[Netflix](https://github.com/Netflix)/[**Hystrix**](https://github.com/Netflix/Hystrix)

Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable.

## What Is Hystrix?

In a distributed environment, inevitably some of the many service dependencies will fail. Hystrix is a library that helps you control the interactions between these distributed services by adding latency tolerance and fault tolerance logic. Hystrix does this by isolating points of access between the services, stopping cascading failures across them, and providing fallback options, all of which improve your system’s overall resiliency.

#### History of Hystrix

Hystrix evolved out of resilience engineering work that the Netflix API team began in 2011. In 2012, Hystrix continued to evolve and mature, and many teams within Netflix adopted it. Today tens of billions of thread-isolated, and hundreds of billions of semaphore-isolated calls are executed via Hystrix every day at Netflix. This has resulted in a dramatic improvement in uptime and resilience.

The following links provide more context around Hystrix and the challenges that it attempts to address:

# Making the Netflix API More Resilient

The API brokers catalog and subscriber metadata between internal services and Netflix applications on hundreds of device types. If any of these internal services fail there is a risk that the failure could propagate to the API and break the user experience for members.

To provide the best possible streaming experience for our members, it is critical for us to keep the API online and serving traffic at all times. Maintaining high availability and resiliency for a system that handles a billion requests a day is one of the goals of the API team, and we have made great progress toward achieving this goal over the last few months.

### Principles of Resiliency

Here are some of the key principles that informed our thinking as we set out to make the API more resilient.

1. A failure in a service dependency should not break the user experience for members
2. The API should automatically take corrective action when one of its service dependencies fails
3. The API should be able to show us what’s happening right now, in addition to what was happening 15–30 minutes ago, yesterday, last week, etc.

### Keep the Streams Flowing

As stated in the first principle above, we want members to be able to continue instantly watching movies and TV shows streaming from Netflix when server failures occur, even if the experience is slightly degraded and less personalized. To accomplish this we’ve restructured the API to enable graceful fallback mechanisms to kick in when a service dependency fails. We decorate calls to service dependencies with code that tracks the result of each call. When we detect that a service is failing too often we stop calling it and serve fallback responses while giving the failing service time to recover. We then periodically let some calls to the service go through and if they succeed then we open traffic for all calls.

If this pattern sounds familiar to you, you’re probably thinking of the CircuitBreaker pattern from Michael Nygard’s book “[Release It! Design and Deploy Production-Ready Software](http://pragprog.com/book/mnee/release-it)”, which influenced the implementation of our service dependency decorator code. Our implementation goes a little further than the basic CircuitBreaker pattern in that fallbacks can be triggered in a few ways:

1. A request to the remote service times out
2. The thread pool and bounded task queue used to interact with a service dependency are at 100% capacity
3. The client library used to interact with a service dependency throws an exception

These buckets of failures factor into a service’s overall error rate and when the error rate exceeds a defined threshold then we “trip” the circuit for that service and immediately serve fallbacks without even attempting to communicate with the remote service.

Each service that’s wrapped by a circuit breaker implements a fallback using one of the following three approaches:

1. **Custom fallback** — in some cases a service’s client library provides a fallback method we can invoke, or in other cases we can use locally available data on an API server (eg, a cookie or local JVM cache) to generate a fallback response
2. **Fail silent** — in this case the fallback method simply returns a null value, which is useful if the data provided by the service being invoked is optional for the response that will be sent back to the requesting client
3. **Fail fast** — used in cases where the data is required or there’s no good fallback and results in a client getting a 5xx response. This can negatively affect the device UX, which is not ideal, but it keeps API servers healthy and allows the system to recover quickly when the failing service becomes available again.

Ideally, all service dependencies would have custom fallbacks as they provide the best possible user experience (given the circumstances). Although that is our goal, it’s also very challenging to maintain complete fallback coverage for many service dependencies. So the fail silent and fail fast approaches are reasonable alternatives.

### Real-time Stats Drive Software and Diagnostics

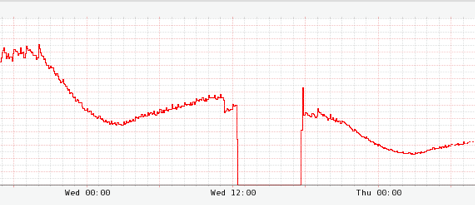
I mentioned that our circuit breaker/fallback code tracks and acts on requests to service dependencies. This code counts requests to each service dependency over a 10 second rolling window. The window is rolling in the sense that request stats that are older than 10 seconds are discarded; only the results of requests over the last 10 seconds matter to the code. We also have a dashboard that’s wired up to these same stats that shows us the state of our service dependencies for the last 10 seconds, which comes in really handy for diagnostics.

You might ask, “Do you really need a dashboard that shows you the state of your service dependencies for the last 10 seconds?” The Netflix API receives around 20,000 requests per second at peak traffic. At that rate, 10 seconds translates to 200,000 requests from client devices, which can easily translate to 1,000,000+ requests from the API into upstream services. A lot can happen in 10 seconds, and we want our software to base its decision making on what just happened, not what was happening 10 or 15 minutes ago. These real-time insights can also help us identify and react to issues before they become member-facing problems. (Of course, we have charts for identifying trends beyond the 10 second window, too.)

