Handling Overload

Written by Alejandro Forero Cuervo
Edited by Sarah Chavis

Avoiding overload is a goal of load balancing policies. But no matter how efficient your load balancing policy, *eventually* some part of your system will become overloaded. Gracefully handling overload conditions is fundamental to running a reliable serving system.

One option for handling overload is to serve degraded responses: responses that are not as accurate as or that contain less data than normal responses, but that are easier to compute. For example:

- Instead of searching an entire corpus to provide the best available results to a search query, search only a small percentage of the candidate set.
- Rely on a local copy of results that may not be fully up to date but that will be cheaper to use than going against the canonical storage.

However, under extreme overload, the service might not even be able to compute and serve degraded responses. At this point it may have no immediate option but to serve errors. One way to mitigate this scenario is to balance traffic across datacenters such that no datacenter receives more traffic than it has the capacity to process. For example, if a datacenter runs 100 backend tasks and each task can process up to 500 requests per second, the load balancing algorithm will not allow more than 50,000 queries per second to be sent to that datacenter. However, even this constraint can prove insufficient to avoid overload when you're operating at scale. At the end of the day, it's best to build clients and backends to handle resource

restrictions gracefully: redirect when possible, serve degraded results when necessary, and handle resource errors transparently when all else fails.

The Pitfalls of "Queries per Second"

Different queries can have vastly different resource requirements. A query's cost can vary based on arbitrary factors such as the code in the client that issues them (for services that have many different clients) or even the time of the day (e.g., home users versus work users; or interactive end-user traffic versus batch traffic).

We learned this lesson the hard way: modeling capacity as "queries per second" or using static features of the requests that are believed to be a proxy for the resources they consume (e.g., "how many keys are the requests reading") often makes for a poor metric. Even if these metrics perform adequately at one point in time, the ratios can change. Sometimes the change is gradual, but sometimes the change is drastic (e.g., a new version of the software suddenly made some features of some requests require significantly fewer resources). A moving target makes a poor metric for designing and implementing load balancing.

A better solution is to measure capacity directly in available resources. For example, you may have a total of 500 CPU cores and 1 TB of memory reserved for a given service in a given datacenter. Naturally, it works much better to use those numbers directly to model a datacenter's capacity. We often speak about the *cost* of a request to refer to a normalized measure of how much CPU time it has consumed (over different CPU architectures, with consideration of performance differences).

In a majority of cases (although certainly not in all), we've found that simply using CPU consumption as the signal for provisioning works well, for the following reasons:

- In platforms with garbage collection, memory pressure naturally translates into increased CPU consumption.
- In other platforms, it's possible to provision the remaining resources in such a way that they're very unlikely to run out before CPU runs out.

In cases where over-provisioning the non-CPU resources is prohibitively expensive, we take each system resource into account separately when considering resource consumption.

Per-Customer Limits

One component of dealing with overload is deciding what to do in the case of *global* overload. In a perfect world, where teams coordinate their launches carefully with the owners of their backend dependencies, global overload never happens and backend services always have enough capacity to serve their customers. Unfortunately, we don't live in a perfect world. Here in reality, global overload occurs quite frequently (especially for internal services that tend to have many clients run by many teams).

When global overload *does* occur, it's vital that the service only delivers error responses to misbehaving customers, while other customers remain unaffected. To achieve this outcome, service owners provision their capacity based on the negotiated usage with their customers and define per-customer quotas according to these agreements.

For example, if a backend service has 10,000 CPUs allocated worldwide (over various datacenters), their per-customer limits might look something like the following:

- Gmail is allowed to consume up to 4,000 CPU seconds per second.
- Calendar is allowed to consume up to 4,000 CPU seconds per second.
- Android is allowed to consume up to 3,000 CPU seconds per second.
- Google+ is allowed to consume up to 2,000 CPU seconds per second.
- Every other user is allowed to consume up to 500 CPU seconds per second.

Note that these numbers may add up to more than the 10,000 CPUs allocated to the backend service. The service owner is relying on the fact that it's unlikely for *all* of their customers to hit their resource limits simultaneously.

We aggregate global usage information in real time from all backend tasks, and use that data to push effective limits to individual backend tasks. A closer look at the system that implements this logic is outside of the scope of this discussion, but we've written significant code to implement this in our backend tasks. An interesting part of the puzzle is computing in real time the amount of resources—specifically CPU—consumed by each individual request. This computation is particularly tricky for servers that don't implement a thread-per-request model,

where a pool of threads just executes different parts of all requests as they come in, using nonblocking APIs.

Client-Side Throttling

When a customer is out of quota, a backend task should reject requests quickly with the expectation that returning a "customer is out of quota" error consumes significantly fewer resources than actually processing the request and serving back a correct response. However, this logic doesn't hold true for all services. For example, it's almost equally expensive to reject a request that requires a simple RAM lookup (where the overhead of the request/response protocol handling is significantly larger than the overhead of producing the response) as it is to accept and run that request. And even in the case where rejecting requests saves significant resources, those requests *still* consume some resources. If the amount of rejected requests is significant, these numbers add up quickly. In such cases, the backend can become overloaded even though the vast majority of its CPU is spent just rejecting requests!

Client-side throttling addresses this problem.[106] When a client detects that a significant portion of its recent requests have been rejected due to "out of quota" errors, it starts self-regulating and caps the amount of outgoing traffic it generates. Requests above the cap fail locally without even reaching the network.

We implemented client-side throttling through a technique we call *adaptive throttling*. Specifically, each client task keeps the following information for the last two minutes of its history:

`requests`

: The number of requests attempted by the application layer(at the client, on top of the adaptive throttling system)

`accepts`

: The number of requests accepted by the backend

Under normal conditions, the two values are equal. As the backend starts rejecting traffic, the number of `accepts` becomes smaller than the number of `requests`. Clients can continue to

issue requests to the backend until `requests` is K times as large as `accepts`. Once that cutoff is reached, the client begins to self-regulate and new requests are rejected locally (i.e., at the client) with the probability calculated in Client request rejection probability.

Client request rejection probability

$$\max(0, \frac{\text{requests} - K \times \text{accepts}}{\text{requests} + 1})$$

As the client itself starts rejecting requests, `requests` will continue to exceed `accepts`. While it may seem counterintuitive, given that locally rejected requests aren't actually propagated to the backend, this is the preferred behavior. As the rate at which the application attempts requests to the client grows (relative to the rate at which the backend accepts them), we want to increase the probability of dropping new requests.

For services where the cost of processing a request is very close to the cost of rejecting that request, allowing roughly half of the backend resources to be consumed by rejected requests can be unacceptable. In this case, the solution is simple: modify the accepts multiplier K (e.g., 2) in the client request rejection probability (Client request rejection probability). In this way:

- Reducing the multiplier will make adaptive throttling behave more aggressively
- Increasing the multiplier will make adaptive throttling behave less aggressively

For example, instead of having the client self-regulate when `requests = 2 * accepts`, have it self-regulate when `requests = 1.1 * accepts`. Reducing the modifier to 1.1 means only one request will be rejected by the backend for every 10 requests accepted.

We generally prefer the 2x multiplier. By allowing more requests to reach the backend than are expected to actually be allowed, we waste more resources at the backend, but we also speed up the propagation of state from the backend to the clients. For example, if the backend decides to stop rejecting traffic from the client tasks, the delay until all client tasks have detected this change in state is shorter.

We've found adaptive throttling to work well in practice, leading to stable rates of requests overall. Even in large overload situations, backends end up rejecting one request for each request they actually process. One large advantage of this approach is that the decision is

made by the client task based entirely on local information and using a relatively simple implementation: there are no additional dependencies or latency penalties.

