Achieving Scale with Messaging & the Cloud

Tony Garnock-Jones <tonyg@lshift.net>

Outline

- Messaging – what, why

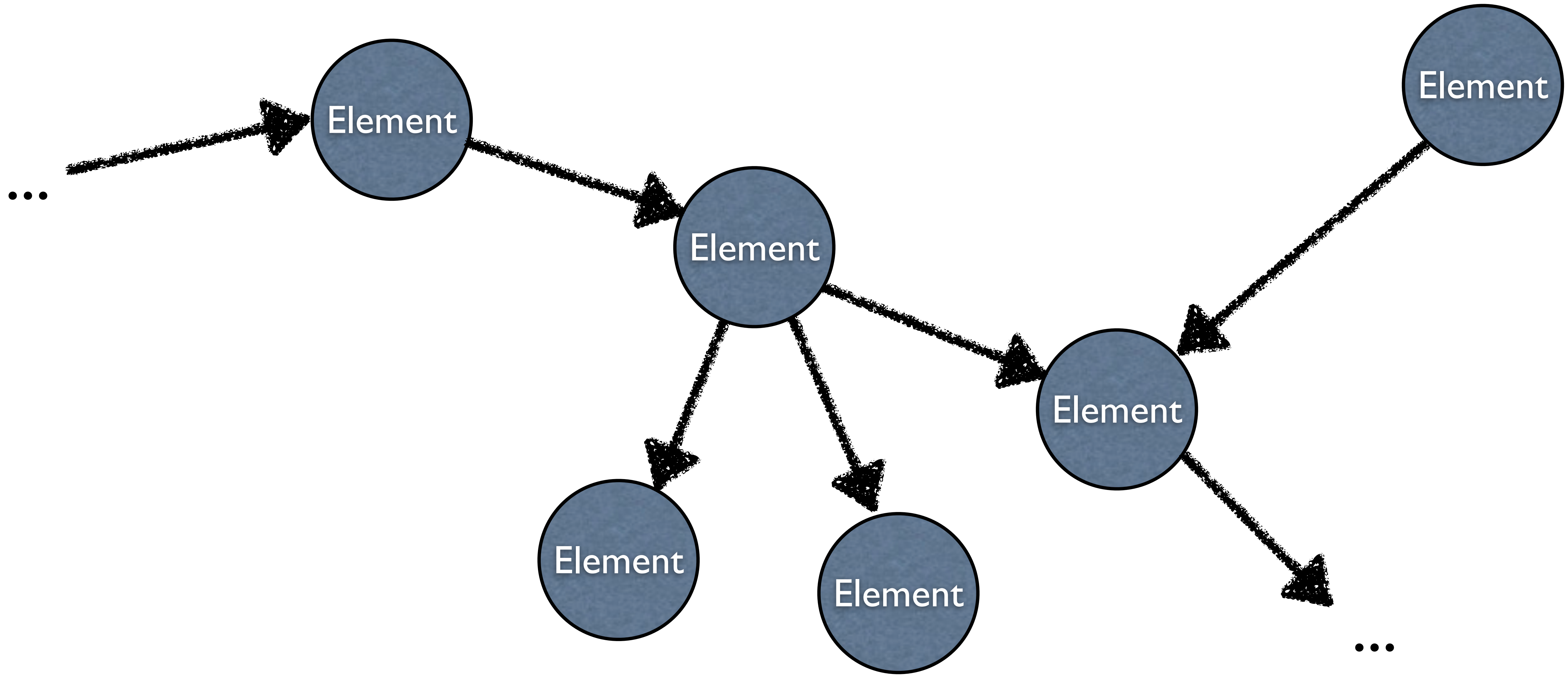
- Cloud Messaging – some examples

- Achieving Scale – what, why, how

Messaging

Messaging Elements

...relaying, filtering, multicasting, forwarding, buffering, distribution/scheduling, subscription, topics, streaming, ...



Messaging is used to structure distributed applications. There are lots of messaging patterns in use: for example, plain old request-response RPC; point-to-point messaging, like email or IM; multicast messaging, like mailing lists and newsgroups; and general publish/subscribe (essentially multicast plus filtering) and queueing (store-and-forward).

What is it good for?

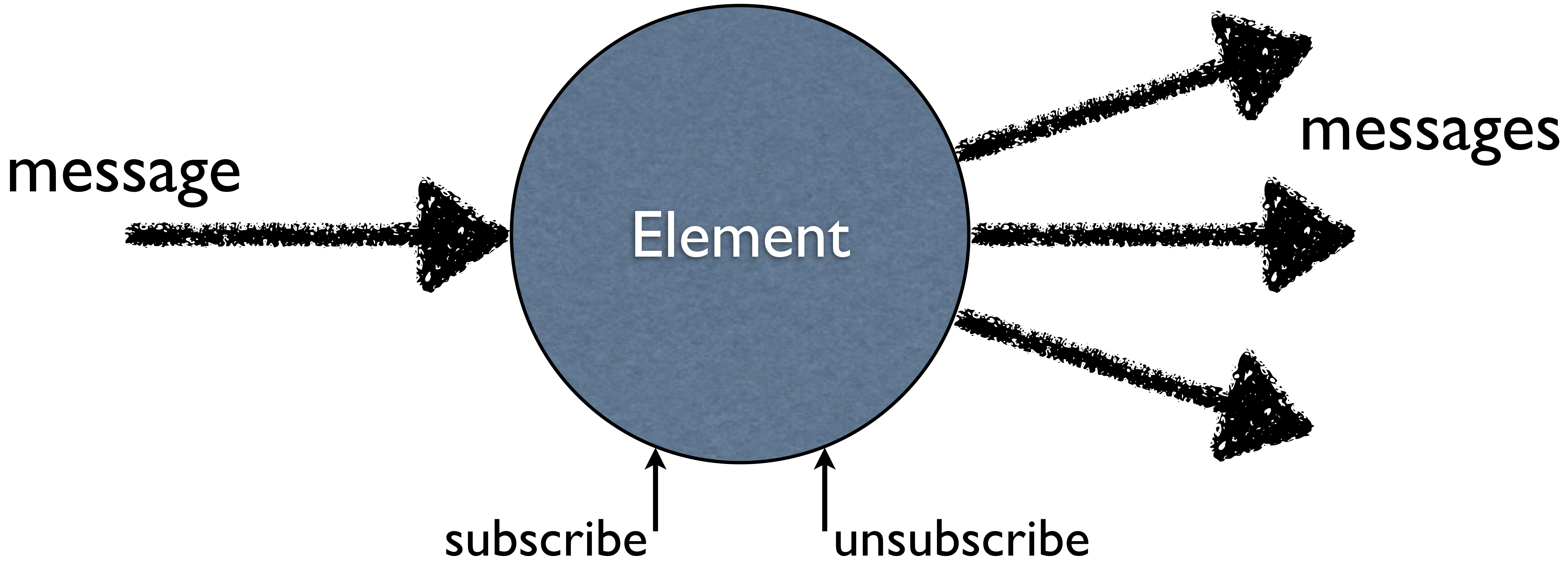
Decoupling communication from behaviour, enabling:

- debuggability
- manageability
- monitorability
- rolling upgrades
- load-balancing
- cross-language integration
- fault-tolerance
- store-and-forward
- ...

Decouples patterns of communication from the behaviour of the communicating elements

Messaging Elements

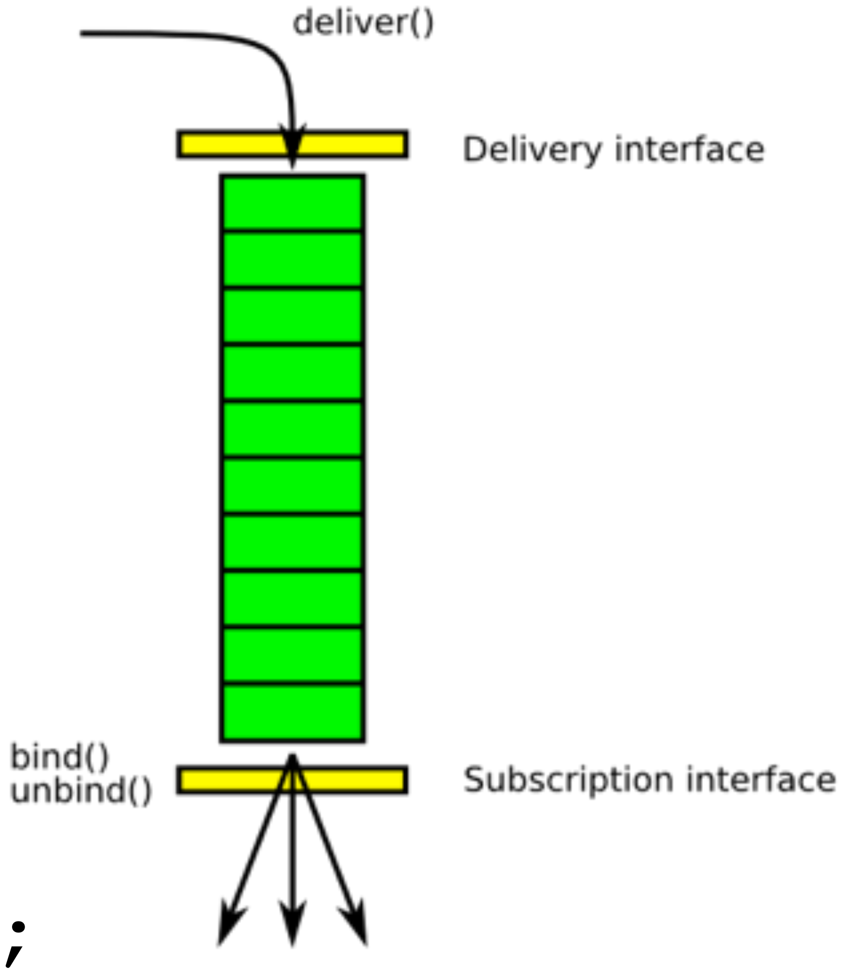
...relaying, filtering, multicasting, forwarding, buffering, distribution/scheduling, subscription, topics, streaming, ...



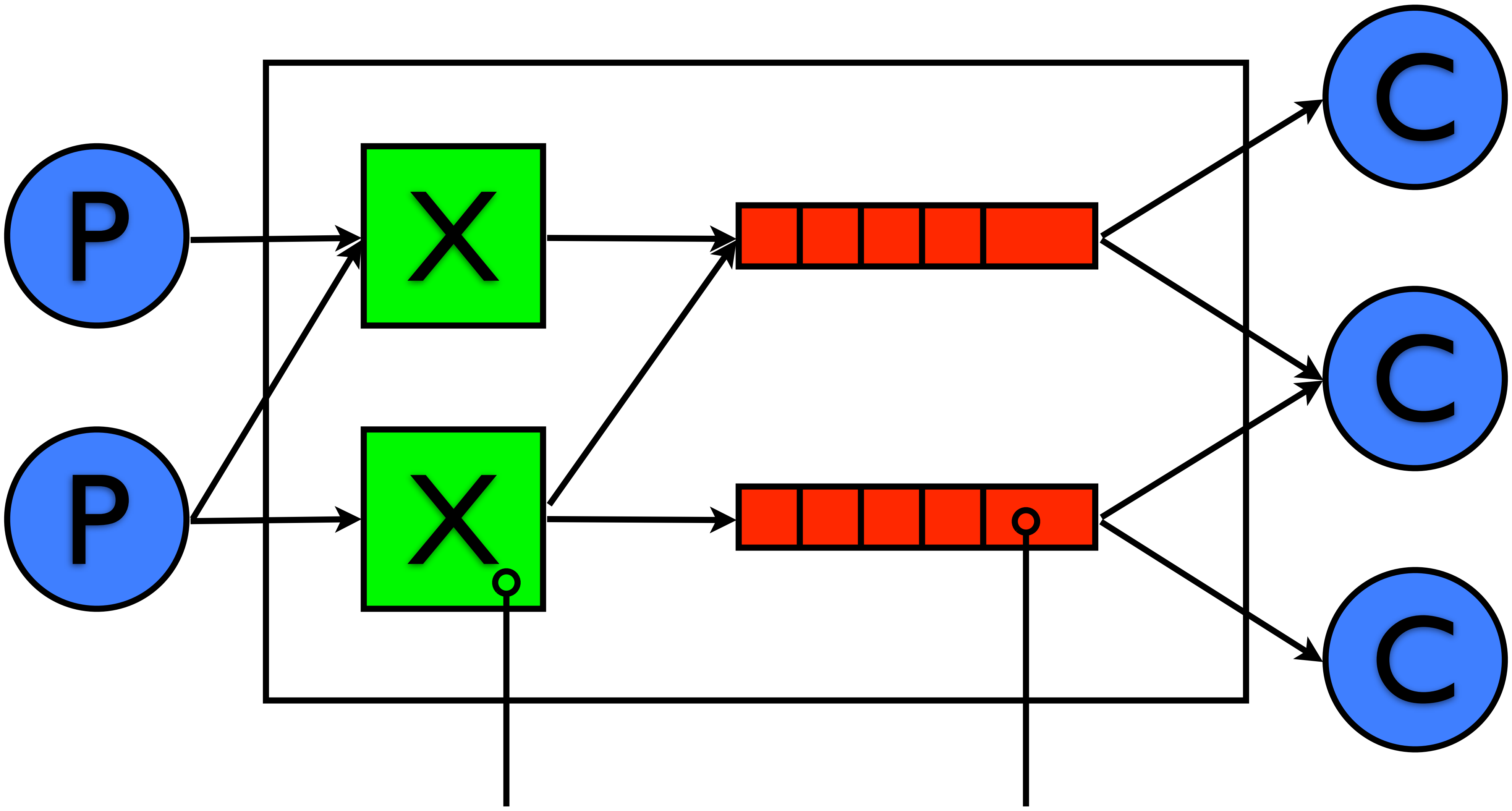
One really common pattern in a messaging-based system is various incarnations of a generalised publish/subscribe service, where messages flow in, are processed by the node, and are forwarded on to subscribers. For example, with email mailing lists, joining the list is a subscription, and your delivery settings - whether you want a digest or not, how frequently you want the digest etc - are used by the mailing-list manager in making decisions about what and how to forward incoming messages on to subscribers.

