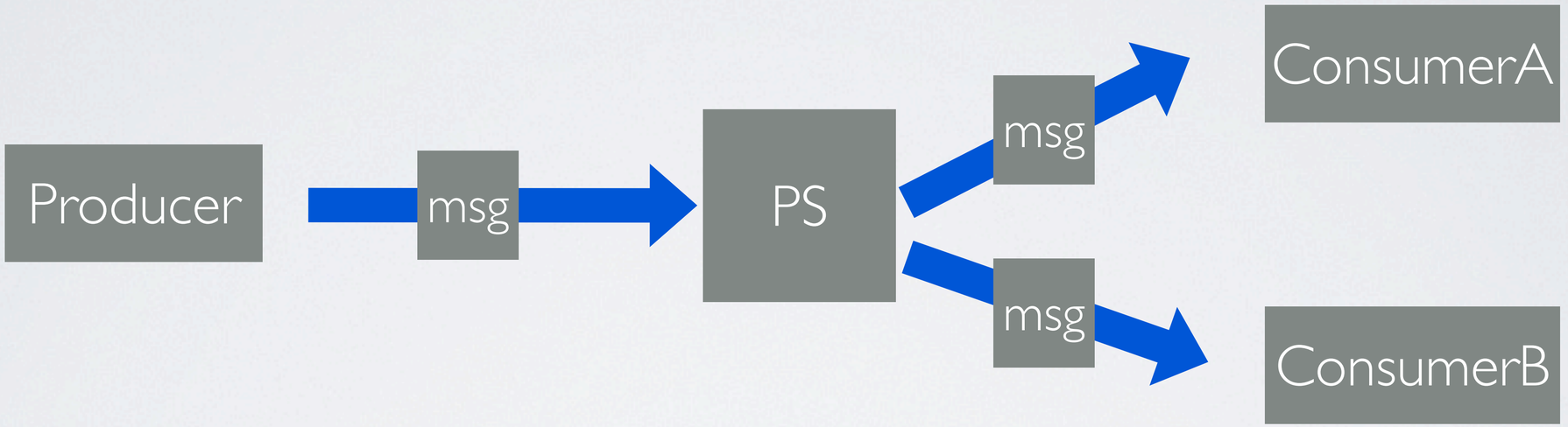
- Backfill new systems
- Repair busted systems
- Ripe for batch processing

ARCHIVAL GOODIES

- Backfill new systems
- Repair busted systems
- Ripe for batch processing
- Include timestamps in your messages!

COMPOSITION BUILDING BLOCKS

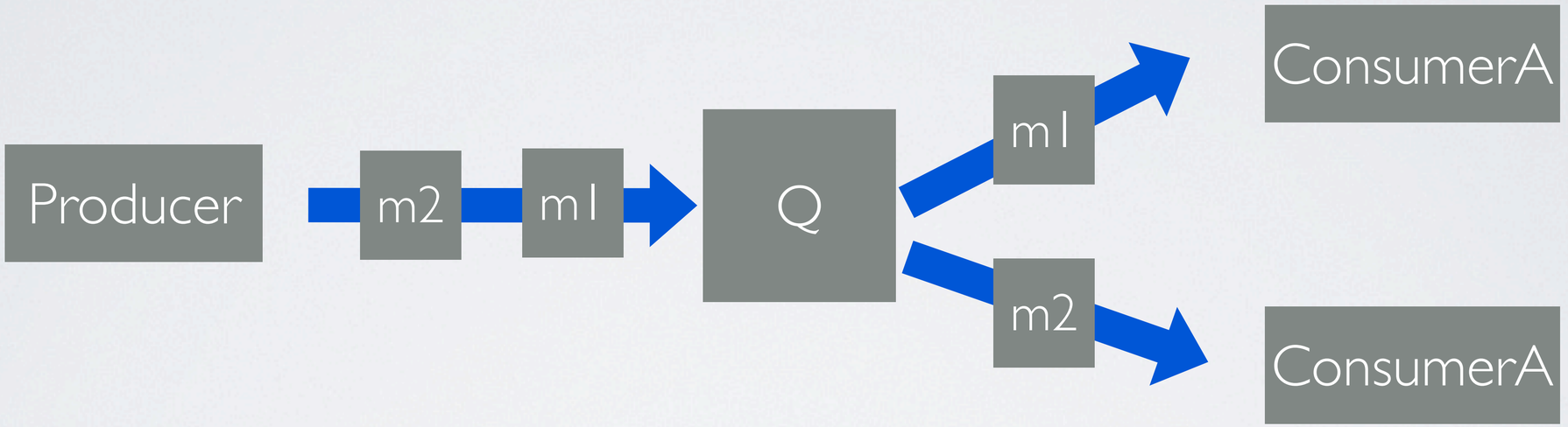
Pubsub / Multicast Model



Messages duplicated to multiple consumers

Decouple independent stream operations

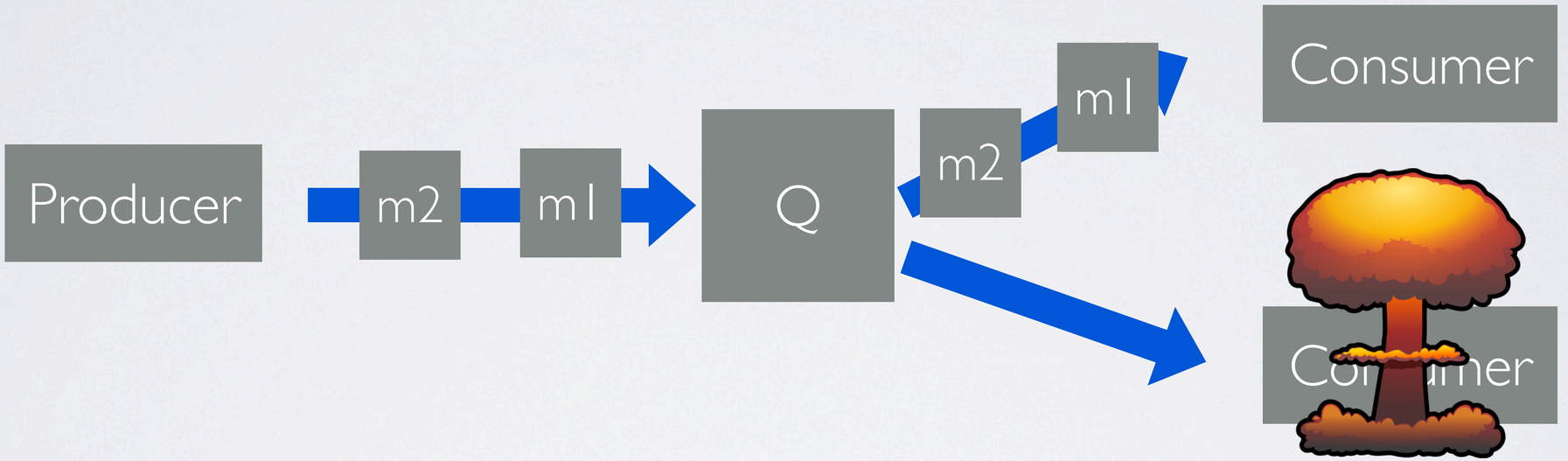
Distribution Model



Messages distributed among consumers

Horizontally scale workers to achieve desired throughput

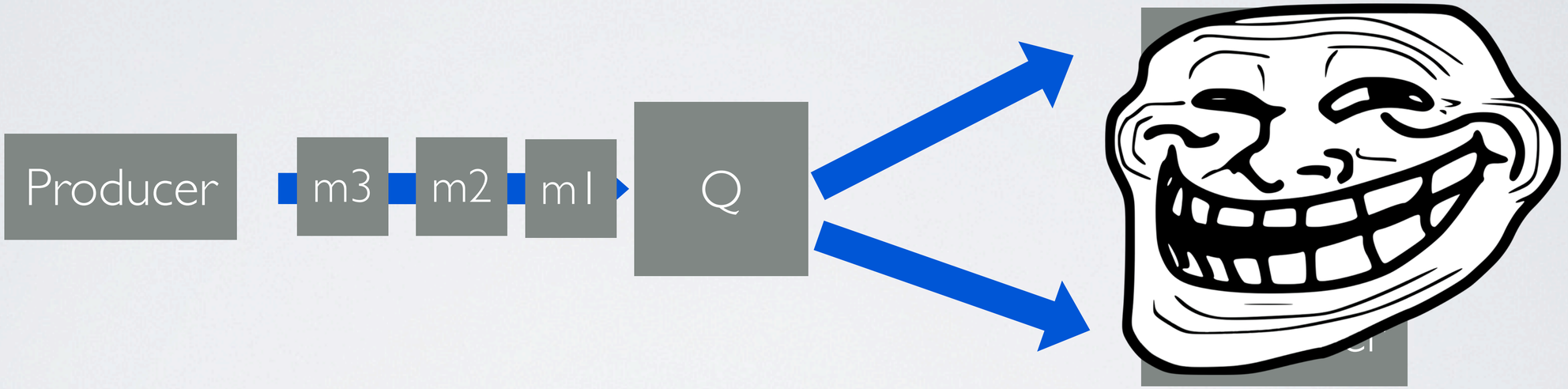
Distribution Model



Fault Tolerance:

In face of consumer failure, other consumers (try to) pick up the slack

Buffered Model



Buffering:

If consumers cannot keep up with producers, the queue is able to hold onto messages so they can be processed later

MAKE IT WEBSCALE!!!

what does this have to do with my webapp?

MAKE IT WEBSCALE!!!

what does this have to do with my webapp?

Web requests are serialized as event messages

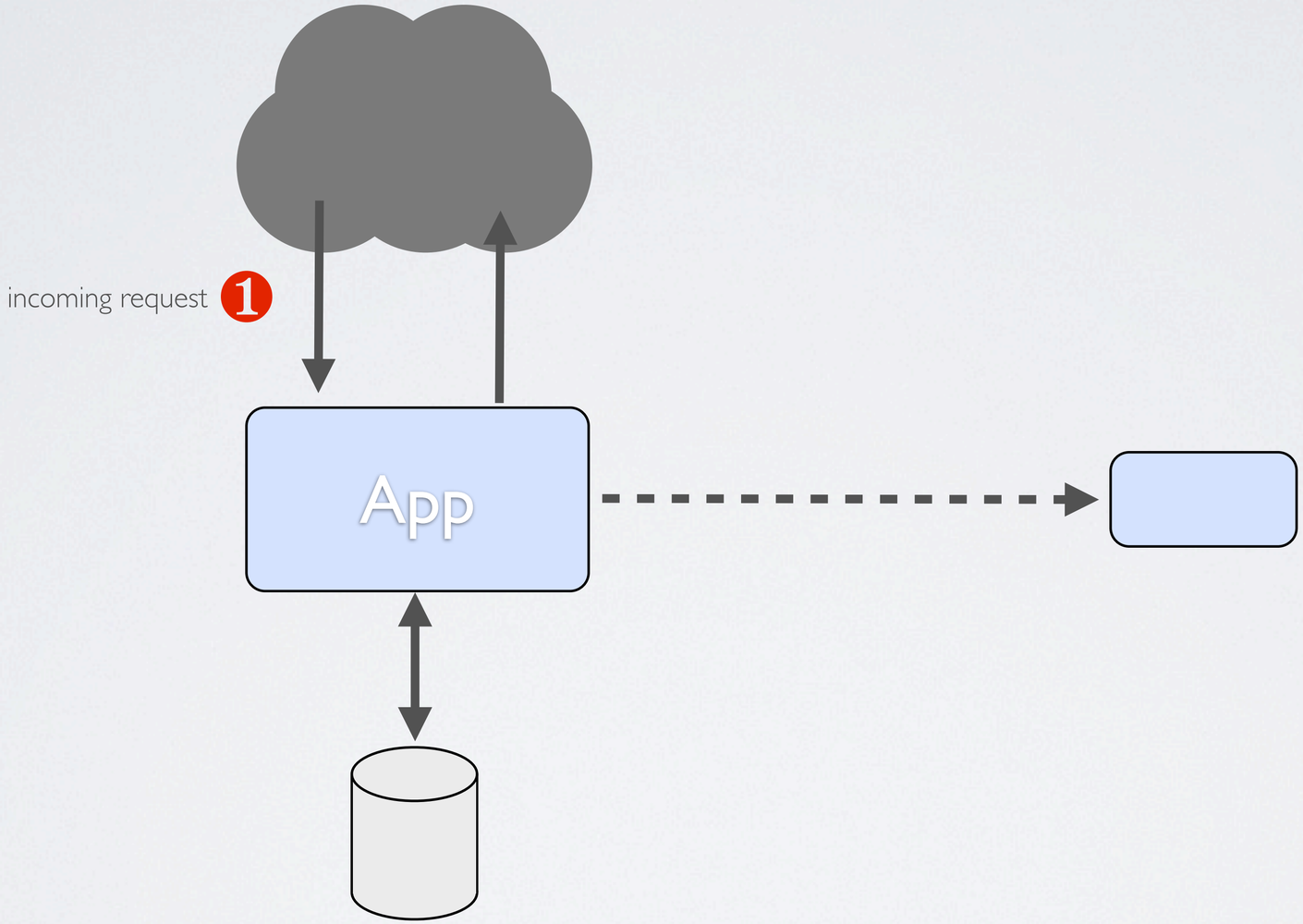
MAKE IT WEBSCALE!!!

what does this have to do with my webapp?

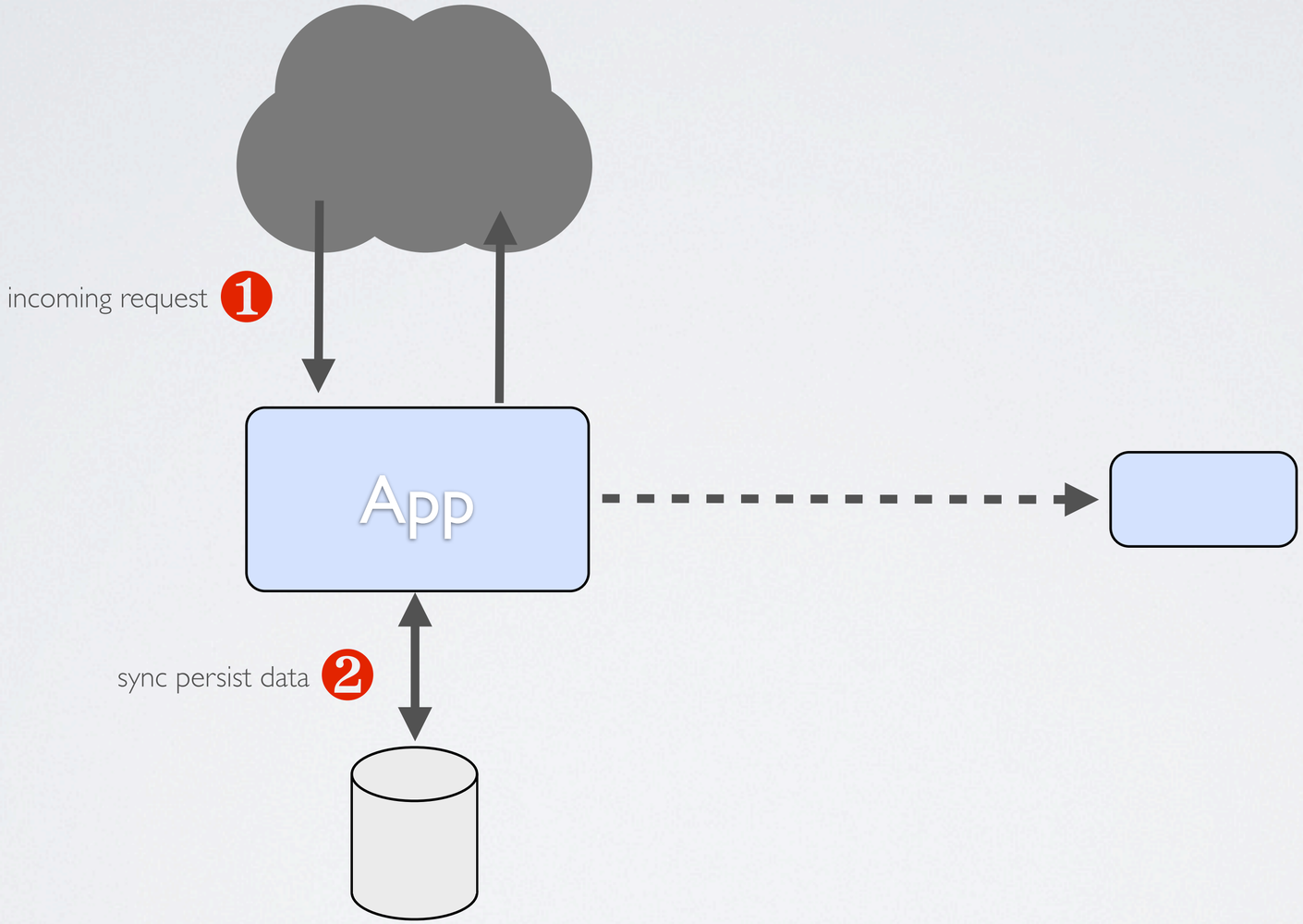
Web requests are serialized as event messages

Messages make up a stream that can be processed elsewhere in your distributed application

