A photograph of a tall palm tree with dense green fronds, standing on a sandy beach. The background is a bright blue sky with wispy white clouds.

Resilience with Hystrix

An introduction

Uwe Friedrichsen, codecentric AG, 2014

@ufried



A photograph of a long, narrow hallway. The walls are painted a vibrant blue. The floor is made of large, light-colored tiles. A person wearing a blue shirt and white pants is walking away from the camera, towards the end of the hallway. The ceiling is white with several rectangular fluorescent light fixtures. The perspective of the hallway creates a sense of depth.

It's all about production!

Production



Availability



Resilience



Fault Tolerance

re•sil•ience (rī'zil yəns) also **re•sil'iен•cy**, n.

1. the power or ability to return to the original form, position, etc., after being bent, compressed, or stretched; elasticity.
2. ability to recover readily from illness, depression, adversity, or the like; buoyancy.

Random House Kernerman Webster's College Dictionary, © 2010 K Dictionaries Ltd.
Copyright 2005, 1997, 1991 by Random House, Inc. All rights reserved.

PUBLIC



Netflix / Hystrix

Star

1,580

Fork

219

Home

Pages

History

Home



HYSTRIX

DEFEND YOUR APP

Page History

Clone URL

- [Home](#)
- [Getting Started](#)
- [How To Use](#)
 - [Hello World!](#)
 - [Synchronous Execution](#)
 - [Asynchronous Execution](#)
 - [Reactive Execution](#)
 - [Fallback](#)
 - [Error Propagation](#)
 - [Command Name](#)
 - [Command Group](#)
 - [Command Thread Pool](#)
 - [Request Cache](#)
 - [Request Collapsing](#)
 - [Request Context Setup](#)
 - [Common Patterns](#)
 - [Migrating to Hystrix](#)
- [How It Works](#)
 - [Execution Flow](#)
 - [Circuit Breaker](#)
 - [Isolation](#)
 - [Request Collapsing](#)
 - [Request Caching](#)

What is Hystrix?

In a distributed environment, failure of any given service is inevitable. Hystrix is a library designed to control the interactions between these distributed services providing greater latency and fault tolerance. Hystrix does this by isolating points of access between the services, stopping cascading failures across them, and providing fallback options, all of which improve the system's overall resiliency.

Hystrix evolved out of resilience engineering work that the Netflix API team began in 2011. Over the course of 2012, Hystrix continued to evolve and mature, eventually leading to adoption