Messaging Elements

- Delivery:
  - **`deliver(target, message);`**
- Subscription:
  - **`bind(source, topic, target);`**
  - **`unbind(source, topic, target);`**



AMQP for Messaging



- Exchanges perform relaying, copying, and filtering
- Queues perform buffering and round-robin delivery

AMQP is a messaging protocol that’s designed to capture a lot of the common utility elements in messaging systems. It can be seen as combining pub/sub with queueing. [Explanation of the diagram]

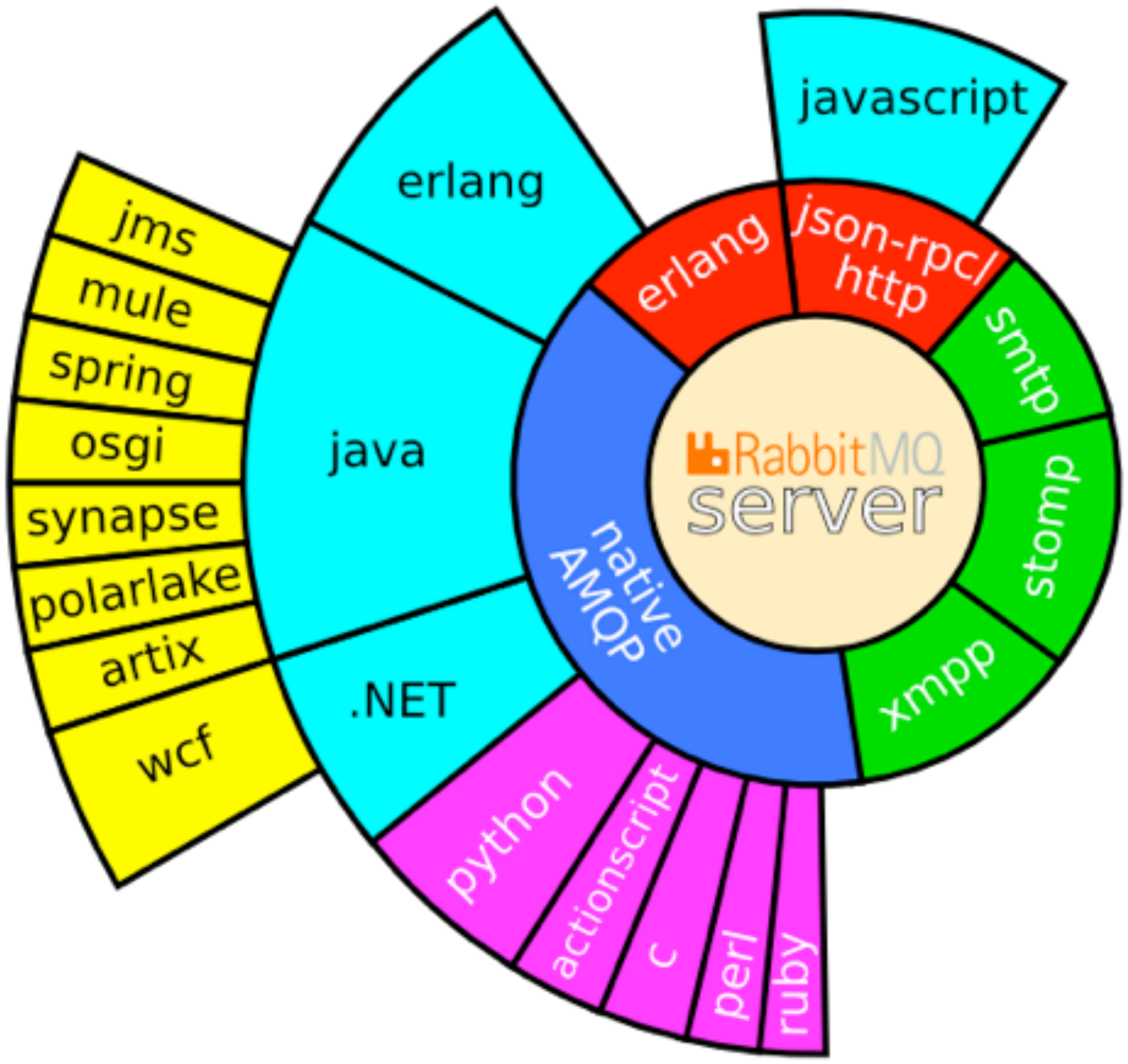
It’s Easy

```
byte[] body = ...;
ch.basicPublish(exchange, routingKey, null, body);
```

```
ch.queueBind(queueName, exchange, routingKeyPattern);
```

```
QueueingConsumer consumer = new QueueingConsumer(ch);
ch.basicConsume(queueName, consumer);
while (true) {
    QueueingConsumer.Delivery delivery =
      consumer.nextDelivery();
    // ... use delivery.getBody(), etc ...
    ch.basicAck(delivery.getEnvelope().getDeliveryTag(),
                false);
}
```

RabbitMQ Universe



🟥 🟦 Transport
🟩 Gateway
🟪 🟦 Client
🟨 Adapter

...plus new developments like RabbitHub, BBC FeedsHub, Trixx, ...

Naturally, since I work on RabbitMQ, I'm going to be focussing on systems built with it, but generally a lot of the techniques for scaling out messaging work with other AMQP systems and other kinds of messaging network.

RabbitMQ is an AMQP implementation with a server ("broker") written in Erlang/OTP and clients in many programming languages for many different environments. There are also quite a few gateways to other kinds of messaging network, such as HTTP pubsub and various flavours of XMPP.

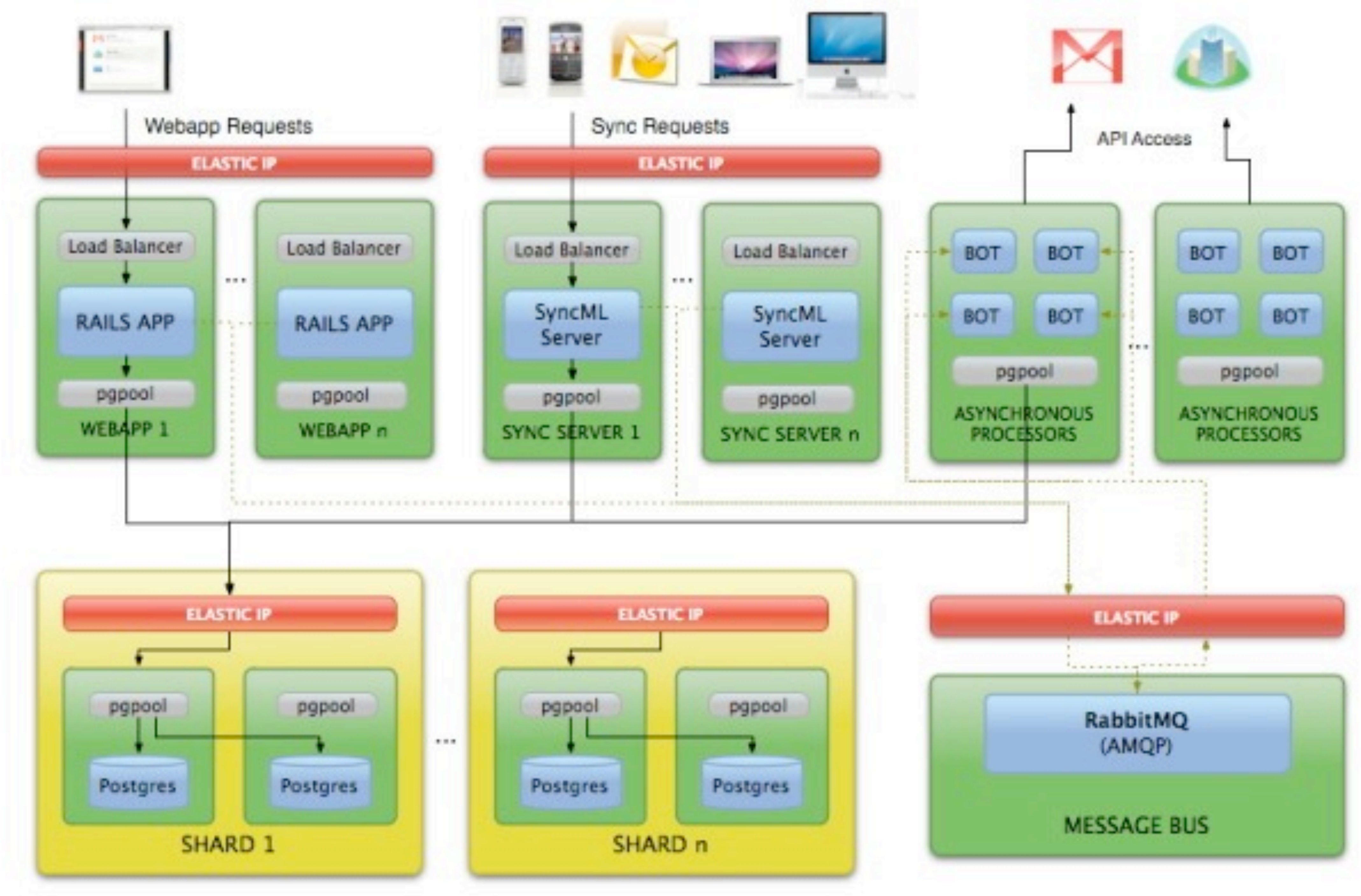
Cloud Messaging

Examples

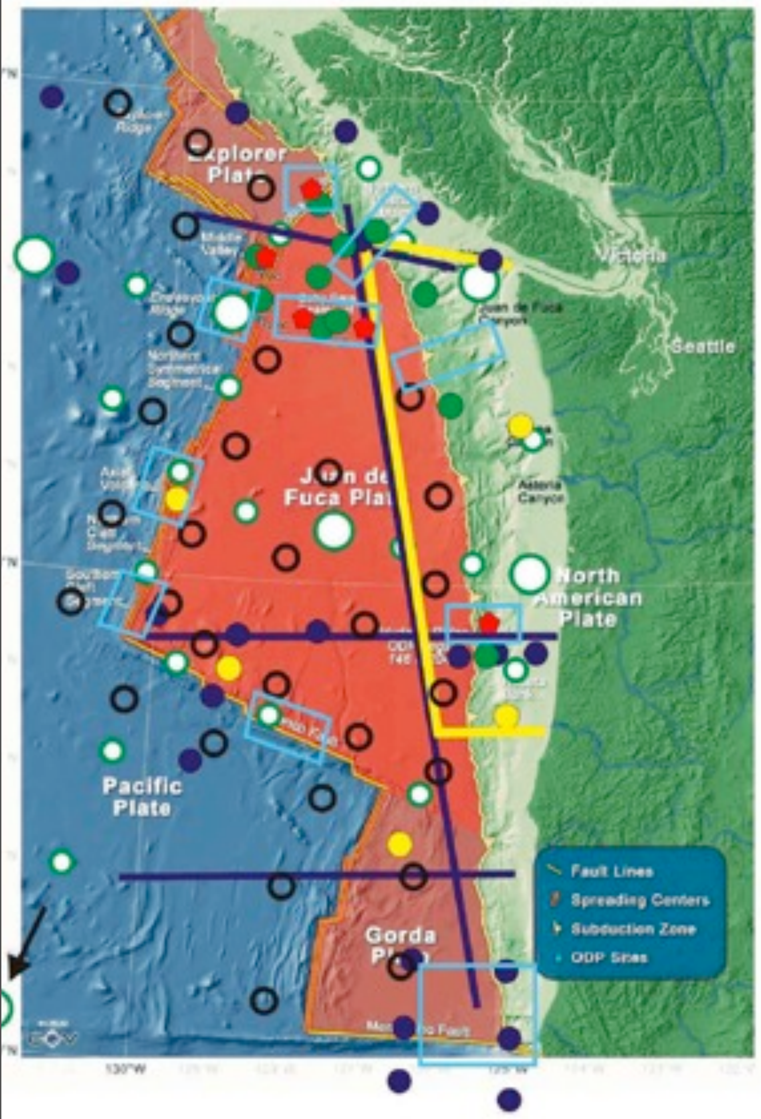
- Soocial are using RabbitMQ + EC2 to coordinate their contacts-database synchronisation application
- The Ocean Observatories Initiative is using RabbitMQ + relays + gateways for a global sensing & data distribution network

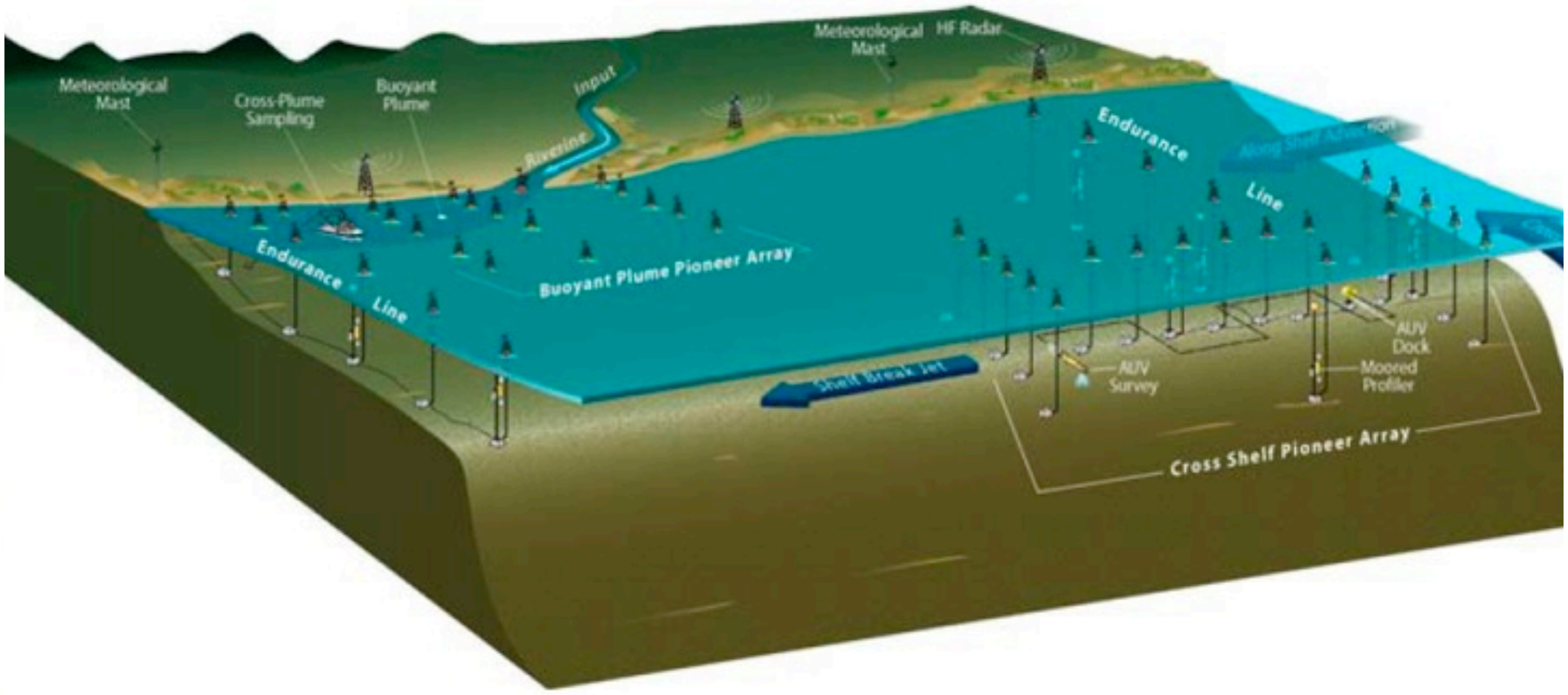
And of course there are many, many applications using different kinds of messaging, such as SQS, XMPP and so forth, as well.