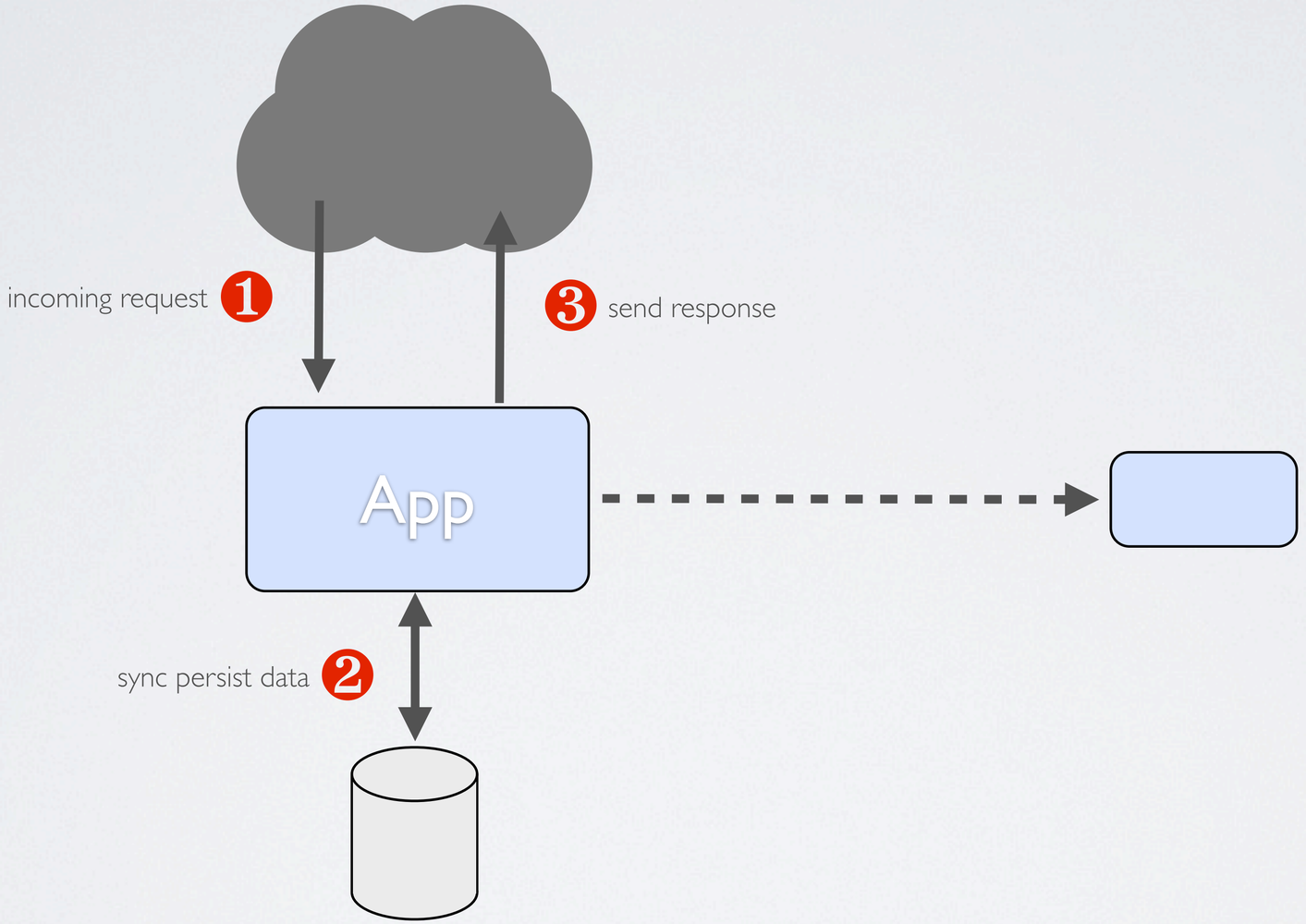
ASYNC DATA FLOW



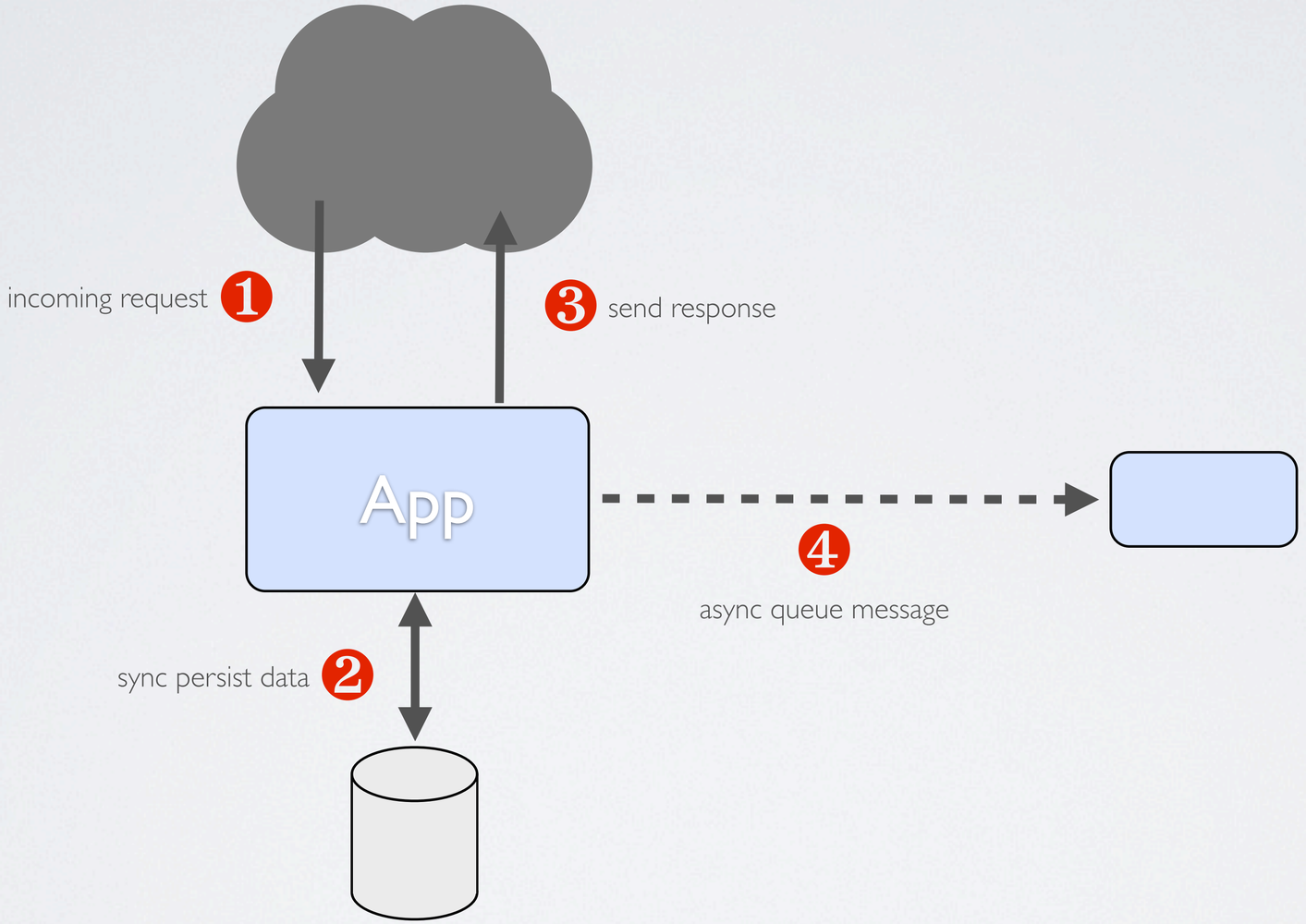
ASYNC DATA FLOW



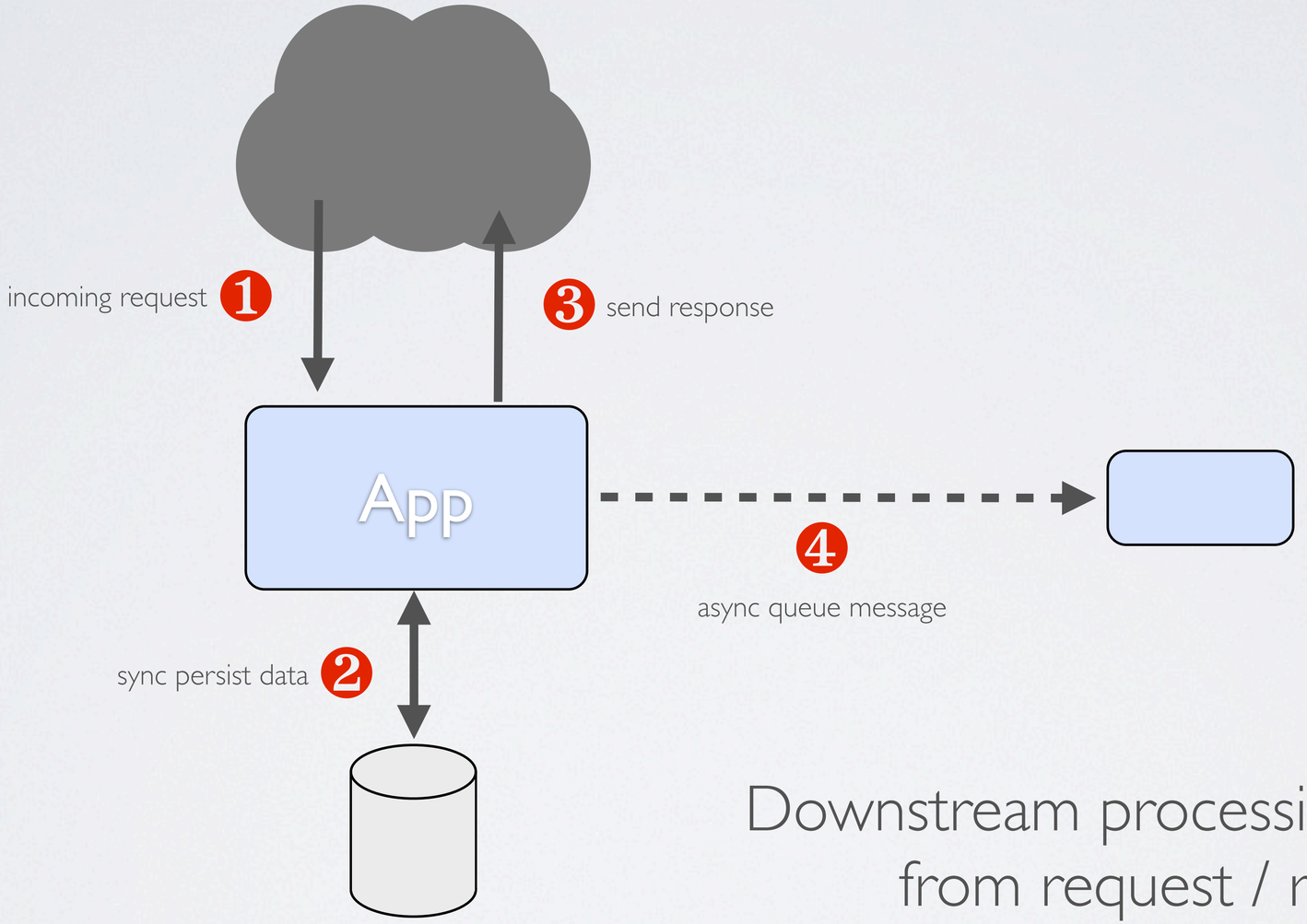
ASYNC DATA FLOW



ASYNC DATA FLOW



ASYNC DATA FLOW

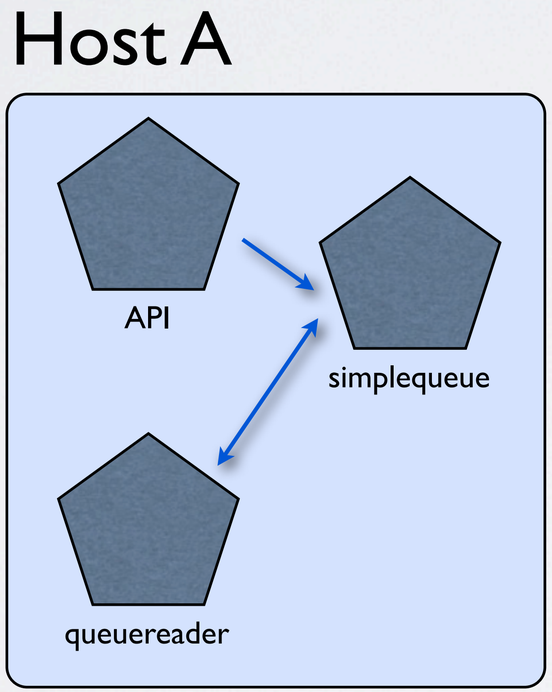


Downstream processing decoupled
from request / response

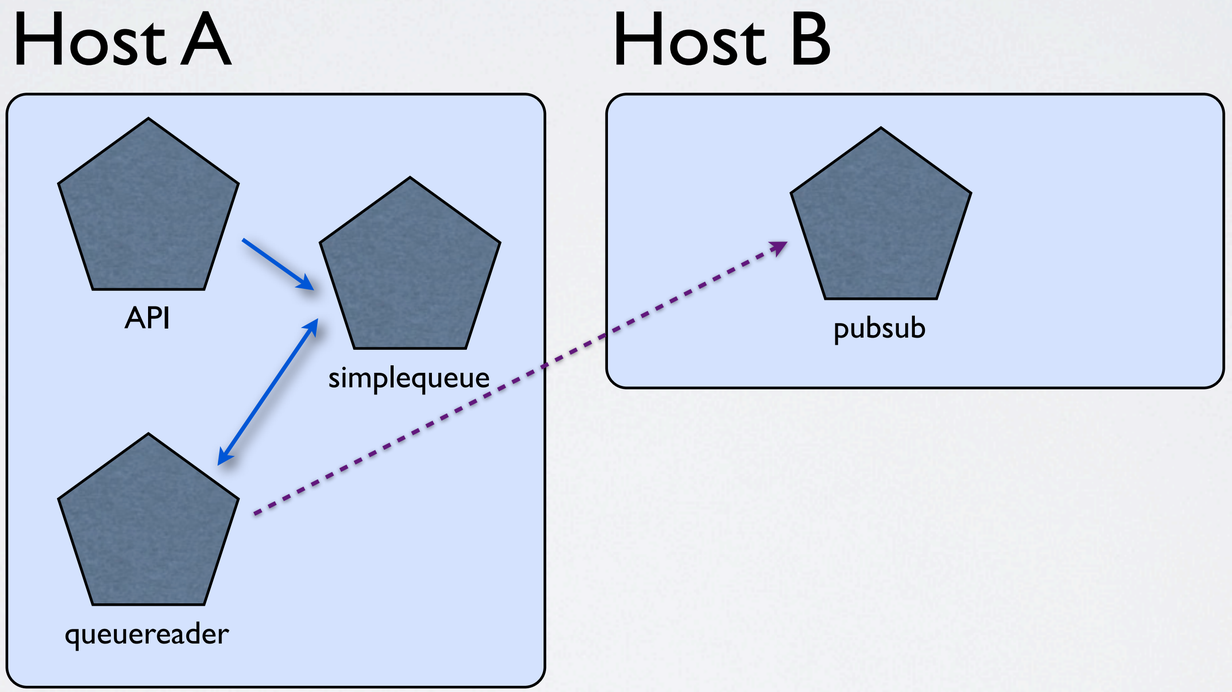
IT'S NICE BUT

- Stringing together queues and pubsubs implementing these models a pain
- Single conduit for messages a SPOF
- Single queue leads to rigid dependencies between services