One additional consideration is that client-side throttling may not work well with clients that only very sporadically send requests to their backends. In this case, the view that each client has of the state of the backend is reduced drastically, and approaches to increment this visibility tend to be expensive.

Criticality

Criticality is another notion that we've found very useful in the context of global quotas and throttling. A request made to a backend is associated with one of four possible criticality values, depending on how critical we consider that request:

`CRITICAL_PLUS`

> Reserved for the most critical requests, those that will result in serious user-visible impact if they fail.

`CRITICAL`

> The default value for requests sent from production jobs. These requests will result in user-visible impact, but the impact may be less severe than those of `CRITICAL_PLUS`. Services are expected to provision enough capacity for all expected `CRITICAL` and `CRITICAL_PLUS` traffic.

`SHEDDABLE_PLUS`

> Traffic for which partial unavailability is expected. This is the default for batch jobs, which can retry requests minutes or even hours later.

`SHEDDABLE`

> Traffic for which frequent partial unavailability and occasional full unavailability is expected.

We found that four values were sufficiently robust to model almost every service. We've had various discussions on proposals to add more values, because doing so would allow us to classify requests more finely. However, defining additional values would require more resources to operate various criticality-aware systems.

We've made criticality a first-class notion of our RPC system and we've worked hard to integrate it into many of our control mechanisms so it can be taken into account when reacting to overload situations. For example:

- When a customer runs out of global quota, a backend task will only reject requests of a given criticality if it's already rejecting all requests of all lower criticalities (in fact, the per-customer limits that our system supports, described earlier, can be set per criticality).
- When a task is itself overloaded, it will reject requests of lower criticalities sooner.
- The adaptive throttling system also keeps separate stats for each criticality.

The criticality of a request is orthogonal to its latency requirements and thus to the underlying network quality of service (QoS) used. For example, when a system displays search results or suggestions while the user is typing a search query, the underlying requests are highly sheddable (if the system is overloaded, it's acceptable to not display these results), but tend to have stringent latency requirements.

We've also significantly extended our RPC system to propagate criticality automatically. If a backend receives request A and, as part of executing that request, issues outgoing request B and request C to other backends, request B and request C will use the same criticality as request A by default.

In the past, many systems at Google had evolved their own ad hoc notions of criticality that were often incompatible across services. By standardizing and propagating criticality as a part of our RPC system, we are now able to consistently set the criticality at specific points. This means we can be confident that overloaded dependencies will abide by the desired high-level criticality as they reject traffic, regardless of how deep down the RPC stack they are. Our practice is thus to set the criticality as close as possible to the browsers or mobile clients—typically in the HTTP frontends that produce the HTML to be returned—and only override the criticality in specific cases where it makes sense at specific points in the stack.

Utilization Signals

Our implementation of task-level overload protection is based on the notion of *utilization*. In many cases, the utilization is just a measurement of the CPU rate (i.e., the current CPU rate divided by the total CPUs reserved for the task), but in some cases we also factor in measurements such as the portion of the memory reserved that is currently being used. As utilization approaches configured thresholds, we start rejecting requests based on their criticality (higher thresholds for higher criticalities).

The utilization signals we use are based on the state local to the task (since the goal of the signals is to protect the task) and we have implementations for various signals. The most generally useful signal is based on the "load" in the process, which is determined using a system we call *executor load average*.

To find the executor load average, we count the number of active threads in the process. In this case, "active" refers to threads that are currently running or ready to run and waiting for a free processor. We smooth this value with exponential decay and begin rejecting requests as the number of active threads grows beyond the number of processors available to the task. That means that an incoming request that has a very large fan-out (i.e., one that schedules a burst of a very large number of short-lived operations) will cause the load to spike very briefly, but the smoothing will mostly swallow that spike. However, if the operations are not short-lived (i.e., the load increases and remains high for a significant amount of time), the task will start rejecting requests.

While the executor load average has proven to be a very useful signal, our system can plug in any utilization signal that a particular backend may need. For example, we might use memory pressure—which indicates whether the memory usage in a backend task has grown beyond normal operational parameters—as another possible utilization signal. The system can also be configured to combine multiple signals and reject requests that would surpass the combined (or individual) target utilization thresholds.

Handling Overload Errors

In addition to handling load gracefully, we've put a significant amount of thought into how clients should react when they receive a load-related error response. In the case of overload errors, we distinguish between two possible situations.

A large subset of backend tasks in the datacenter are overloaded.

> If the cross-datacenter load balancing system is working perfectly (i.e., it can propagate state and react instantaneously to shifts in traffic), this condition will not occur.

A small subset of backend tasks in the datacenter are overloaded.

> This situation is typically caused by imperfections in the load balancing inside the datacenter. For example, a task may have very recently received a very expensive request. In this case, it is very likely that the datacenter has remaining capacity in other tasks to handle the request.

If a large subset of backend tasks in the datacenter are overloaded, requests should not be retried and errors should bubble up all the way to the caller (e.g., returning an error to the end user). It's much more typical that only a small portion of tasks become overloaded, in which case the preferred response is to retry the request immediately. In general, our cross-datacenter load balancing system tries to direct traffic from clients to their nearest available backend datacenters. In a few cases, the nearest datacenter is far away (e.g., a client may have its nearest available backend in a different continent), but we usually manage to situate clients close to their backends. That way, the additional latency of retrying a request—just a few network round trips—tends to be negligible.

From the point of view of our load balancing policies, retries of requests are indistinguishable from new requests. That is, we don't use any explicit logic to ensure that a retry actually goes to a different backend task; we just rely on the likely probability that the retry will land on a different backend task simply by virtue of the number of participating backends in the subset. Ensuring that all retries actually go to a different task would incur more complexity in our APIs than is worthwhile.

Even if a backend is only slightly overloaded, a client request is often better served if the backend rejects retry and new requests equally and quickly. These requests can then be retried immediately on a different backend task that may have spare resources. The consequence of treating retries and new requests identically at the backend is that retrying requests in different

tasks becomes a form of organic load balancing: it redirects load to tasks that may be better suited for those requests.

Deciding to Retry

When a client receives a "task overloaded" error response, it needs to decide whether to retry the request. We have a few mechanisms in place to avoid retries when a significant portion of the tasks in a cluster are overloaded.

First, we implement a *per-request retry budget* of up to three attempts. If a request has already failed three times, we let the failure bubble up to the caller. The rationale is that if a request has already landed on overloaded tasks three times, it's relatively unlikely that attempting it again will help because the whole datacenter is likely overloaded.

Secondly, we implement a *per-client retry budget*. Each client keeps track of the ratio of requests that correspond to retries. A request will only be retried as long as this ratio is below 10%. The rationale is that if only a small subset of tasks are overloaded, there will be relatively little need to retry.

As a concrete example (of the worst-case scenario), let's assume a datacenter is accepting a small amount of requests and rejecting a large portion of requests. Let X be the total rate of requests attempted against the datacenter according to the client-side logic. Due to the number of retries that will occur, the number of requests will grow significantly, to somewhere just below 3X. Although we've effectively capped the growth caused by retries, a threefold increase in requests is significant, especially if the cost of rejecting versus processing a request is considerable. However, layering on the per-client retry budget (a 10% retry ratio) reduces the growth to just 1.1x in the general case—a significant improvement.

A third approach has clients include a counter of how many times the request has already been tried in the request metadata. For instance, the counter starts at 0 in the first attempt and is incremented on every retry until it reaches 2, at which point the per-request budget causes it to stop being retried. Backends keep histograms of these values in recent history. When a backend needs to reject a request, it consults these histograms to determine the likelihood that other backend tasks are also overloaded. If these histograms reveal a significant amount of retries (indicating that other backend tasks are likely also overloaded), they return an "overloaded; don't retry" error response instead of the standard "task overloaded" error that triggers retries.

Figure 21-1 shows the number of attempts in each request received by a given backend task in various example situations, over a sliding window (corresponding to 1,000 initial requests, not counting retries). For simplicity, the per-client retry budget is ignored (i.e., these numbers assume that the only limit to retries is the retry budget of three attempts per request), and subsetting could alter these numbers somewhat.

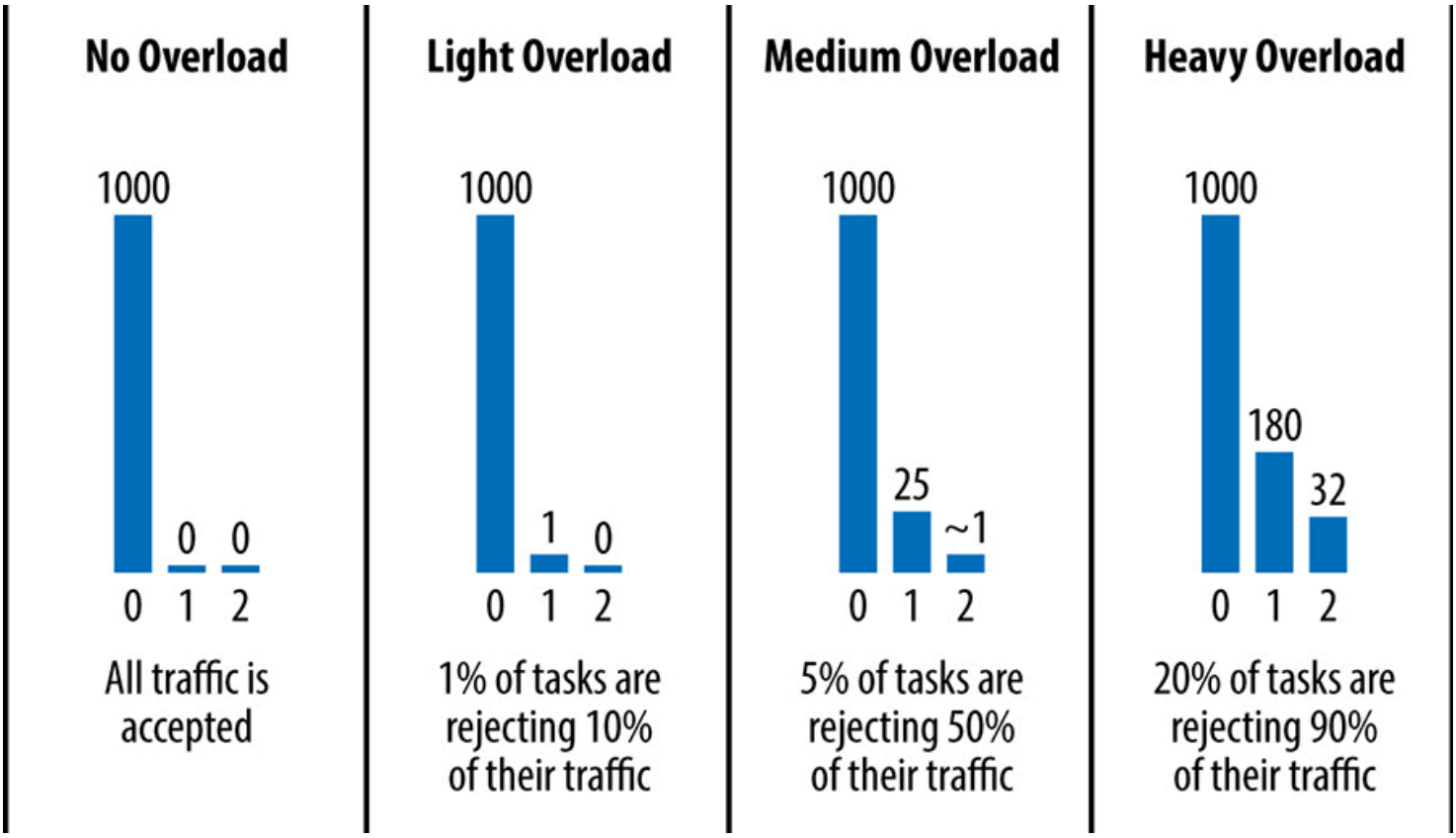

Figure 21-1. Histograms of attempts in various conditions

Our larger services tend to be deep stacks of systems, which may in turn have dependencies on each other. In this architecture, requests should only be retried at the layer immediately above the layer that is rejecting them. When we decide that a given request can't be served and shouldn't be retried, we use an "overloaded; don't retry" error and thus avoid a combinatorial retry explosion.

Consider the example from Figure 21-2 (in practice, our stacks are often significantly more complex). Imagine that the DB Frontend is currently overloaded and rejects a request. In that case:

- Backend B will then retry the request according to the preceding guidelines.
- However, once Backend B determines that the request to the DB Frontend can't be served (for example, because the request has already been attempted and rejected three times), Backend B has to return to Backend A either an "overloaded; don't retry" error or a degraded response (assuming that it can produce some moderately useful response even when its request to the DB Frontend failed).
- Backend A has exactly the same options for the request it received from the Frontend, and proceeds accordingly.

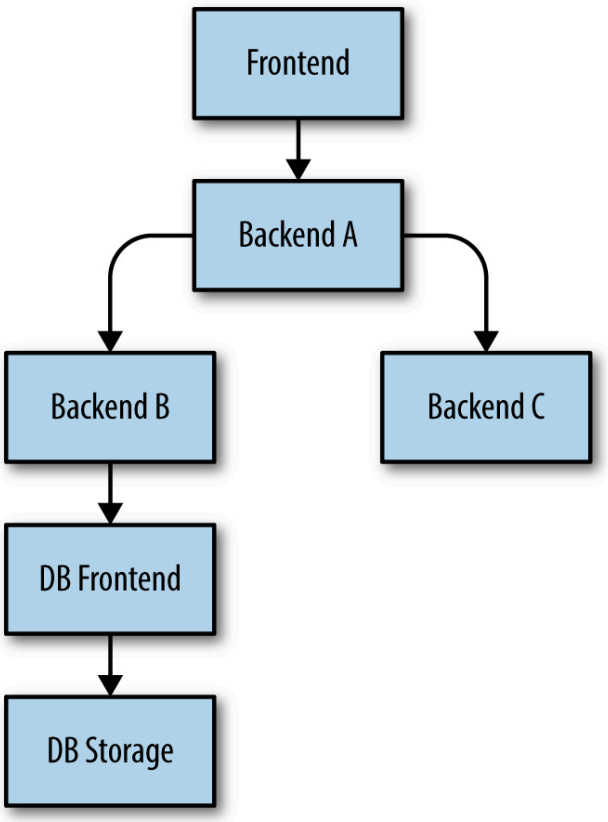

Figure 21-2. A stack of dependencies

The key point is that a failed request from the DB Frontend should only be retried by Backend B, the layer immediately above it. If multiple layers retried, we'd have a combinatorial explosion.

Load from Connections

The load associated with connections is one last factor worth mentioning. We sometimes only take into account load at the backends that is caused directly by the requests they receive (which is one of the problems with approaches that model load based upon queries per second). However, doing so overlooks the CPU and memory costs of maintaining a large pool of connections or the cost of a fast rate of churn of connections. Such issues are negligible in small systems, but quickly become problematic when running very large-scale RPC systems.

