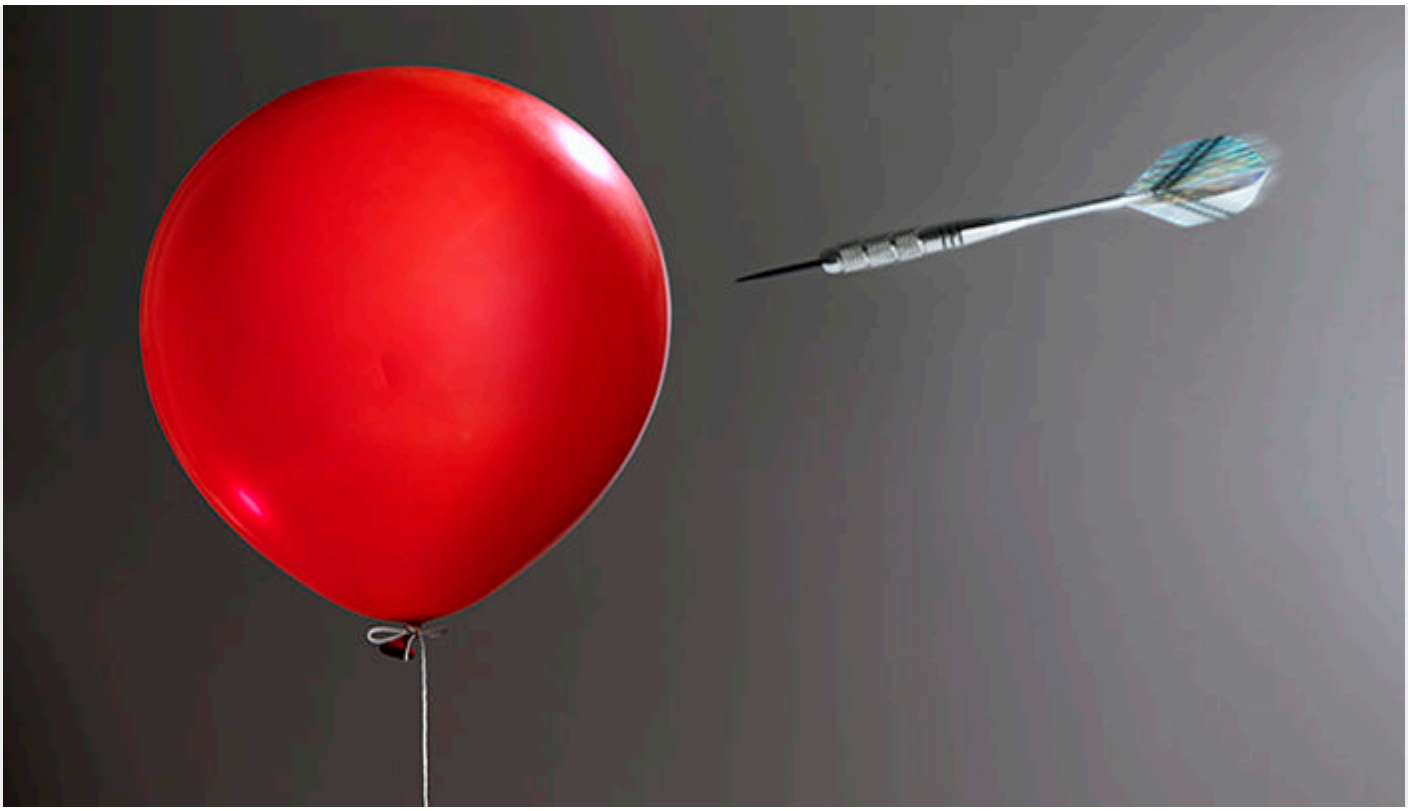
WHAT ARE CASCADING FAILURES?



In complex computer systems, there remains a perpetual risk of small errors. Sometimes these errors mind their own business and everything quickly returns to normal. Other times, an error causes another related problem to follow it, which in turn feeds back into the original error or perhaps additional ones in another part of the system. The resulting domino effect of increasing failures due to positive feedback mechanisms is what is called a cascading failure.

Feedback Mechanisms

Stand-alone errors pose much less of a threat to systems than those that contribute to feedback loops. In such a feedback loop, distributed software systems might first experience a relatively small reduction in capacity, a spike in errors, or an increase in latency. As other parts of the system try to respond to and correct the original problem, they end up making it worse.

For example, one server in a set of servers working together could experience an excess load which causes it to fail. Regularly, it would offload a small excess onto a neighboring server that acts as a backup while still maintaining its own load. But this second server cannot handle two entire loads after the complete failure of the first server, so it fails as well.

The excess loads continue passing down the line of neighboring servers, snowballing into larger and larger excess loads until the entire system crumbles. This type of cascading failure is a mix of overload and proximity-based failover feedback triggers, as mentioned below.

Feedback Triggers of Cascading Failures

Several triggers can account for the typical cascading failure, both individually and in conjunction with one another, including:

- Load Spikes and Overload
- Proximity-Based Failover
- Process Death/Query of Death
- Excessive Client Retry Behavior
- Resource Exhaustion
- Resource Limits
- Resource Interdependence
- CPU Issues
- Memory Issues
- Thread Starvation
- New Rollouts and Changes
- Organic Growth
- Long Startup Times
- Shifted Traffic Mix

If your system has never experienced a cascading failure, do not let yourself be lulled into a false sense of security. At any time, your system may try to operate outside its limits, leading to an unexpected problem.

Some of these triggers are avoidable through proper structuring and management of a system. Others are essentially unavoidable, but can be prepared for and mitigated by studying how they work before they create significant problems.

For instance, spikes in load are an expected part of system operation, and they are the most common feedback trigger for cascading failures. You can't reasonably stop them, but you can manage how your system reacts to them. While the task scheduling system and load balancer manage typical loads in a certain way that usually works, they should also be prepared for abnormal loads. When the load increases beyond the average, they may need to implement new tactics to prevent a cascading failure.

Recovering From a Cascading Failure

Regardless of the type of trigger, recovering from a cascading failure is almost always extremely challenging. Frequently, the system requires a full shutdown because it can no longer restabilize itself without manual intervention.

Why Is It So Difficult?

If your system contains the right conditions for a cascading failure, you likely have no idea and the event will strike almost instantly without any warning. And once it's started, there is little chance of slowing or stopping it. The state of the system will usually get worse and worse without self-healing in any way.

Many teams may have an instinct to try to recover the system by adding more capacity. But this, too, will likely just contribute to the positive feedback of the failure. As you add a new healthy instance, it will immediately get hit with the excess load from other failing instances. Such a huge load saturates the new instance, making it fail before it has a chance to help, discarding the possibility that you can reach a point where you have sufficient serving capacity to address the load.

Setting up the system beforehand to behave in a manner that doesn't send all requests to only healthy instances might prevent this type of problem, slowing down or halting a cascading failure. But without such foresight, often the only way to recover from a cascading failure is to take down the entire service and then reintroduce the load.

Tactics for Addressing a Failure

If your system is already experiencing a cascading failure, there are a few steps you can take to try to immediately address it before resorting to a total system shutdown. First, check that any overloaded capacity is not due to idle resources. Adding idle resources into the active tasks can quickly recover a system if the load is not too large.

If all resources are actively participating, then changing health check behavior can stop the spiral. As the system checks the health of tasks, it will find that many tasks are unhealthy and others are restarting. Thus, it will typically send disproportionate loads to the few remaining healthy instances causing them to also become unhealthy.

Temporarily disabling or changing the behavior of these types of self-health checks might help the system restabilize. One option is to have the system make a distinction between completely unresponsive and temporarily unresponsive tasks, so it can include the latter types in the load distribution.

A more drastic response to a cascading failure is to drop traffic and enter degraded modes. This tactic requires some aggression to work, with a significant reduction in traffic to something like just 1% of regular rates. As the servers recover, more traffic can gradually pass through. As the load returns to normal levels, the caches can warm up and establish connections, but users will have a very negative experience.

Reducing traffic in this way needs to be implemented simultaneously with a repair of the underlying problem. Otherwise, there may be a second cascading failure as soon as the recovery is complete and traffic returns to normal. If the computer ran out of memory or the system had insufficient capacity, then be sure to add more memory or tasks before trying to reintroduce traffic.

Additionally, systems with the capability to differentiate between various types of traffic can specifically eliminate less important or bad traffic. This must already be part of the system capabilities, but it can allow prioritization of the traffic getting through to reduce negative consequences.

Similarly, discriminating against less important load batches can reduce the load on tasks with minimized negative results. Pausing sources such as statistics gathering, data copies, and index updates can contribute to a recovery.

If all of these strategies fail to stop the cascading failure, a full system shutdown is likely the only remaining option.

Preventing Cascading Failures Before They Begin

Prevent Overloaded Servers

Because overload is the most common trigger, focusing on overload prevention can go a long way in avoiding cascading failures. To avoid overload, consider limiting the number of incoming requests to the system. Accepting an unlimited number of requests can create huge queues of requests and concurrent threads attempting to execute during a cascading failure.

This buildup causes the system to become unresponsive, frequently requiring an intervention or restart. To avoid it, moderate client retry behavior to avoid floods of retried requests, set limits on the

load on each instance, and consider load shedding at a load-balancer. The latter sets some limits on your service, but so would the alternative of a cascading failure.

Conduct Testing

When it comes to other potential feedback triggers, sometimes you don't even know that your system has an inadequate setup or response. The best way to find out how it handles problematic situations is to test it against them.

Fuzz testing can help detect programs that crash on bad input, also known as Queries of Death. Rather than crashing, a program should be able to recognize if its internal state poses safety concerns and react in a manner that does not cause a crash. This type of testing is particularly useful for services that frequently experience inputs from outside the organization.

In addition to load, test client behavior and noncritical backends. Clients should be able to queue their work while the system is down, and they should exponentially back off on errors. Meanwhile, the unavailability of backends during a failure should not interfere with the more critical parts of the system.

While conducting any testing, be sure to consider both gradual and impulse pattern changes. The system may respond well to gradual increases in load, but crash during a sudden spike. And once a failure occurs in the system, continue testing to see if and how it is able to recover.

After you identify the issues unique to your system, you can implement appropriate changes. Altering system setup and improving response can mitigate and even entirely prevent cascading failures.