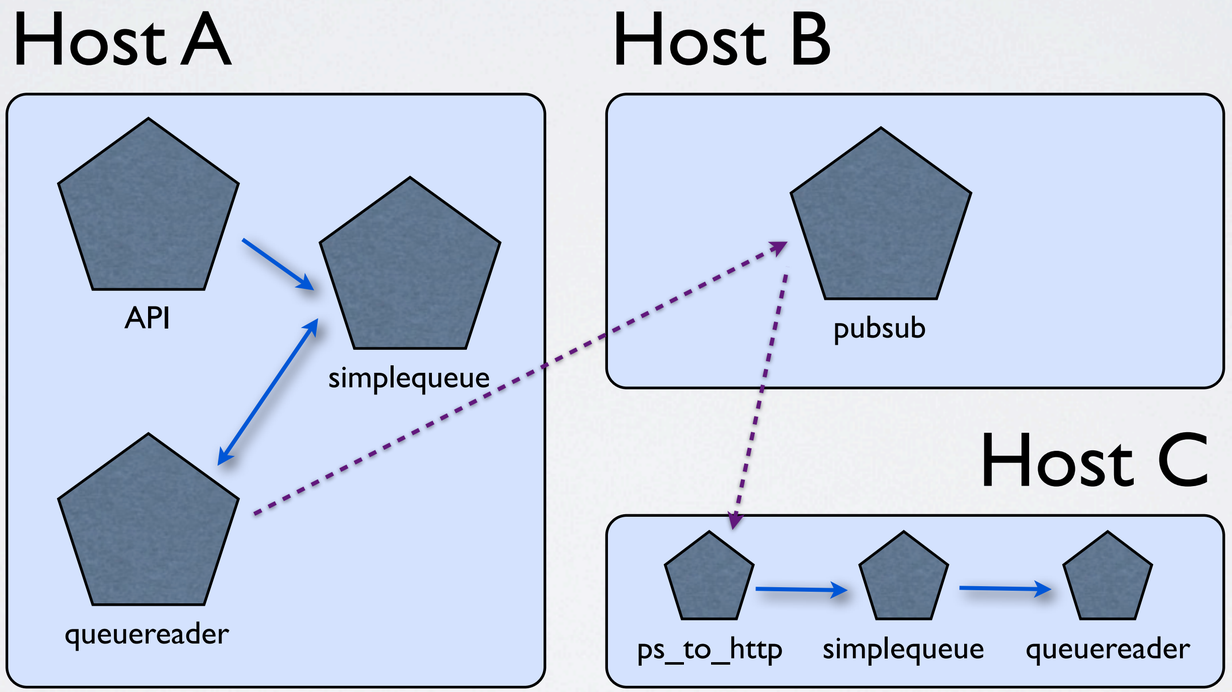
TYPICAL (OLD) ARCHITECTURE



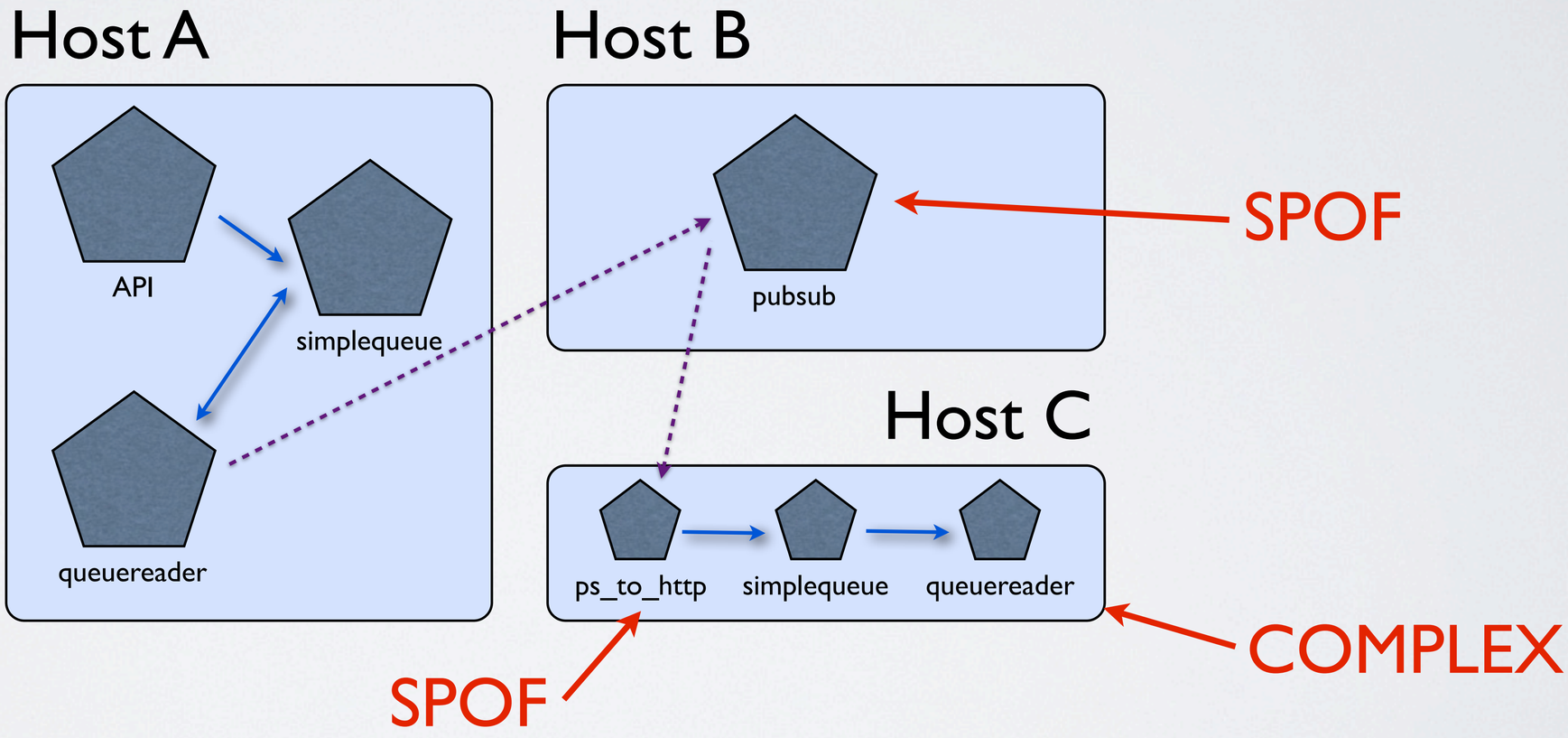
TYPICAL (OLD) ARCHITECTURE



TYPICAL (OLD) ARCHITECTURE

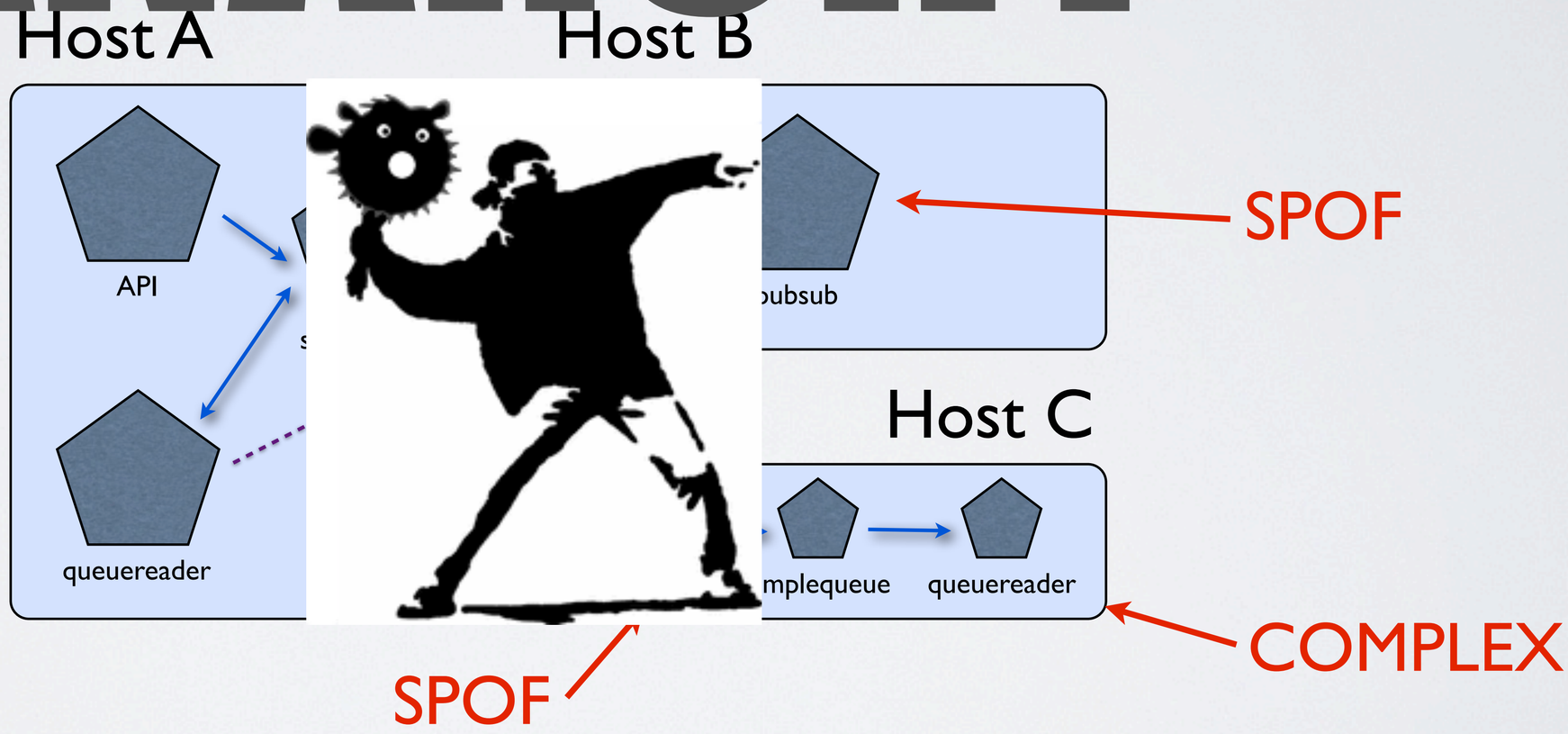


TYPICAL (OLD) ARCHITECTURE



TYPICAL (OLD) ARCHITECTURE

ANARCHY



I WANT IT ALL


NSQ
BILLIONS AND BILLIONS
OF MESSAGES SERVED

NSQ Core Features

NSQ Core Features

Queue daemon facilitates multicast,
distribution, **and** buffering

NSQ Core Features

Queue daemon facilitates multicast,
distribution, **and** buffering

Fully distributed and decentralized

NSQ Core Features

Queue daemon facilitates multicast,
distribution, **and** buffering

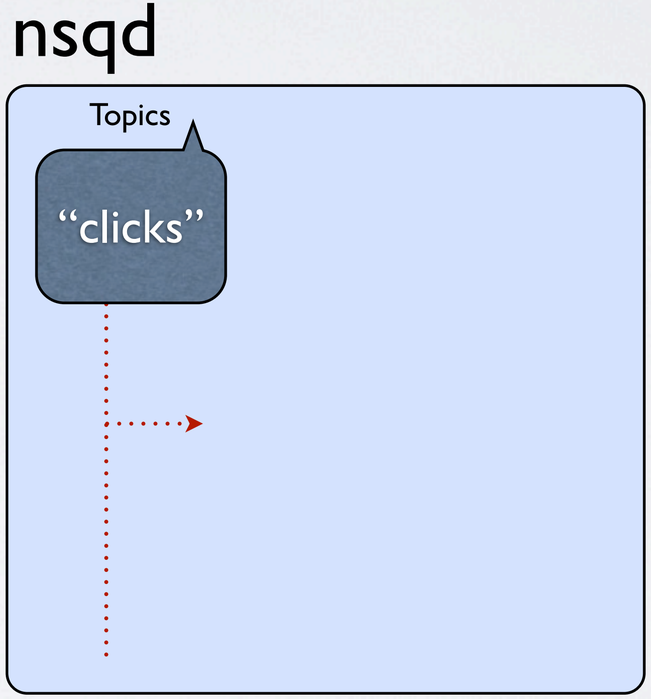
Fully distributed and decentralized

Lookup service simplifies configuration
and allows topology to change dynamically

MULTICAST **AND** BUFFERING, YOU SAY?

NSQ CONCEPTS AND MESSAGE FLOW

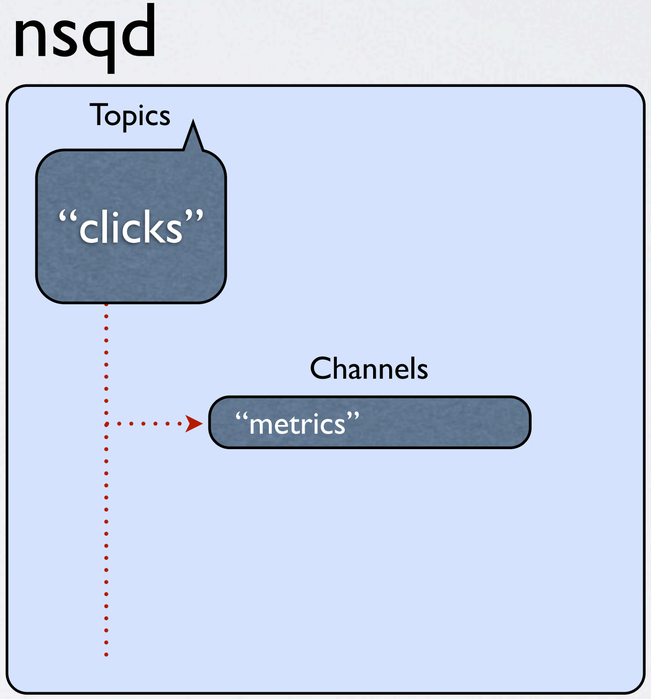
- a **topic** is a distinct stream of messages (a single **nsqd** instance can have multiple **topics**)
- a **channel** is an independent queue for a **topic** (a **topic** can have multiple **channels**)
- consumers discover producers by querying **nsqlookupd** (a discovery service for topics)
- **topics** and **channels** are created at runtime (just start publishing/subscribing)



MULTICAST **AND** BUFFERING, YOU SAY?

NSQ CONCEPTS AND MESSAGE FLOW

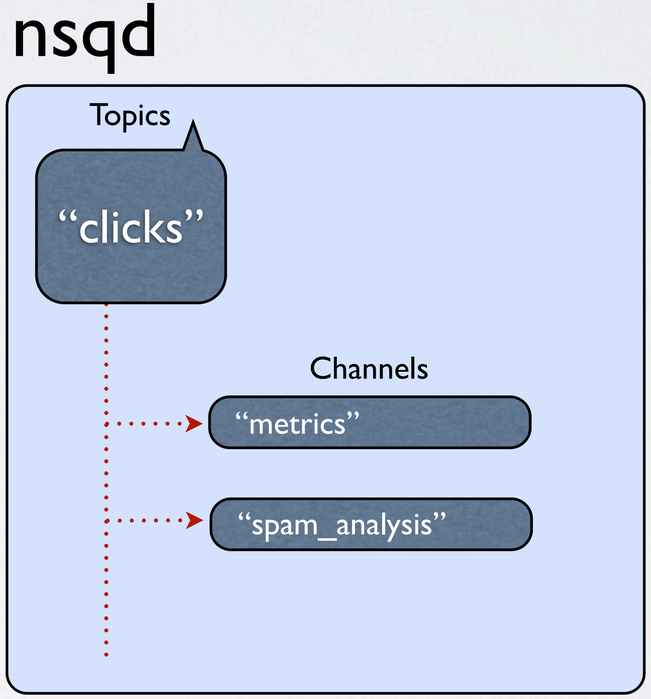
- a **topic** is a distinct stream of messages (a single **nsqd** instance can have multiple **topics**)
- a **channel** is an independent queue for a **topic** (a **topic** can have multiple **channels**)
- consumers discover producers by querying **nsqlookupd** (a discovery service for topics)
- **topics** and **channels** are created at runtime (just start publishing/subscribing)



MULTICAST **AND** BUFFERING, YOU SAY?

NSQ CONCEPTS AND MESSAGE FLOW

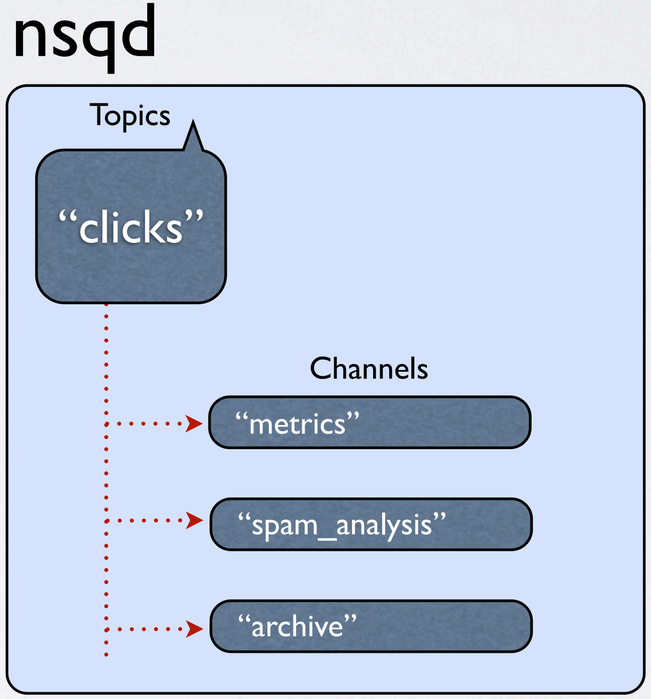
- a **topic** is a distinct stream of messages (a single **nsqd** instance can have multiple **topics**)
- a **channel** is an independent queue for a **topic** (a **topic** can have multiple **channels**)
- consumers discover producers by querying **nsqlookupd** (a discovery service for topics)
- **topics** and **channels** are created at runtime (just start publishing/subscribing)



MULTICAST **AND** BUFFERING, YOU SAY?

NSQ CONCEPTS AND MESSAGE FLOW

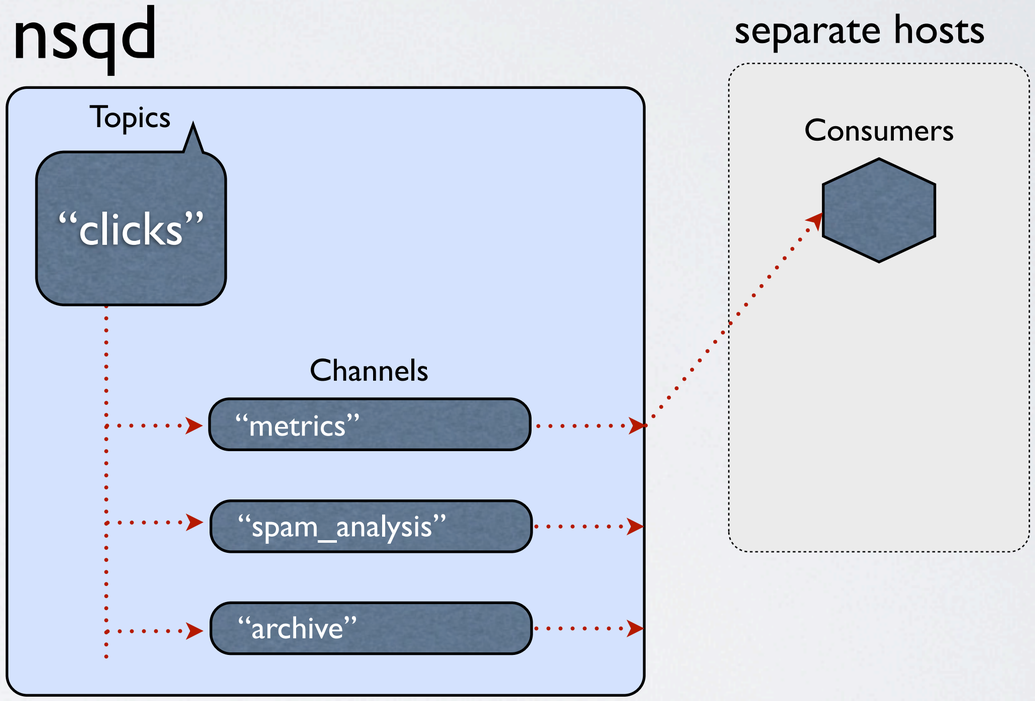
- a **topic** is a distinct stream of messages (a single **nsqd** instance can have multiple **topics**)
- a **channel** is an independent queue for a **topic** (a **topic** can have multiple **channels**)
- consumers discover producers by querying **nsqlookupd** (a discovery service for topics)
- **topics** and **channels** are created at runtime (just start publishing/subscribing)



MULTICAST **AND** BUFFERING, YOU SAY?