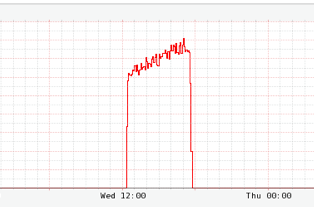
### Circuit Breaker in Action

Now that I’ve described the basics of the service dependency decorator layer that we’ve built, here’s a real world example that demonstrates the value it can provide.The data that the API uses to respond to certain requests is stored in a database but it’s also cached as a Java object in a shared cache. Upon receiving one of these requests, the API looks in the shared cache and if the object isn’t there it queries the database. One day we discovered a bug where the API was occasionally loading a Java object into the shared cache that wasn’t fully populated, which had the effect of intermittently causing problems on certain devices.

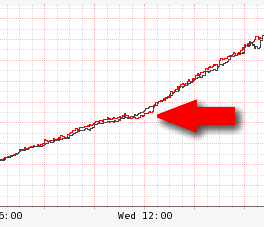
Once we discovered the problem, we decided to bypass the shared cache and go directly to the database while we worked on a patch. The following chart shows cache hits and that disabling the cache had the expected effect of dropping hits to zero.



What we weren’t counting on was getting throttled by our database for sending it too much traffic. Fortunately, we implemented custom fallbacks for database selects and so our service dependency layer started automatically tripping the corresponding circuit and invoking our fallback method, which checked a local query cache on database failure. This next chart shows the spike in fallback responses.

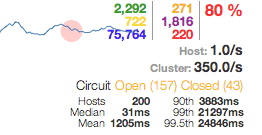


The fallback query cache had most of our active data set and so the overall impact to member experience was very low as can be seen by the following chart, which shows a minimal effect on overall video views. (The red line is video views per second this week and the black line is the same metric last week.)



#### Show Me, Don’t Tell Me

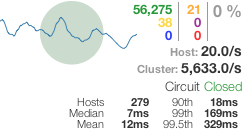
While this was happening, we were able to see exactly what the system was doing by looking at our dashboard, which processes a data stream that includes the same stats used by the circuit breaker code. Here’s an excerpt that shows what the dashboard looked like during the incident.



The red 80% in the upper right shows the overall error rate for our database select circuit, and the “Open” and “Closed” counts show that the majority of server instances (157 of 200) were serving fallback responses. The blue count is the number of short-circuited requests that were never sent to the database server.

The dashboard is based on the classic green, yellow, red traffic light status page pattern and is designed to be quickly scannable. Each circuit (we have ~60 total at this point) has a circle to the left that encodes call volume (size of the circle — bigger means more traffic) and health (color of the circle — green is healthy and red indicates a service that’s having problems). The sparkline indicates call volume over a 2 minute rolling window (though the stats outside of the 10 second window are just used for display and don’t factor into the circuit breaker logic).

Here’s an example of what a healthy circuit looks like.



# Fault Tolerance in a High Volume, Distributed System

by Ben Christensen

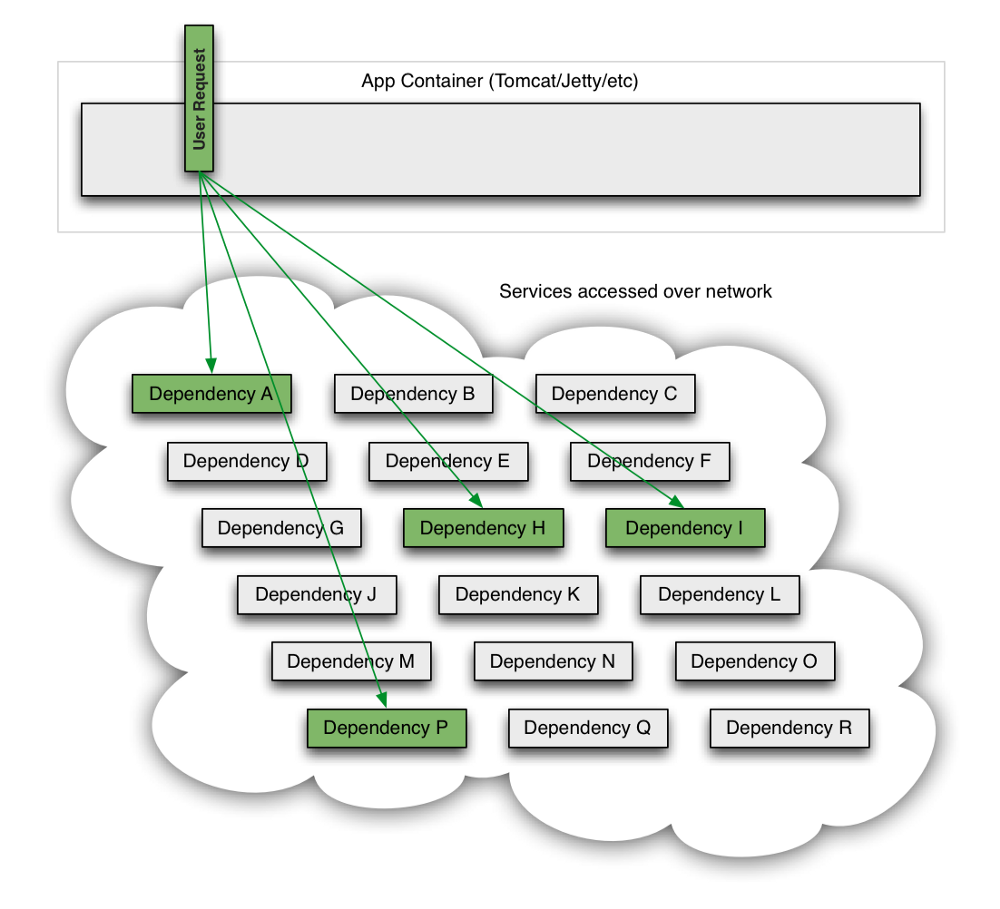
In an [earlier post](https://medium.com/@Netflix_Techblog/making-the-netflix-api-more-resilient-a8ec62159c2d) by [Ben Schmaus](http://twitter.com/schmaus), we shared the principles behind our circuit-breaker implementation. In that post, Ben discusses how the Netflix API interacts with dozens of systems in our service-oriented architecture, which makes the API inherently more vulnerable to any system failures or latencies underneath it in the stack.

