Avoiding overload in distributed systems by putting the smaller service in control

Joe Magerramov



Avoiding overload in distributed systems by putting the smaller service in control
Copyright © 2020 Amazon Web Services, Inc. and/or its affiliates. All rights reserved

At Amazon, we build large-scale distributed systems that are composed of smaller independent services, each with a narrow set of responsibilities. These services interact with each other over well-defined APIs, allowing us to scale, evolve, and operate each one of them independently. Within AWS, a common pattern is to split the system into services that are responsible for executing customer requests (the *data plane*), and services that are responsible for managing and vending customer configuration (the *control plane*). In this article, I discuss a number of different ways the data plane and the control plane interact with each other to avoid system overload. In many of these architectures the larger data plane fleet calls the smaller control plane fleet, but I also want to share the success we've had at Amazon when we put the smaller fleet in control.

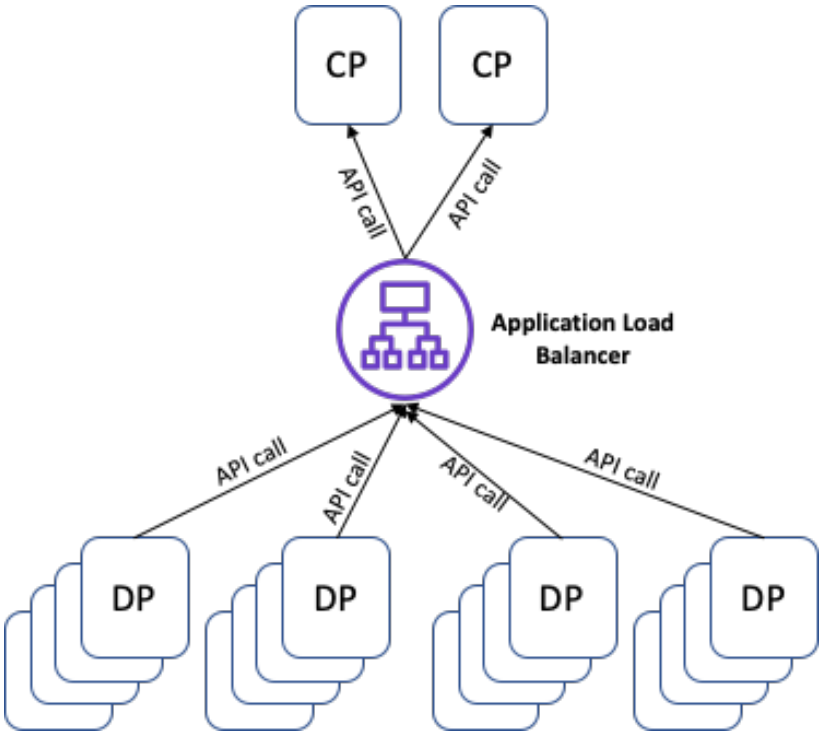
Amazon Elastic Compute Cloud (EC2) is an example of an architecture that includes a data plane and a control plane. The data plane consists of physical servers where customers' Amazon EC2 instances run. The control plane consists of a number of services that interact with the data plane, performing functions such as these:

- Telling each server about the EC2 instances that it needs to run.
- Keeping running EC2 instances up to date with Amazon Virtual Private Cloud (VPC) configuration.
- Receiving metering data, logs, and metrics emitted by the servers.
- Deploying new software to the servers.

Although the actual names and responsibilities can vary across similar architectures, these systems have two things in common:

- The data plane and the control plane need to stay in sync with each other. The data plane needs to receive configuration updates from the control plane, and the control plane needs to receive operational state from the data plane.
- The size of the data plane fleet exceeds the size of the control plane fleet, frequently by a factor of 100 or more.

When building such architectures at Amazon, one of the decisions we carefully consider is the direction of API calls. Should the larger data plane fleet call the control plane fleet? Or should the smaller control plane fleet call the data plane fleet? For many systems, having the data plane fleet call the control plane tends to be the simpler of the two approaches. In such systems, the control plane exposes APIs that can be used to retrieve configuration updates and to push operational state. The data plane servers can then call those APIs either periodically or on demand. Each data plane server initiates the API request, so the control plane does not need to keep track of the data plane fleet. For each request, a control plane server responds with updated configuration (usually by querying its durable data store), and then forgets about it. To provide fault tolerance and horizontal scalability, control plane servers are placed behind a load balancer like Application Load Balancer. This is one of the simpler types of distributed systems to build, and it's illustrated by the following diagram.

