programming is terrible
lessons learned from a life wasted

2017-06-28: How do you cut a monolith in half?

It depends.

The problem with distributed systems, is that no matter what the question is, the answer is inevitably 'It Depends'.

When you cut a larger service apart, where you cut depends on latency, resources, and access to state, but it also depends on error handling, availably and recovery processes. It depends, but you probably don't want to depend on a message broker.

Using a message broker to distribute work is like a cross between a load balancer with a database, with the disadvantages of both and the advantages of neither.

Message brokers, or persistent queues accessed by publish-subscribe, are a popular way to pull components apart over a network. They're popular because they often have a low setup cost, and provide easy service discovery, but they can come at a high operational cost, depending where you put them in your systems.

In practice, a message broker is a service that transforms network errors and machine failures into filled disks. Then you add more disks. The advantage of publish-subscribe is that it isolates components from each other, but the problem is usually gluing them together.

For short-lived tasks, you want a load balancer

For short-lived tasks, publish-subscribe is a convenient way to build a system quickly, but you inevitably end up implementing a new protocol atop. You have publish-subscribe, but you really want request-response. If you want something computed, you'll probably want to know the result.

Starting with publish-subscribe makes work assignment easy: jobs get added to the queue, workers take turns to remove them. Unfortunately, it makes finding out what happened quite hard, and you'll need to add another queue to send a result back.

Once you can handle success, it is time to handle the errors. The first step is often adding code to retry the request a few times. After you DDoS your system, you put a call to sleep(). After you slowly DDoS your system, each retry waits twice as long as the previous.

(Aside: Accidental synchronisation is still a problem, as waiting to retry doesn't prevent a lot of things happening at once.)

As workers fail to keep up, clients give up and retry work, but the earlier request is still waiting to be processed. The solution is to move some of the queue back to clients, asking them to hold onto work until work has been accepted: back-pressure, or acknowledgements.

Although the components interact via publish-subscribe, we've created a request-response protocol atop. Now the message broker is really only doing two useful things: service discovery, and load balancing. It is also doing two not-so-useful thing: enqueuing requests, and persisting them.

For short-lived tasks, the persistence is unnecessary: the client sticks around for as long as the work needs to be done, and handles recovery. The queuing isn't that necessary either.

queues inevitably run in two states: full, or empty. If your queue is running full, you haven't pushed enough work to the edges, and if it is running empty, it's working as a slow load balancer.

A mostly empty queue is still first-come-first-served, serving as point of contention for requests. A broker often does nothing but wait for workers to poll for new messages. If your queue is meant to run empty, why wait to forward on a request.

(Aside: Something like random load balancing will work, but join-idle-queue is well worth your time investigating)

For distributing short-lived tasks, you can use a message broker, but you'll be building a load balancer, along with an ad-hoc RPC system, with extra latency.

For long-lived tasks, you'll need a database

A load balancer with service discovery won't help you with long running tasks, or work that outlives the client, or manage throughput. You'll want persistence, but not in your message broker. For long-lived tasks, you'll want a database instead.

Although the persistence and queueing were obstacles for short-lived tasks, the disadvantages are less obvious for long-lived tasks, but similar things can go wrong.

If you care about the result of a task, you'll want to store that it is needed somewhere other than in the persistent queue. If the task is run but fails midway, something will have to take responsibility for it, and the broker will have forgotten. This is why you want a database.

Duplicates in a queue often cause more headaches, as long-lived tasks have more opportunities to overlap. Although we're using the broker to distribute work, we're also using it implicitly as a mutex. To stop work from overlapping, you implement a lock atop. After it breaks a couple of times, you replace it with leases, adding timeouts.

(Note: This is not why you want a database, using transactions for long running tasks is suffering. Long running processes are best modelled as state machines.)

When the database becomes the primary source of truth, you can handle a broker going offline, or a broker losing the contents of a queue, by backfilling from the database. As a result, you don't need to directly enqueue work with the broker, but mark it as required in the database, and wait for something else to handle it.

Assuming that something else isn't a human who has been paged.

A message pump can scan the database periodically and send work requests to the broker. Enqueuing work in batches can be an effective way of making an expensive database call survivable. The pump responsible for enqueuing the work can also track if it has completed, and so handle recovery or retries too.

Backlog is still a problem, so you'll want to use back-pressure to keep the queue fairly empty, and only fill from the database when needed. Although a broker can handle temporary overload, back-pressure should mean it never has to.

At this point the message broker is really providing two things: service discovery, and work assignment, but really you need a scheduler. A scheduler is what scans a database, works out which jobs need to run, and often where to run them too. A scheduler is what takes responsibility for handling errors.

(Aside: Writing a scheduler is hard. It is much easier to have 1000 while loops waiting for the right time, than one while loop waiting for which of the 1000 is first. A scheduler can track when it last ran something, but the work can't rely on that being the last time it ran. Idempotency isn't just your friend, it is your saviour.)

You can use a message broker for long-lived tasks, but you'll be building a lock manager, a database, and a scheduler, along with yet another home-brew request-response system.

Publish-Subscribe is about isolating components

The problem with running tasks with publish-subscribe is that you really want request-response. The problem with using queues to assign work is that you don't want to wait for a worker to ask.

The problem with relying on a persistent queue for recovery, is that recovery must get handled elsewhere, and the problem with brokers is nothing else makes service discovery so trivial.

Message brokers can be misused, but it isn't to say they have no use. Brokers work well when you need to cross system boundaries.

Although you want to keep queues empty between components, it is convenient to have a buffer at the edges of your system, to hide some failures from external clients. When you handle external faults at the edges, you free the insides from handling them. The inside of your system can focus on handling internal problems, of which there are many.

A broker can be used to buffer work at the edges, but it can also be used as an optimisation, to kick off work a little earlier than planned. A broker can pass on a notification that data has been changed, and the system can fetch data through another API.

(Aside: If you use a broker to speed up a process, the system will grow to rely on it for performance. People use caches to speed up database calls, but there are many systems that simply do not work fast enough until the cache is warmed up, filled with data. Although you are not relying on the message broker for reliability, relying on it for performance is just as treacherous.)

Sometimes you want a load balancer, sometimes you'll need a database, but sometimes a message broker will be a good fit.

Although persistence can't handle many errors, it is convenient if you need to restart with new code or settings, without data loss. Sometimes the error handling offered is just right.

Although a persistent queue offers some protection against failure, it can't take responsibility for when things go wrong halfway through a task. To be able to recover from failure you have to stop hiding it, you must add acknowledgements, back-pressure, error handling, to get back to a working system.

A persistent message queue is not bad in itself, but relying on it for recovery, and by extension, correct behaviour, is fraught with peril.

Systems grow by pushing responsibilities to the edges

Performance isn't easy either. You don't want queues, or persistence in the central or underlying layers of your system. You want them at the edges.

It's slow is the hardest problem to debug, and often the reason is that something is stuck in a queue. For long and short-lived tasks, we used back-pressure to keep the queue empty, to reduce latency.

When you have several queues between you and the worker, it becomes even more important to keep the queue out of the centre of the network. We've spent decades on tcp congestion control to avoid it.

If you're curious, the history of tcp congestion makes for interesting reading. Although the ends of a tcp connection were responsible for failure and retries, the routers were responsible for congestion: drop things when there is too much.

The problem is that it worked until the network was saturated, and similar to backlog in queues, when it broke, errors cascaded. The solution was similar: back-pressure. Similar to sleeping twice as long on errors, tcp sends half as many packets, before gradually increasing the amount as things improve.

Back-pressure is about pushing work to the edges, letting the ends of the conversation find stability, rather than trying to optimise all of the links in-between in isolation. Congestion control is about using back-pressure to keep the queues in-between as empty as possible, to keep latency down, and to increase throughput by avoiding the need to drop packets.

Pushing work to the edges is how your system scales. We have spent a lot of time and a considerable amount of money on IP-Multicast, but nothing has been as effective as BitTorrent. Instead of relying on smart routers to work out how to broadcast, we rely on smart clients to talk to each other.

Pushing recovery to the outer layers is how your system handles failure. In the earlier examples, we needed to get the client, or the scheduler to handle the lifecycle of a task, as it outlived the time on the queue.

Error recovery in the lower layers of a system is an optimisation, and you can't push work to the centre of a network and scale. This is the end-to-end principle, and it is one of the most important ideas in system design.

The end-to-end principle is why you can restart your home router, when it crashes, without it having to replay all of the websites you wanted to visit before letting you ask for a new page. The browser (and your computer) is responsible for recovery, not the computers in between.