[**Making the Netflix API More Resilient**  
Maintaining high availability and resiliency for a system that handles a billion requests a day.medium.com](https://medium.com/@Netflix_Techblog/making-the-netflix-api-more-resilient-a8ec62159c2d)

The rest of this post provides a more technical deep-dive into how our API and other systems isolate failure, shed load and remain resilient to failures.

#### Fault Tolerance is a Requirement, Not a Feature

The Netflix API receives more than 1 billion incoming calls per day which in turn fans out to several billion outgoing calls (averaging a ratio of 1:6) to dozens of underlying subsystems with peaks of over 100k dependency requests per second.

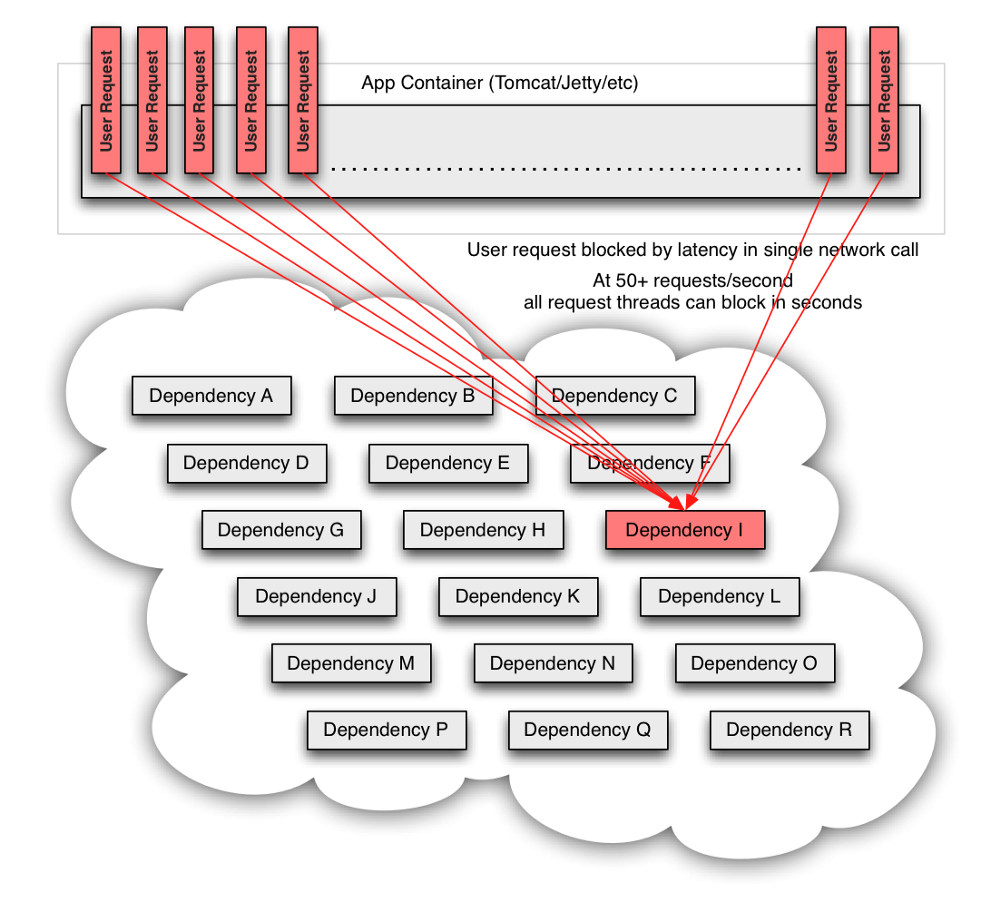


This all occurs in the cloud across thousands of EC2 instances.