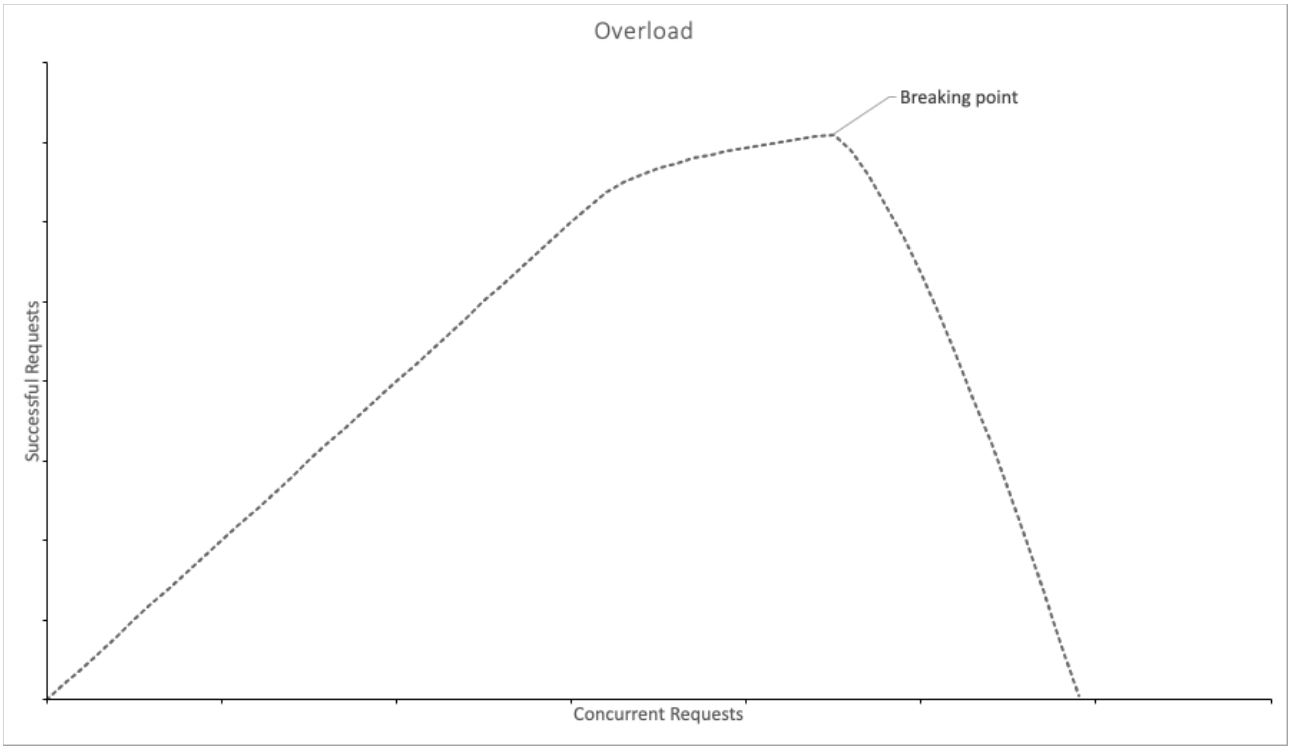


The simplicity of this architecture gives it inherent availability advantages. However, at Amazon we have also learned that when the scale of the data plane fleet exceeds the scale of the control plane fleet by a factor of 100 or more, this type of distributed system requires careful fine-tuning to avoid the risk of overload. In the steady state, the larger data plane fleet periodically calls control plane APIs, with requests from individual servers arriving at uncorrelated times. The load on the control plane is evenly distributed, and the system hums along.

However, unexpected shifts in the calling pattern of the data plane can increase the load on the control plane to the point of overloading it. Here are some situations that could cause such shifts:

- Code or configuration bugs in the data plane that increase the frequency of API calls.
- Recovery from an outage, where all the data plane servers try to call control plane APIs at the same time.
- A small number of failed API calls that cause retries, which causes increased load on the control plane, which causes even more failures and more retries, and so on.

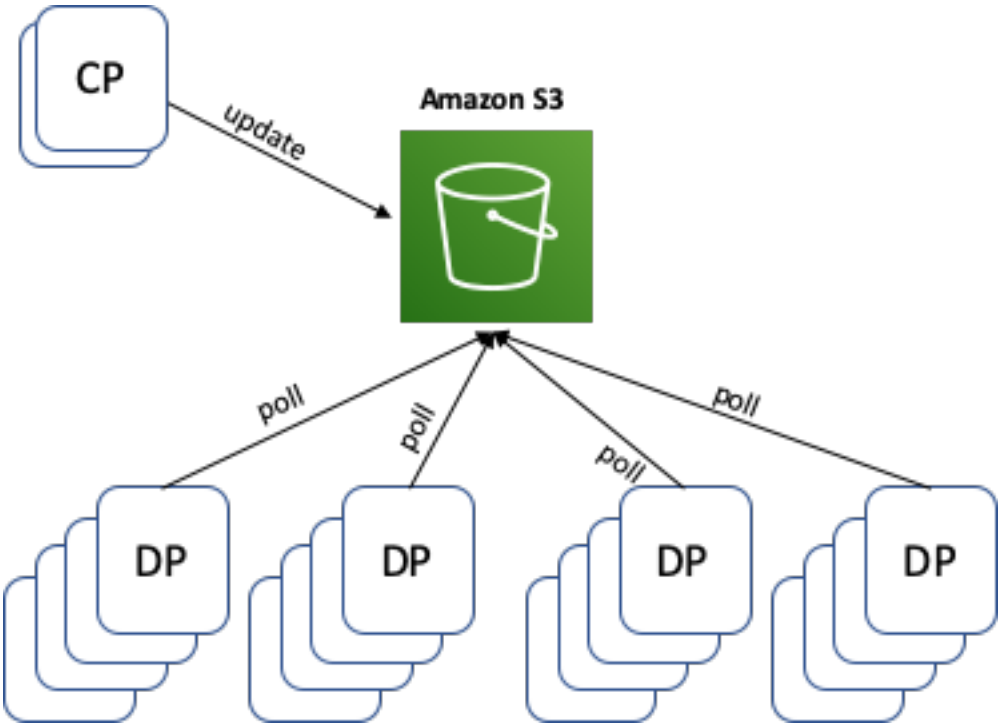
By examining a number of these situations, we've observed that what they have in common is a small change in the environment that causes the clients to act in a correlated manner and start making concurrent requests at the same time. Without careful fine-tuning, this shift in behavior can overwhelm the smaller control plane. This change in the volume of concurrent requests can cause the load on the system to increase beyond its breaking point, after which its *goodput* (the amount of useful work the system performs) quickly drops to zero, as shown in the following graph.



To avoid overloading the system, both the control plane and the data plane need to be carefully tuned to make sure the load on the control plane never exceeds the breaking point. On the control plane, mechanisms like load shedding can help the system continue to make progress, even in the face of unexpected load shifts. (For more information on load shedding, see the article Using load shedding to avoid overload.) On the data plane, carefully tuning the frequency of requests, and adding back-off and jitter to retries can help reduce the risk of correlated increases in the request rates. (For more information on retries, see the article Timeouts, retries, and back-off with jitter.)

Solving the problem of scale mismatch

The biggest challenge with this architecture is scale mismatch. The control plane fleet is badly outnumbered by the data plane fleet. At AWS, we looked to our storage services for help. When it comes to serving content at scale, the service we commonly use at Amazon is Amazon Simple Storage Service (S3). Instead of exposing APIs directly to the data plane, the control plane can periodically write updated configuration into an Amazon S3 bucket. Data plane servers then poll this bucket for updated configuration and cache it locally. Similarly, to stay up to date on the data plane's operational state, the control plane can poll an Amazon S3 bucket into which data plane servers periodically write that information. This architecture is illustrated in the following diagram.



This architecture has several advantages. It's simple to implement, and Amazon S3 is scaled to support even the largest client fleets. Additionally, as the size of the data plane fleet grows, the control plane fleet can still stay relatively small. And if the control plane has an outage, the data plane can continue running with the last known configuration, even as servers come in or out of service. This property, called *static stability*, is a desirable attribute in distributed systems.

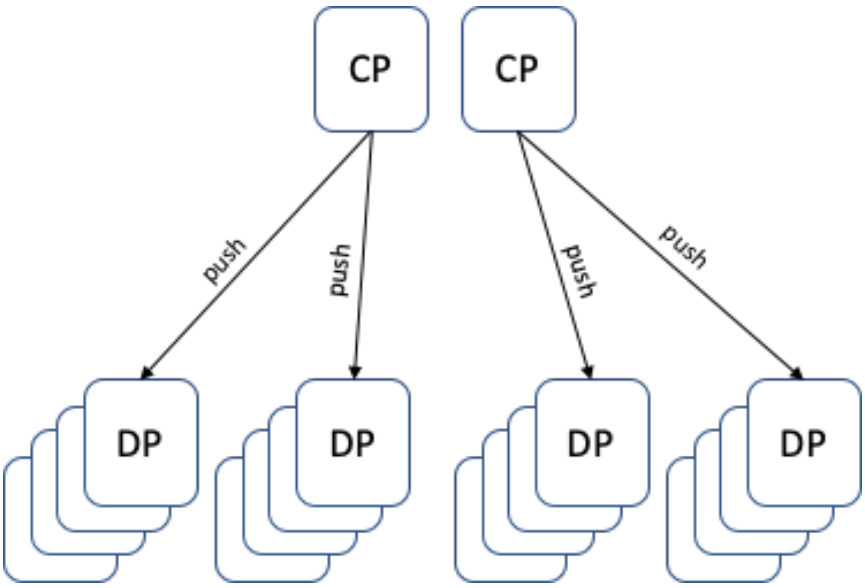
Because of these advantages, this architecture is a popular choice at Amazon. An example of a system using this architecture is AWS Hyperplane, the internal Amazon network function virtualization system behind AWS services and resources like Network Load Balancer, NAT Gateway, and AWS PrivateLink. The AWS Hyperplane data plane contains devices that process customer traffic, and that need to know about configuration of individual Network Load Balancers, NAT Gateways, and AWS PrivateLink connections. A periodic task within the AWS Hyperplane control plane scans its Amazon DynamoDB tables containing customer configuration and writes that configuration into several Amazon S3 files. The data plane then periodically downloads these files and uses their content to update internal routing configuration.

Despite its popularity, this architecture does not work in all cases. In systems with a large amount of dynamic configuration it might be impractical to continuously recompute the entire configuration and write it into Amazon S3. An example of such system is Amazon EC2, where a large number of data plane servers need to know about the constantly changing configuration of instances that they need to run, their VPCs, IAM roles, and Amazon Elastic Block Store (EBS) volumes.

In other systems, it's important for changes in the control plane configuration to be reflected in the data plane in single digit seconds or faster. An example of this situation is a container

service that notifies a server about a new container it needs to run. In such systems, polling periodically updated files in Amazon S3 can make it impossible to achieve acceptable propagation latencies.

At Amazon when faced with such scenarios, we look for other approaches where the small fleet can be in control of the pace at which requests flow through the system. One such approach is to reverse the flow of API calls and have the smaller control plane fleet push configuration changes to the larger data plane fleet. This architecture is illustrated in the following diagram.



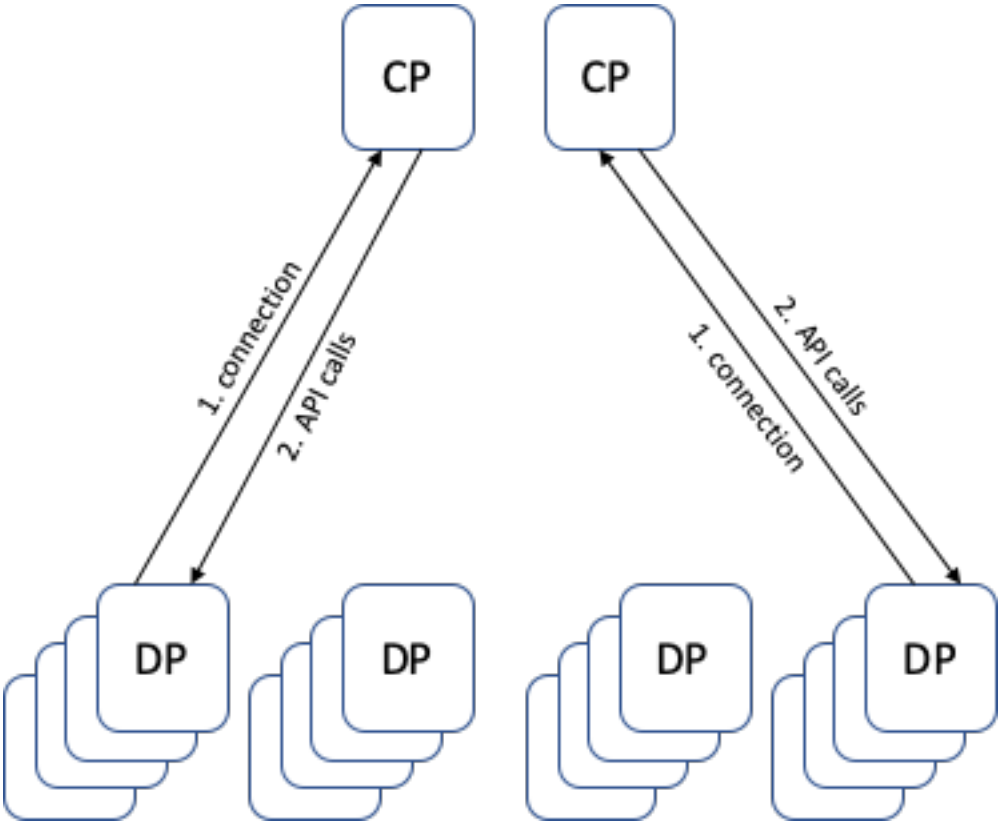
An architecture in which the smaller control plane fleet dictates the pace of work is more resilient when overloaded. If the control plane is overloaded or underscaled, it still continues making progress, even though it's at a slower pace. However, this approach also comes with several downsides that make it challenging to implement:

- The control plane needs to keep an up-to-date inventory of all available data plane servers, so that it can push configuration updates to each one.
- At any point in time, some data plane servers will be unreachable, and the control plane needs to deal with that.
- The control plane needs a mechanism to make sure that every data plane server receives configuration updates from at least one control plane server and a mechanism to ensure that no control plane server is responsible for too many data plane servers.

The last point can be particularly challenging as control plane servers come in and out of service. One approach we use at Amazon is to make each control plane server responsible for a portion of the data plane fleet, as determined by consistent hashing. Using an approach that bears a lot of similarities to leader election, described in the article Leader election in distributed systems, every control plane server sends a heartbeat into a shared data store (for example, DynamoDB) to indicate that it's alive. Independently, each control plane server scans that data store for a list of other control plane servers with a recent heartbeat. It then

deterministically executes a consistent hashing algorithm to identify a subset of data plane servers that it's responsible for.

To avoid the three complications I discussed earlier, this scheme can be architected to decouple the direction of discovery from the direction of the control flow. Instead of control plane servers initiating the connection, each server in the larger data plane fleet opens a long-lived connection to a single control plane server. This can be done by putting control plane servers behind a load balancer like Network Load Balancer. After the connection is established, the control plane server takes over and begins making API calls over that connection, similar to the HTTP server push mechanism. If the control plane server is unavailable or busy, it can reject the connection. The data plane server then tries a different control plane server. The data plane server will also try a different control plane server if the first connection is terminated for any reason. The architecture that decouples the direction of discovery from the direction of the control flow is illustrated in the following diagram.



Because control plane servers dictate the pace at which API calls are made, this approach has resilience to overloads similar to the approach in which the smaller control plane fleet pushes configuration changes to the larger data plane fleet. However, unlike that previous approach, it doesn't require the control plane to keep up-to-date inventory of all the data plane servers, or to implement consistent hashing or any other distributed algorithm. All decisions are made

using only local state. However, a downside of this approach is that it requires a more sophisticated communication protocol between control and data plane servers.

In cases of extreme scale mismatch, where the data plane fleet outnumbers the control plane fleet by a factor of 1000 or more, instead of initiating the connection, each data plane server could send a small UDP request and wait for the control plane to initiate the connection. Because the cost of handling such UDP requests is significantly lower than performing TCP and TLS handshakes, this pattern can help avoid situations where the influx of connection requests alone overloads the control plane. A variation of this approach is used by Amazon EC2 control planes to push configuration updates to the Nitro system.

Conclusion

We've learned that if we don't pay attention to the relative scale of a service and its clients, distributed systems can be at risk of overload. In such situations, a subtle change in the behavior of the clients can increase the load on the service to the point of overwhelming it. Such changes in behavior can be hard to predict, and protecting against them requires careful fine-tuning of both the service and its clients.

At Amazon, we pay attention to situations where clients outnumber a service by a factor of 100 or more, and look for approaches where the smaller fleet can control the pace at which changes flow through the system. The simplest approach uses a large-scale service like Amazon S3 as an intermediary between the small fleet and its clients. When that is not practical, we look to inversion of scale, an approach where the smaller fleet calls the larger one. Although these approaches can add implementation complexity, we find that they make it easier to scale and operate the smaller fleet, even as the larger fleet continues to grow in size.