Soocial.com

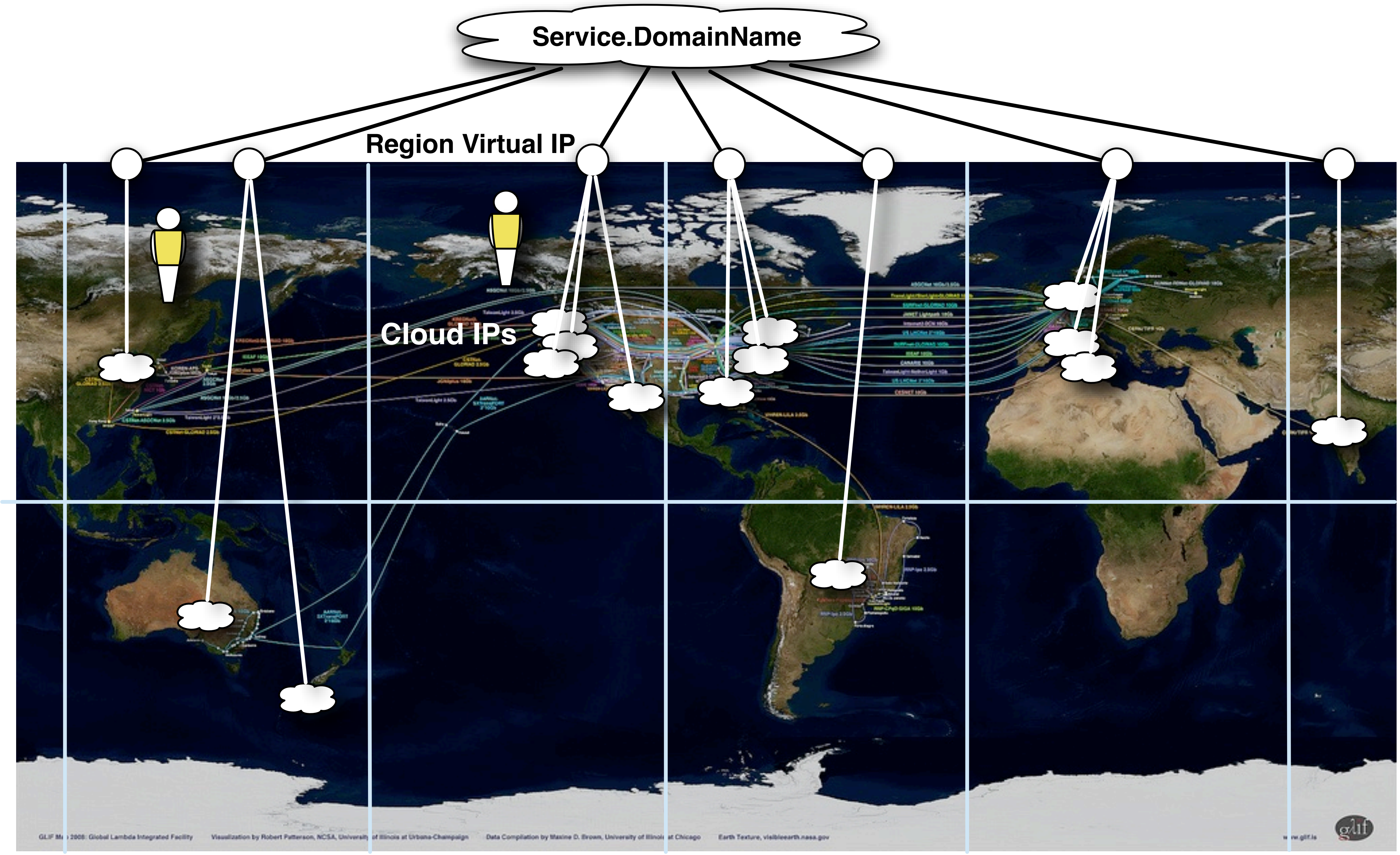

Webapp Requests
Sync Requests
API Access
ELASTIC IP
ELASTIC IP
Load Balancer
RAILS APP
pgpool
WEBAPP 1
Load Balancer
RAILS APP
pgpool
WEBAPP n
Load Balancer
SyncML Server
pgpool
SYNC SERVER 1
Load Balancer
SyncML Server
pgpool
SYNC SERVER n
BOT
BOT
BOT
BOT
pgpool
ASYNCHRONOUS PROCESSORS
BOT
BOT
BOT
BOT
pgpool
ASYNCHRONOUS PROCESSORS
ELASTIC IP
pgpool
Postgres
pgpool
Postgres
SHARD 1
ELASTIC IP
pgpool
Postgres
pgpool
Postgres
SHARD n
ELASTIC IP
RabbitMQ
(AMQP)
MESSAGE BUS

OOI – Ocean Observatories Initiative





OOI – Ocean Observatories Initiative



“Twitter for data”; applying a social networking model to applications in a very large scientific distributed system. Messaging holds the core of the system together.

Nanite

- Developed by Ezra Zygmuntowicz at Engineyard
- Presence

(by pinging; we’re still working out the best way of putting presence into RabbitMQ itself)

- Self-assembling compute fabric
- Load-balancing & recovers from failure
- An easy way of deploying to the cloud

Besides those concrete applications that are using messaging in the cloud, there are also frameworks to help you construct your own managed messaging-based cloud applications, like Nanite, and the system that Heroku have built.

In Nanite, Agents do the work; Mappers manage directory, distribution, auto-scaling and load-balancing

VPN^3 (VPN-cubed)

- A secure Virtual Private Network in the cloud
- Uses RabbitMQ as a backend for IP multicast (!) which otherwise doesn't work in EC2
- Uses BGP to arrange routing between nodes
- Can meld your datacentre(s) with EC2

A different kind of framework is support not for applications but for deployments of general networks into the cloud. CohesiveFT use RabbitMQ to stitch together the different components in their Elastic Server product, and they've also produced an impressive system called VPN-cubed that among other things provides support for multicast in an Amazon EC2 environment by using RabbitMQ to distribute the multicast packets.

Achieving Scale

Having touched on a few examples of messaging being used for cloud-based distributed systems, I’m going to get into the technical details of approaches to achieving scale.

Most of the time the simplest of the techniques I’m about to discuss will be more than adequate; for example, RabbitMQ on a single core can cope with several thousand messages per second with latencies of around a millisecond. When that’s not enough, though, it’s nice to know the options for scaling up further.

Paul Baran, *Introduction to Distributed Communications Networks*, 1964

Paul Baran's Networks

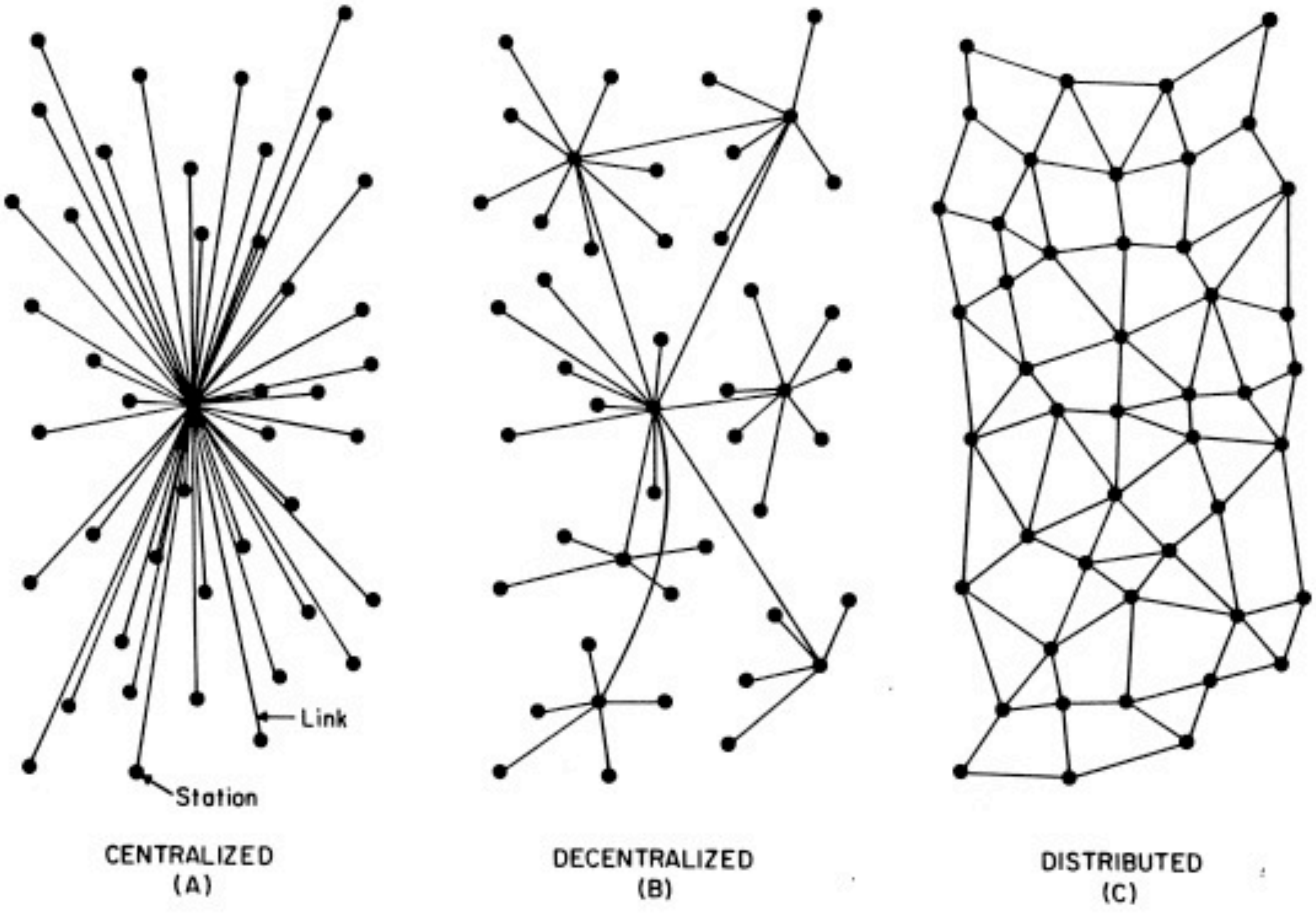


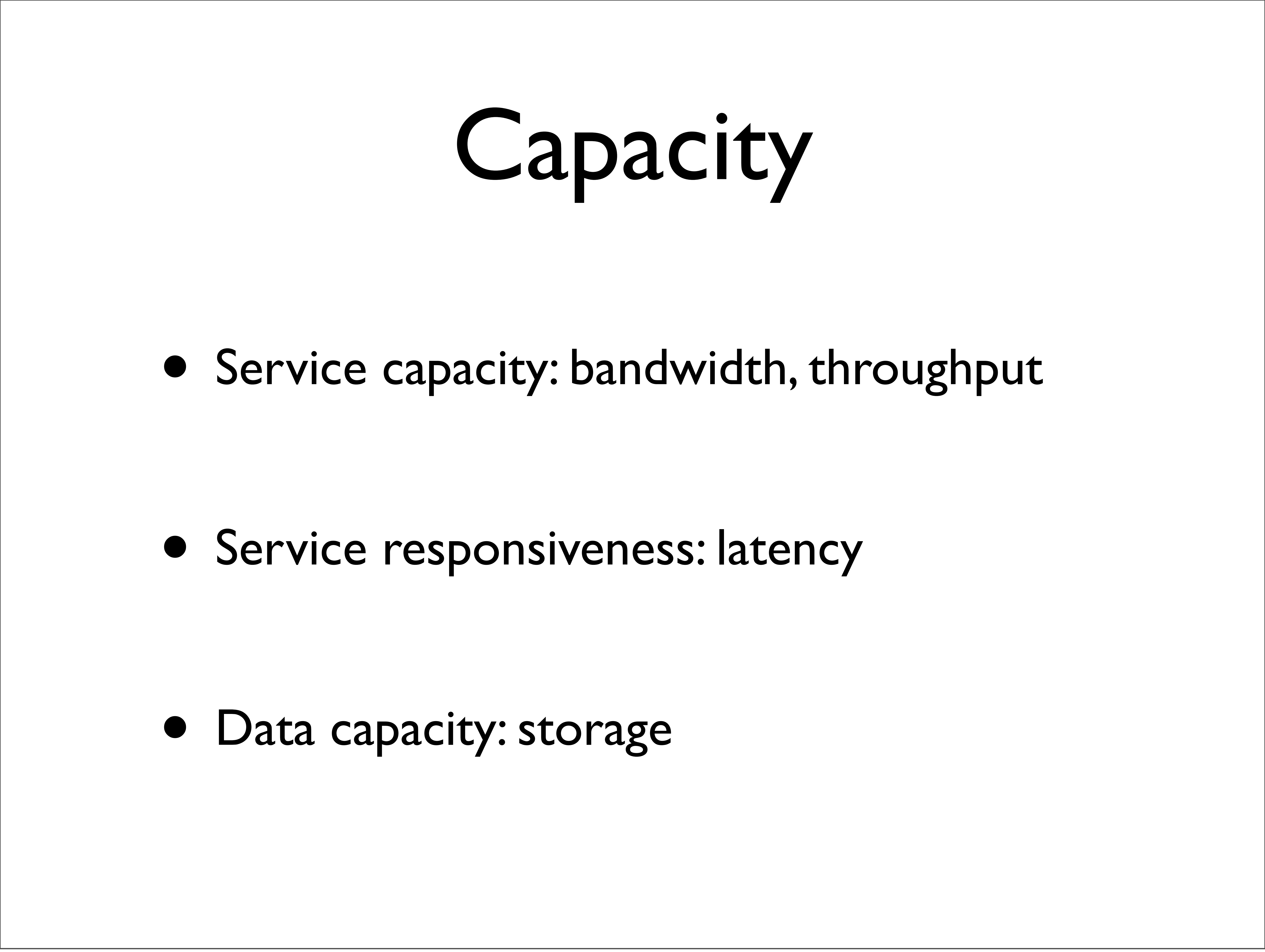
FIG. I — Centralized, Decentralized and Distributed Networks

Back in the sixties, Paul Baran at the Rand Corporation performed the original, ground-breaking research into fault-tolerant distributed systems. This diagram comes from a 1964 memo introducing the whole idea of a distributed network. I find it interesting because I see many systems using messaging start with a centralised setup, which develops over time into a more decentralised picture. Things can go two ways from that point: many networks develop a scale-free aspect, becoming some hybrid of the middle and right pictures, but some - like compute fabrics, workflow systems, and so forth - do develop into mesh-like structures like the one on the right. So to me, reading this diagram left to right represents the stages of scaling up a system.

Achieving Scale

- Capacity – throughput, latency, responsiveness
- Synchronisation & Sharing
- Availability – uptime, data integrity
- Network Management – lots of devices, lots of addresses, authentication & authorisation

There are lots of distinct but related ideas bundled up under the label of “scale”.
Scaling for capacity is one of the things the cloud is great for.
Once you move beyond the single, centralised broker, you often need to have some communication between brokers.
Minimising downtime is important in many systems, and almost every system has at least a few kinds of message that simply Must Not Get Lost, ever.
Finally, once you have a complex system, you start to need tools for cataloguing, exploring, monitoring and debugging your network.