Intermittent failure is guaranteed with this many variables, even if every dependency itself has excellent availability and uptime.

Without taking steps to ensure fault tolerance, 30 dependencies each with 99.99% uptime would result in 2+ hours downtime/month (99.99%30= 99.7% uptime = 2+ hours in a month).

When a single API dependency fails at high volume with increased latency (causing blocked request threads) it can rapidly (seconds or sub-second) saturate all available Tomcat (or other container such as Jetty) request threads and take down the entire API.



Thus, it is a requirement of high volume, high availability applications to build fault tolerance into their architecture and not expect infrastructure to solve it for them.

#### Netflix DependencyCommand Implementation

The service-oriented architecture at Netflix allows each team freedom to choose the best transport protocols and formats (XML, JSON, Thrift, Protocol Buffers, etc) for their needs so these approaches may vary across services.

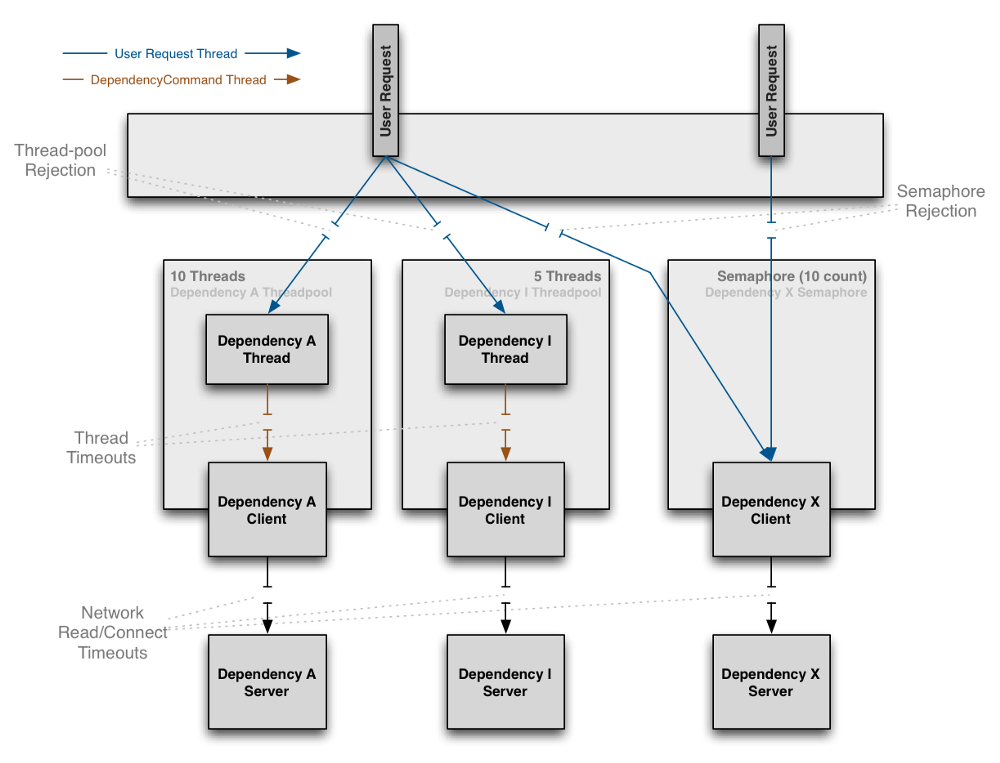
In most cases the team providing a service also distributes a Java client library.

Because of this, applications such as API in effect treat the underlying dependencies as 3rd party client libraries whose implementations are “black boxes”. This in turn affects how fault tolerance is achieved.

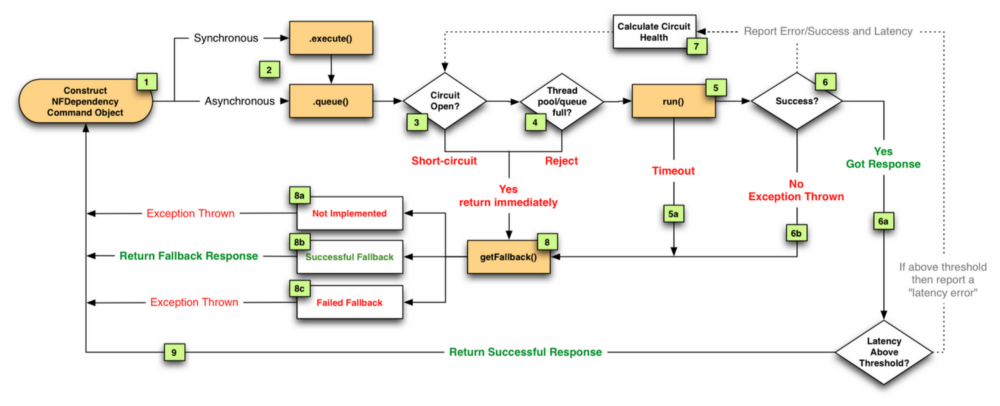
In light of the above architectural considerations we chose to implement a solution that uses a combination of fault tolerance approaches:

* network timeouts and retries
* separate threads on per-dependency thread pools
* semaphores (via a [tryAcquire](http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/Semaphore.html#tryAcquire%28%29), not a blocking call)
* circuit breakers

Each of these approaches to fault-tolerance has pros and cons but when combined together provide a comprehensive protective barrier between user requests and underlying dependencies.