As mentioned previously, our RPC protocol requires inactive clients to perform periodic health checks. After a connection has been idle for a configurable amount of time, the client drops its TCP connection and switches to UDP for health checking. Unfortunately, this behavior is problematic when you have a very large number of client tasks that issue a very low rate of requests: health checking on the connections can require more resources than actually serving the requests. Approaches such as carefully tuning the connection parameters (e.g., significantly decreasing the frequency of health checks) or even creating and destroying the connections dynamically can significantly improve this situation.

Handling bursts of new connection requests is a second (but related) problem. We've seen bursts of this type happen in the case of very large batch jobs that create a very large number of worker client tasks all at once. The need to negotiate and maintain an excessive number of new connections simultaneously can easily overload a group of backends. In our experience, there are a couple strategies that can help mitigate this load:

- Expose the load to the cross-datacenter load balancing algorithm (e.g., base load balancing on the utilization of the cluster, rather than just on the number of requests). In this case, load from requests is effectively rebalanced away to other datacenters that have spare capacity.

- Mandate that batch client jobs use a separate set of *batch proxy* backend tasks that do nothing but forward requests to the underlying backends and hand their responses back to the clients in a controlled way. Therefore, instead of "batch client → backend," you have "batch client → batch proxy → backend." In this case, when the very large job starts, only the batch proxy job suffers, shielding the actual backends (and higher-priority clients). Effectively, the batch proxy acts like a fuse. Another advantage of using the proxy is that it typically reduces the number of

connections against the backend, which can improve the load balancing against the backend (e.g., the proxy tasks can use bigger subsets and probably have a better view of the state of the backend tasks).

Conclusions

This chapter and Load Balancing in the Datacenter have discussed how various techniques (deterministic subsetting, Weighted Round Robin, client-side throttling, customer quotas, etc.) can help to spread load over tasks in a datacenter relatively evenly. However, these mechanisms depend on the propagation of state over a distributed system. While they perform reasonably well in the general case, real-world application has resulted in a small number of situations where they work imperfectly.

As a result, we consider it critical to ensure that individual tasks are protected against overload. To state this simply: a backend task provisioned to serve a certain traffic rate should continue to serve traffic at that rate without any significant impact on latency, regardless of how much excess traffic is thrown at the task. As a corollary, the backend task should not fall over and crash under the load. These statements should hold true up to a certain rate of traffic—somewhere above 2x or even 10x what the task is provisioned to process. We accept that there might be a certain point at which a system begins to break down, and raising the threshold at which this breakdown occurs becomes relatively difficult to achieve.

The key is to take these degradation conditions seriously. When these degradation conditions are ignored, many systems will exhibit terrible behavior. And as work piles up and tasks eventually run out of memory and crash (or end up burning almost all their CPU in memory thrashing), latency suffers as traffic is dropped and tasks compete for resources. Left unchecked, the failure in a subset of a system (such as an individual backend task) might trigger the failure of other system components, potentially causing the entire system (or a considerable subset) to fail. The impact from this kind of cascading failure can be so severe that it's critical for any system operating at scale to protect against it; see Addressing Cascading Failures.

It's a common mistake to assume that an overloaded backend should turn down and stop accepting all traffic. However, this assumption actually goes counter to the goal of robust load balancing. We actually want the backend to continue accepting as much traffic as possible, but

to only accept that load as capacity frees up. A well-behaved backend, supported by robust load balancing policies, should accept only the requests that it can process and reject the rest gracefully.

While we have a vast array of tools to implement good load balancing and overload protections, there is no magic bullet: load balancing often requires deep understanding of a system and the semantics of its requests. The techniques described in this chapter have evolved along with the needs of many systems at Google, and will likely continue to evolve as the nature of our systems continues to change.

---

[106] For example, see Doorman, which provides a cooperative distributed client-side throttling system.

← PREVIOUS
Chapter 20 - Load Balancing in the Datacenter

NEXT
Chapter 22 - Addressing Cascading Failures