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 Netflix API More Resilient"
- "Fault Tolerance in a High Volume, Distributed System"
- "Performance and Fault Tolerance for the Netflix API"
- "Application Resilience in a Service-oriented Architecture"
- ["Application Resilience Engineering & Operations at Netflix"] (https://speakerdeck.com/benjchristensen/application-resilience-engineering-and-operations-at-netflix)

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.

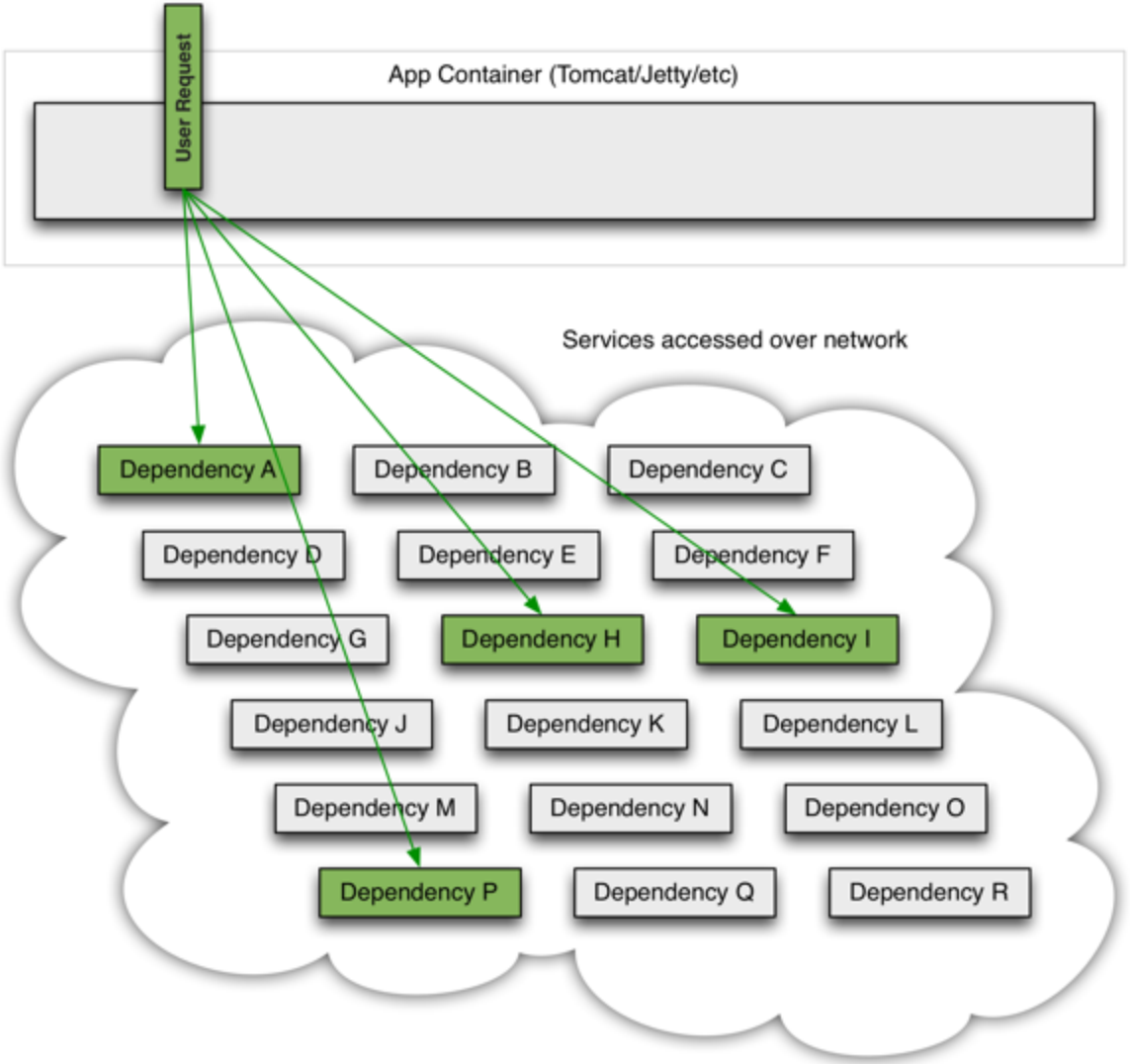
For example, for an application that depends on 30 services where each service has 99.99% uptime, here is what you can expect:

> 99.99^{30} = 99.7% uptime
> 0.3% of 1 billion requests = 3,000,000 failures
> 2+ hours downtime/month even if all dependencies have excellent uptime.