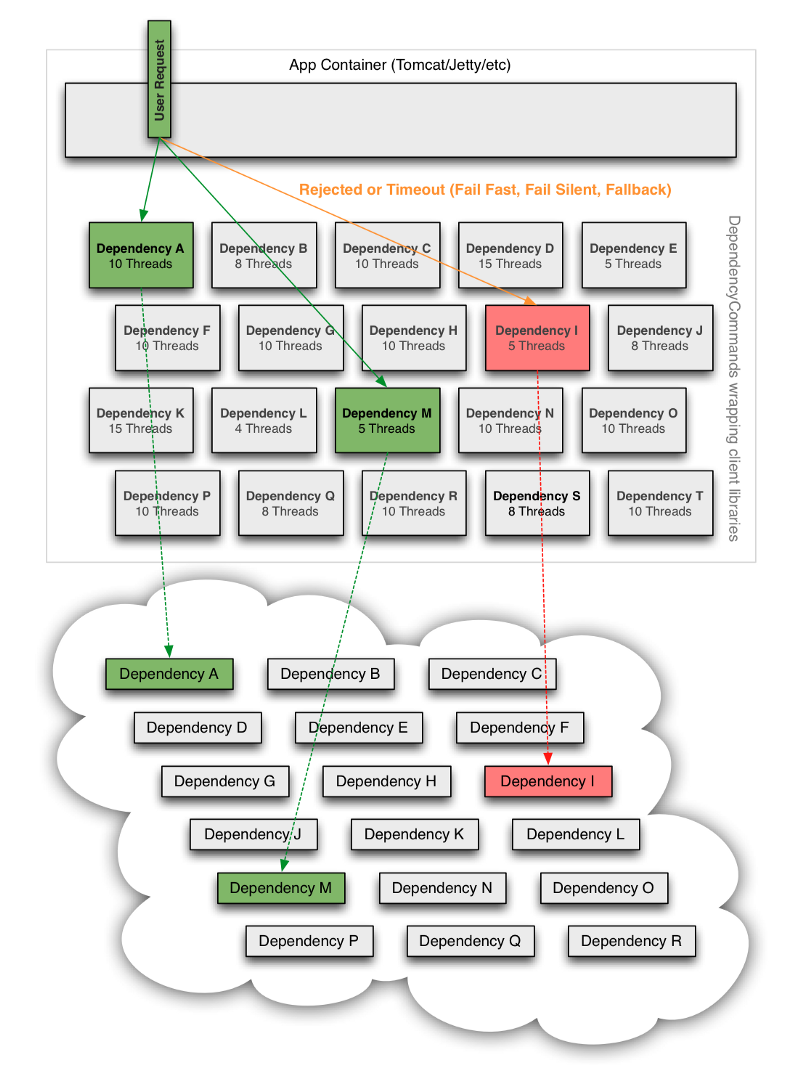


The Netflix DependencyCommand implementation wraps a network-bound dependency call with a preference towards executing in a separate thread and defines fallback logic which gets executed (step 8 in flow chart below) for any failure or rejection (steps 3, 4, 5a, 6b below) regardless of which type of fault tolerance (network or thread timeout, thread pool or semaphore rejection, circuit breaker) triggered it.

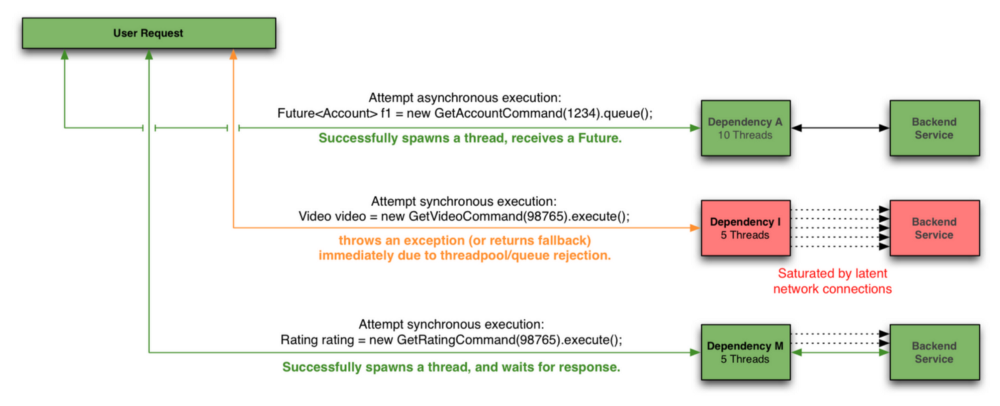


We decided that the benefits of isolating dependency calls into separate threads outweighs the drawbacks (in most cases). Also, since the API is progressively [moving towards increased concurrency](https://medium.com/@Netflix_Techblog/redesigning-the-netflix-api-db5a7221fcff) it was a win-win to achieve both fault tolerance and performance gains through concurrency with the same solution. In other words, the overhead of separate threads is being turned into a positive in many use cases by leveraging the concurrency to execute calls in parallel and speed up delivery of the Netflix experience to users.