NSQ CONCEPTS AND MESSAGE FLOW

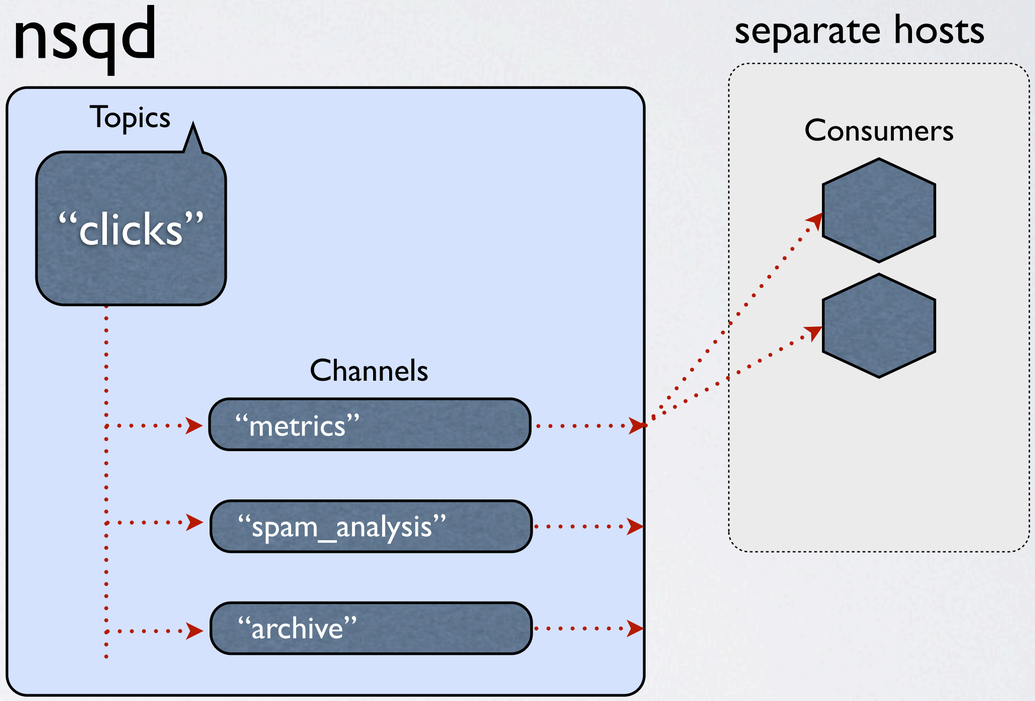
- a **topic** is a distinct stream of messages (a single **nsqd** instance can have multiple **topics**)
- a **channel** is an independent queue for a **topic** (a **topic** can have multiple **channels**)
- consumers discover producers by querying **nsqlookupd** (a discovery service for topics)
- **topics** and **channels** are created at runtime (just start publishing/subscribing)



MULTICAST **AND** BUFFERING, YOU SAY?

NSQ CONCEPTS AND MESSAGE FLOW

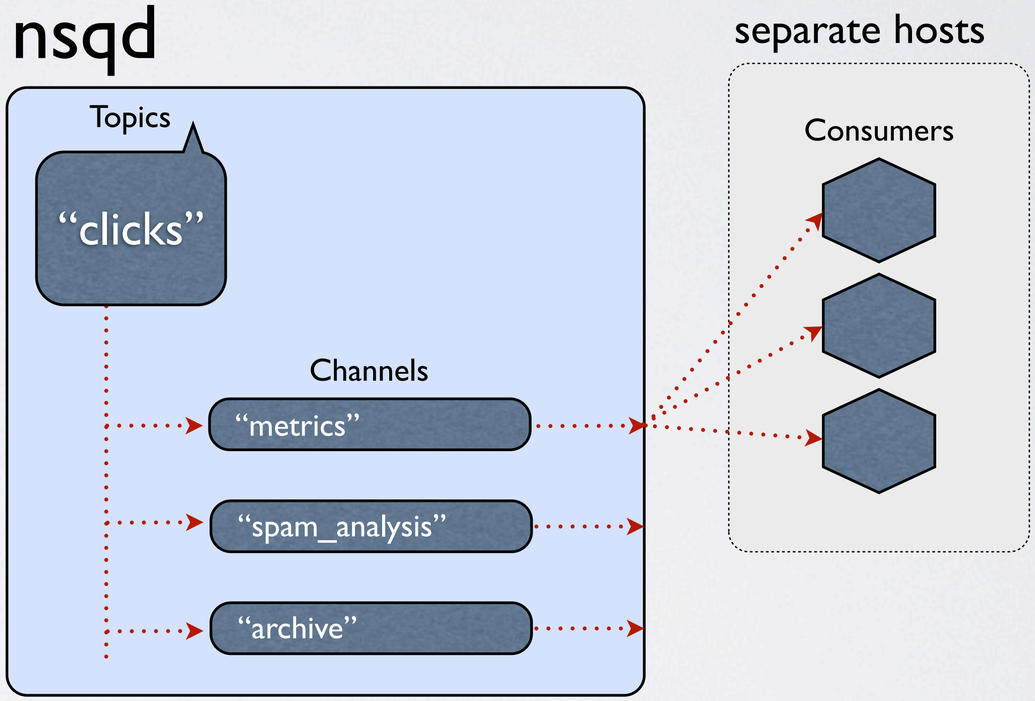
- a **topic** is a distinct stream of messages (a single **nsqd** instance can have multiple **topics**)
- a **channel** is an independent queue for a **topic** (a **topic** can have multiple **channels**)
- consumers discover producers by querying **nsqlookupd** (a discovery service for topics)
- **topics** and **channels** are created at runtime (just start publishing/subscribing)



MULTICAST **AND** BUFFERING, YOU SAY?

NSQ CONCEPTS AND MESSAGE FLOW

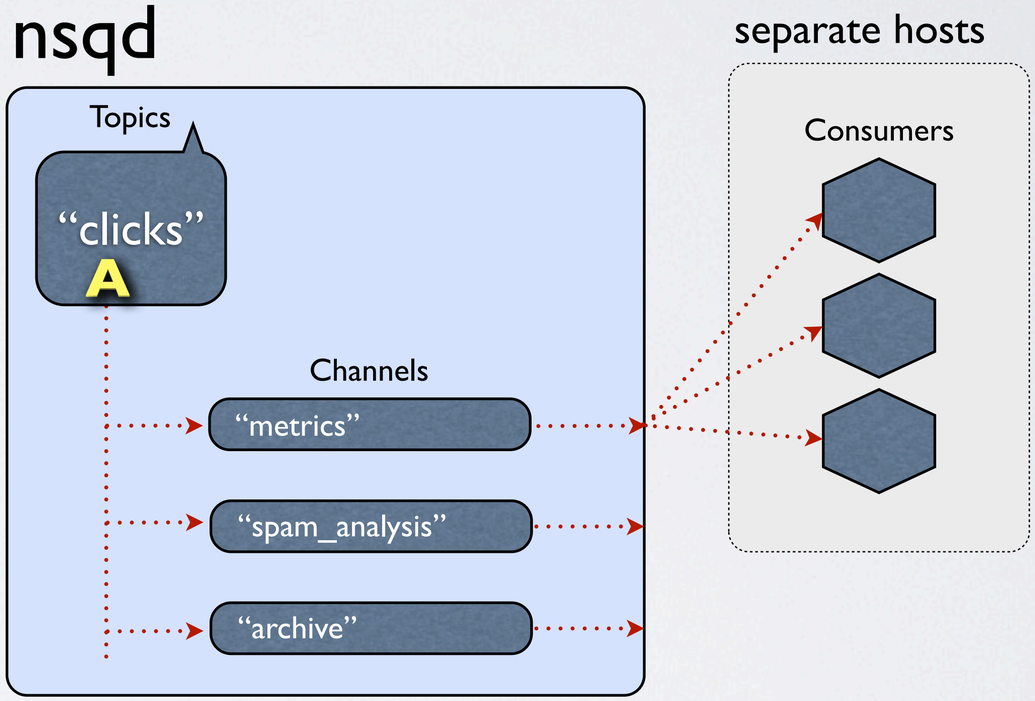
- a **topic** is a distinct stream of messages (a single **nsqd** instance can have multiple **topics**)
- a **channel** is an independent queue for a **topic** (a **topic** can have multiple **channels**)
- consumers discover producers by querying **nsqlookupd** (a discovery service for topics)
- **topics** and **channels** are created at runtime (just start publishing/subscribing)



MULTICAST **AND** BUFFERING, YOU SAY?

NSQ CONCEPTS AND MESSAGE FLOW

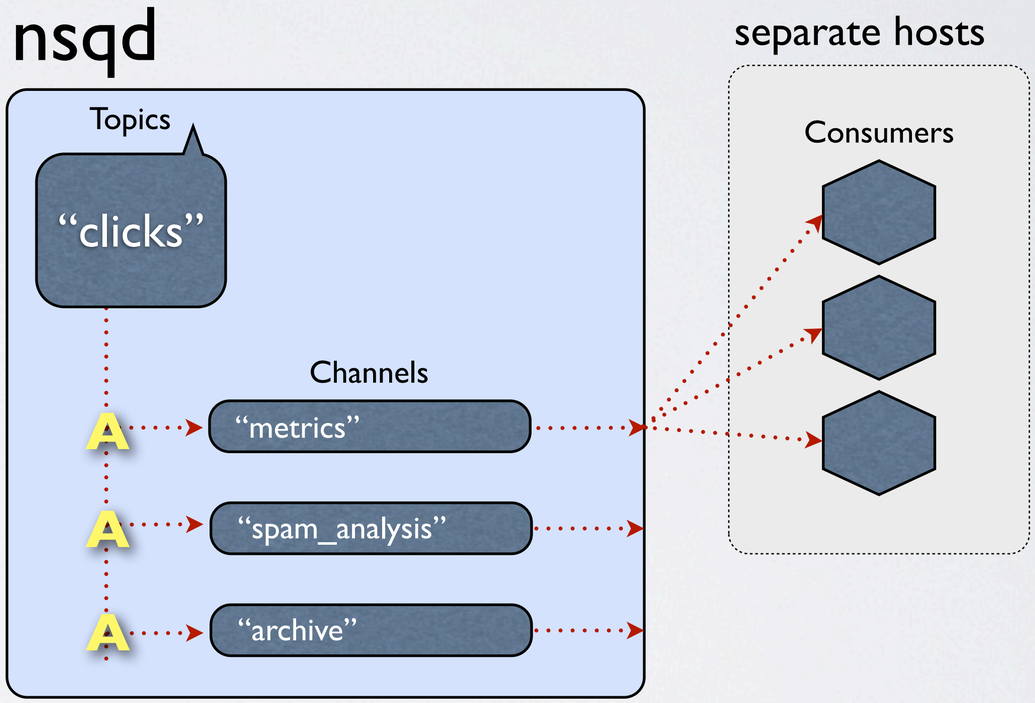
- a **topic** is a distinct stream of messages (a single **nsqd** instance can have multiple **topics**)
- a **channel** is an independent queue for a **topic** (a **topic** can have multiple **channels**)
- consumers discover producers by querying **nsqlookupd** (a discovery service for topics)
- **topics** and **channels** are created at runtime (just start publishing/subscribing)



MULTICAST **AND** BUFFERING, YOU SAY?

NSQ CONCEPTS AND MESSAGE FLOW

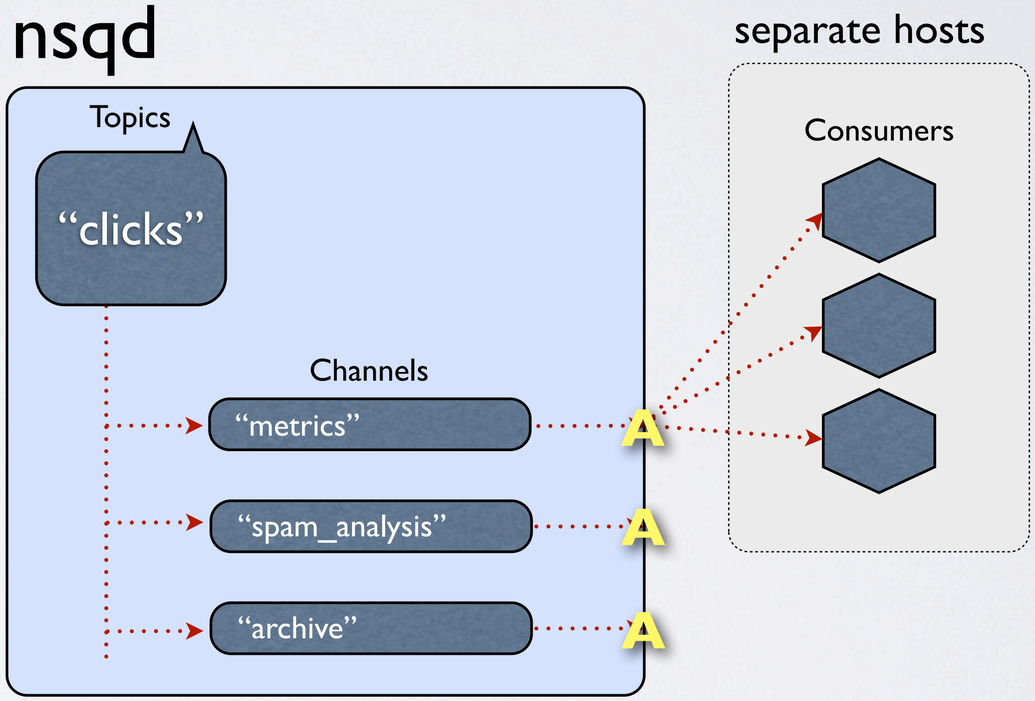
- a **topic** is a distinct stream of messages (a single **nsqd** instance can have multiple **topics**)
- a **channel** is an independent queue for a **topic** (a **topic** can have multiple **channels**)
- consumers discover producers by querying **nsqlookupd** (a discovery service for topics)
- **topics** and **channels** are created at runtime (just start publishing/subscribing)



MULTICAST **AND** BUFFERING, YOU SAY?

NSQ CONCEPTS AND MESSAGE FLOW

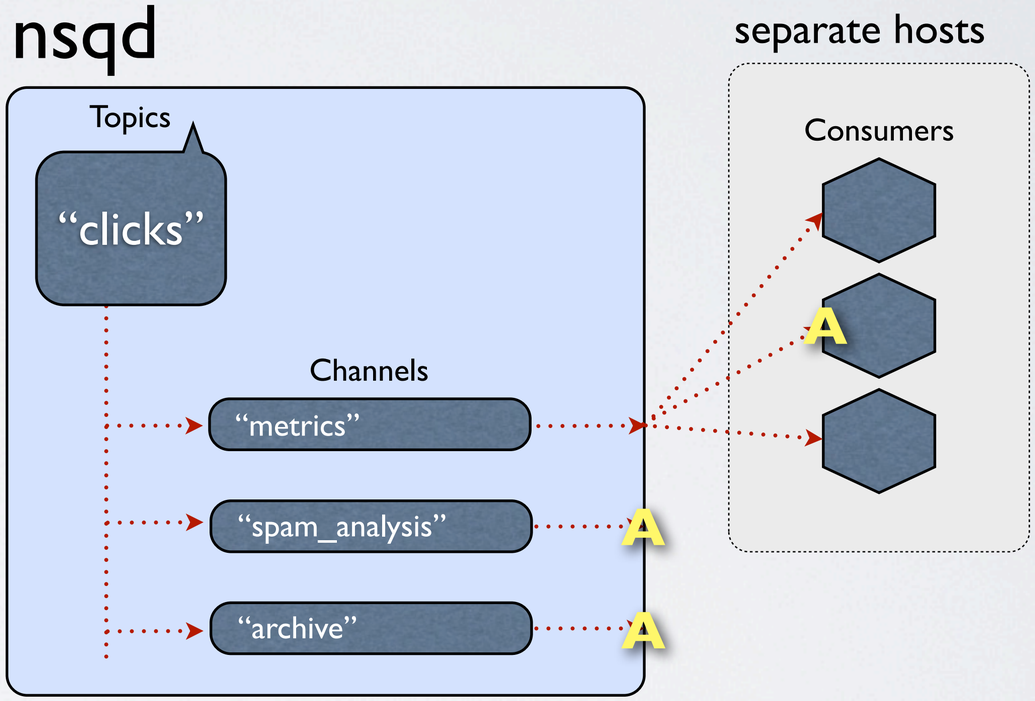
- a **topic** is a distinct stream of messages (a single **nsqd** instance can have multiple **topics**)
- a **channel** is an independent queue for a **topic** (a **topic** can have multiple **channels**)
- consumers discover producers by querying **nsqlookupd** (a discovery service for topics)
- **topics** and **channels** are created at runtime (just start publishing/subscribing)



MULTICAST **AND** BUFFERING, YOU SAY?

NSQ CONCEPTS AND MESSAGE FLOW

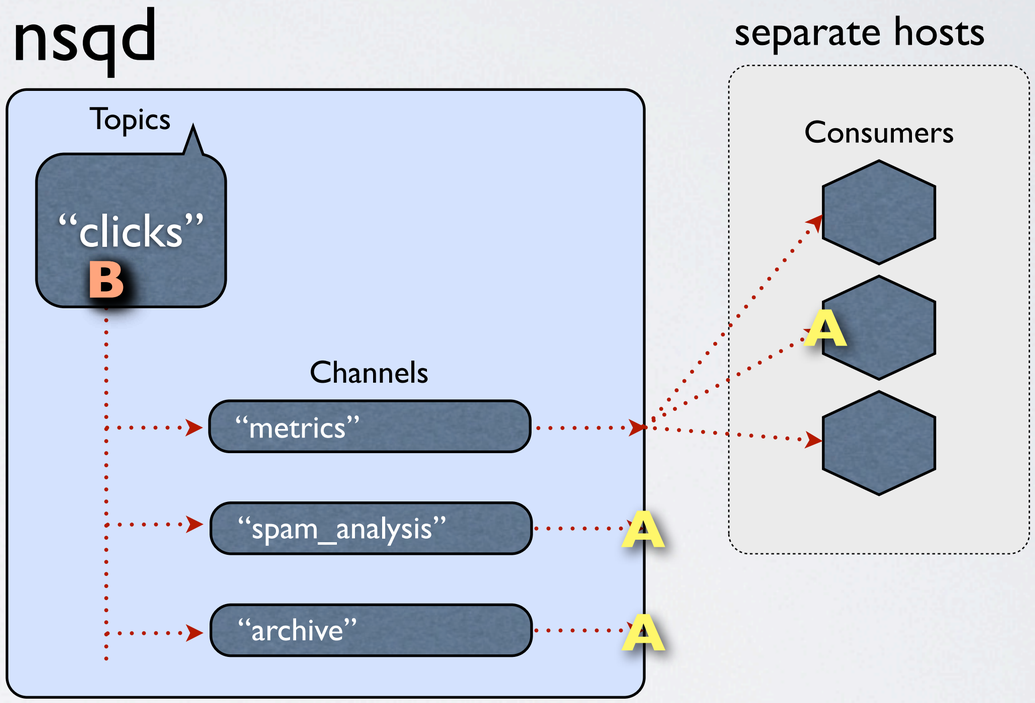
- a **topic** is a distinct stream of messages (a single **nsqd** instance can have multiple **topics**)
- a **channel** is an independent queue for a **topic** (a **topic** can have multiple **channels**)
- consumers discover producers by querying **nsqlookupd** (a discovery service for topics)
- **topics** and **channels** are created at runtime (just start publishing/subscribing)



MULTICAST **AND** BUFFERING, YOU SAY?

NSQ CONCEPTS AND MESSAGE FLOW

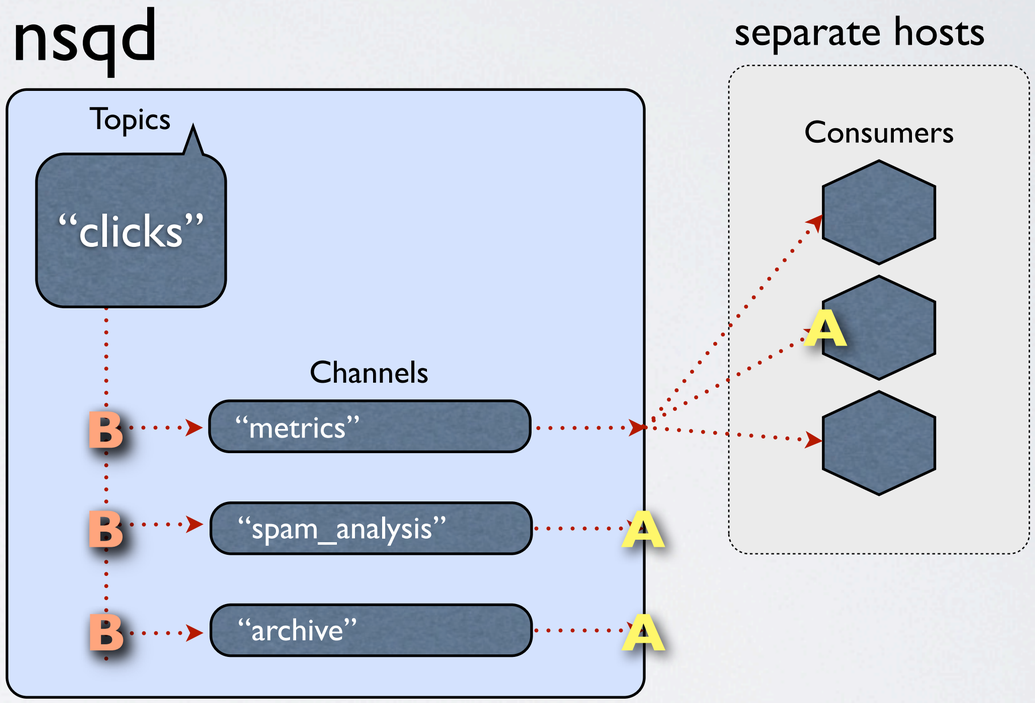
- a **topic** is a distinct stream of messages (a single **nsqd** instance can have multiple **topics**)
- a **channel** is an independent queue for a **topic** (a **topic** can have multiple **channels**)
- consumers discover producers by querying **nsqlookupd** (a discovery service for topics)
- **topics** and **channels** are created at runtime (just start publishing/subscribing)



MULTICAST **AND** BUFFERING, YOU SAY?

NSQ CONCEPTS AND MESSAGE FLOW

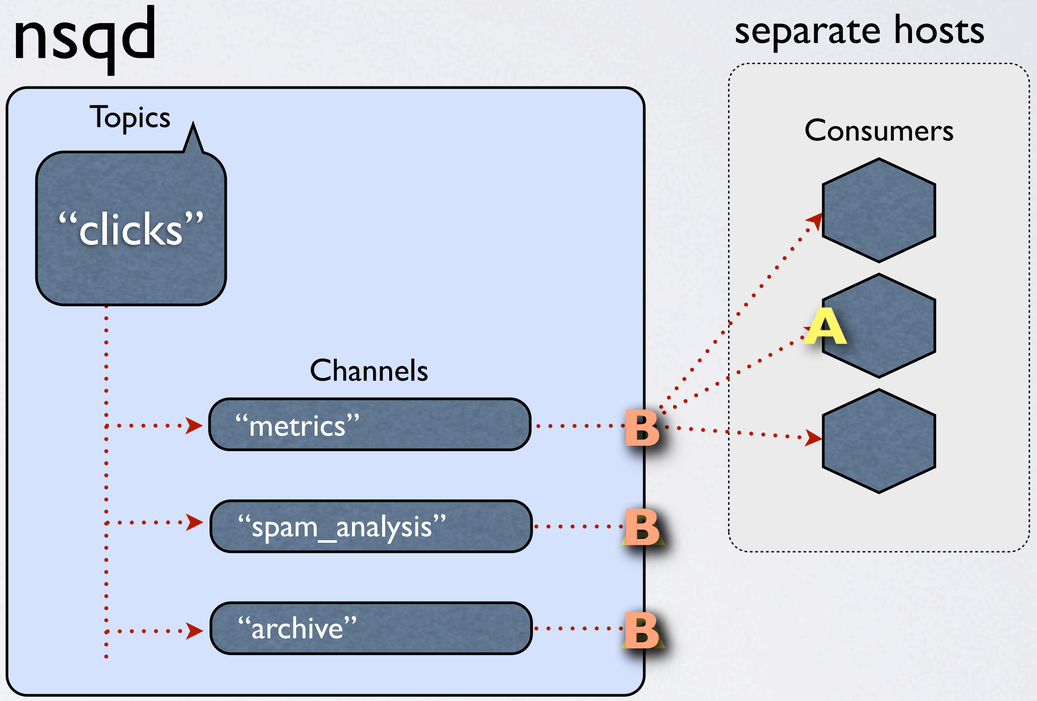
- a **topic** is a distinct stream of messages (a single **nsqd** instance can have multiple **topics**)
- a **channel** is an independent queue for a **topic** (a **topic** can have multiple **channels**)
- consumers discover producers by querying **nsqlookupd** (a discovery service for topics)
- **topics** and **channels** are created at runtime (just start publishing/subscribing)



MULTICAST **AND** BUFFERING, YOU SAY?

NSQ CONCEPTS AND MESSAGE FLOW

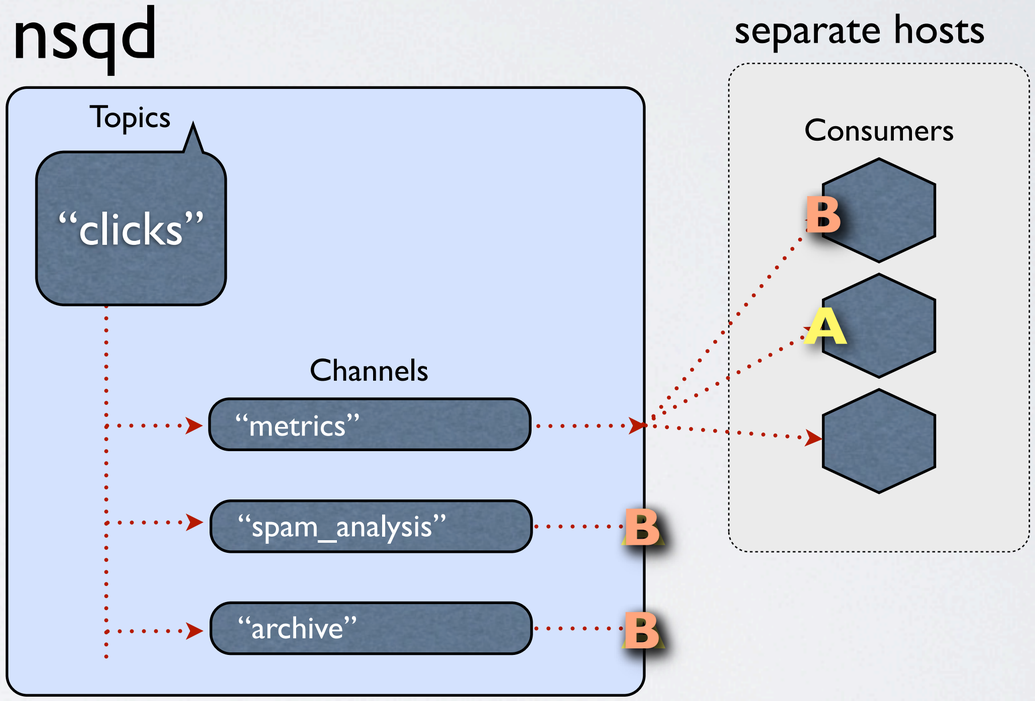
- a **topic** is a distinct stream of messages (a single **nsqd** instance can have multiple **topics**)
- a **channel** is an independent queue for a **topic** (a **topic** can have multiple **channels**)
- consumers discover producers by querying **nsqlookupd** (a discovery service for topics)
- **topics** and **channels** are created at runtime (just start publishing/subscribing)



MULTICAST **AND** BUFFERING, YOU SAY?

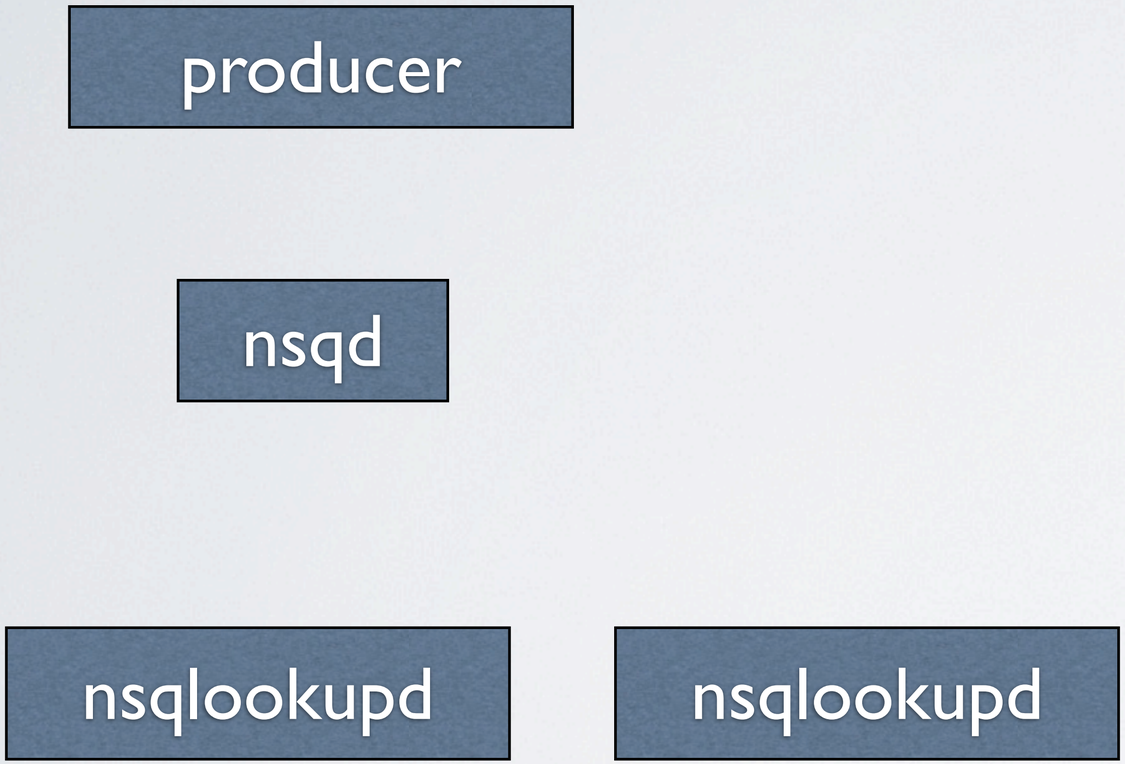
NSQ CONCEPTS AND MESSAGE FLOW

- a **topic** is a distinct stream of messages (a single **nsqd** instance can have multiple **topics**)
- a **channel** is an independent queue for a **topic** (a **topic** can have multiple **channels**)
- consumers discover producers by querying **nsqlookupd** (a discovery service for topics)
- **topics** and **channels** are created at runtime (just start publishing/subscribing)



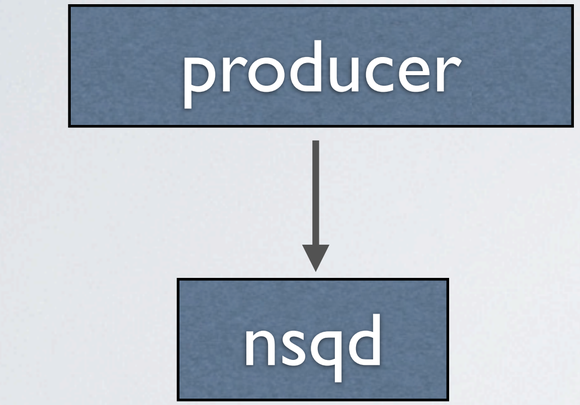
DISCOVERY

remove the need for publishers and consumers to know about each other



DISCOVERY

remove the need for publishers and consumers to know about each other



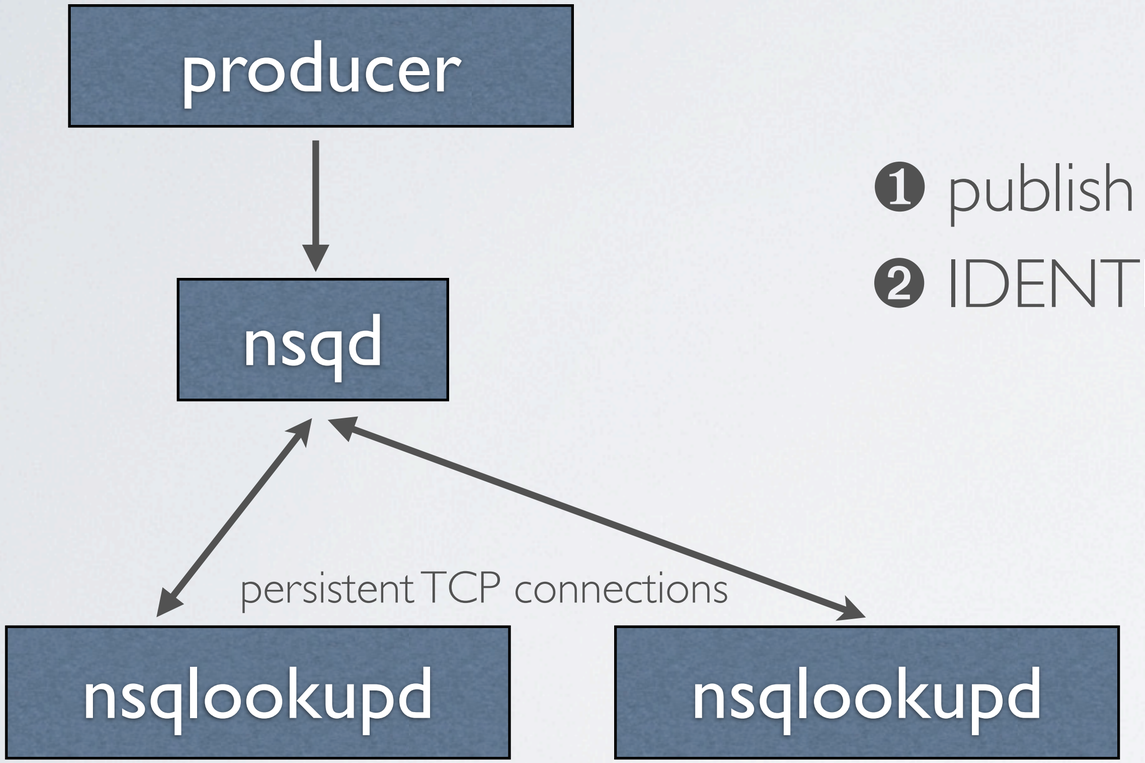
❶ publish msg (specifying topic)

nsqlookupd

nsqlookupd

DISCOVERY

remove the need for publishers and consumers to know about each other

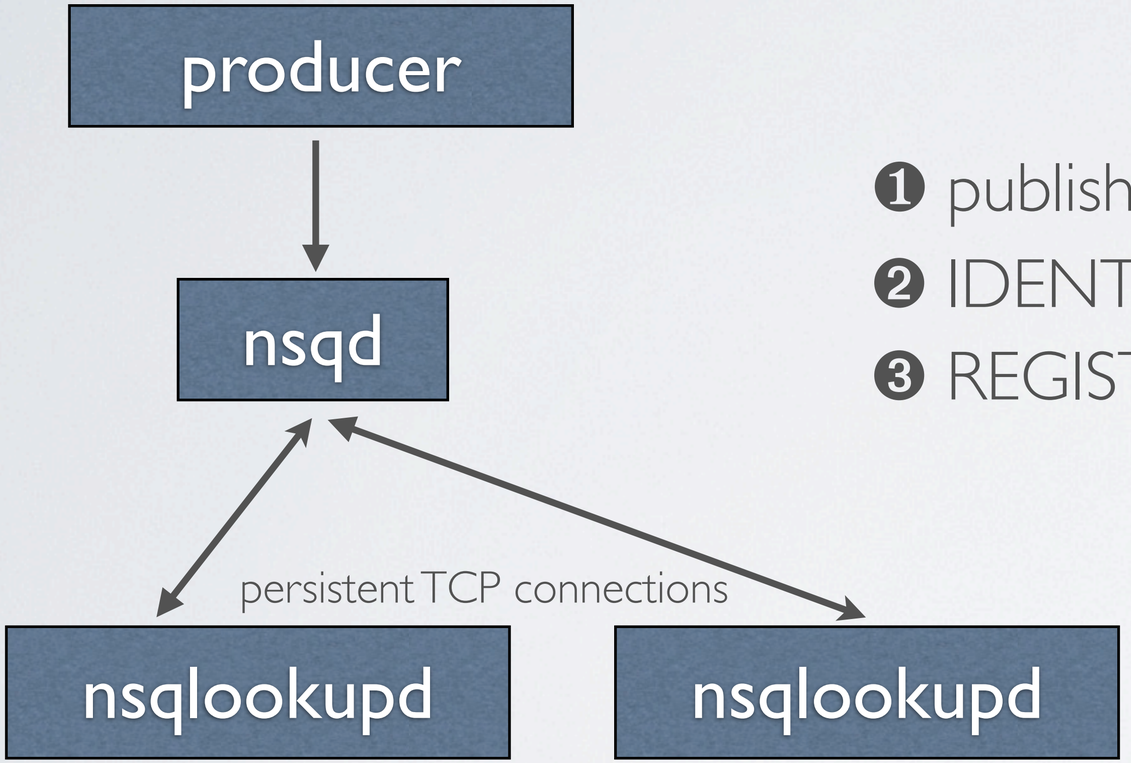


❶ publish msg (specifying topic)

❷ IDENTIFY

DISCOVERY

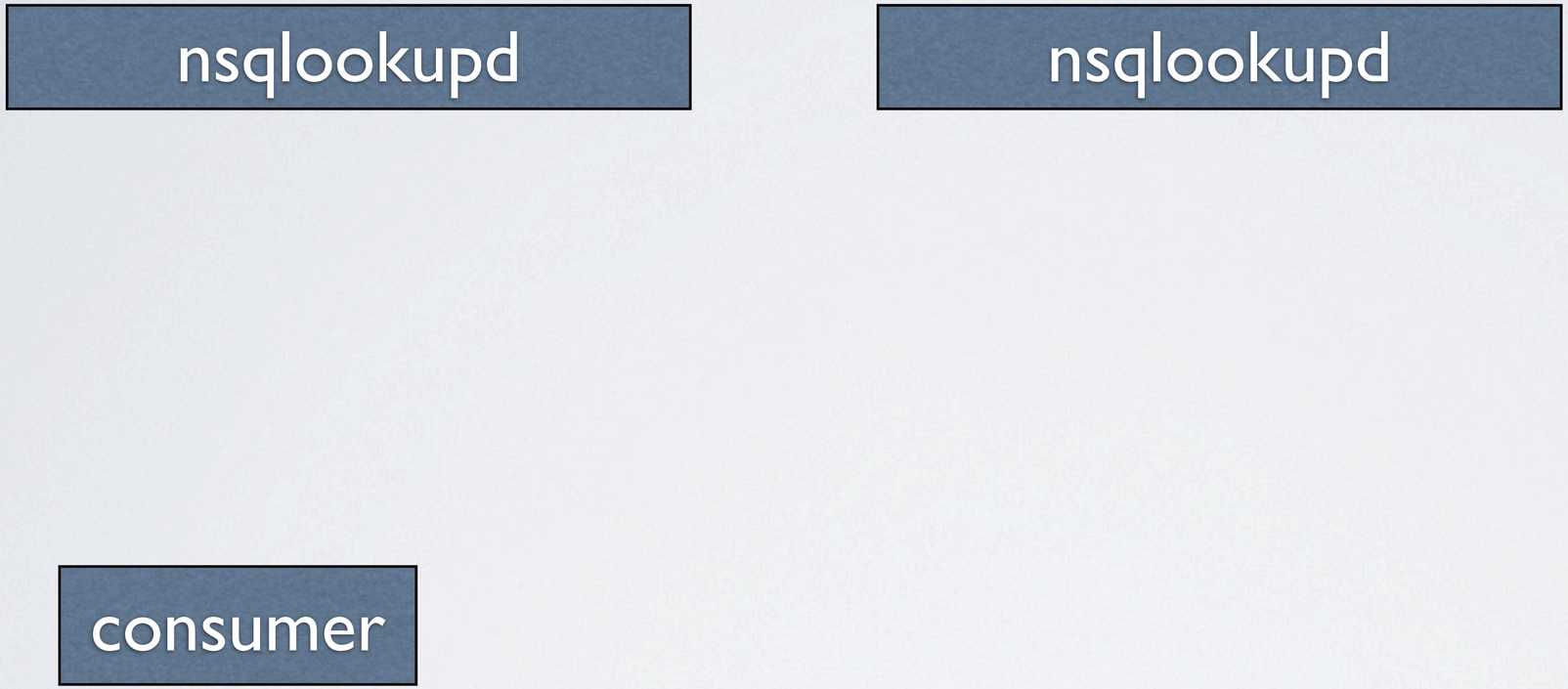
remove the need for publishers and consumers to know about each other



❶ publish msg (specifying topic)
❷ IDENTIFY
❸ REGISTER (topic/channel)

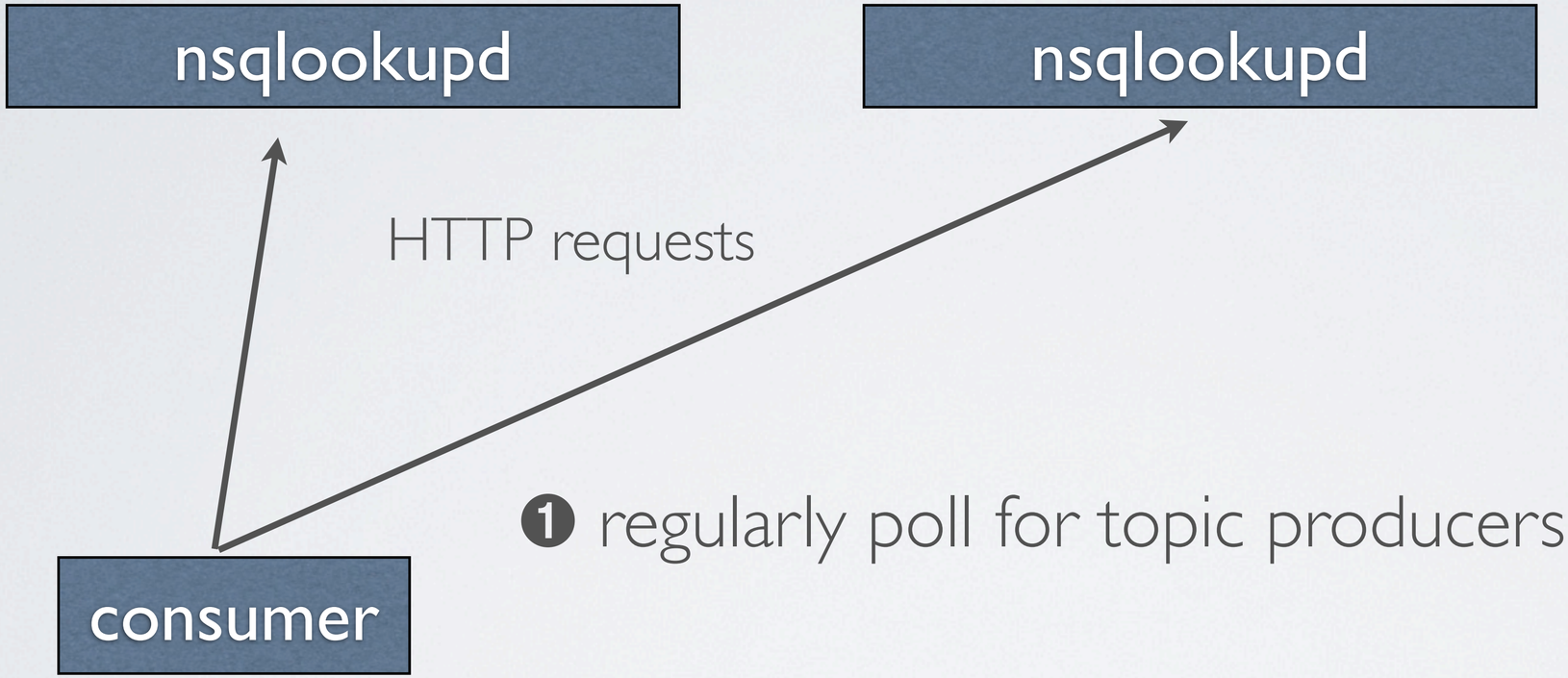
DISCOVERY (CLIENT)

remove the need for publishers and consumers to know about each other



DISCOVERY (CLIENT)

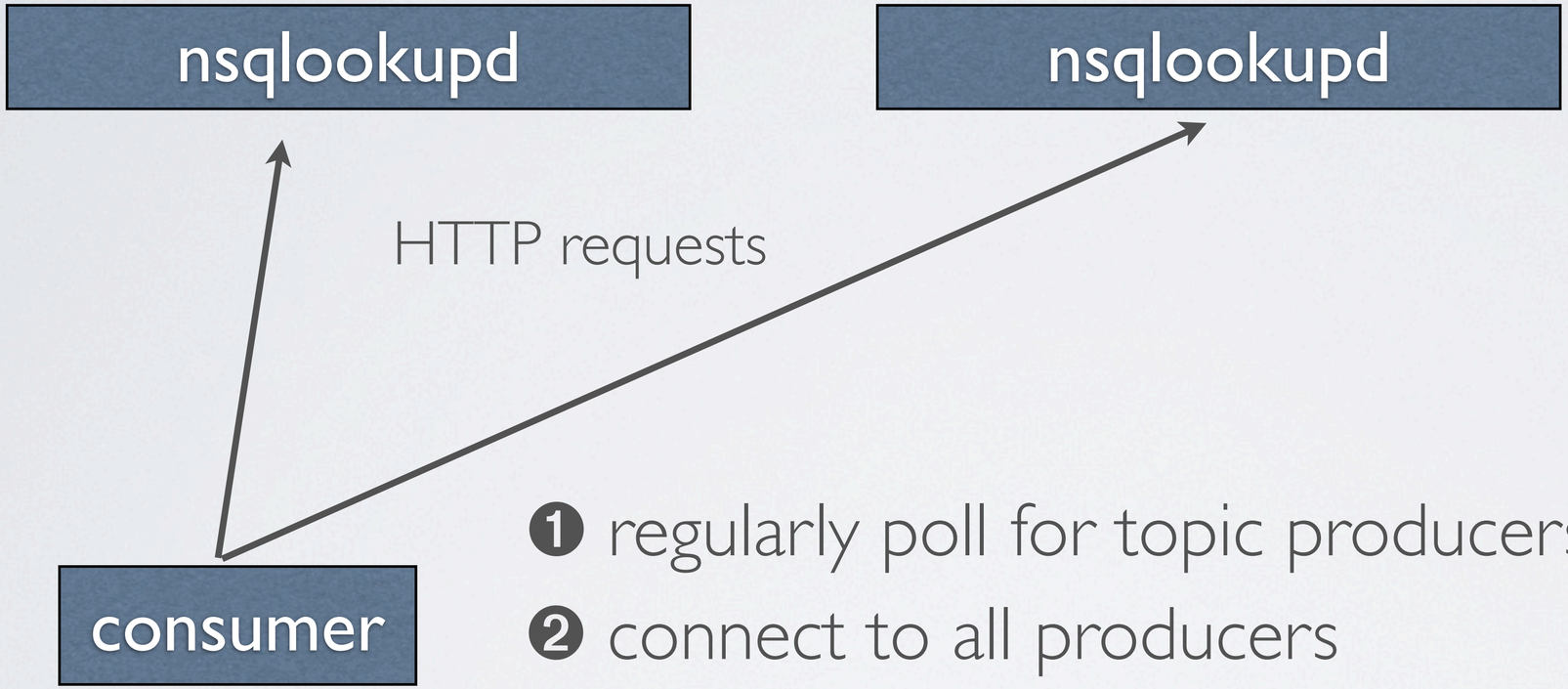
remove the need for publishers and consumers to know about each other



❶ regularly poll for topic producers

DISCOVERY (CLIENT)

remove the need for publishers and consumers to know about each other



❶ regularly poll for topic producers

❷ connect to all producers

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)

ELIMINATE ALL THE SPOF

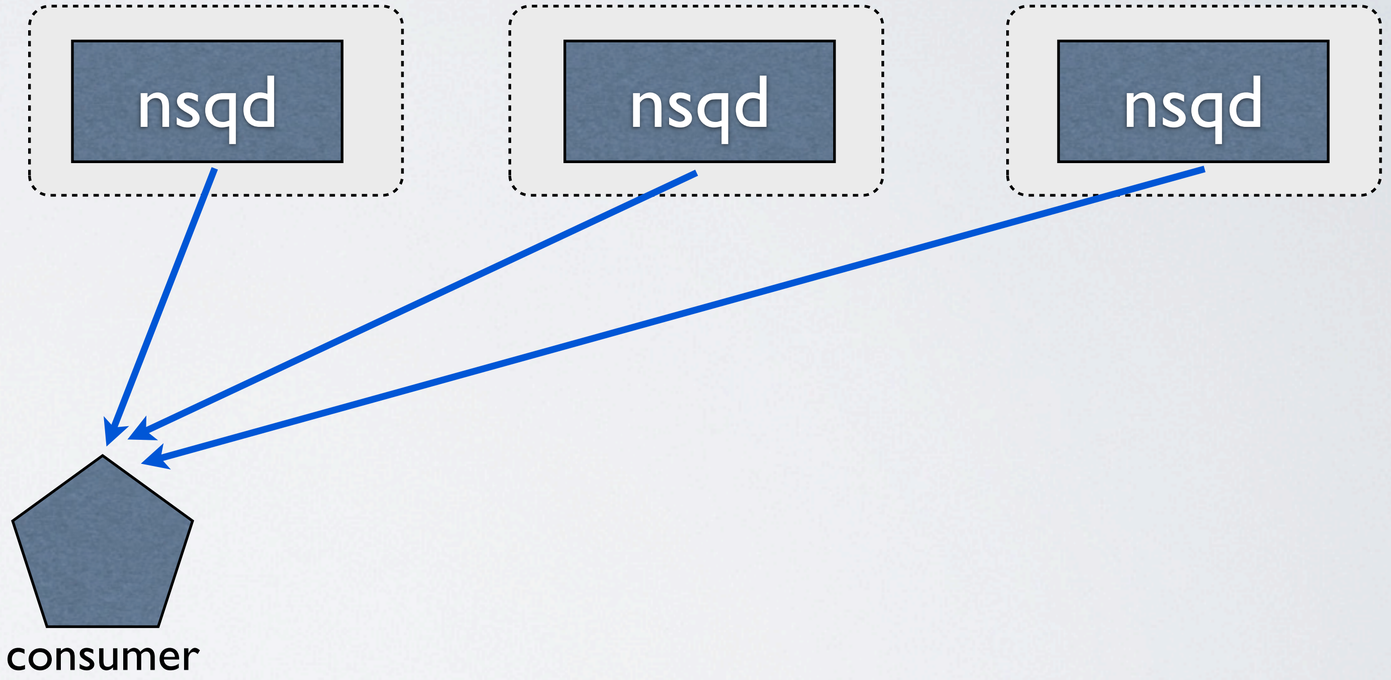
- easily enable *distributed* and *decentralized* topologies
- no brokers
- consumers connect to all producers
- messages are *pushed* to consumers
- **nsqlookupd** instances are *independent* and require no coordination (run a few for HA)

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)

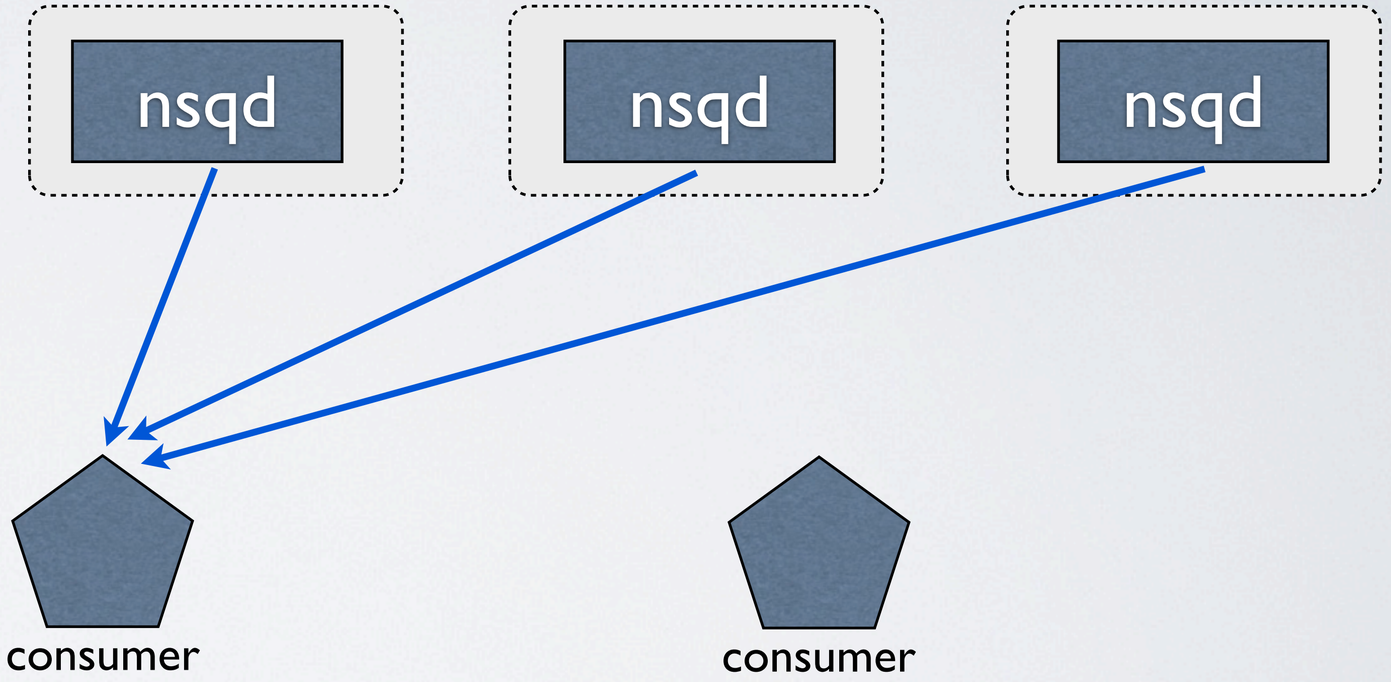
ELIMINATE ALL THE SPOF

- 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)



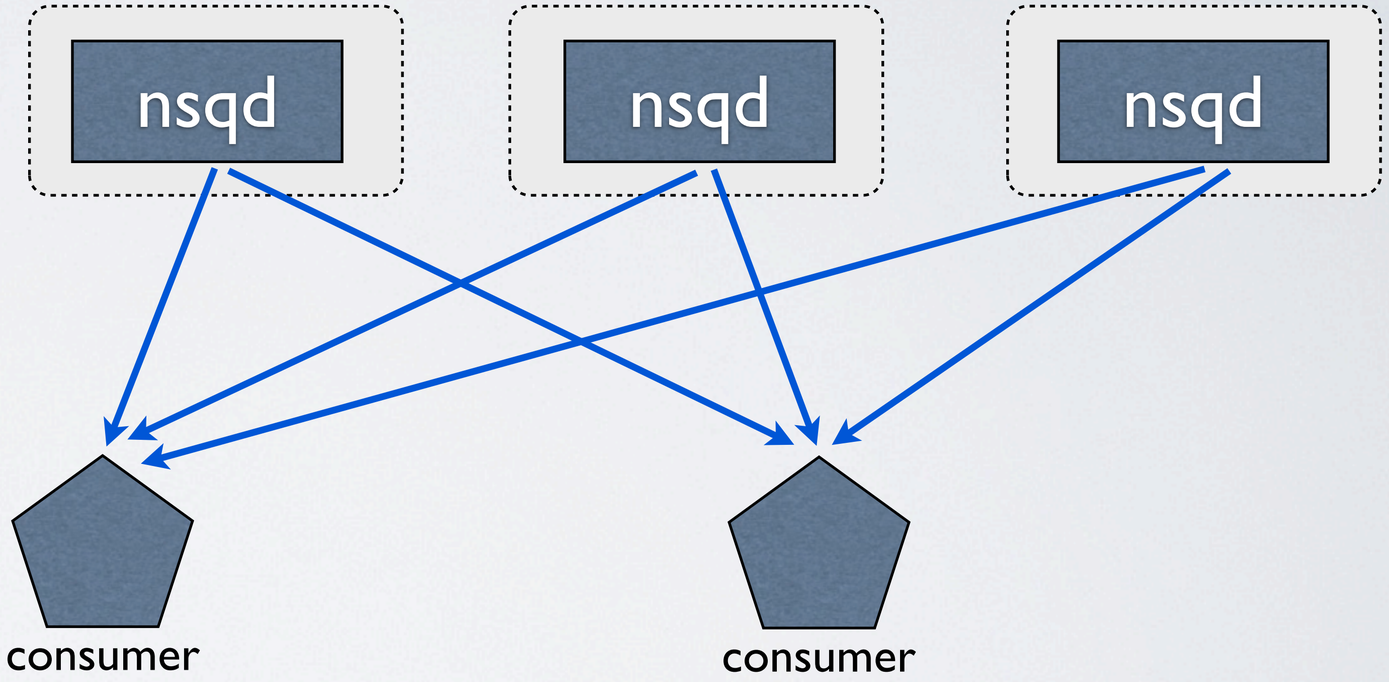
ELIMINATE ALL THE SPOF

- 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)

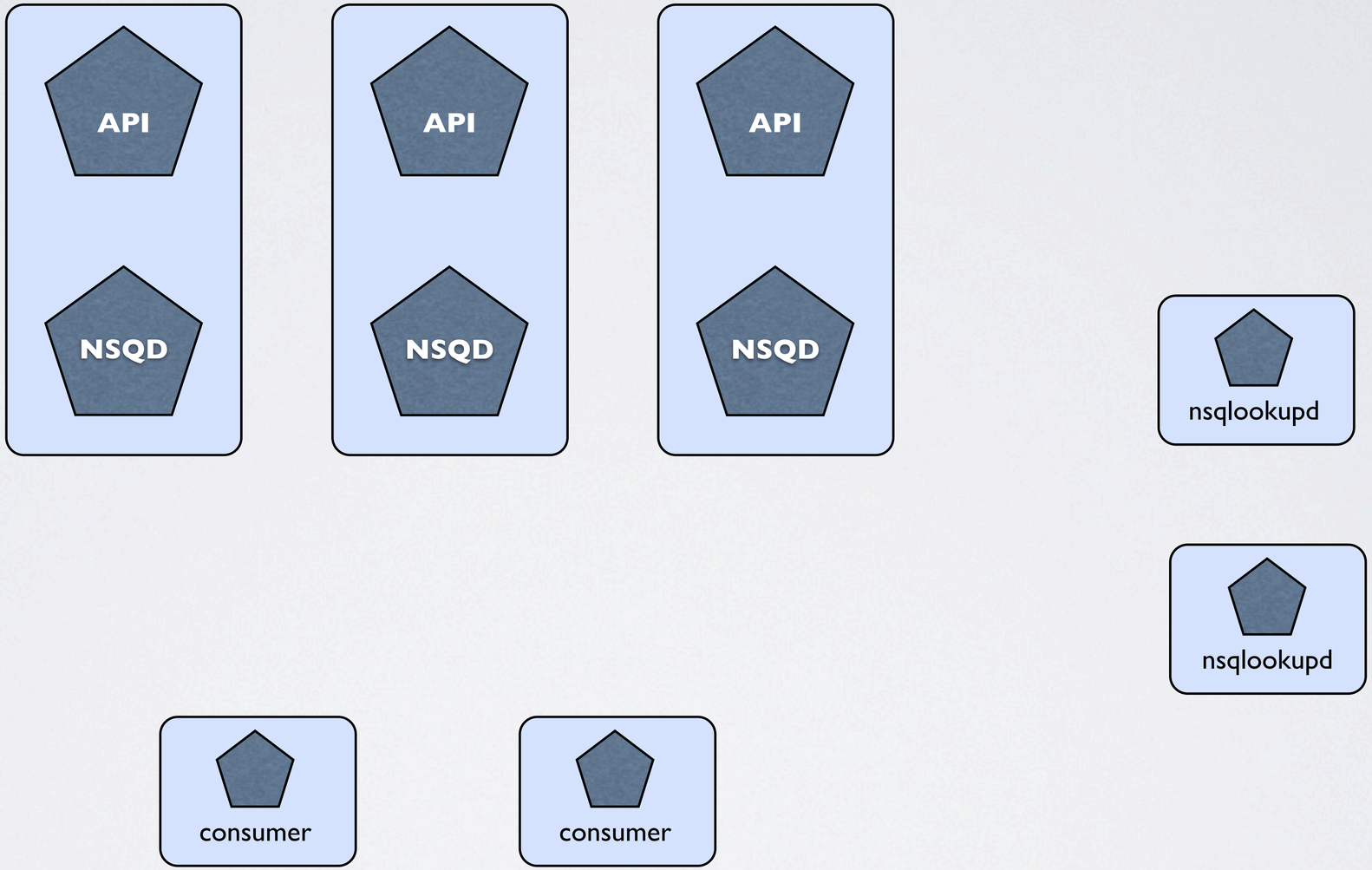


ELIMINATE ALL THE SPOF

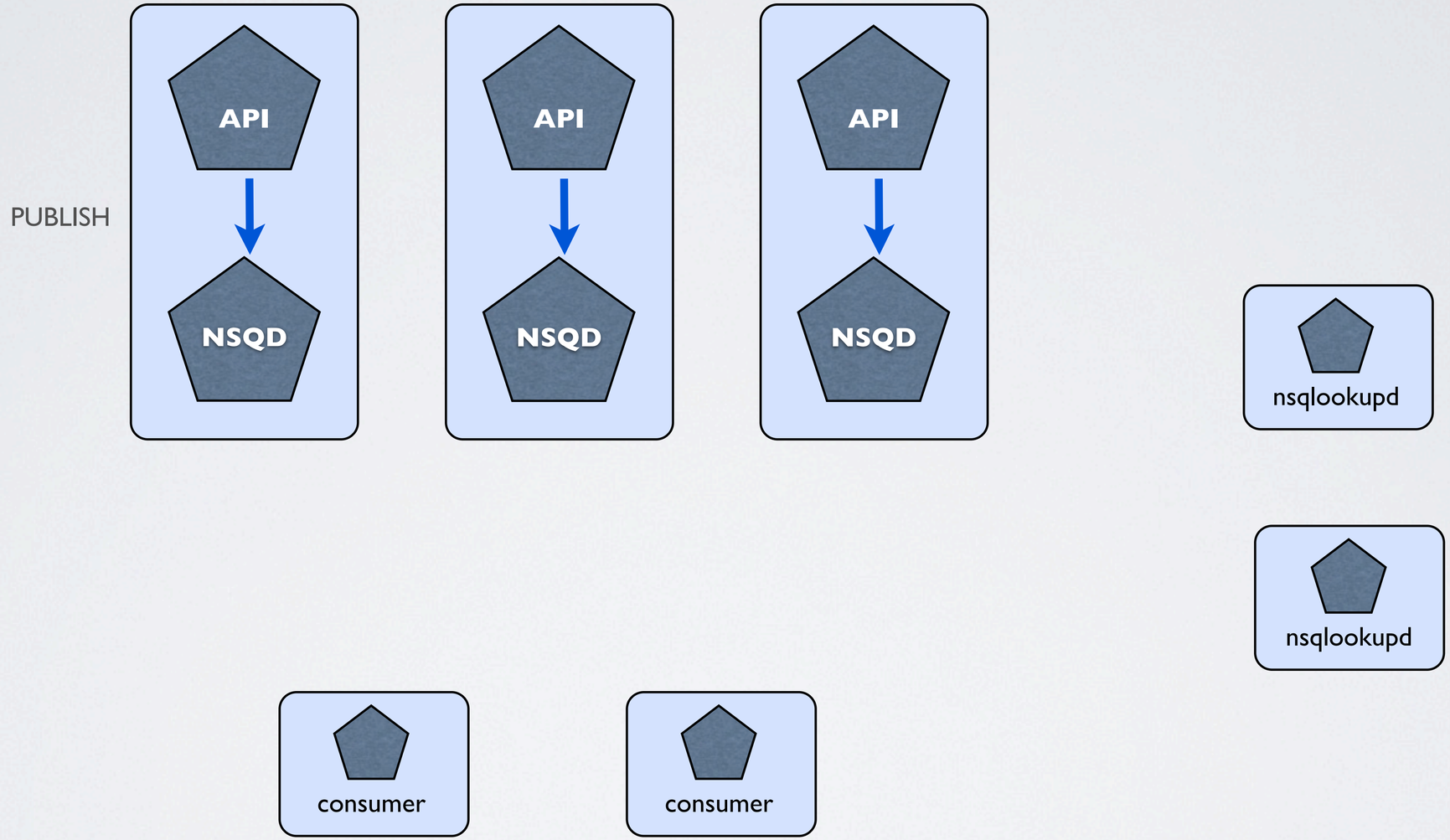
- easily enable *distributed* and *decentralized* topologies
- no brokers
- consumers connect to all producers
- messages are *pushed* to consumers
- **nsqlookupd** instances are *independent* and require no coordination (run a few for HA)



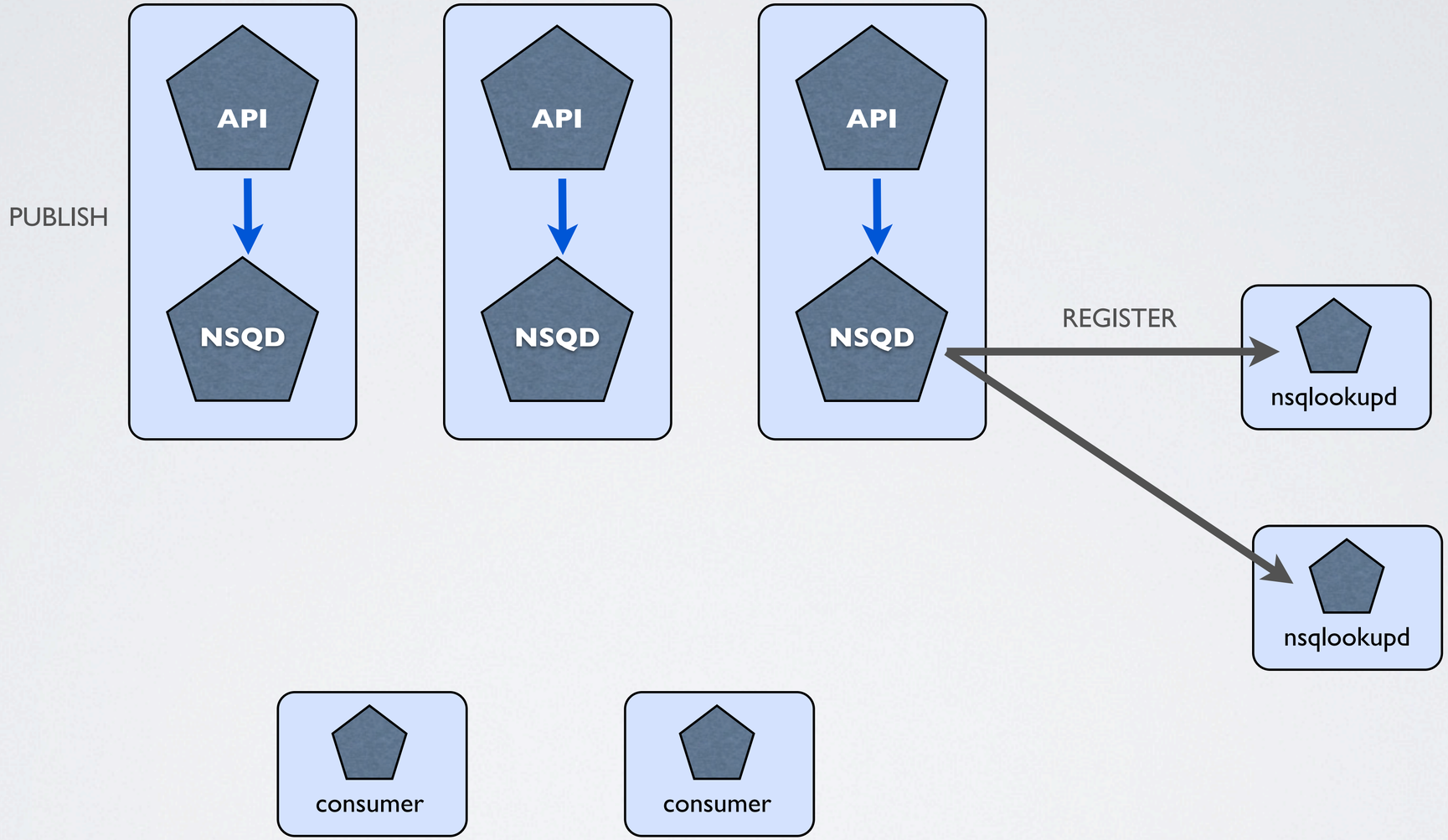
EXAMPLE NSQ ARCHITECTURE



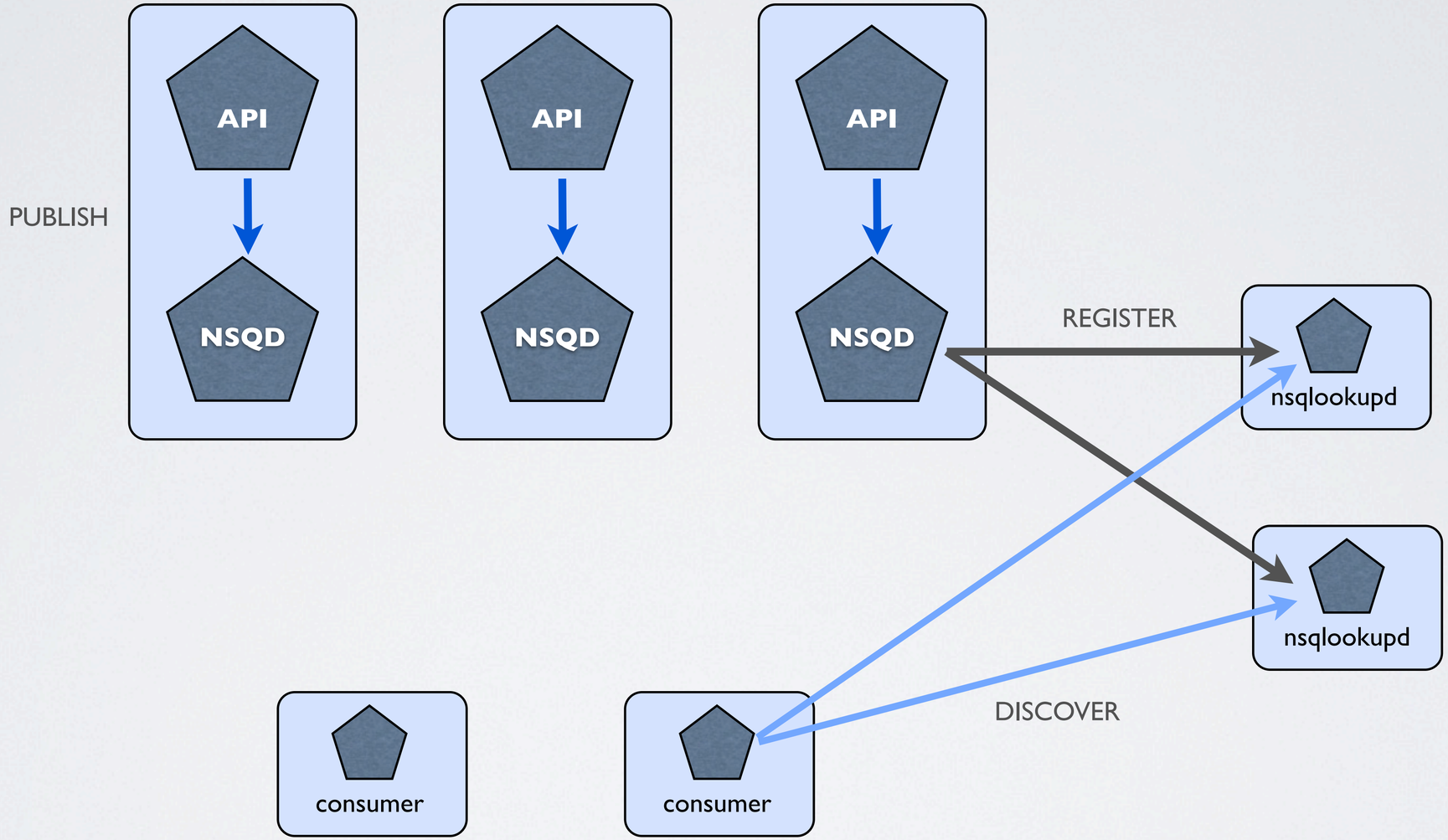
EXAMPLE NSQ ARCHITECTURE



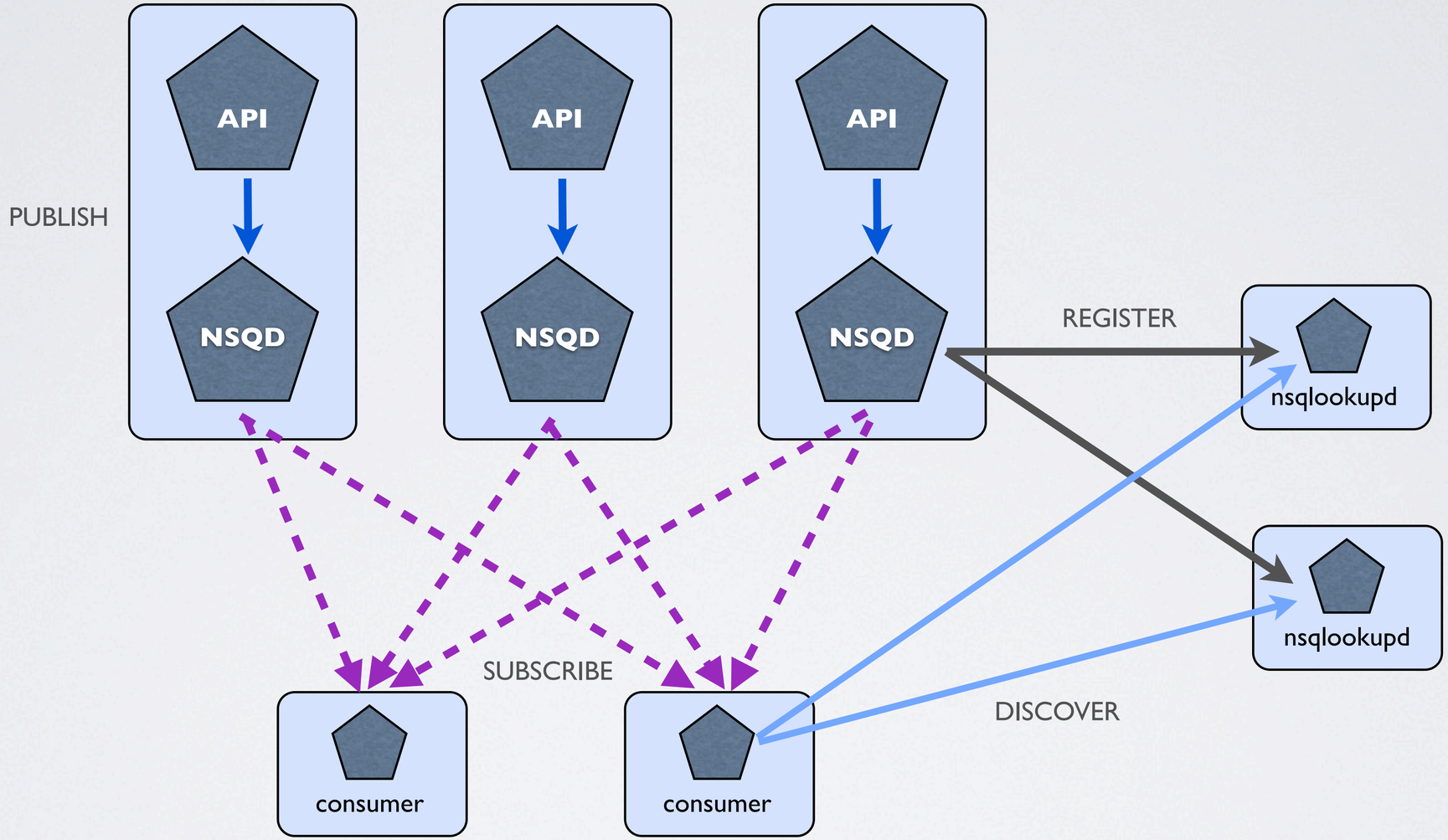
EXAMPLE NSQ ARCHITECTURE



EXAMPLE NSQ ARCHITECTURE



EXAMPLE NSQ ARCHITECTURE



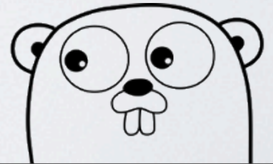
A WORD ON ERRORS

- If a reader does not reply to confirm completion of a message within a timeout, the message is requeued.
- Abandoned after configurable number of requeues
- Allows for recovery in face of transient problems without getting hung up on bad messages

OTHER NSQ NICETIES

- Admin interface: server-side channel pausing, admin action notifications
- Configurable high-water mark on memory usage
- Ephemeral channels for stream sampling

github.com/bitly/nsq



DISTRIBUTED MESSAGING CAVEATS

DISTRIBUTED MESSAGING CAVEATS

- Messages in order? Fuggedaboudit!*

DISTRIBUTED MESSAGING CAVEATS

- Messages in order? Fuggedaboudit!*
- NSQ protocol guarantees delivery at least once - idempotence is a must! (_ids help)

DISTRIBUTED MESSAGING CAVEATS

- Messages in order? Fuggedaboudit!*
- NSQ protocol guarantees delivery at least once - idempotence is a must! (_ids help)
- Try not to be shocked by effortless recovery from node failure

DISTRIBUTED MESSAGING CAVEATS

- Messages in order? Fuggedaboudit!*
- NSQ protocol guarantees delivery at least once - idempotence is a must! (_ids help)
- Try not to be shocked by effortless recovery from node failure

*See http://bit.ly/life_beyond_transactions

STREAM PROCESSING: WHY NOW?

STREAM PROCESSING: WHY NOW?

- Cheap node distribution: EC2 etc

STREAM PROCESSING: WHY NOW?

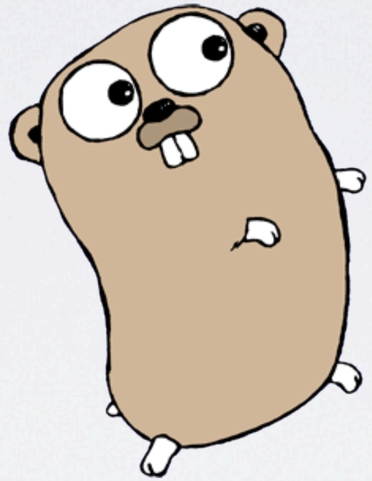
- Cheap node distribution: EC2 etc
- Moore's law, Amdahl's law, battered deceased equines...

STREAM PROCESSING: WHY NOW?

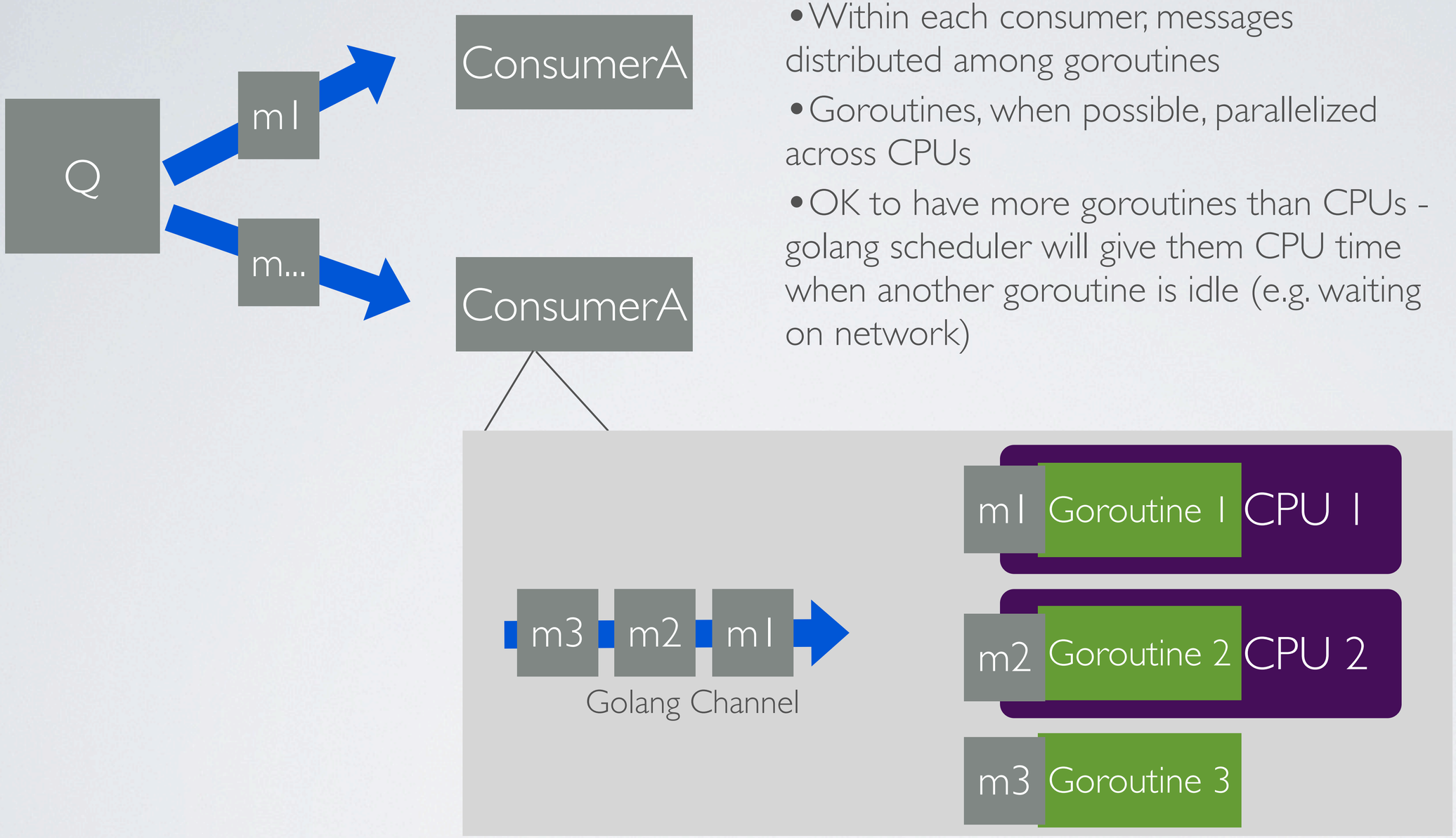
- Cheap node distribution: EC2 etc
- Moore's law, Amdahl's law, battered deceased equines...
- Taking advantage of CPU parallelism the way forward for program efficiency - good thing we just went over a paradigm for distributing tasks among parallel workers!

INTRA-PROGRAM STREAM PROCESSING IN THE WILD

EXAMPLE 1: GOLANG



- **Channels** allow synchronized passage of messages between two **goroutines**
- Goroutine independence (through synchronization) allows stream-like architecture:
- "Don't communicate by sharing memory, share memory by communicating"
- Golang scheduler can parallelize between cores (GOMAXPROCS)
- Channels act like queues. Multicast not really an option
- Queuereader applications are a particularly good fit for goroutine concurrency



- Within each consumer, messages distributed among goroutines
- Goroutines, when possible, parallelized across CPUs
- OK to have more goroutines than CPUs - golang scheduler will give them CPU time when another goroutine is idle (e.g. waiting on network)

EXAMPLE 2

ZMQ FEATURES

- Networking library that provides building blocks discussed earlier
- Unlike golang channels, does support many more complex patterns
- Transport layer abstracted out: same application can connect multiple threads or multiple machines
- Can start by distributing among processes, and scale up to several boxes. Application code doesn't need to know about it!
- All the rage among the webscale set, but unclear what the hell is going on in the community

Change transport by changing one string

```
zmq.bind(“inproc://example_socket”)
```

```
zmq.bind(“tcp://1.2.3.4:5678”)
```

ALMOST DONE I PROMISE

WHAT HAVE WE SEEN HERE?

- Stream processing paradigm is a great tool for writing composed, modular applications
- Fault tolerance and horizontal scalability come in the box
- Your web application is probably better suited to this design than you think
- NSQ is the tool we use to write distributed stream processing applications and it kicks ass at it
- These same paradigms can aid in writing performant applications making use of multicore computer architecture, so you should plan on seeing a lot more of this stuff in the near future, whether you like it or not

THANKS!

Dan Frank
df@bit.ly
@danielhfrank