This isn't a new idea, and Erlang/OTP owes a lot to it. OTP organises a running program into a supervision tree. Each process will often have one process above it, restarting it on failure, and above that, another supervisor to do the same.

(Aside: Pipelines aren't incompatible with process supervision, one way is for each part to spawn the program that reads its output. A failure down the chain can propagate back up to be handled correctly.)

Although each program will handle some errors, the top levels of the supervision tree handle larger faults with restarts. Similarly, it's nice if your webpage can recover from a fault, but inevitably someone will have to hit refresh.

The end-to-end principle is realising that no matter how many exceptions you handle deep down inside your program, some will leak out, and something at the outer layer has to take responsibility.

Although sometimes taking responsibility is writing things to an audit log, and message brokers are pretty good at that.

Aside: But what about replicated logs?

> *"How do I subscribe to the topic on the message broker?"*
>
> *"It's not a message broker, it's a replicated log"*
>
> *"Ok, How do I subscribe to the replicated log"*
>
> From 'I believe I did, Bob', jrecursive

Although a replicated log is often confused with a message broker, they aren't immune from handling failure. Although it's good the components are isolated from each other, they still have to be integrated into the system at large. Both offer a one way stream for sharing, both offer publish-subscribe like interfaces, but the intent is wildly different.

A replicated log is often about auditing, or recovery: having a central point of truth for decisions. Sometimes a replicated log is about building a pipeline with fan-in (aggregating data), or fan-out (broadcasting data), but always building a system where data flows in one direction.

The easiest way to see the difference between a replicated log and a message broker is to ask an engineer to draw a diagram of how the pieces connect.

If the diagram looks like a one-way system, it's a replicated log. If almost every component talks to it, it's a message broker. If you can draw a flow-chart, it's a replicated log. If you take all the arrows away and you're left with a venn diagram of 'things that talk to each other', it's a message broker.

Be warned: A distributed system is something you can draw on a whiteboard pretty quickly, but it'll take hours to explain how all the pieces interact.

You cut a monolith with a protocol

How you cut a monolith is often more about how you are cutting up responsibility within a team, than cutting it into components. It really does depend, and often more on the social aspects than the technical ones, but you are still responsible for the protocol you create.

Distributed systems are messy because of how the pieces interact over time, rather than which pieces are interacting. The complexity of a distributed system does not come from having hundreds of machines, but hundreds of ways for them to interact. A protocol must take into account performance, safety, stability, availability, and most importantly, error handling.

When we talk about distributed systems, we are talking about power structures: how resources are allocated, how work is divided, how control is shared, or how order is kept across systems ostensibly built out of well meaning but faulty components.

A protocol is the rules and expectations of participants in a system, and how they are beholden to each other. A protocol defines who takes responsibility for failure.

The problem with message brokers, and queues, is that no-one does.

Using a message broker is not the end of the world, nor a sign of poor engineering. Using a message broker is a tradeoff. Use them freely knowing they work well on the edges of your system as buffers. Use them wisely knowing that the buck has to stop somewhere else. Use them cheekily to get something working.

I say don't rely on a message broker, but I can't point to easy off-the-shelf answers. HTTP and DNS are remarkable protocols, but I still have no good answers for service discovery.

Lots of software regularly gets pushed into service way outside of its designed capabilities, and brokers are no exception. Although the bad habits around brokers and the relative ease of getting a prototype up and running lead to nasty effects at scale, you don't need to build everything at once.

The complexity of a system lies in its protocol not its topology, and a protocol is what you create when you cut your monolith into pieces. If modularity is about building software, protocol is about how we break it apart.

> *The main task of the engineering analyst is not merely to obtain "solutions" but is rather to understand the dynamic behaviour of the system in such a way that the secrets of the mechanism are revealed, and that if it is built it will have no surprises left for [them]. Other than exhaustive physical experimentations, this is the only sound basis for engineering design, and disregard of this cardinal principle has not infrequently lead to disaster.*
>
> From "Analysis of Nonlinear Control Systems" by Dustan Graham and Duane McRuer, p 436

Protocol is the reason why 'it depends', and the reason why you shouldn't depend on a message broker: You can use a message broker to glue systems together, but never use one to cut systems apart.