Thus, most dependency calls now route through a separate thread-pool as the following diagram illustrates:



If a dependency becomes latent (the worst-case type of failure for a subsystem) it can saturate all of the threads in its own thread pool, but Tomcat request threads will timeout or be rejected immediately rather than blocking.



In addition to the isolation benefits and concurrent execution of dependency calls we have also leveraged the separate threads to enable request collapsing (automatic batching) to increase overall efficiency and reduce user request latencies.

Semaphores are used instead of threads for dependency executions known to not perform network calls (such as those only doing in-memory cache lookups) since the overhead of a separate thread is too high for these types of operations.

We also use semaphores to protect against non-trusted fallbacks. Each DependencyCommand is able to define a fallback function (discussed more below) which is performed on the calling user thread and should not perform network calls. Instead of trusting that all implementations will correctly abide to this contract, it too is protected by a semaphore so that if an implementation is done that involves a network call and becomes latent, the fallback itself won’t be able to take down the entire app as it will be limited in how many threads it will be able to block.

Despite the use of separate threads with timeouts, we continue to aggressively set timeouts and retries at the network level (through interaction with client library owners, monitoring, audits etc).

The timeouts at the DependencyCommand threading level are the first line of defense regardless of how the underlying dependency client is configured or behaving but the network timeouts are still important otherwise highly latent network calls could fill the dependency thread-pool indefinitely.

The tripping of circuits kicks in when a DependencyCommand has passed a certain threshold of error (such as 50% error rate in a 10 second period) and will then reject all requests until health checks succeed.

This is used primarily to release the pressure on underlying systems (i.e. shed load) when they are having issues and reduce the user request latency by failing fast (or returning a fallback) when we know it is likely to fail instead of making every user request wait for the timeout to occur.

#### How do we respond to a user request when failure occurs?

In each of the options described above a timeout, thread-pool or semaphore rejection, or short-circuit will result in a request not retrieving the optimal response for our customers.

An immediate failure (“fail fast”) throws an exception which causes the app to shed load until the dependency returns to health. This is preferable to requests “piling up” as it keeps Tomcat request threads available to serve requests from healthy dependencies and enables rapid recovery once failed dependencies recover.

However, there are often several preferable options for providing responses in a “fallback mode” to reduce impact of failure on users. Regardless of what causes a failure and how it is intercepted (timeout, rejection, short-circuited etc) the request will always pass through the fallback logic (step 8 in flow chart above) before returning to the user to give a DependencyCommand the opportunity to do something other than “fail fast”.

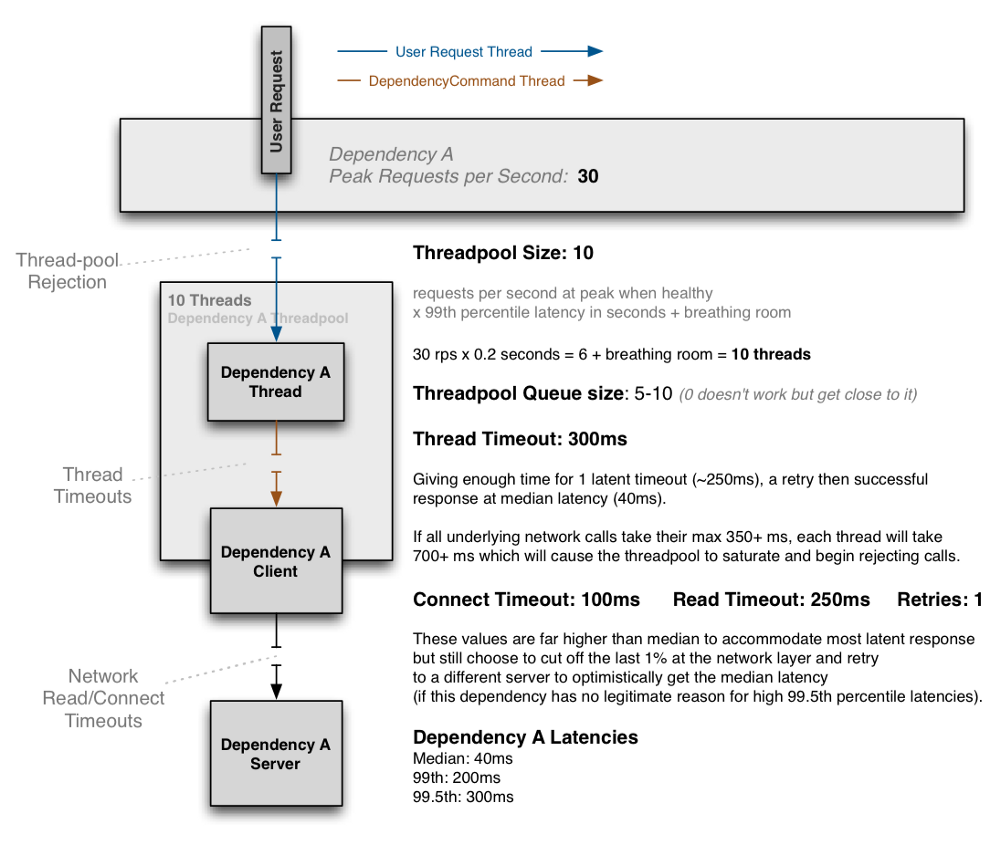
Some approaches to fallbacks we use are, in order of their impact on the user experience:

* Cache: Retrieve data from local or remote caches if the realtime dependency is unavailable, even if the data ends up being stale
* Eventual Consistency: Queue writes (such as in [SQS](http://aws.amazon.com/sqs/)) to be persisted once the dependency is available again
* Stubbed Data: Revert to default values when personalized options can’t be retrieved
* Empty Response (“Fail Silent”): Return a null or empty list which UIs can then ignore

All of this work is to maintain maximum uptime for our users while maintaining the maximum number of features for them to enjoy the richest Netflix experience possible. As a result, our goal is to have the fallbacks deliver responses as close to what the actual dependency would deliver.

#### Example Use Case

Following is an example of how threads, network timeouts and retries combine:



The above diagram shows an example configuration where the dependency has no reason to hit the 99.5th percentile and thus cuts it short at the network timeout layer and immediately retries with the expectation to get median latency most of the time, and accomplish this all within the 300ms thread timeout.

If the dependency has legitimate reasons to sometimes hit the 99.5th percentile (i.e. cache miss with lazy generation) then the network timeout will be set higher than it, such as at 325ms with 0 or 1 retries and the thread timeout set higher (350ms+).

The threadpool is sized at 10 to handle a burst of 99th percentile requests, but when everything is healthy this threadpool will typically only have 1 or 2 threads active at any given time to serve mostly 40ms median calls.

When configured correctly a timeout at the DependencyCommand layer should be rare, but the protection is there in case something other than network latency affects the time, or the combination of connect+read+retry+connect+read in a worst case scenario still exceeds the configured overall timeout.

The aggressiveness of configurations and tradeoffs in each direction are different for each dependency.

Configurations can be changed in realtime as needed as performance characteristics change or when problems are found all without risking the taking down of the entire app if problems or misconfigurations occur.

#### Conclusion

The approaches discussed in this post have had a dramatic effect on our ability to tolerate and be resilient to system, infrastructure and application level failures without impacting (or limiting impact to) user experience.

Despite the success of this new DependencyCommand resiliency system over the past 8 months, there is still a lot for us to do in improving our fault tolerance strategies and performance, especially as we continue to add functionality, devices, customers and international markets.

## What Is Hystrix For?

Hystrix is designed to do the following:

* Give protection from and control over latency and failure from dependencies accessed (typically over the network) via third-party client libraries.
* Stop cascading failures in a complex distributed system.
* Fail fast and rapidly recover.
* Fallback and gracefully degrade when possible.
* Enable near real-time monitoring, alerting, and operational control.

## What Problem Does Hystrix Solve?

Applications in complex distributed architectures have dozens of dependencies, each of which will inevitably fail at some point. If the host application is not isolated from these external failures, it risks being taken down with them.

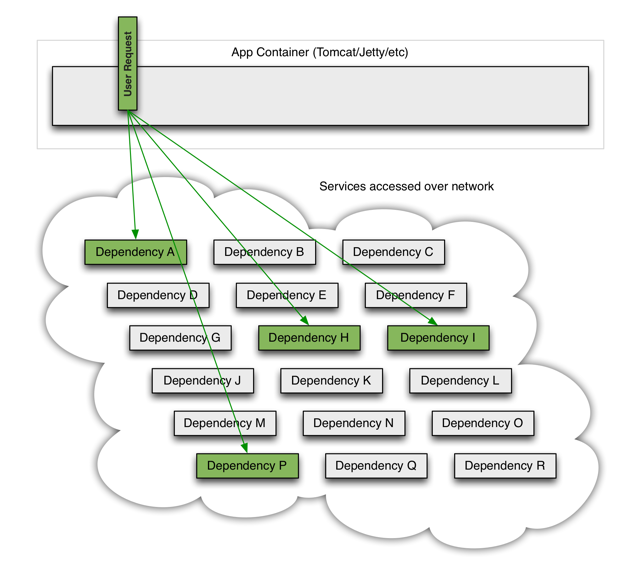
For example, for an application that depends on 30 services where each service has 99.99% uptime, here is what you can expect:

99.9930 = 99.7% uptime  
0.3% of 1 billion requests = 3,000,000 failures  
2+ hours downtime/month even if all dependencies have excellent uptime.