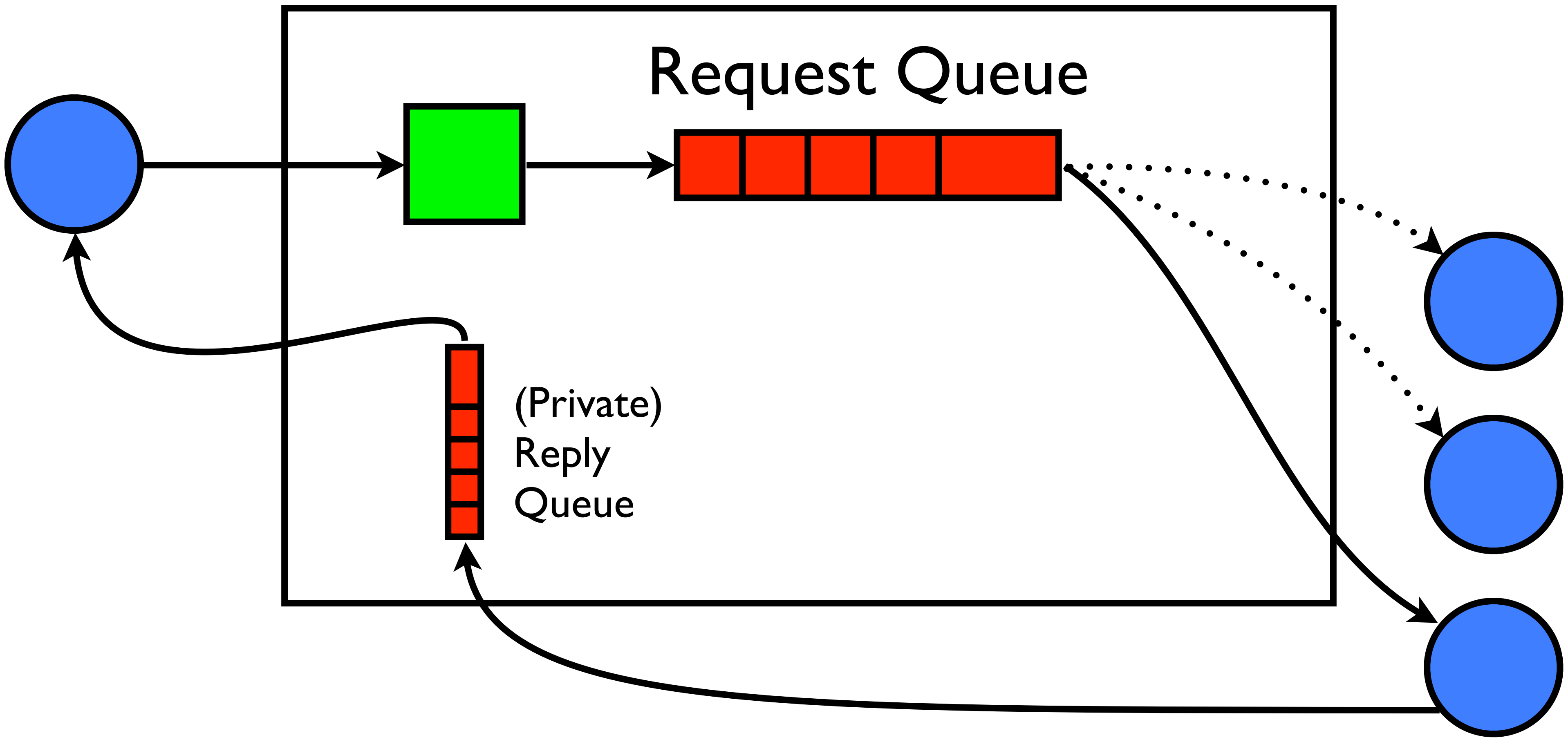


Capacity

- Service capacity: bandwidth, throughput
- Service responsiveness: latency
- Data capacity: storage

Storage out-of-scope: messaging systems aren't intended to store masses of data, but it's worth talking a little bit about what happens in exceptional situations where you need a queue to act like a database temporarily while the rest of the system sorts itself out, ...

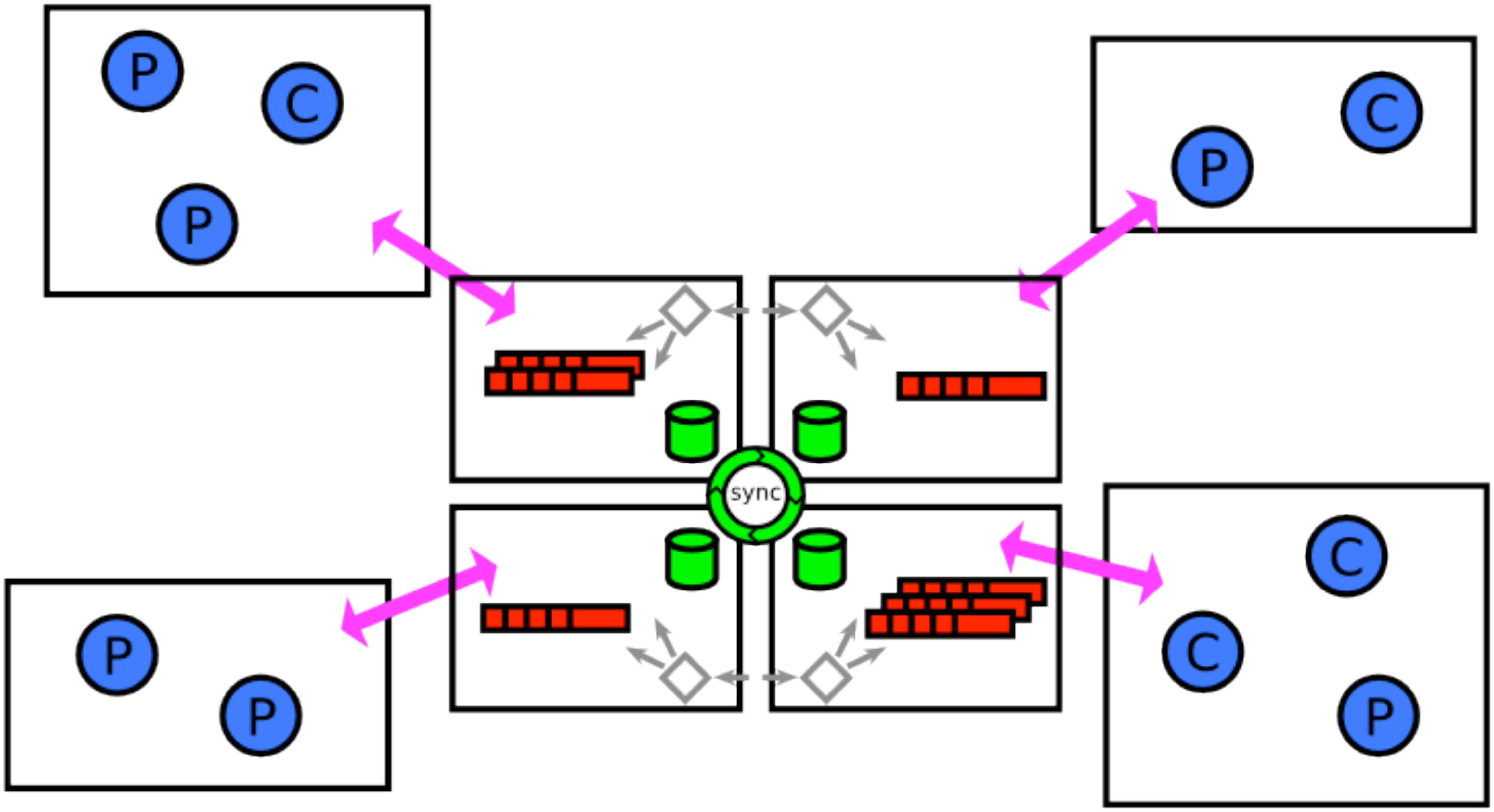
Simple load-balancing



- Shared queue mediates access to service instances
- Load-balancing, live upgrades, fault-tolerance

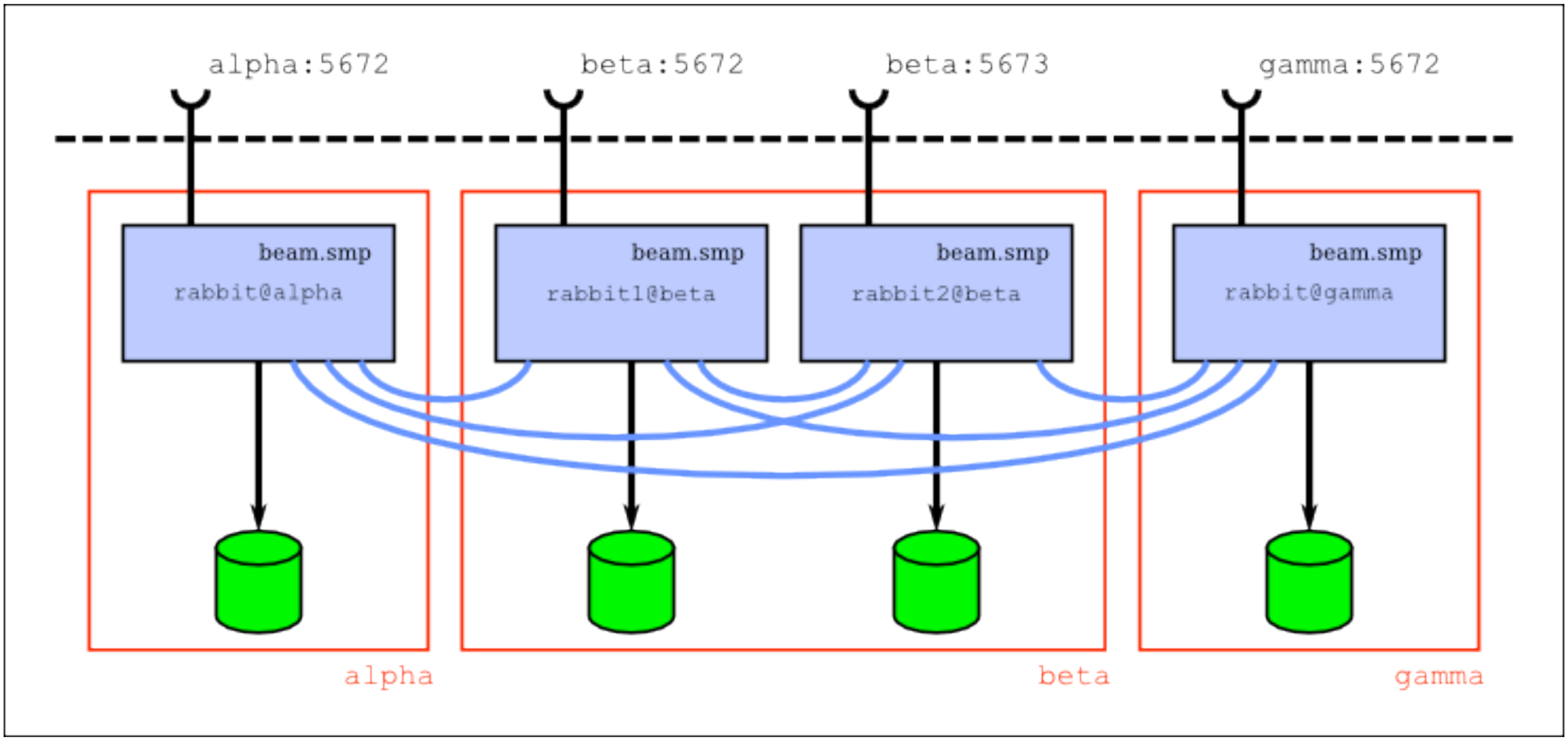
Core RabbitMQ supports this pattern, and systems like Nanite provide easy-to-use frameworks for working with it. This is often enough: frequently, it's the service that's the bottleneck. But what about when the message broker itself becomes a chokepoint?

Clustering



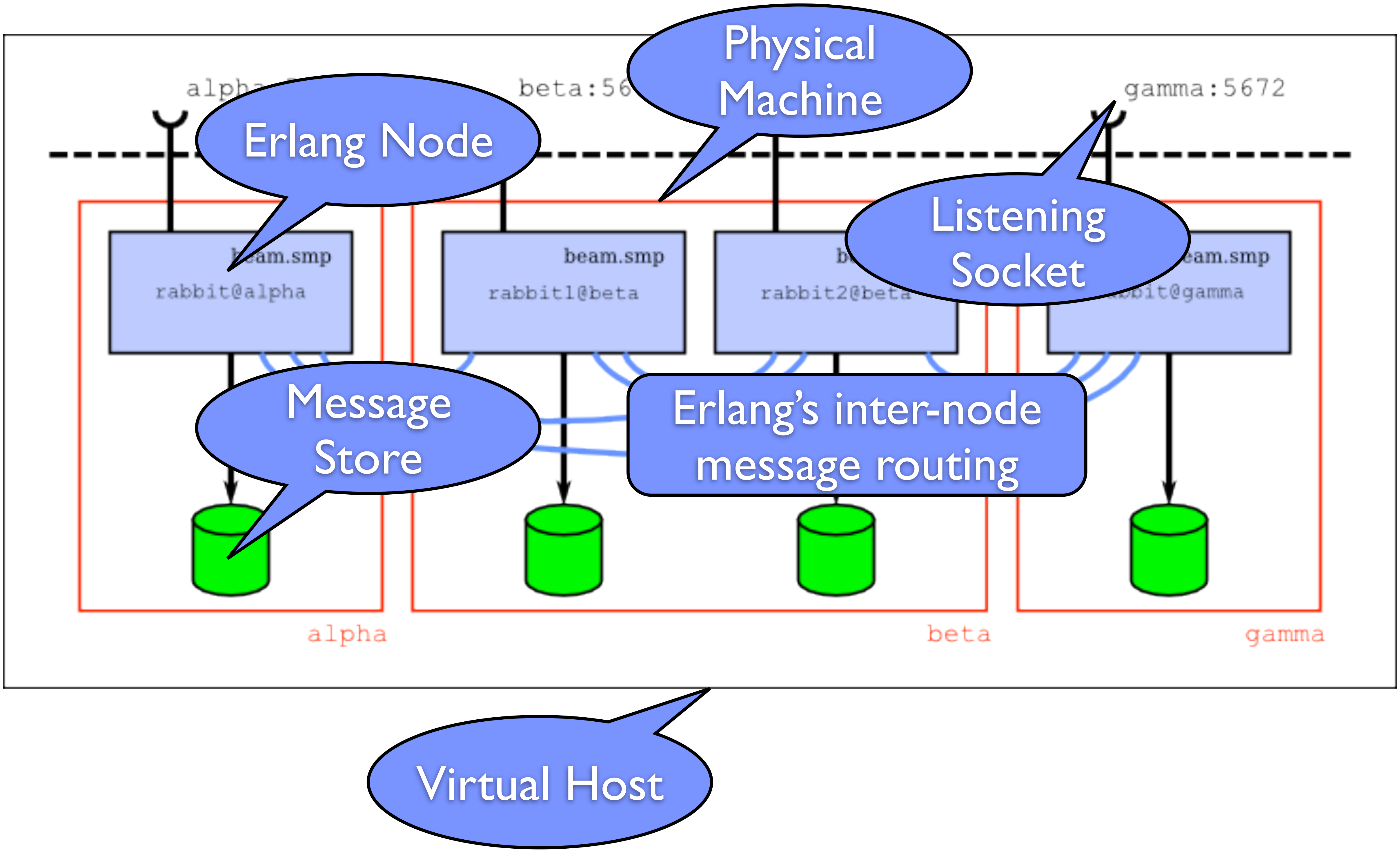
This is where RabbitMQ's clustering comes in. Multiple Erlang virtual machines, all running RabbitMQ, join together to create a giant virtual broker. Connecting clients all see the same picture of the world: to them, it's a single broker; the clustering is hidden away from them.

Clustering



The nodes – Erlang VMs – in a cluster can run on separate cores within a box, or can run on separate boxes connected over a network. Erlang's built-in message passing is used for communication between the nodes, and each node has a private disk-based store of its own. Each node also listens on a separate socket of its own: at the moment, clients need to know all the different socket addresses; DNS SRV records can be used for this.

Clustering



Erlang's built-in mnesia database is used to maintain synchronisation of the routing and filtering setup within the broker. Messages published to any node are transparently routed to where they need to go. One important caveat is that the queues – message buffers, message stores – in the system live on the node on which they were created, so while they're accessible from any other node in the cluster, their state is held only on a single node. We're working on improving this situation; and in the meantime, the whole problem can be made to go away by using some of the techniques for high availability I'll talk about in a little bit.

Divide and Conquer

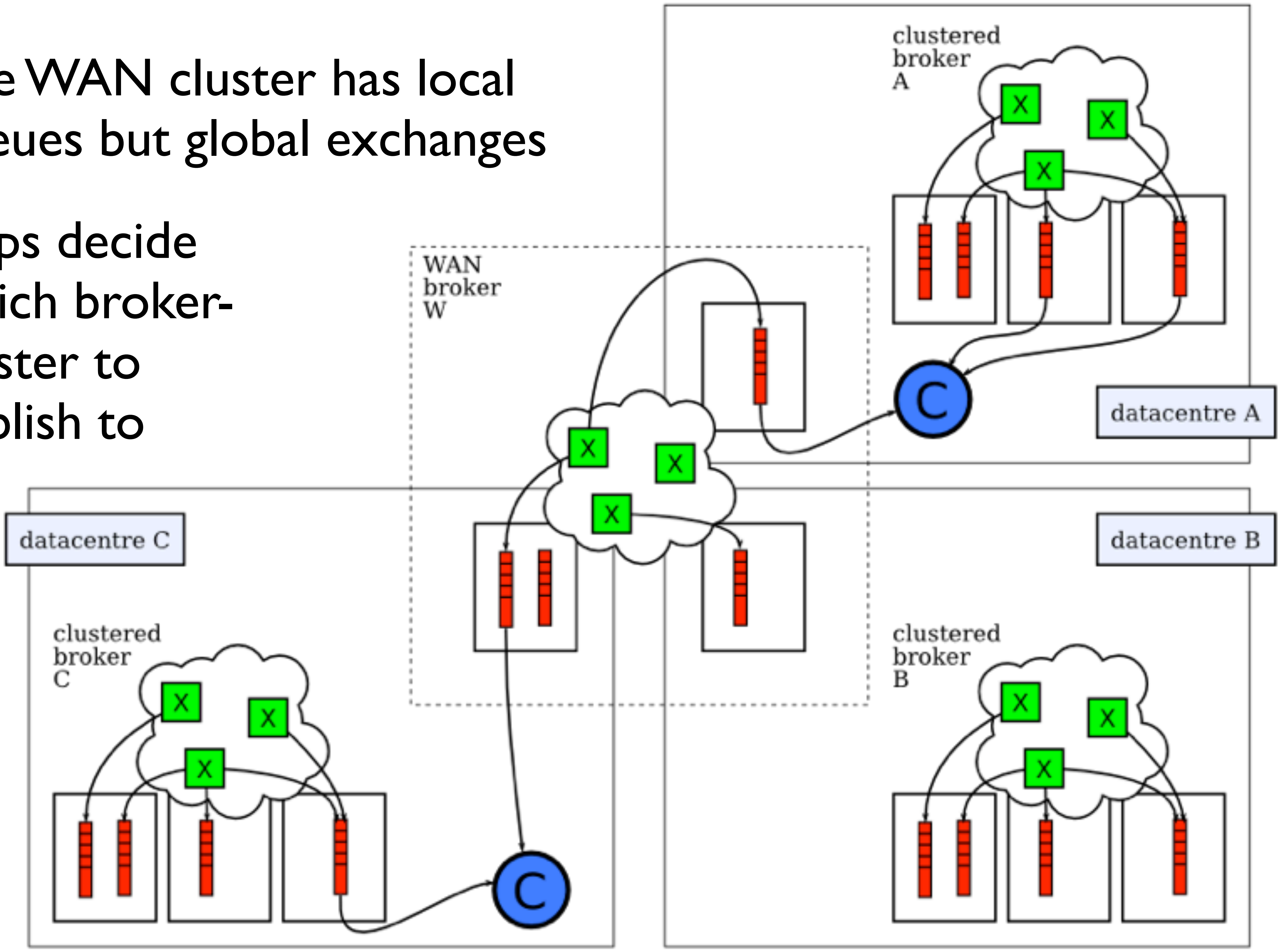
- By record – parallelisable data lets separate broker clusters handle separate *shards* of your data
- By component – each independent part of your application can use separate brokers; “federation of distributed systems”

Once even a clustered single broker no longer copes with the load you’re throwing at it, it’s time to move on to the decentralised network we saw before in Paul Baran’s diagram earlier. If your data parallelises nicely, you can shard your data, effectively running a collection of identical systems side-by-side, each responsible for a subset of the database; and if your application has some serial processing aspect to it, you can let each stage use a separate broker cluster for communicating with the next stage. There’s no intrinsic requirement to route all your messages through a single broker.

WAN Cluster

The WAN cluster has local queues but global exchanges

Apps decide which broker-cluster to publish to



A particularly interesting multi-broker design works well when you have multiple datacentres or multiple branch offices. A RabbitMQ WAN cluster with a node in each location can be used to perform message routing, filtering and delivery between locations, and local clusters can be used for within-location communication. Since queues in a RabbitMQ cluster live on the node they are created on, you get the advantages of a distributed, WAN-scale routing setup with the actual mailboxes holding messages intended for applications nice and close to the applications themselves. It’s not seamless, in that applications to have to know which cluster to talk to for each particular task, but it is simple to set up, to understand, and to use.

Overfeeding the server

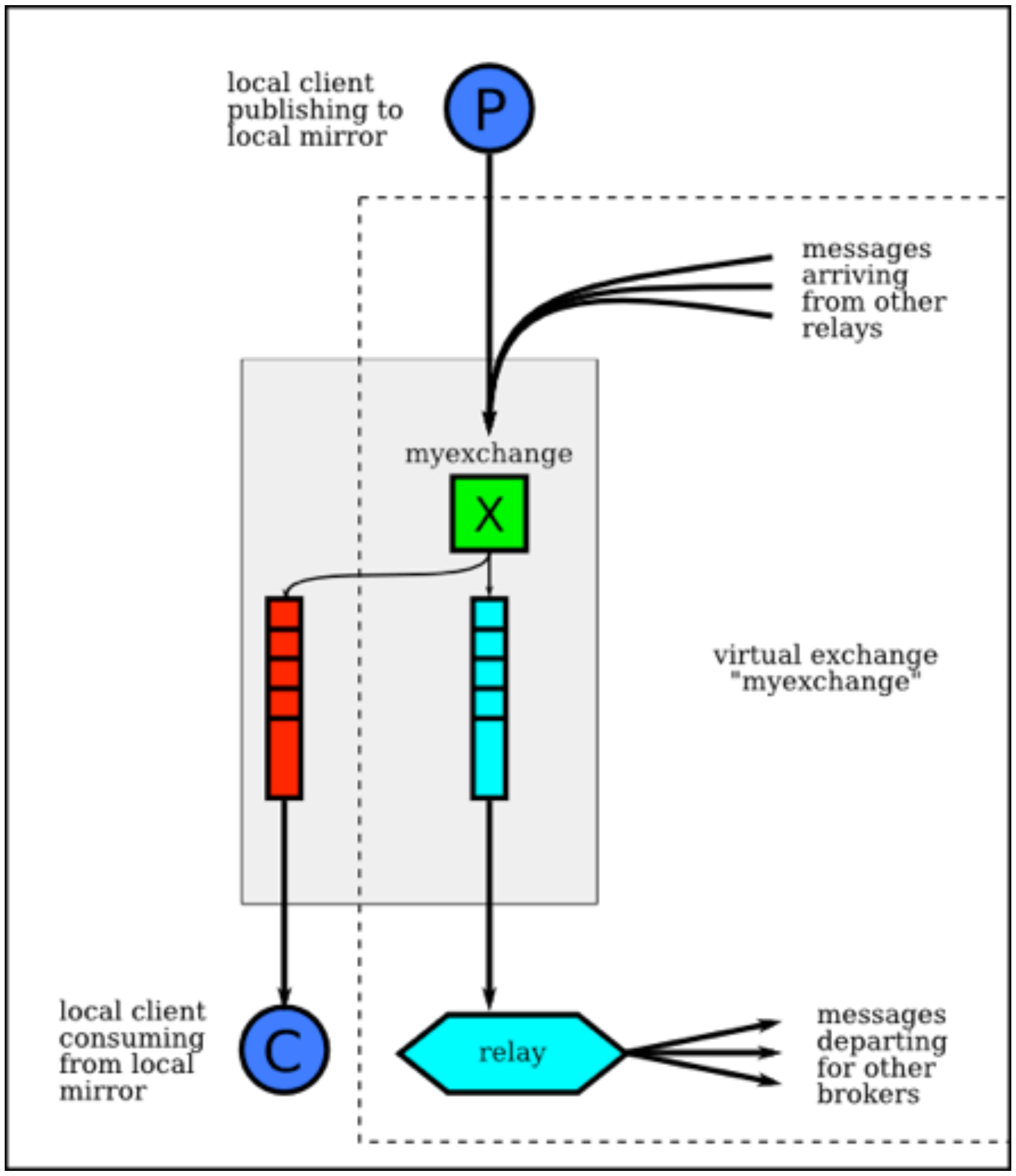
- Message backlogs can cause catastrophe
- Goal: `O(0)` RAM cost per message
- Spool out to disk when memory pressure gets high
- Testing and tuning: should land for RabbitMQ 1.7

Before I move on to talking about synchronisation and relaying, a quick word about data storage capacity...

Synchronisation

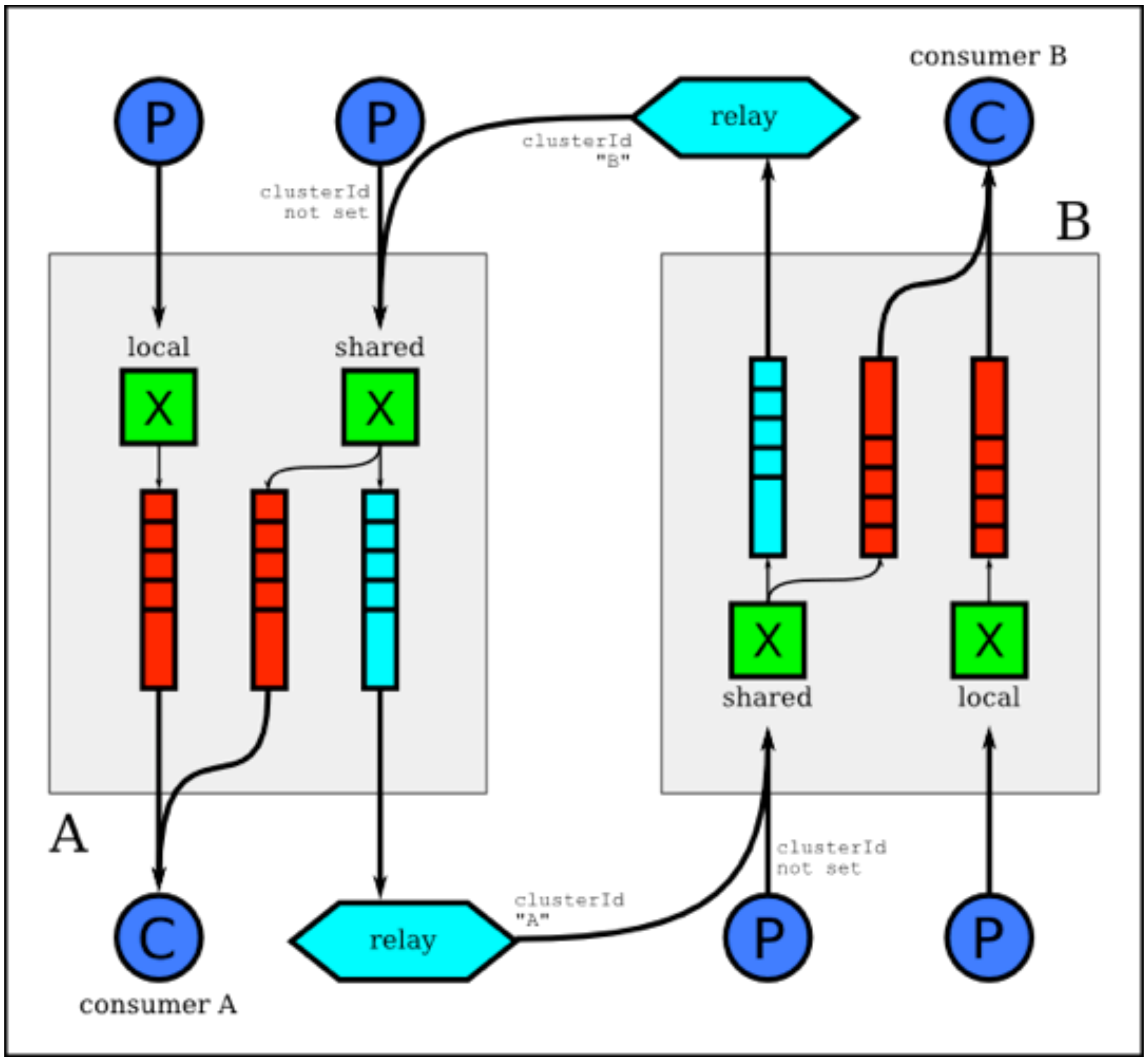
When clustering might not be right:

- huge networks
- intermittent connectivity
- ruling bandwidth with an iron fist
- different administrative domains


local client publishing to local mirror
P
messages arriving from other relays
myexchange
X
virtual exchange "myexchange"
local client consuming from local mirror
C
relay
messages departing for other brokers

So clustering's fine: but in some cases, it's not enough. [Explanation of when clustering might not be right] [Explanation of diagram]

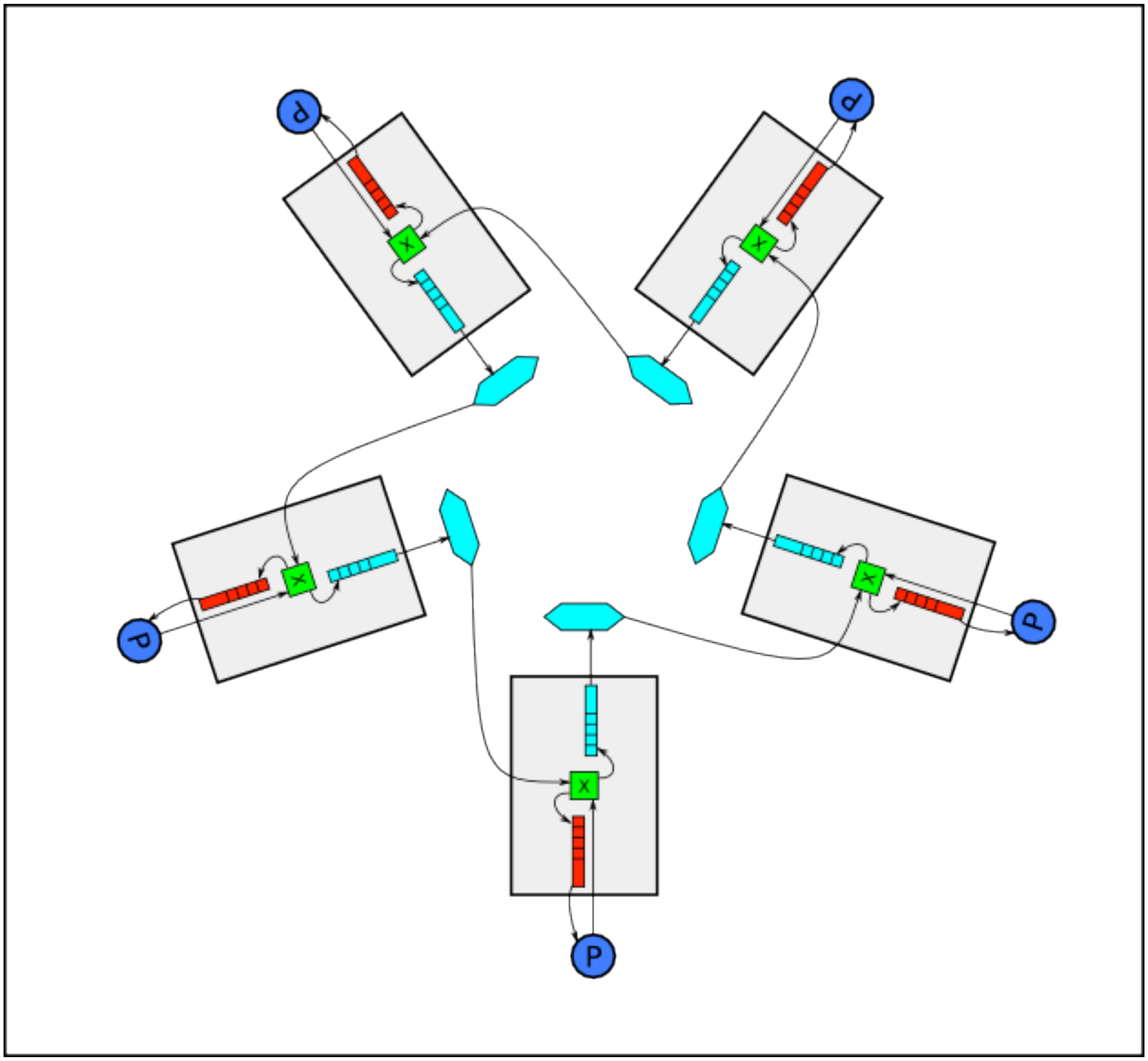
Synchronisation



The important part: Don’t Create Mail Loops!
AMQP has a standard field called “cluster ID” that isn’t formally defined, but that every AMQP broker is required to pass on. This is a really convenient place to put mail-loop-breaking information into: keep a record of who has seen each message, to avoid forwarding it inappropriately.

Ring

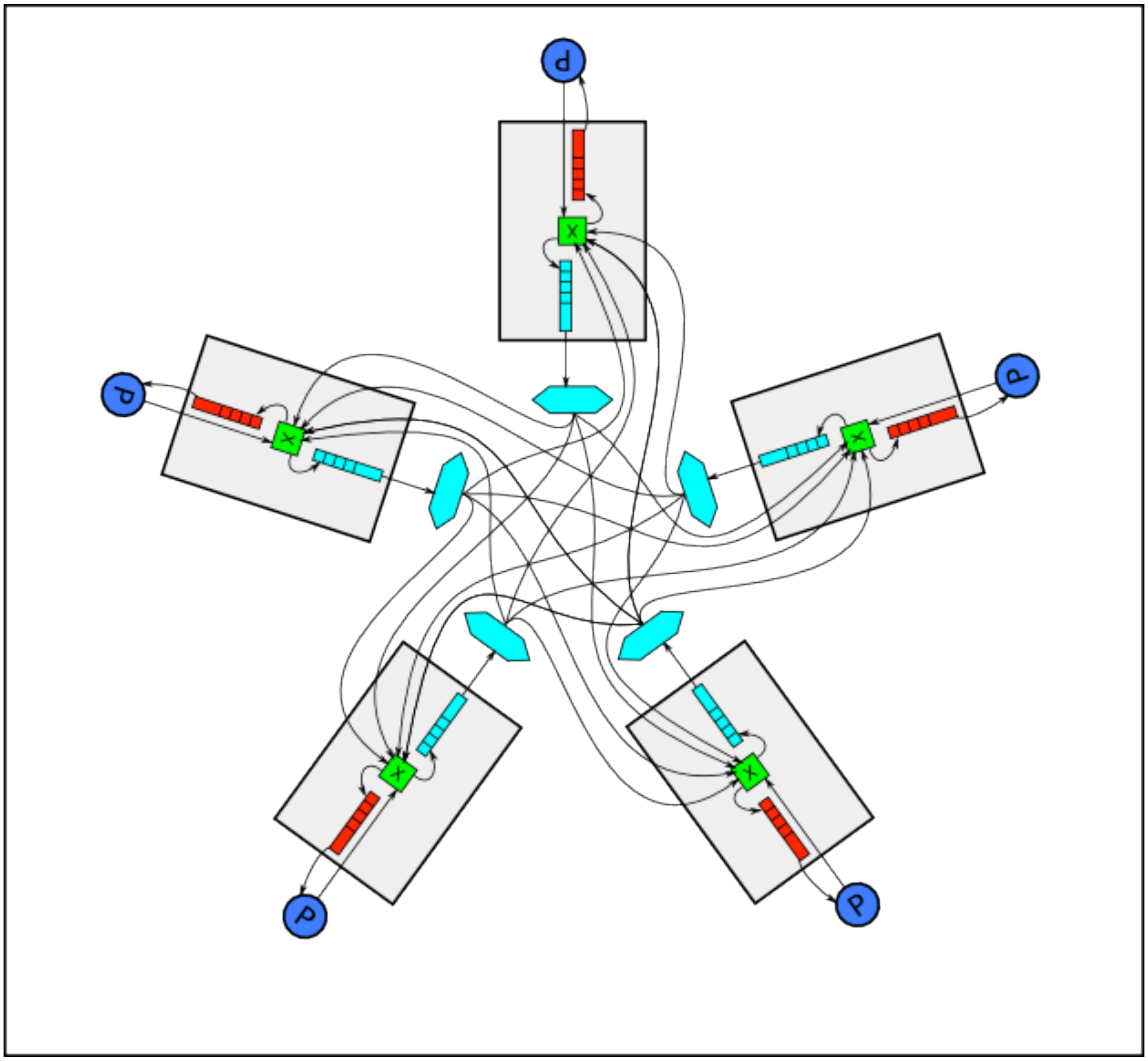
Pass it on to your neighbour if your neighbour's name isn't in the list yet



High latency, economical with bandwidth, poor fault-tolerance

Complete Graph

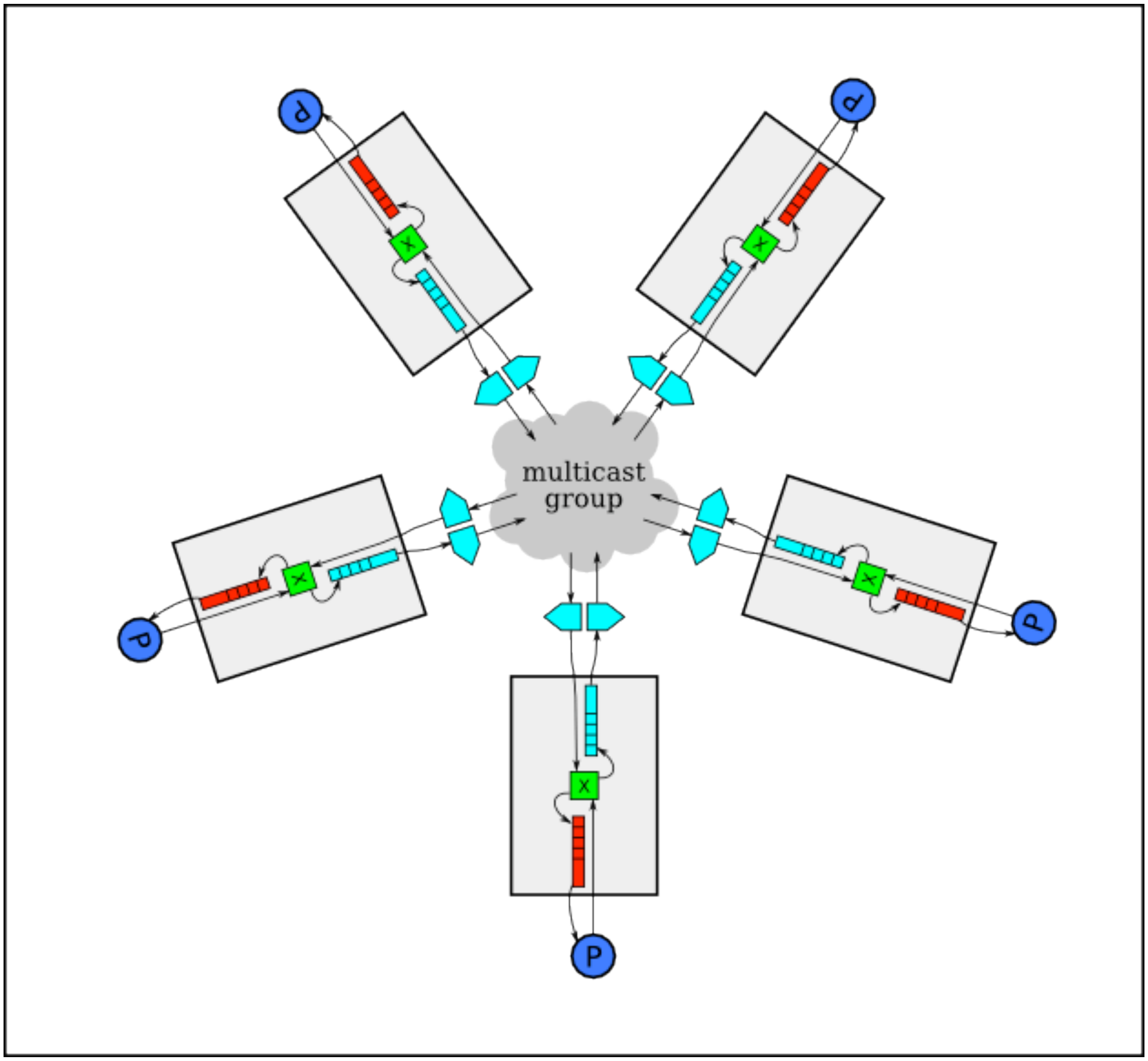
Pass it on to your neighbour if it hasn’t been labelled at all yet



Low latency, wasteful of bandwidth, good fault-tolerance

Multicast

Pass it on to your neighbour if it hasn't been labelled at all yet



Low latency, economical, but of course unreliable!

We've been getting good initial results experimenting with reliable multicast synchronisation groups in which there are three roles: publishers, subscribers, and acknowledger-replayers. Publishers keep retrying until an acknowledger ACKs the delivery. Subscribers are silent unless they detect that they've missed a message, in which case they request it from a replayer using a NACK. There are so many variables to explore here, we're very much just beginning.

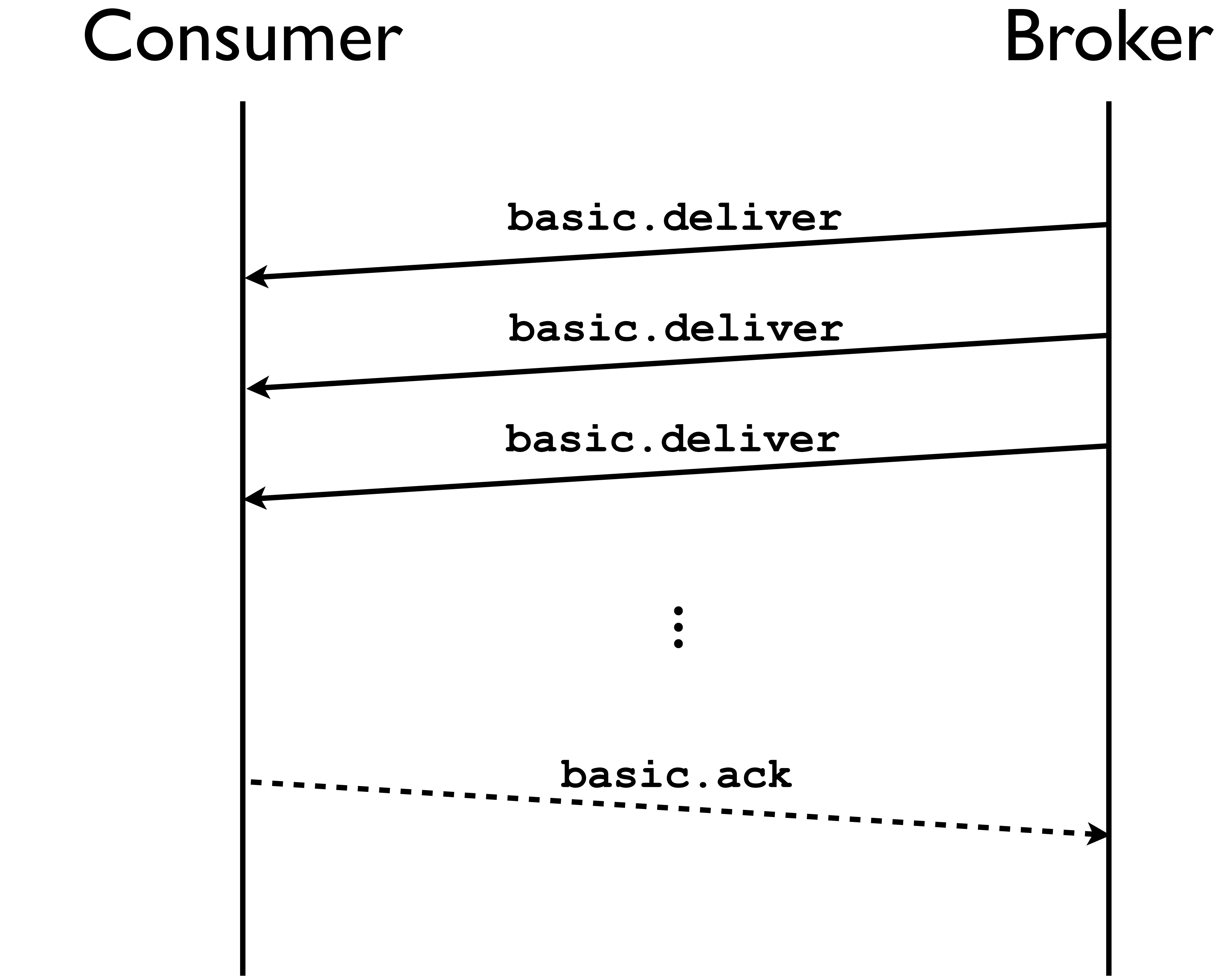
Availability

- Broker Availability
- Application/Service Availability
- Data Availability

Here’s where making the system robust in the face of failure comes in.

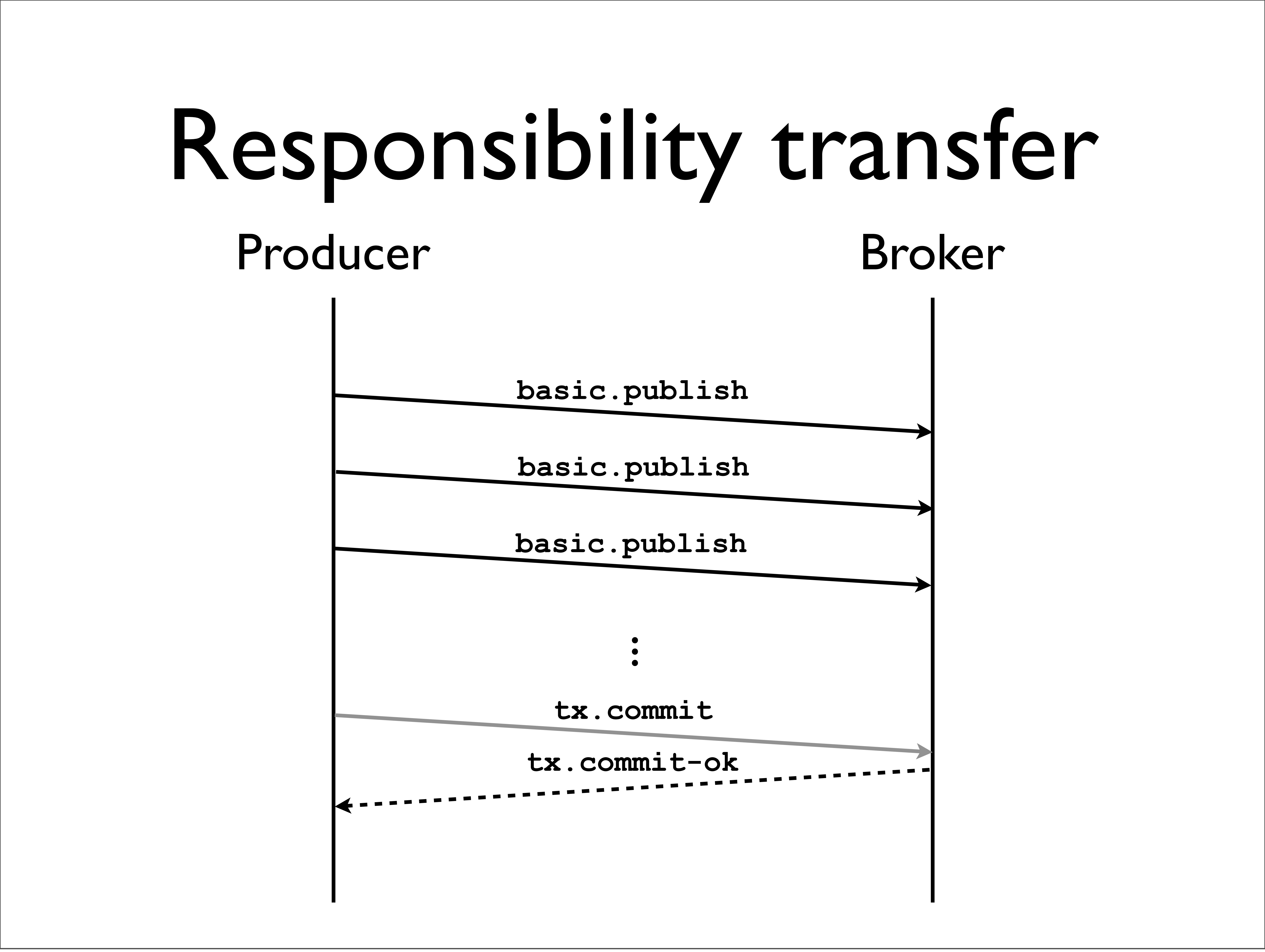
Broker availability – plain old clustering, or running many separate instances (warm/hot standby)
App availability – we’ve already covered the basic idea in the “simple load balancing slide”
Data availability – not losing messages, reliable delivery (better known as responsibility transfer), redundancy for improving the odds

Responsibility transfer



Thursday, 9 July 2009

So to address responsibility transfer first: all that's required is that the receiving party acknowledge acceptance of responsibility for a received message to the sender. In AMQP, that's done with a "basic.ack" message, which lets the server know that the client received the message OK and that it's safe to delete the server's copy.



Responsibility transfer

Producer

Broker

basic.publish

basic.publish

basic.publish

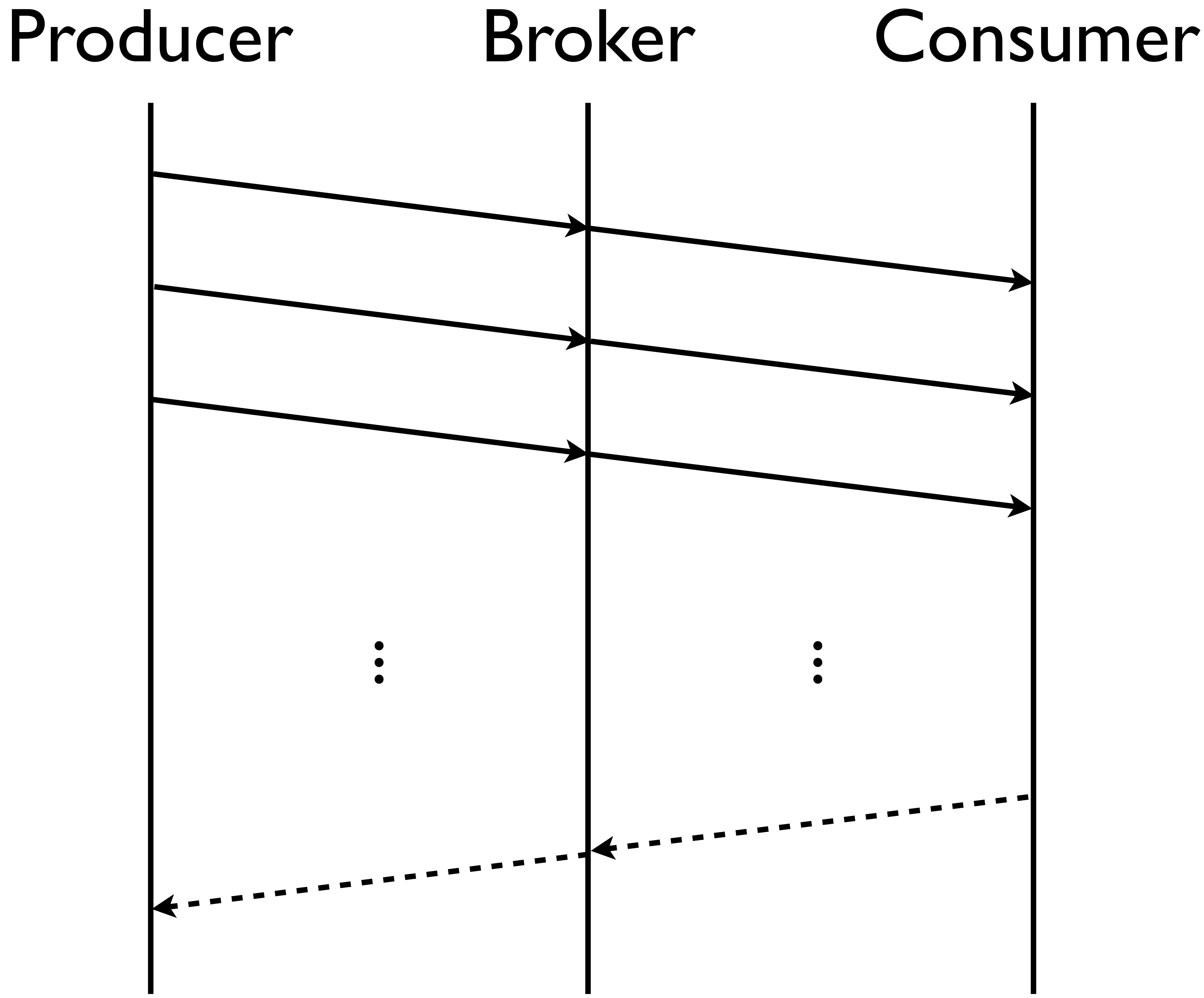
⋮

tx.commit

tx.commit-ok

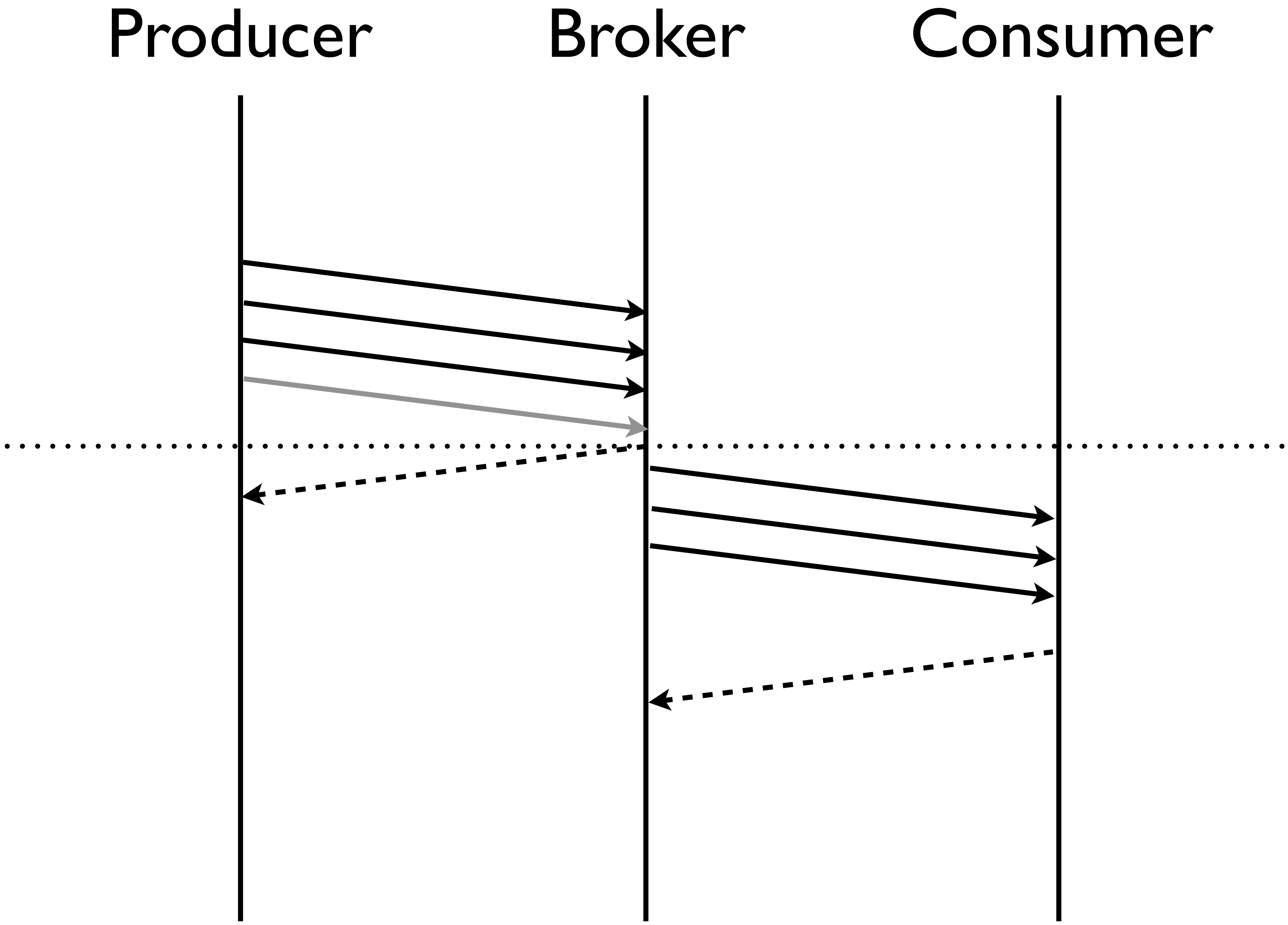
Things are a little awkward on the publishing side with AMQP, though. The protocol isn’t symmetric – yet! – so you have to use a transactional delivery mode. The “commit-ok” response coming back from the server acts as the acknowledgement for the published messages.

Responsibility transfer



Ideally, AMQP would permit this kind of situation, where the broker passes messages on to consumers as they arrive and acknowledges them to the sender once the ultimate consumer takes responsibility for them, but...

Responsibility transfer



... because of the grouping induced by the use of transactional mode, this is the closest you can get by staying within the letter of the specification. The server isn't free to pass on the messages it's received until it hears that transaction commit request from the publisher.

Redundancy for HA

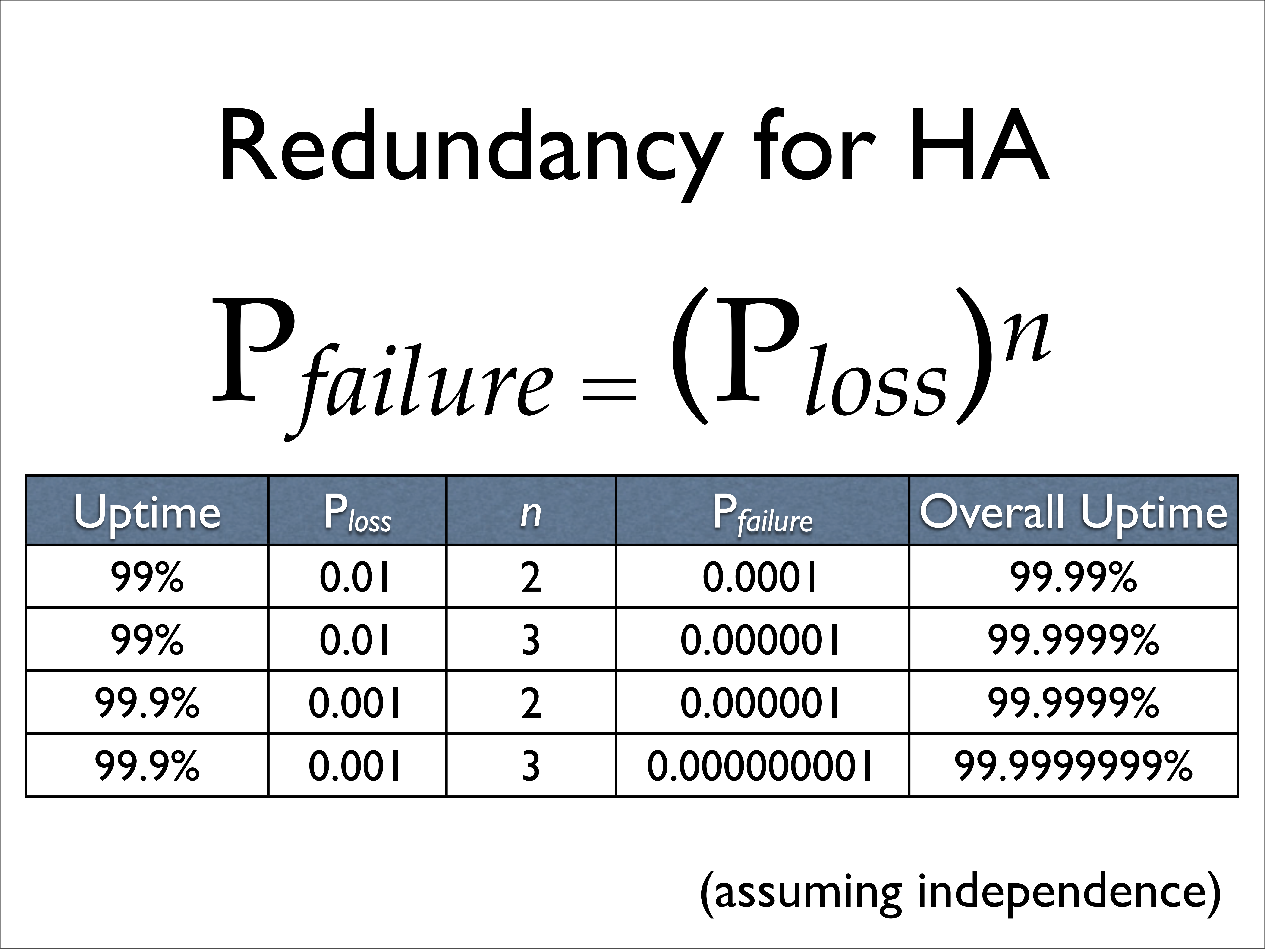
$$P_{failure} = (P_{loss})^n$$

(assuming independence)

The next aspect of Availability I want to mention is the use of redundant intermediate systems for broker availability.

The probability of total failure decreases exponentially with each independent redundant path for data through the system. Just to illustrate that, here's a back-of-the-envelope table; you can see how redundant data paths really pay off very quickly.

Of course, this simple equation only applies if the probabilities of each delivery failing are independent; usually they'll not be completely independent. The idea is to make each delivery path as independent as possible for maximum reliability.



Redundancy for HA

$$P_{failure} = (P_{loss})^n$$

Uptime	P_{loss}	n	$P_{failure}$	Overall Uptime
99%	0.01	2	0.0001	99.99%
99%	0.01	3	0.000001	99.9999%
99.9%	0.001	2	0.000001	99.9999%
99.9%	0.001	3	0.000000001	99.9999999%

(assuming independence)

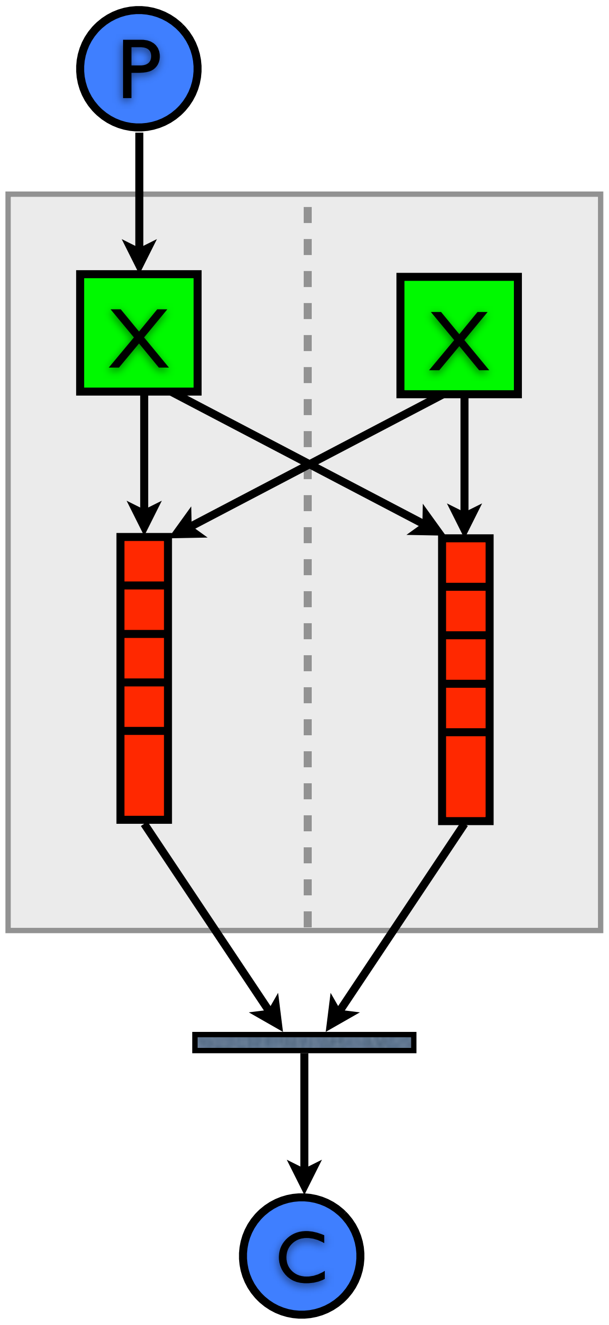
The next aspect of Availability I want to mention is the use of redundant intermediate systems for broker availability.

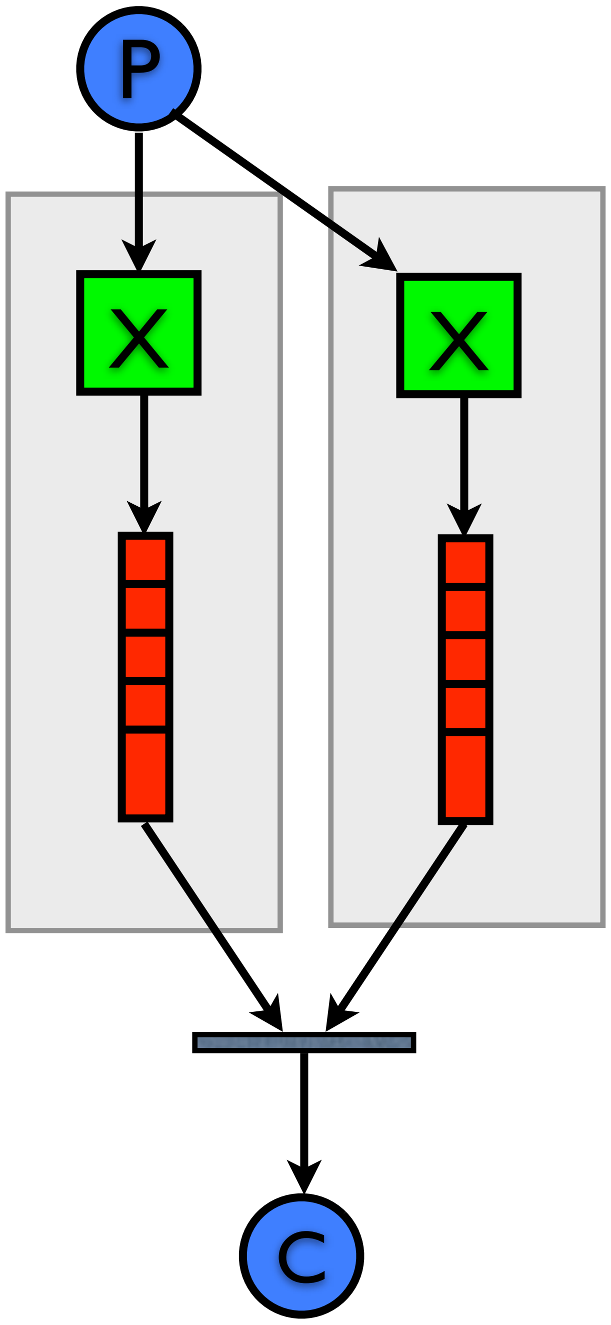
The probability of total failure decreases exponentially with each independent redundant path for data through the system. Just to illustrate that, here’s a back-of-the-envelope table; you can see how redundant data paths really pay off very quickly.

Of course, this simple equation only applies if the probabilities of each delivery failing are independent; usually they’ll not be completely independent. The idea is to make each delivery path as independent as possible for maximum reliability.

Redundancy for HA

One broker, two nodes each





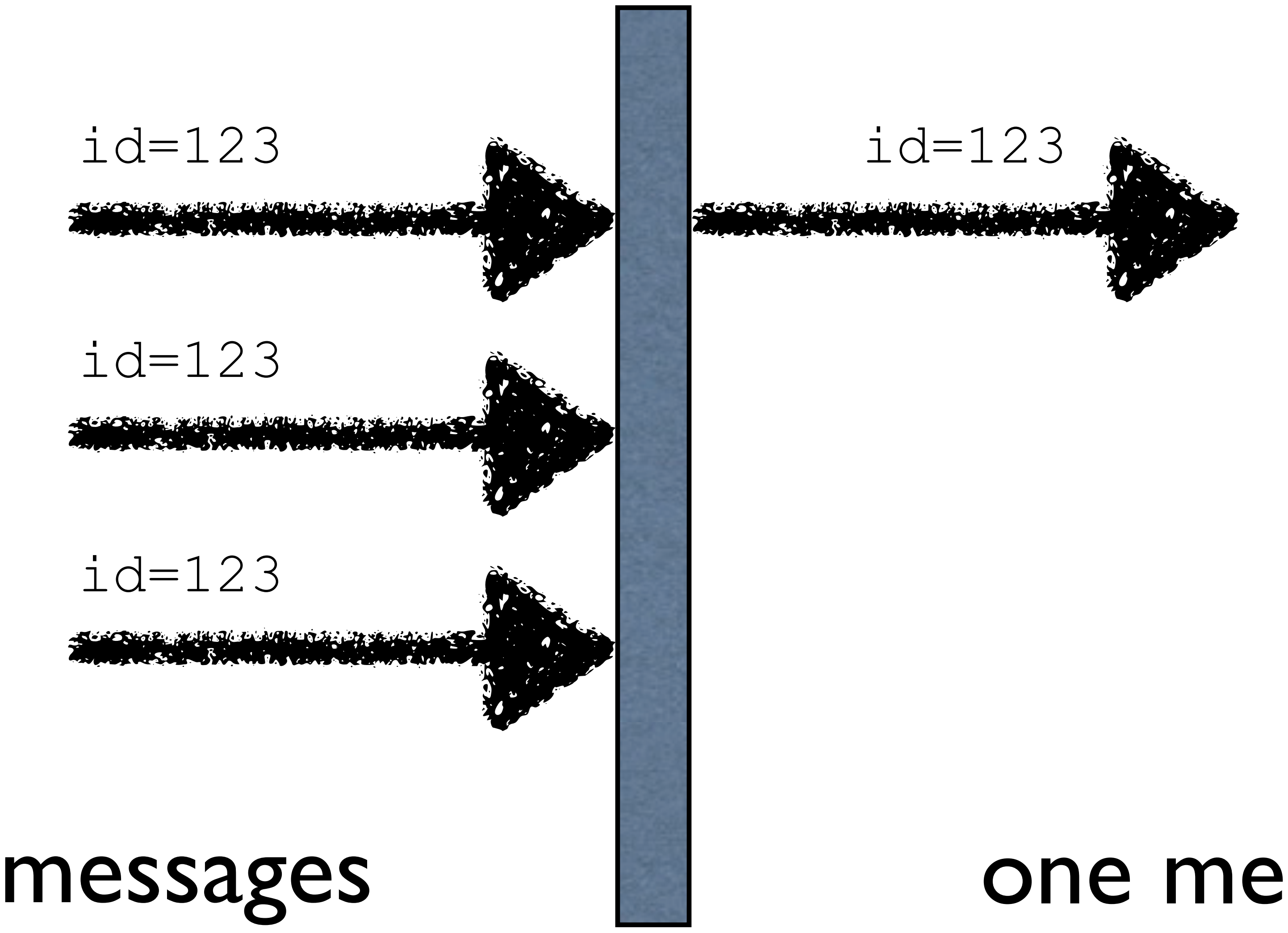
Two brokers, one node each

With redundant paths through the broker, individual outages can be tolerated. The main difference between the two configurations shown here is that on the right, the producer needs to explicitly send every message to both broker instances. On the left, the built-in RabbitMQ clustering takes care of that for you; on the other hand, the server admin tasks involved in managing a cluster are slightly more complex than managing separate broker instances, so there's a tradeoff there.

At the bottom of the two pictures here, there's a horizontal bar, representing a deduplication filter.

Deduplication

(“Idempotency Barrier”)



id=123

id=123

id=123

id=123

many messages enter

one message leaves

The basic idea here is ...

Need a predicate for determining message identity: a simple, workable solution is to attach UUIDs to messages, but sometimes hashing (e.g. SHA-1) the content is a good solution

Fault tolerance, but also exactly-once delivery: the server is free to redeliver previously-attempted messages in certain circumstances, and a deduplication filter can be used to keep things sane for the message receivers.

(((Needs a memory of what it’s seen. Can expire things: keep them for the duration of the longest *automatically-recovered* service outage you want to support, plus a bit. So long as longer outages are manually-recovered, the queues can be purged and the system monitored to ensure that nothing’s lost or duplicated.)))

Network Management

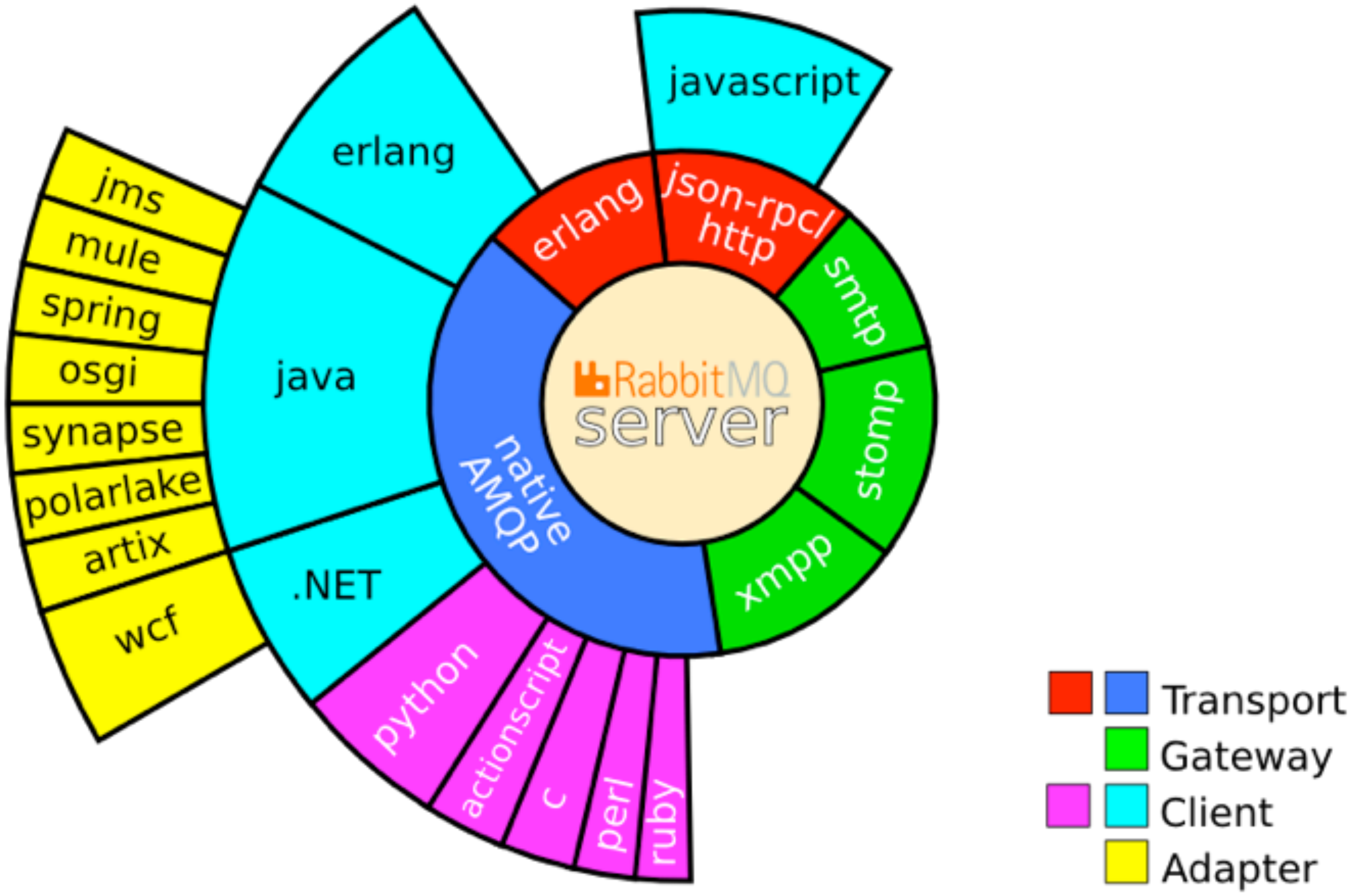
- Small systems can be managed by hand
- Large systems need automation:
    - Naming & address management
    - Service directories
    - Authentication & authorization
    - Logging, monitoring & alerting

Finally, a few words about management and monitoring of large networks.

Large systems: e.g. the internet itself, the cellular network (all those iPhones!), auto-scaling systems running in the cloud

Messaging can be applied to management and monitoring tasks just as well as to application tasks. Brokers can provide rich support for all these tasks; systems like Nanite, which uses RabbitMQ for managing agents as well as letting them communicate, are helping to discover and define the kinds of things that can be moved into the broker in these areas.

The Future



We’re working on making it as easy as possible to use the design patterns I’ve been talking about. In particular, solid support for synchronising relays, for gateway plugins, and for deduplication (in support of redundancy-for-HA and so forth) is high on our to-do list.

People are already building things around RabbitMQ and making it easy to deploy in the cloud; I’m looking forward to seeing what people do with it.

Questions?

http://www.rabbitmq.com/how