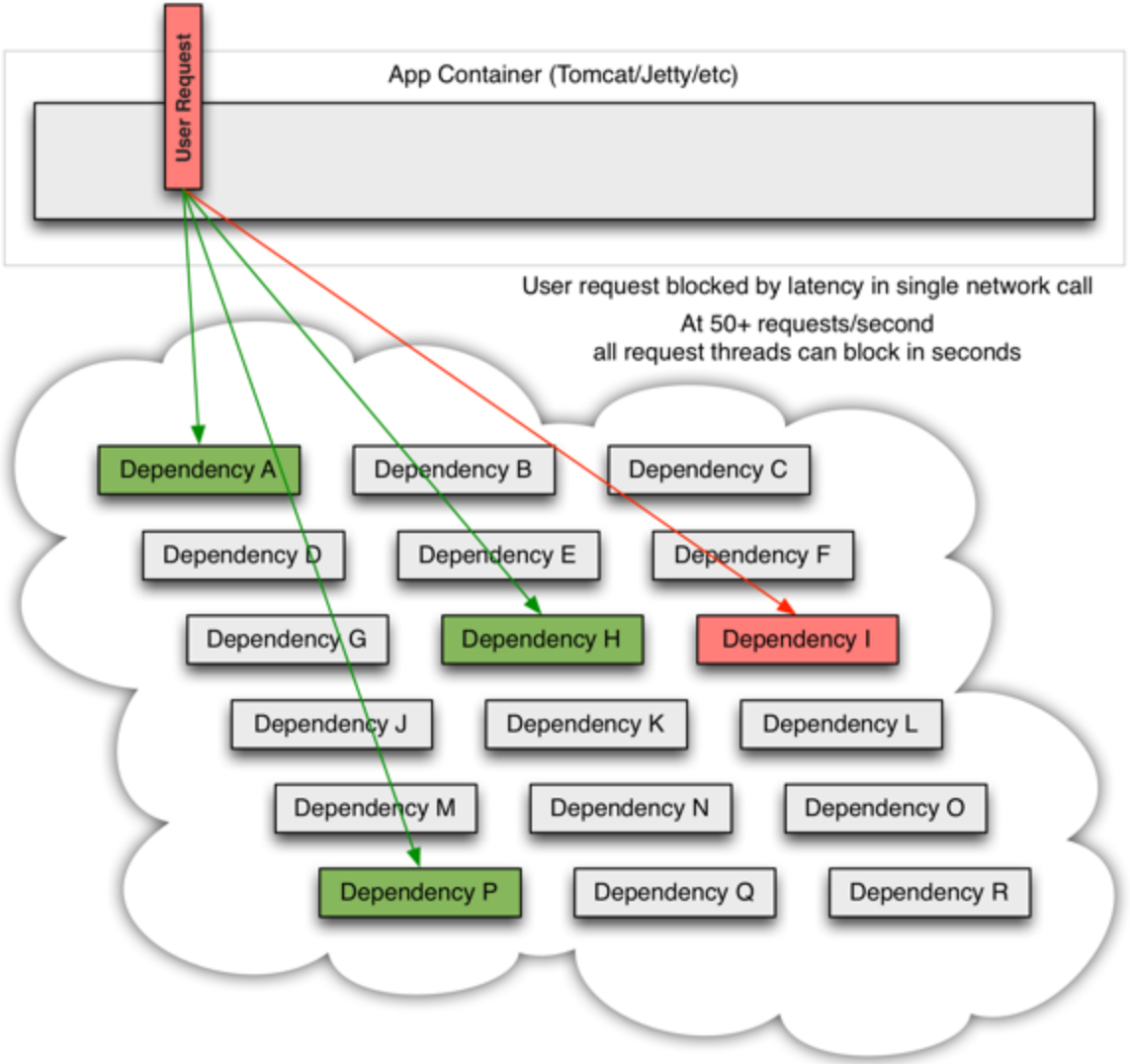
Reality is generally worse.

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:

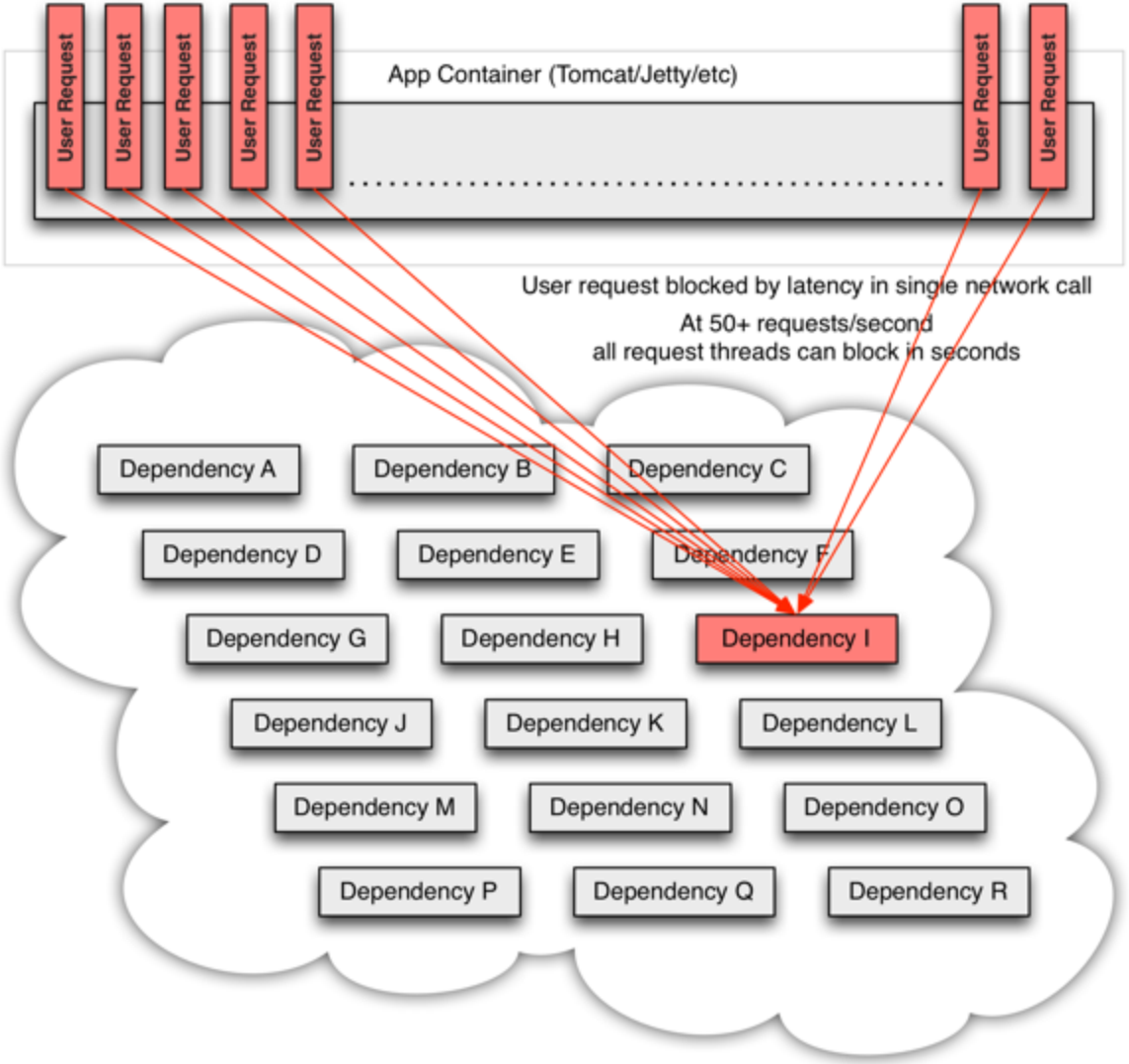


When one of many backend systems becomes latent it can block the entire user request:



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.

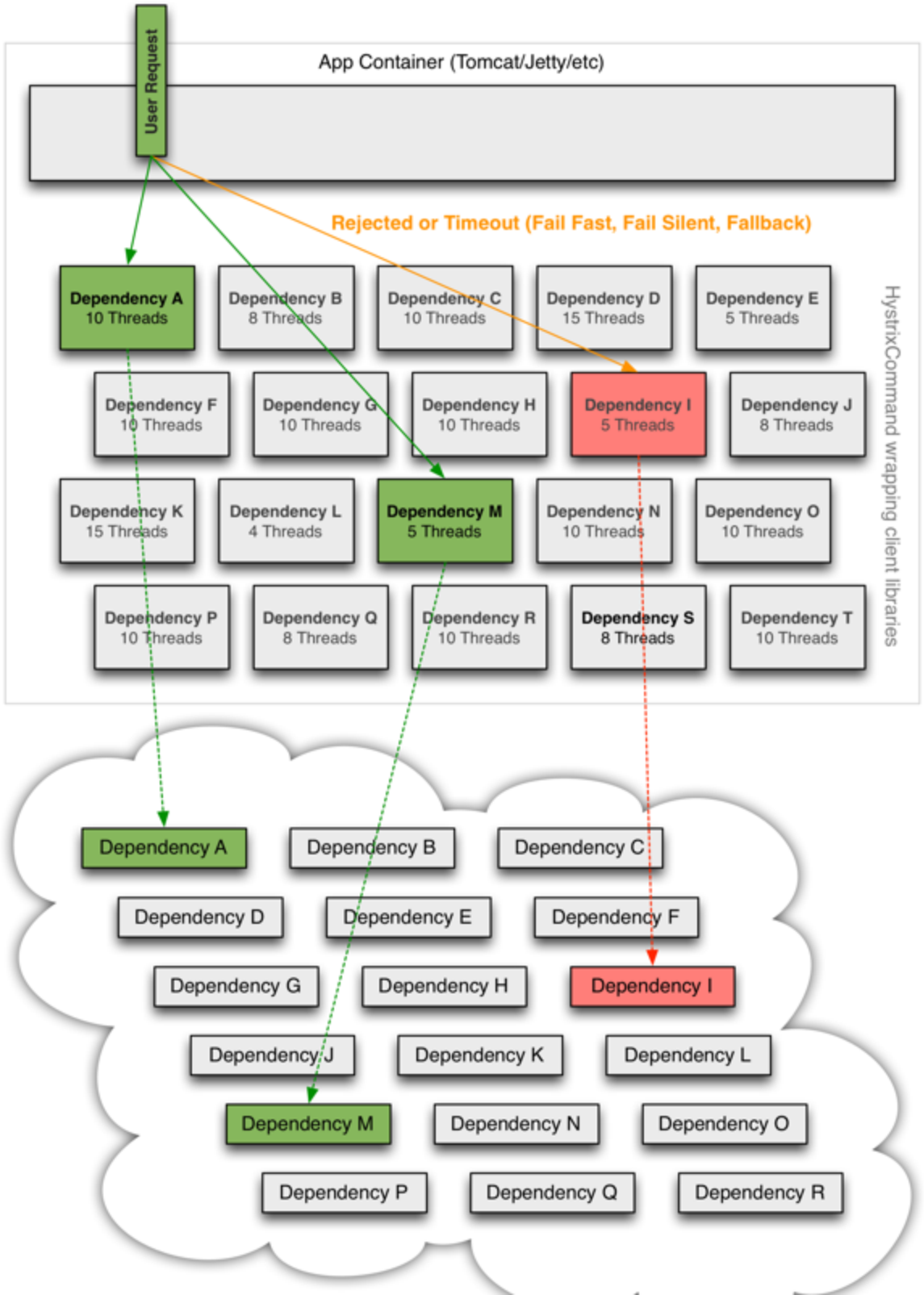
How Does Hystrix Accomplish Its Goals?

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).
- 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:


User Request
App Container (Tomcat/Jetty/etc)
Rejected or Timeout (Fail Fast, Fail Silent, Fallback)
Dependency A
10 Threads
Dependency B
8 Threads
Dependency C
10 Threads
Dependency D
15 Threads
Dependency E
5 Threads
Dependency F
10 Threads
Dependency G
10 Threads
Dependency H
10 Threads
Dependency I
5 Threads
Dependency J
8 Threads
Dependency K
15 Threads
Dependency L
4 Threads
Dependency M
5 Threads
Dependency N
10 Threads
Dependency O
10 Threads
Dependency P
10 Threads
Dependency Q
8 Threads
Dependency R
10 Threads
Dependency S
8 Threads
Dependency T
10 Threads
HystrixCommand wrapping client libraries
Dependency A
Dependency B
Dependency C
Dependency D
Dependency E
Dependency F
Dependency G
Dependency H
Dependency I
Dependency J
Dependency K
Dependency L
Dependency M
Dependency N
Dependency O
Dependency P
Dependency Q
Dependency R