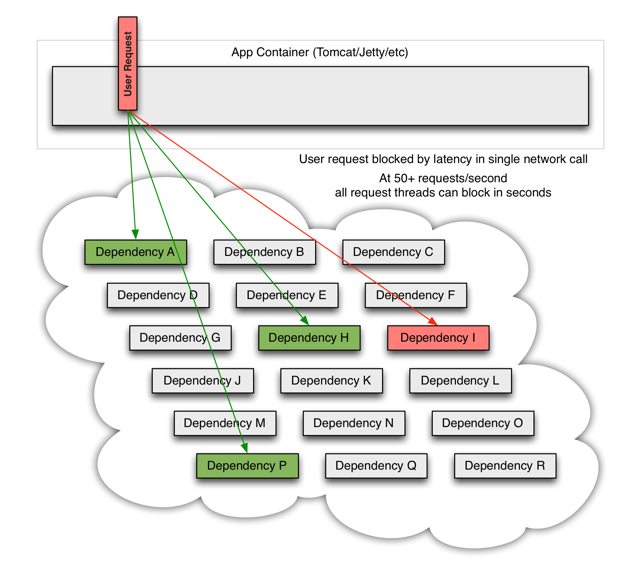
Reality is generally worse.

Even when all dependencies perform well the aggregate impact of even 0.01% downtime on each of dozens of services equates to potentially hours a month of downtime **if you do not engineer the whole system for resilience**.

When everything is healthy the request flow can look like this:

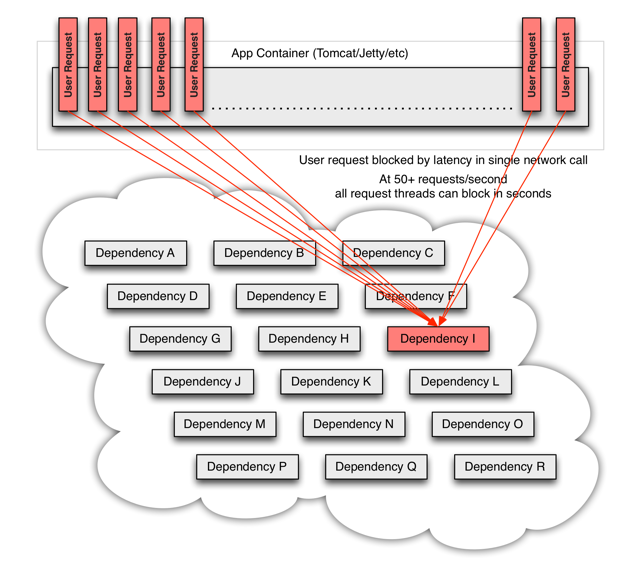


When one of many backend systems becomes latent it can block the entire user request:



With high volume traffic a single backend dependency becoming latent can cause all resources to become saturated in seconds on all servers.

Every point in an application that reaches out over the network or into a client library that might result in network requests is a source of potential failure. Worse than failures, these applications can also result in increased latencies between services, which backs up queues, threads, and other system resources causing even more cascading failures across the system.



These issues are exacerbated when network access is performed through a third-party client — a “black box” where implementation details are hidden and can change at any time, and network or resource configurations are different for each client library and often difficult to monitor and change.

Even worse are transitive dependencies that perform potentially expensive or fault-prone network calls without being explicitly invoked by the application.

Network connections fail or degrade. Services and servers fail or become slow. New libraries or service deployments change behavior or performance characteristics. Client libraries have bugs.

All of these represent failure and latency that needs to be isolated and managed so that a single failing dependency can’t take down an entire application or system.

## What Design Principles Underlie Hystrix?

Hystrix works by:

* Preventing any single dependency from using up all container (such as Tomcat) user threads.
* Shedding load and failing fast instead of queueing.
* Providing fallbacks wherever feasible to protect users from failure.
* Using isolation techniques (such as bulkhead, swimlane, and circuit breaker patterns) to limit the impact of any one dependency.
* Optimizing for time-to-discovery through near real-time metrics, monitoring, and alerting
* Optimizing for time-to-recovery by means of low latency propagation of configuration changes and support for dynamic property changes in most aspects of Hystrix, which allows you to make real-time operational modifications with low latency feedback loops.
* Protecting against failures in the entire dependency client execution, not just in the network traffic.

## How Does Hystrix Accomplish Its Goals?

Hystrix does this by:

* Wrapping all calls to external systems (or “dependencies”) in a HystrixCommand or HystrixObservableCommand object which typically executes within a separate thread (this is an example of the [command pattern](http://en.wikipedia.org/wiki/Command_pattern)).
* Timing-out calls that take longer than thresholds you define. There is a default, but for most dependencies you custom-set these timeouts by means of “properties” so that they are slightly higher than the measured 99.5th percentile performance for each dependency.
* Maintaining a small thread-pool (or semaphore) for each dependency; if it becomes full, requests destined for that dependency will be immediately rejected instead of queued up.
* Measuring successes, failures (exceptions thrown by client), timeouts, and thread rejections.
* Tripping a circuit-breaker to stop all requests to a particular service for a period of time, either manually or automatically if the error percentage for the service passes a threshold.
* Performing fallback logic when a request fails, is rejected, times-out, or short-circuits.
* Monitoring metrics and configuration changes in near real-time.

When you use Hystrix to wrap each underlying dependency, the architecture as shown in diagrams above changes to resemble the following diagram. Each dependency is isolated from one other, restricted in the resources it can saturate when latency occurs, and covered in fallback logic that decides what response to make when any type of failure occurs in the dependency:

