Avoiding insurmountable queue backlogs

David Yanacek

Algorithms imitating life

Ever since my first computer science course in college, I've been interested in how algorithms play out in the real world. When we think about some of the things that happen in the real world, we can come up with algorithms that mimic those things. I especially do this when I'm stuck in a line, like in the grocery store, in traffic, or at the airport. I've found that being bored while standing in lines provides great opportunities to ponder queueing theory.

Over a decade ago, I spent a day working in an Amazon fulfillment center. I was guided by an algorithm, picking items off of shelves, moving items from one box into another, moving bins around. Working in parallel with so many other people, I found it beautiful to be a part of what is essentially an amazingly orchestrated physical merge sort.

In queueing theory, the behavior of queues when they are short is relatively uninteresting. After all, when a queue is short, everyone is happy. It's only when the queue is backlogged, when the line to an event goes out the door and around the corner, which people start thinking about throughput and prioritization.

In this article, I discuss strategies we use at Amazon to deal with queue backlog scenarios – design approaches we take to drain queues quickly and to prioritize workloads. Most importantly, I describe how to prevent queue backlogs from building up in the first place. In the first half, I describe scenarios that lead to backlogs, and in the second half, I describe many approaches used at Amazon to avoid backlogs or deal with them gracefully.

The duplicitous nature of queues

Queues are powerful tools for building reliable asynchronous systems. Queues allow one system to accept a message from another system, and persist the message until it is fully processed, even in the face of long outages, server failures, or problems with dependent systems. Rather than dropping messages when a failure occurs, the queue re-drives the messages until they are successfully processed. In the end, a queue increases a system's durability and availability, at the price of occasional increased latency due to retries.

At Amazon, we build many asynchronous systems that take advantage of queues. Some of these systems process workflows that might take a long time and that involve physical things moving around in the world, like fulfilling orders placed on amazon.com. Other systems coordinate steps that might take a non-trivial amount of time. For example, Amazon RDS requests EC2 instances, waits for them to launch, and then configures your databases for you. Other systems take advantage of batching. For example, systems involved in ingesting CloudWatch metrics and logs pull in a bunch of data, and then aggregate and "flatten" it in chunks.

While it may be easy to see the benefits of a queue for processing messages asynchronously, the risks of using a queue are more subtle. We have found over the years that queueing that is meant to improve availability can backfire. In fact, it can dramatically increase the recovery time after an outage.

In a queue-based system, when processing stops but messages keep arriving, the message debt can accumulate into a large backlog, driving up processing time. Work can be completed too late for the results to be useful, essentially causing the availability hit that queueing was meant to guard against.

Putting it another way, a queue-based system has two modes of operation, or bimodal behavior. When there is no backlog in the queue, the system's latency is low, and the system is in fast mode. But if a failure or unexpected load pattern causes the arrival rate to exceed the processing rate, it quickly flips into a more sinister operating mode. In this mode, the end-to-end latency grows higher and higher, and it can take a great deal of time to work through the backlog to return to the fast mode.

Queue-based systems

To illustrate queue-based systems in this article, I touch on how two AWS services work under the hood: AWS Lambda, a service that executes your code in response to events without you having worry about the infrastructure that it runs on; and AWS IoT Core, a managed service that lets connected devices easily and securely interact with cloud applications and other devices.

With AWS Lambda, you upload your function code, and then invoke your functions one of two ways:

- Synchronously: where the output of your function is returned to you in the HTTP response
- Asynchronously: where the HTTP response returns immediately, and your function is executed and retried behind the scenes

Lambda makes sure that your function is run, even in the face of server failures, so it needs a durable queue in which to store your request. With a durable queue, your request can be re-driven if your function fails the first time.

With AWS IoT Core, your devices and applications connect and can subscribe to PubSub message topics. When a device or application publishes a message, the applications with matching subscriptions receive their own copy of the message. Much of this PubSub messaging happens asynchronously, because a constrained IoT device does not want to spend its limited resources waiting to ensure that all of the subscribed devices, applications, and systems receive a copy. This is especially important because a subscribed device might be offline when another device publishes a message that it's interested in. When the offline device reconnects, it expects to be brought back up to speed first, and then have its messages delivered to it later (for information on coding your system to manage message delivery after reconnect, see MQTT Persistent Sessions in the *AWS IoT Developer Guide*). There are a variety of flavors of persistence and asynchronous processing that go on behind the scenes to make this happen.

Queue-based systems like these are often implemented with a durable queue. SQS offers durable, scalable, at-least-once message delivery semantics, so Amazon teams including Lambda and IoT regularly use it when building their scalable asynchronous systems. In queue-based systems, a component *produces* data by putting messages into the queue, and another component *consumes* that data by periodically asking for messages, processing messages, and finally deleting them once it's done.

Failures in asynchronous systems

In AWS Lambda, if an invocation of your function is slower than normal (for example, because of a dependency), or if it fails transiently, no data is lost, and Lambda retries your function. Lambda queues up your invoke calls, and when the function starts working again, Lambda works through your function's backlog. But let's consider how long it takes to work through the backlog and return to normal.

Imagine a system that experiences an hour-long outage while processing messages. Regardless of the given rate and processing capacity, recovering from the outage requires double the system's capacity for another hour after the recovery. In practice, the system might have more than double the capacity available, especially with elastic services like Lambda, and recovery could happen faster. On the other hand, other systems that your function interacts with might not be prepared to handle a huge increase in processing as you work through the backlog. When this happens, it can take even longer to catch up. Asynchronous services build up backlogs during outages, leading to long recovery times, unlike synchronous services, which drop requests during outages but have quicker recover times.

Over the years, when thinking about queueing, we've sometimes been tempted to think that latency isn't important for asynchronous systems. Asynchronous systems are often built for durability, or to isolate the immediate caller from latency. However, in practice we've seen that processing time does matter, and often even asynchronous systems are expected to have subsecond or better latency. When queues are introduced for durability, it is easy to miss the tradeoff that causes such high processing latency in the face of a backlog. The hidden risk with asynchronous systems is dealing with large backlogs.

How we measure availability and latency

This discussion of trading off latency for availability raises an interesting question: how do we measure and set goals around latency and availability for an asynchronous service? Measuring error rates from the *producer* perspective gives us part of the availability picture, but not much of it. The producer availability is proportional to the queue availability of the system we're using. So when we build on SQS, our producer availability matches the SQS availability.

On the other hand, if we measure availability on the *consumer* side, that can make the availability of the system look worse than it actually is, because failures might be retried and then succeed on the next attempt.

We also get availability measurements from dead-letter queues (DLQ). If a message runs out of retries, it is dropped or put into a DLQ. A DLQ is simply a separate queue used to store messages that cannot be processed for later investigation and intervention. The rate of dropped or DLQ messages is a good availability measurement, but can detect the problem too late. Though it's a good idea to alarm on DLQ volumes, DLQ information would arrive too late for us to rely on it exclusively to detect problems.

What about latency? Again, *producer*-observed latency mirrors the latency of our queue service itself. Therefore, we focus more on measuring the age of messages that are in the queue. This quickly catches cases where the systems are behind, or are frequently erroring out and causing retries. Services like SQS provide the timestamp of when each message reached the queue. With the timestamp information, every time we take a message off the queue, we can log and produce metrics on how far behind our systems are.

The latency problem can be a little more nuanced though. After all, backlogs are to be expected, and in fact are okay for some messages. For example, in AWS IoT, there are times when a device is expected to go offline or to be slow to read its messages. This is because many IoT devices are low-powered and have spotty internet connectivity. As operators of AWS IoT Core, we need to be able to tell the difference between an expected small backlog caused by devices being offline or choosing to read messages slowly, and an unexpected system-wide backlog.

In AWS IoT, we instrumented the service with another metric: *AgeOfFirstAttempt*. This measurement records *now* minus *message enqueue time*, but only if this was the first time that AWS IoT attempted to deliver a message to a device. This way when devices are backed up, we have a clean metric that isn't polluted with devices retrying messages or enqueueing. To make the metric even cleaner, we emit a second metric – *AgeOfFirstSubscriberFirstAttempt*. In a PubSub system like AWS IoT, there is no practical limit to how many devices or applications can subscribe to a particular topic, so the latency is higher when sending the message to a million devices than when sending to a single device. To give ourselves a stable metric, we emit a timer metric on the first attempt to publish a message to the first subscriber to that topic. We then have other metrics to measure the progress of the system on publishing the remaining messages.

The AgeOfFirstAttempt metric serves as an early warning for a system-wide problem, in large part because it filters out the noise from devices that are choosing to read their messages more slowly. It's worth mentioning that systems like AWS IoT are instrumented with many more metrics than this. But with all of the latency-related metrics available, the strategy of categorizing latency of first attempts separate from the latency of retry attempts is commonly used across Amazon.

Measuring latency and availability of asynchronous systems is challenging, and debugging can also be tricky, because requests bounce around between servers and can be delayed in places outside of each system. To help with distributed tracing, we propagate a *request id* around in our queued messages so that we can piece things together. We commonly use systems like X-Ray to help with this too.

Backlogs in multitenant asynchronous systems

Many asynchronous systems are multitenant, handling work on behalf of many different customers. This adds a complicating dimension to managing latency and availability. The benefit to multitenancy is that it saves us the operational overhead of having to separately operate multiple fleets, and it lets us run combined workloads at much higher resource utilization. However, customers expect it to behave like their very own single-tenant system, with predictable latency and high availability, regardless of the workloads of other customers.

AWS services do not expose their internal queues directly for callers to put messages into. Instead, they implement lightweight APIs to authenticate callers and append caller information to each message before enqueueing. This is similar to the Lambda architecture described earlier: when you invoke a function asynchronously, Lambda puts your message in a Lambda-owned queue and returns right away, rather than exposing Lambda's internal queues directly to you.

These lightweight APIs also allow us to add fairness throttling. Fairness in a multitenant system is important so that no customer's workload impacts another customer. A common way that AWS implements fairness is by setting per-customer rate-based limits, with some flexibility for bursting. In many of our systems, for example in SQS itself, we increase per-customer limits as customers grow organically. The limits serve as guardrails for unexpected spikes, allowing us time to make

provisioning adjustments behind the scenes.

In some ways, fairness in asynchronous systems works just like throttling in synchronous systems. However, we think that it's even more important to think about in asynchronous systems because of the large backlogs that can build up so quickly.

To illustrate, consider what would happen if an asynchronous system didn't have enough noisy neighbor protections built in. If one customer of the system suddenly spiked their traffic that went unthrottled, and generated a system-wide backlog, it might take on the order of 30 minutes for an operator to be engaged, to figure out what's going on, and to mitigate the problem. During that 30 minutes, the producer side of the system might have scaled well and queued up all of the messages. But if the volume of queued messages was 10x the capacity that the consumer side was scaled to, this means it would take 300 minutes for the system to work through the backlog and recover. Even short load spikes can result in multi-hour recovery times, and therefore cause multi-hour outages.

In practice, systems in AWS have numerous compensating factors to minimize or prevent negative impacts from queue backlogs. For example, automatic scaling helps mitigate issues when load increases. But it's helpful to look at queueing affects alone, without considering compensating factors, because this helps design systems that are reliable in multiple layers. Here are a few design patterns that we have found can help to avoid large queue backlogs and long recovery times:

- **Protection at every layer is important in asynchronous systems.** Because synchronous systems don't tend to build up backlogs, we protect them with front-door throttling and admission control. In asynchronous systems, each component of our systems needs to protect itself from overload, and prevent one workload from consuming an unfair share of the resources. There will always be some workload that gets around the front-door admission control, so we need a belt, suspenders, and a pocket protector to keep services from becoming overloaded.
- **Using more than one queue helps shape traffic.** In some ways, a single queue and multitenancy are at odds with each other. By the time work is queued up in a shared queue, it's hard to isolate one workload from another.
- **Real time systems often are implemented with FIFO-ish queues, but prefer LIFO-ish behavior.** We hear from our customers that when faced with a backlog, they prefer to see their fresh data processed immediately. Any data accumulated during an outage or surge can then be processed as capacity is available.

Amazon's strategies for creating resilient multitenant asynchronous systems

There are several patterns that -systems at Amazon use to make their multitenant asynchronous systems resilient to changes in workloads. These are many techniques, but there are also many systems used throughout Amazon, each with its own set of liveness and durability requirements. In the following section, I describe some of the patterns we use, and that AWS customers tell us they use in their systems.

Separating workloads into separate queues

Instead of sharing one queue across all customers, in some systems we give each customer its own

queue. Adding a queue for each customer or workload is not always cost-effective, because services will need to spend resources polling all queues. But in systems with a handful of customers or adjacent systems, this simple solution can be helpful. On the other hand, if a system has even tens or hundreds of customers, separate queues can start to get unwieldy. For example, AWS IoT does not use a separate queue for every IoT device in the universe. Polling costs would not scale well in that case.

Shuffle-sharding

AWS Lambda is an example of a system where polling a separate queue for every Lambda customer would be too costly. However, having a single queue could result in some of the issues described in this article. So rather than using one queue, AWS Lambda provisions a fixed number of queues, and hashes each customer to a small number of queues. Before enqueueing a message, it checks to see which of those targeted queues contains the fewest messages, and enqueues into that one. When one customer's workload increases, it will drive a backlog in its mapped queues, but other workloads will automatically be routed away from those queues. It doesn't take a large number of queues to build in some magical resource isolation. This is only one of the many protections built into Lambda, but it is a technique that is also used in other services at Amazon.

Sidelining excess traffic to a separate queue

In some ways, when a backlog has built up in a queue, it's too late to prioritize traffic. However, if processing the message is relatively expensive or time consuming, it may still be worthwhile to be able to move messages to a separate, spillover queue. In some systems in Amazon, the consumer service implements distributed throttling, and when they dequeue messages for a customer that has gone over a configured rate, they enqueue those excess messages into separate spillover queues, and delete the messages from the primary queue. The system still works on the messages in the spillover queue as soon as resources are available. In essence, this approximates a priority queue. Similar logic is sometimes implemented on the producer side. This way, if a system accepts a large volume of requests from a single workload, that workload doesn't crowd out other workloads in the hot path queue.

Sidelining old traffic to a separate queue

Similar to sidelining excess traffic, we can also sideline old traffic. When we dequeue a message, we can check how old it is. Rather than just logging the age, we can use the information to decide whether to move the message into a backlog queue that we work through only after we're caught up on the live queue. If there's a load spike where we ingest a lot of data, and we get behind, we can sideline that wave of traffic into a different queue as quickly as we can dequeue and re-enqueue the traffic. This frees up consumer resources to work on fresh messages more quickly than if we had simply worked the backlog in order. This is one way to approximate LIFO ordering.

Dropping old messages (message time-to-live)

Some systems can tolerate very old messages being dropped. For example, some systems process deltas to systems quickly, but also do full synchronization periodically. We often call these periodic

synchronization systems anti-entropy sweepers. In these cases, instead of sidelining old queued up traffic, we can cheaply drop it if it came in before the most recent sweep.

Limiting threads (and other resources) per workload

Much as in our synchronous services, we design our asynchronous systems to prevent one workload from using more than its fair share of threads. One aspect of AWS IoT that we haven't talked about yet is the rules engine. Customers can configure AWS IoT to route messages from their devices to a customer-owned Amazon Elasticsearch cluster, Kinesis Stream, and so on. If the latency to those customer-owned resources becomes slow, but the incoming message rate remains constant, the amount of concurrency in the system increases. And because the amount of concurrency a system can handle is limited at any instant in time, the rules engine prevents any one workload from consuming more than its fair share of concurrency-related resources.

The force at work is described by Little's Law, which states that the concurrency in a system is equal to the arrival rate multiplied by the average latency of each request. For example, if a server was processing 100 messages / sec at 100 ms average, it would consume 10 threads on average. If the latency suddenly spiked to 10 seconds, it would suddenly use 1,000 threads (on average, so it could be more in practice), which could easily exhaust a thread pool.

The rules engine uses several techniques to prevent this from happening. It uses non-blocking I/O to avoid thread exhaustion, though there are still other limits to how much work a given server has (for example, memory, and file descriptors when the client is churning through connections and the dependency is timing out). A second concurrency guard that can be used is a semaphore that measures and limits the amount of concurrency that can be used for any single workload at any instant in time. The rules engine also uses rate-based fairness limiting. However, because it's perfectly normal for workloads to change over time, the rules engine also automatically scales limits over time to adapt to changes in workloads. And because the rules engine is queue-based, it serves as a buffer between IoT devices and the automatic scaling of resources and safeguard-limits behind the scenes.

Across services at Amazon, we use separate thread pools for each workload to avoid one workload from consuming all of the available threads. We also use an *AtomicInteger* for each workload to limit the allowed concurrency for each, and rate-based throttling approaches for isolating rate-based resources.

Sending backpressure upstream

If a workload is driving an unreasonable backlog that the consumer is unable to keep up with, many of our systems automatically start rejecting work more aggressively in the producer. It is easy to build up a day-long backlog for a workload. Even if that workload is isolated, it may be accidental, and expensive to churn through. An implementation of this approach could be as simple as occasionally measuring the queue depth of a workload (assuming a workload is on its own queue), and scaling an inbound throttle limit (inversely) proportionally to backlog size.

In cases where we share an SQS queue for multiple workloads, this approach gets tricky. While there's an SQS API that returns the number of messages in the queue, there's no API that can return the number of messages in the queue with a particular attribute. We could still measure queue depth and apply backpressure accordingly, but it would unfairly put backpressure on innocent workloads that happened to share the same queue. Other systems like Amazon MQ have finer-grained backlog

visibility.

Backpressure is not suitable for all systems at Amazon. For example, in systems that perform order processing for amazon.com, we tend to prefer to accept orders even if a backlog builds up, rather than preventing new orders from being accepted. But of course this is accompanied with plenty of prioritization behind the scenes so that the most urgent orders are handled first.

Using delay queues to put off work until later

When systems have a sense that the throughput for a particular workload needs to be reduced, we try to use a *back off* strategy on that workload. To implement this, we often use an SQS feature that delays the delivery of a message until later. When we process a message and decide to save it for later, we sometimes re-enqueue that message into a separate *surge queue*, but set the delay parameter so that the message stays hidden on the delay queue for several minutes. This gives the system a chance to work on fresher data instead.

Avoiding too many in-flight messages

Some queue services like SQS have limits as to how many in-flight messages can be delivered to the consumer of the queue. This is different from the number of messages that can be in the queue (for which there is no practical limit), but rather is the number of messages that the consumer fleet is working on at once. This number can be inflated if a system dequeues messages, but then fails to delete them. For example, we have seen bugs where code fails to catch an exception while processing a message and forgets to delete the message. In these cases, the message remains in-flight from the perspective of SQS for the *VisibilityTimeout* of the message. When we design our error handling and overload strategy, we keep these limits in mind, and tend to favor moving the excess messages to a different queue instead of letting them remain visible.

SQS FIFO queues have a similar but subtle limit. With SQS FIFO, systems consume your messages in-order for a given *message group*, but messages of different groups are processed in any order. So if we develop a small backlog in one message group, we continue to process messages in other groups. However, SQS FIFO only polls the most recent unprocessed 20k messages. So if there are more that 20k unprocessed messages in a subset of message groups, other message groups with fresh messages will be starved out.

Using dead-letter queues for messages that can't be processed

Messages that cannot be processed can contribute to system overload. If a system enqueues a message that can't be processed (perhaps because it triggers an input validation edge case), SQS can help by moving these messages automatically into a separate queue with the dead-letter queue (DLQ) feature. We alarm if there are any messages in this queue, because it means we have a bug that we need to fix. The benefit of the DLQ is that it lets us re-process the messages after the bug is fixed.

Ensuring additional buffer in polling threads per workload

If a workload is driving enough throughput to a point that polling threads are busy all the time even

during steady state, then the system might have reached a point where there is no buffer to absorb a surge in traffic. In this state, a small spike in the incoming traffic will lead to a sustained amount of unprocessed backlog, resulting in higher latency. We plan for additional buffer in polling threads in order to absorb such bursts. One measurement is to track the number of polling attempts that result in empty responses. If every polling attempt is retrieving one more message, then we either have just the right number of polling threads, or possibly not enough to keep up with incoming traffic.

Heartbeating long-running messages

When a system processes an SQS message, SQS gives that system a certain amount of time to finish processing the message before it assumes that the system crashed, and to deliver the message to another consumer to try again. If the code keeps running and forgets about this deadline, the same message can be delivered multiple times in parallel. While the first processor is still churning away on a message after its timeout, a second processor will pick it up and similarly churn away past the timeout, and then a third, and so on. This potential for cascading brownouts is why we implement our message processing logic to stop work when a message expires, or to continue to heartbeat that message to remind SQS that we're still working on it. This concept is similar to leases in leader election.

This is an insidious problem, because we see that a system's latency is likely to increase during an overload, perhaps from queries to a database taking longer, or from servers simply taking on more work than they can handle. When system latency crosses that VisibilityTimeout threshold, it causes an already overloaded service to essentially fork-bomb itself.

Plan for cross-host debugging

Understanding failures in a distributed system is already difficult. The related article about instrumentation describes several of our approaches for instrumenting asynchronous systems, from recording queue depths periodically, to propagating "trace ids" and integrating with X-Ray. Or, when our systems have a complicated asynchronous workflow beyond a trivial SQS queue, we often use a different asynchronous workflow service like Step Functions, which provide visibility into workflow and simplifies distributed debugging.

Conclusion

In an asynchronous system, it's easy to forget how important it is to think about latency. After all, asynchronous systems are supposed to occasionally take longer, because they are fronted by a queue for performing reliable retries. However, overload and failure scenarios can build up huge insurmountable backlogs from which a service cannot recover in a reasonable amount of time. These backlogs can come from one workload or customer enqueueing at an unexpectedly high rate, from workloads that become more expensive than predicted to process, or from latency or failures in a dependency.

When building an asynchronous system, we need to focus on and anticipate these backlog scenarios, and minimize them by using techniques like prioritization, sidelining, and backpressure.

Further reading

- Queueing theory
- Little's law
- Amdahl's law
- Little A Proof for the Queuing Formula: L = λW, Case Western, 1961
- McKenney, Stochastic Fairness Queuing, IBM, 1990
- Nichols and Jacobson, Controlling Queue Delay, PARC, 2011