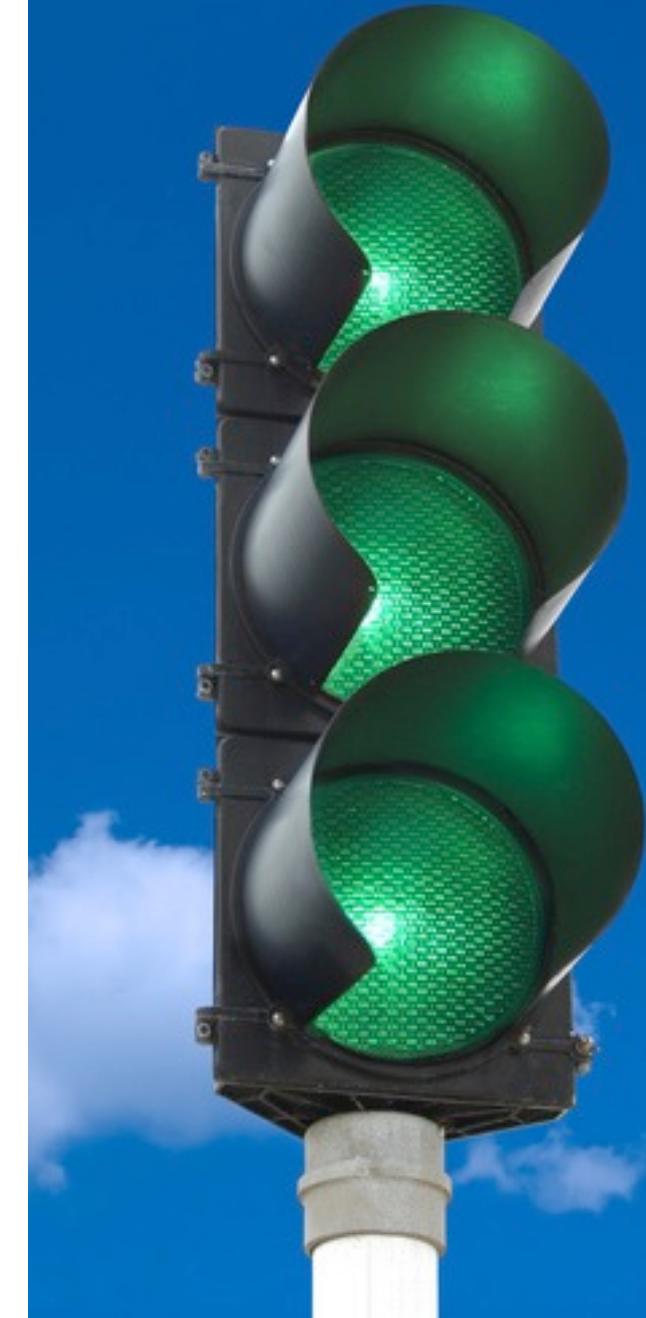
Implemented patterns



- Timeout
- Circuit breaker
- Load shedder

Supported patterns

- Bulkheads
(a.k.a. Failure Units)
- Fail fast
- Fail silently
- Graceful degradation of service
- Failover
- Escalation
- Retry
- ...

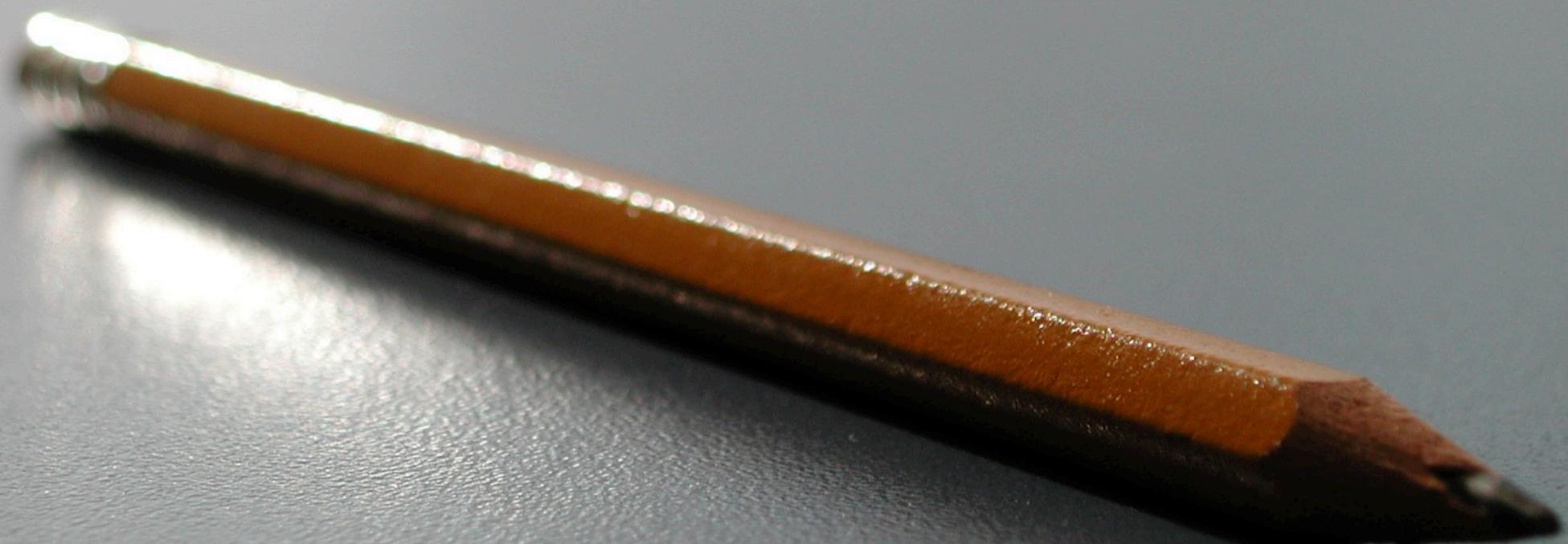


A photograph showing a person's hands holding a large, sprawling pile of old computer keyboards. The keyboards are mostly white or light-colored plastic, with various keys and some cables visible. The background is a plain, light-colored wall.

That's been enough theory

Give us some code – now!

Basics



Hello, world!

```
public class HelloCommand extends HystrixCommand<String> {
    private static final String COMMAND_GROUP = "default";
    private final String name;

    public HelloCommand(String name) {
        super(HystrixCommandGroupKey.Factory.asKey(COMMAND_GROUP));
        this.name = name;
    }

    @Override
    protected String run() throws Exception {
        return "Hello, " + name;
    }
}
```

```
@Test
public void shouldGreetWorld() {
    String result = new HelloCommand("World").execute();
    assertEquals("Hello, World", result);
}
```

Synchronous execution

```
@Test
public void shouldExecuteSynchronously() {
    String s = new HelloCommand("Bob").execute();
    assertEquals("Hello, Bob", s);
}
```

Asynchronous execution

```
@Test
public void shouldExecuteAsynchronously() {
    Future<String> f = new HelloCommand("Alice").queue();

    String s = null;
    try {
        s = f.get();
    } catch (InterruptedException | ExecutionException e) {
        // Handle Future.get() exceptions here
    }

    assertEquals("Hello, Alice", s);
}
```

Reactive execution (simple)

```
@Test
public void shouldExecuteReactiveSimple() {
    Observable<String> observable =
        new HelloCommand("Alice").observe();
    String s = observable.toBlockingObservable().single();

    assertEquals("Hello, Alice", s);
}
```

Reactive execution (full)

```
public class CommandHelloObserver implements Observer<String> {
    // Needed for communication between observer and test case
    private final AtomicReference<String> aRef;
    private final Semaphore semaphore;

    public CommandHelloObserver(AtomicReference<String> aRef,
                                Semaphore semaphore) {
        this.aRef = aRef;
        this.semaphore = semaphore;
    }

    @Override
    public void onCompleted() { // Not needed here }

    @Override
    public void onError(Throwable e) { // Not needed here }

    @Override
    public void onNext(String s) {
        aRef.set(s);
        semaphore.release();
    }
}
```

```
@Test
public void shouldExecuteReactiveFull() {
    // Needed for communication between observer and test case
    AtomicReference<String> aRef = new AtomicReference<>();
    Semaphore semaphore = new Semaphore(0);

    Observable<String> observable = new HelloCommand("Bob").observe();
    Observer<String> observer =
        new CommandHelloObserver(aRef, semaphore);
    observable.subscribe(observer);

    // Wait until observer received a result
    try {
        semaphore.acquire();
    } catch (InterruptedException e) {
        // Handle exceptions here
    }

    String s = aRef.get();
    assertEquals("Hello, Bob", s);
}
```

Fallback

```
public class FallbackCommand extends HystrixCommand<String> {
    private static final String COMMAND_GROUP = "default";

    public FallbackCommand() {
        super(HystrixCommandGroupKey.Factory.asKey(COMMAND_GROUP));
    }

    @Override
    protected String run() throws Exception {
        throw new RuntimeException("I will always fail");
    }

    @Override
    protected String getFallback() {
        return "Powered by fallback";
    }
}
```

```
@Test
public void shouldGetFallbackResponse() {
    String s = new FallbackCommand().execute();
    assertEquals("Powered by fallback", s);
}
```

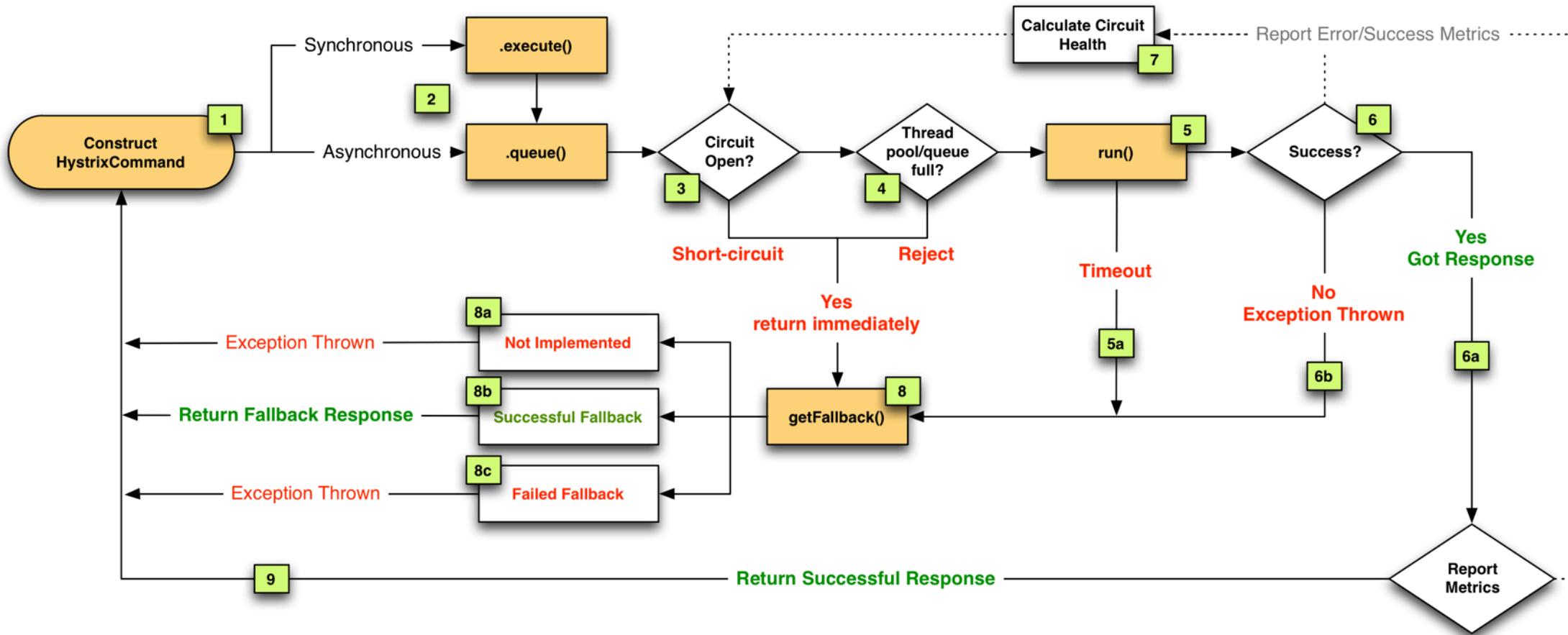
Error propagation

```
public class ErrorPropagationCommand extends HystrixCommand<String> {  
    private static final String COMMAND_GROUP = "default";  
  
    protected ErrorPropagationCommand() {  
        super(HystrixCommandGroupKey.Factory.asKey(COMMAND_GROUP));  
    }  
  
    @Override  
    protected String run() throws Exception {  
        throw new HystrixBadRequestException("I fail differently",  
            new RuntimeException("I will always fail"));  
    }  
  
    @Override  
    protected String getFallback() {  
        return "Powered by fallback";  
    }  
}
```

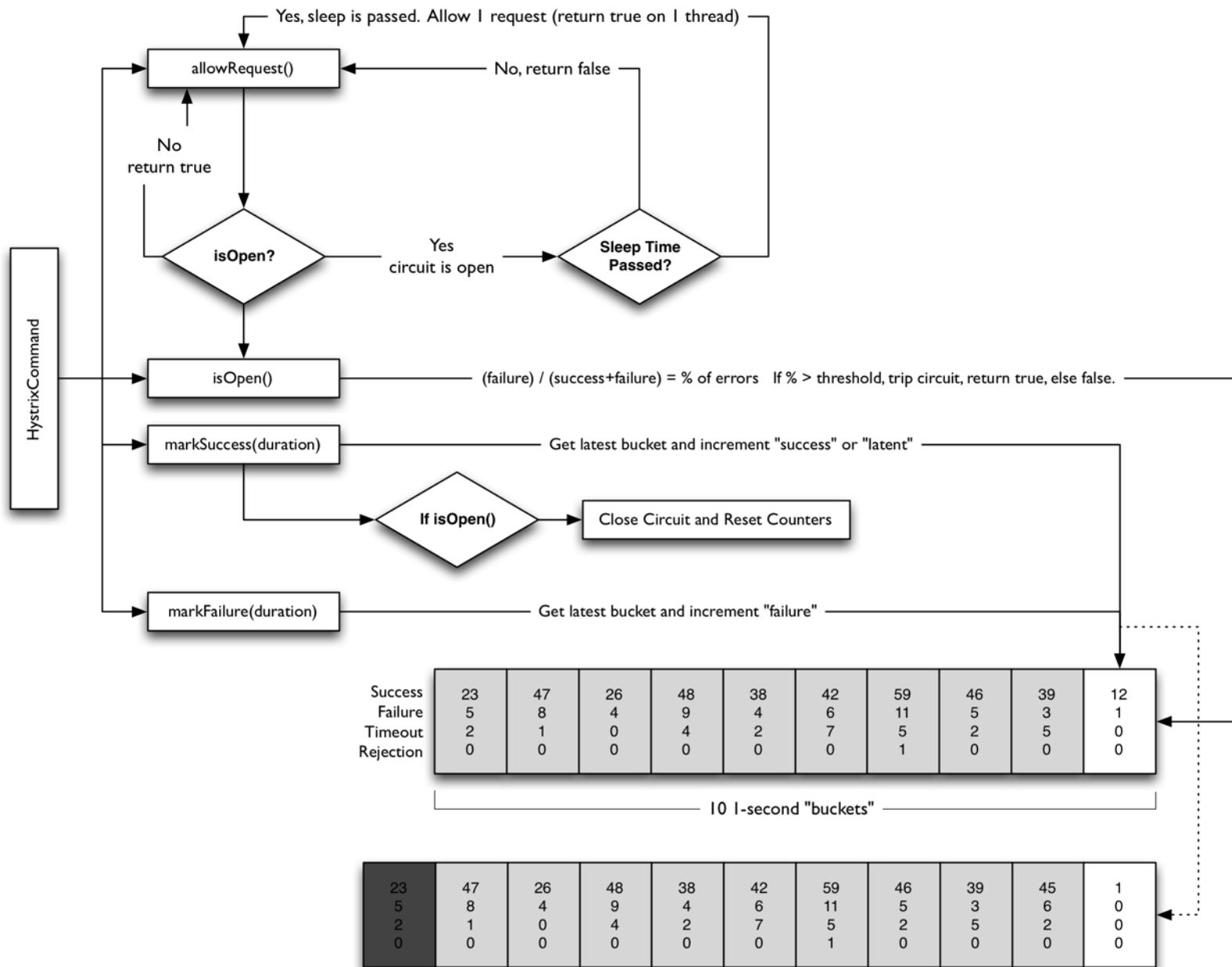
```
@Test
public void shouldGetFallbackResponse() {
    String s = null;
    try {
        s = new ErrorPropagationCommand().execute();
        fail(); // Just to make sure
    } catch (HystrixBadRequestException e) {
        assertEquals("I will fail differently", e.getMessage());
        assertEquals("I will always fail", e.getCause().getMessage());
    }
    assertNull(s); // Fallback is not triggered
}
```

How it works

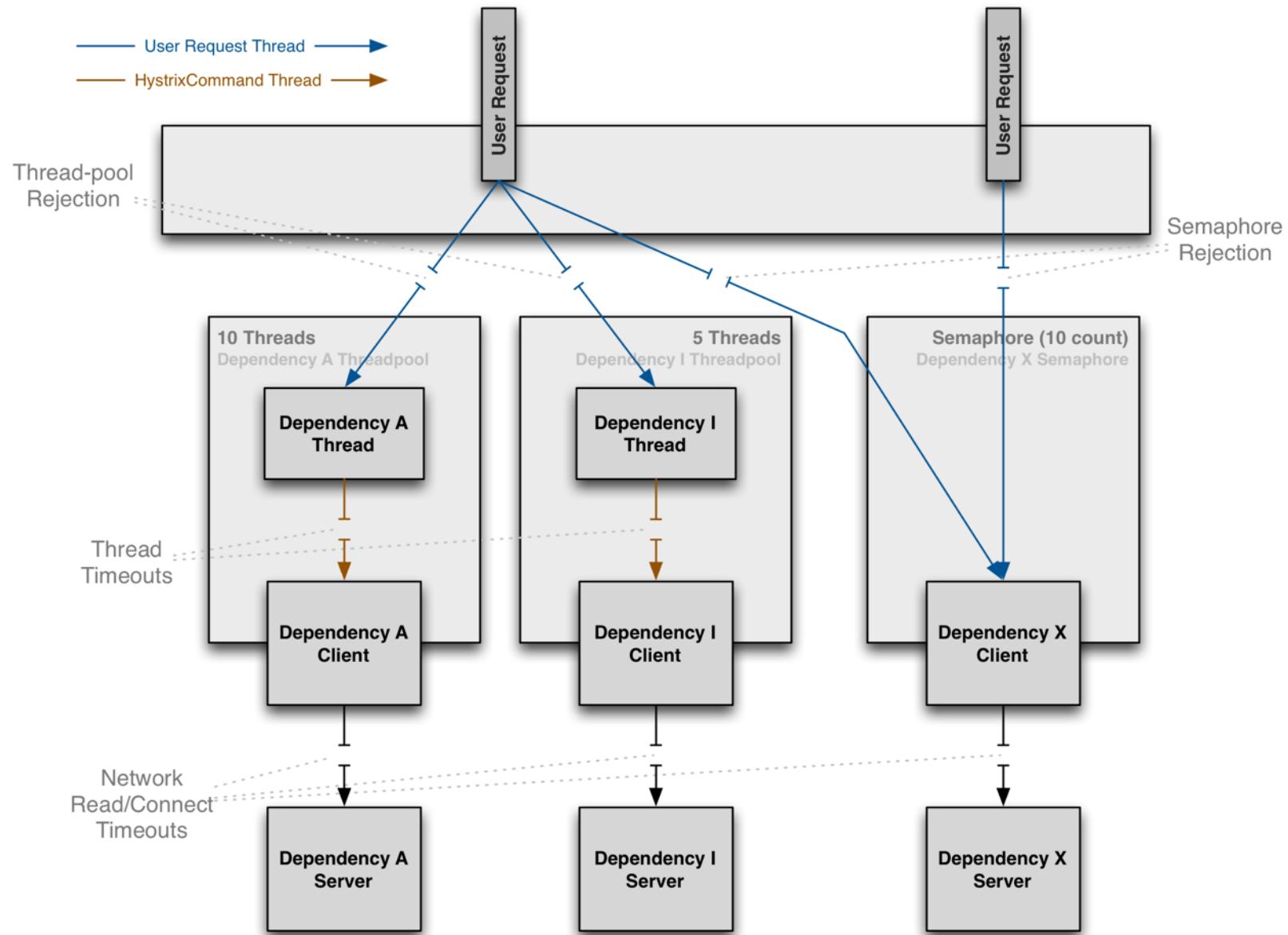




Source: <https://github.com/Netflix/Hystrix/wiki/How-it-Works>



On "getLatestBucket" if the 1-second window is passed a new bucket is created, the rest slid over and the oldest one dropped.



Source: <https://github.com/Netflix/Hystrix/wiki/How-it-Works>



Configuration

Configuration

- Based on Archaius by Netflix
which extends the Apache Commons Configuration Library
- Four levels of precedence
 1. Global default from code
 2. Dynamic global default property
 3. Instance default from code
 4. Dynamic instance property

Provides many configuration & tuning options



Configuration – Some examples

- `hystrix.command.*.execution.isolation.strategy`
Either "THREAD" or "SEMAPHORE" (Default: THREAD)
- `hystrix.command.*.execution.isolation.thread.timeoutInMilliseconds`
Time to wait for the run() method to complete (Default: 1000)
- `h.c.*.execution.isolation.semaphore.maxConcurrentRequests`
Maximum number of concurrent requests when using semaphores (Default: 10)
- `hystrix.command.*.circuitBreaker.errorThresholdPercentage`
Error percentage at which the breaker trips open (Default: 50)
- `hystrix.command.*.circuitBreaker.sleepWindowInMilliseconds`
Time to wait before attempting to reset the breaker after tripping (Default: 5000)

* must be either "default" or the command key name

Configuration – More examples

- `hystrix.threadpool.*.coreSize`

Maximum number of concurrent requests when using thread pools (Default: 10)

- `hystrix.threadpool.*.maxQueueSize`

Maximum LinkedBlockingQueue size - -1 for using SynchronousQueue (Default: -1)

- `hystrix.threadpool.default.queueSizeRejectionThreshold`

Queue size rejection threshold (Default: 5)

... and many more

* must be either "default" or the thread pool key name



Patterns

Timeout

```
public class LatentResource {
    private final long latency;

    public LatentResource(long latency) {
        this.latency = ((latency < 0L) ? 0L : latency);
    }

    public String getData() {
        addLatency(); // This is the important part
        return "Some value";
    }

    private void addLatency() {
        try {
            Thread.sleep(latency);
        } catch (InterruptedException e) {
            // We do not care about this here
        }
    }
}
```

```
public class TimeoutCommand extends HystrixCommand<String> {
    private static final String COMMAND_GROUP = "default";
    private final LatentResource resource;

    public TimeoutCommand(int timeout, LatentResource resource) {
        super(Setter.withGroupKey(
            HystrixCommandGroupKey.Factory.asKey(COMMAND_GROUP))
            .andCommandPropertiesDefaults(
                HystrixCommandProperties.Setter()
                    .withExecutionIsolationThreadTimeoutInMilliseconds(
                        timeout)));
        this.resource = resource;
    }

    @Override
    protected String run() throws Exception {
        return resource.getData();
    }

    @Override
    protected String getFallback() {
        return "Resource timed out";
    }
}
```

```
@Test
public void shouldGetNormalResponse() {
    LatentResource resource = new LatentResource(50L);
    TimeoutCommand command = new TimeoutCommand(100, resource);
    String s = command.execute();

    assertEquals("Some value", s);
}

@Test
public void shouldGetFallbackResponse() {
    LatentResource resource = new LatentResource(150L);
    TimeoutCommand command = new TimeoutCommand(100, resource);
    String s = command.execute();

    assertEquals("Resource timed out", s);
}
```

Circuit breaker

```
public class CircuitBreakerCommand extends HystrixCommand<String> {  
    private static final String COMMAND_GROUP = "default";  
  
    public CircuitBreakerCommand(String name, boolean open) {  
        super(Setter.withGroupKey(  
            HystrixCommandGroupKey.Factory.asKey(COMMAND_GROUP))  
            .andCommandKey(HystrixCommandKey.Factory.asKey(name))  
            .andCommandPropertiesDefaults(  
                HystrixCommandProperties.Setter()  
                .withCircuitBreakerForceOpen(open)));  
    }  
  
    @Override  
    protected String run() throws Exception {  
        return("Some result");  
    }  
  
    @Override  
    protected String getFallback() {  
        return "Fallback response";  
    }  
}
```

```
@Test
public void shouldExecuteWithCircuitBreakerClosed() {
    CircuitBreakerCommand command = new
        CircuitBreakerCommand("ClosedBreaker", false);

    String s = command.execute();
    assertEquals("Some result", s);

    HystrixCircuitBreaker breaker =
        HystrixCircuitBreaker.Factory
            .getInstance(command.getCommandKey());

    assertTrue(breaker.allowRequest());
}
```

```
@Test
public void shouldExecuteWithCircuitBreakerOpen() {
    CircuitBreakerCommand command = new
        CircuitBreakerCommand("OpenBreaker", true);

    String s = command.execute();
    assertEquals("Fallback response", s);

    HystrixCircuitBreaker breaker =
        HystrixCircuitBreaker.Factory
            .getInstance(command.getCommandKey());

    assertFalse(breaker.allowRequest());
}
```

Load shedder (Thread pool)

```
public class LatentCommand extends HystrixCommand<String> {
    private static final String COMMAND_GROUP = "default";

    private final LatentResource resource;

    public LatentCommand(LatentResource resource) {
        super(HystrixCommandGroupKey.Factory.asKey(COMMAND_GROUP));
        this.resource = resource;
    }

    @Override
    protected String run() throws Exception {
        return resource.getData();
    }

    @Override
    protected String getFallback() {
        return "Fallback triggered";
    }
}
```

```
@Test
public void shouldShedLoad() {
    List<Future<String>> l = new LinkedList<>();
    LatentResource resource = new LatentResource(500L); // Make latent

    // Use up all available threads
    for (int i = 0; i < 10; i++)
        l.add(new LatentCommand(resource).queue());

    // Call will be rejected as thread pool is exhausted
    String s = new LatentCommand(resource).execute();
    assertEquals("Fallback triggered", s);

    // All other calls succeed
    for (Future<String> f : l)
        assertEquals("Some value", get(f)); // wrapper for f.get()
}
```

Load shedder (Semaphore)

```
public class SemaphoreCommand extends HystrixCommand<String> {
    private static final String COMMAND_GROUP = "default";
    private final LatentResource resource;

    public SemaphoreCommand(LatentResource resource) {
        super(Setter.withGroupKey(
            HystrixCommandGroupKey.Factory.asKey(COMMAND_GROUP))
            .andCommandPropertiesDefaults(
                HystrixCommandProperties.Setter()
                    .withExecutionIsolationStrategy(HystrixCommandProperties
                        .ExecutionIsolationStrategy.SEMAPHORE)));
        this.resource = resource;
    }

    @Override
    protected String run() throws Exception {
        return resource.getData();
    }

    @Override
    protected String getFallback() {
        return "Fallback triggered";
    }
}
```

```
private class SemaphoreCommandInvocation implements
        java.util.concurrent.Callable<String> {
    private final LatentResource resource;

    public SemaphoreCommandInvocation(LatentResource resource) {
        this.resource = resource;
    }

    @Override
    public String call() throws Exception {
        return new SemaphoreCommand(resource).execute();
    }
}
```

```
@Test
public void shouldShedLoad() {
    List<Future<String>> l = new LinkedList<>();
    LatentResource resource = new LatentResource(500L); // Make latent

    ExecutorService e = Executors.newFixedThreadPool(10);

    // Use up all available semaphores
    for (int i = 0; i < 10; i++)
        l.add(e.submit(new SemaphoreCommandInvocation(resource)));

    // Wait a moment to make sure all commands are started
    pause(250L); // Wrapper around Thread.sleep()

    // Call will be rejected as all semaphores are in use
    String s = new SemaphoreCommand(resource).execute();
    assertEquals("Fallback triggered", s);

    // All other calls succeed
    for (Future<String> f : l)
        assertEquals("Some value", get(f));
}
```

Fail fast

```
public class FailFastCommand extends HystrixCommand<String> {
    private static final String COMMAND_GROUP = "default";

    private final boolean preCondition;

    public FailFastCommand(boolean preCondition) {
        super(HystrixCommandGroupKey.Factory.asKey(COMMAND_GROUP));
        this.preCondition = preCondition;
    }

    @Override
    protected String run() throws Exception {
        if (!preCondition)
            throw new RuntimeException("Fail fast");
        return "Some value";
    }
}
```

```
@Test
public void shouldSucceed() {
    FailFastCommand command = new FailFastCommand(true);
    String s = command.execute();

    assertEquals("Some value", s);
}

@Test
public void shouldFailFast() {
    FailFastCommand command = new FailFastCommand(false);
    try {
        String s = command.execute();
        fail("Did not fail fast");
    } catch (Exception e) {
        assertEquals(HystrixRuntimeException.class, e.getClass());
        assertEquals("Fail fast", e.getCause().getMessage());
    }
}
```

Fail silent

```
public class FailSilentCommand extends HystrixCommand<String> {
    private static final String COMMAND_GROUP = "default";

    private final boolean preCondition;

    public FailSilentCommand(boolean preCondition) {
        super(HystrixCommandGroupKey.Factory.asKey(COMMAND_GROUP));
        this.preCondition = preCondition;
    }

    @Override
    protected String run() throws Exception {
        if (!preCondition)
            throw new RuntimeException("Fail fast");
        return "Some value";
    }

    @Override
    protected String getFallback() {
        return null; // Turn into silent failure
    }
}
```

```
@Test
public void shouldSucceed() {
    FailSilentCommand command = new FailSilentCommand(true);
    String s = command.execute();

    assertEquals("Some value", s);
}

@Test
public void shouldFailSilent() {
    FailSilentCommand command = new FailSilentCommand(false);
    String s = "Test value";
    try {
        s = command.execute();
    } catch (Exception e) {
        fail("Did not fail silent");
    }
    assertNull(s);
}
```

Static fallback

```
public class StaticFallbackCommand extends HystrixCommand<String> {
    private static final String COMMAND_GROUP = "default";

    private final boolean preCondition;

    public StaticFallbackCommand(boolean preCondition) {
        super(HystrixCommandGroupKey.Factory.asKey(COMMAND_GROUP));
        this.preCondition = preCondition;
    }

    @Override
    protected String run() throws Exception {
        if (!preCondition)
            throw new RuntimeException("Fail fast");
        return "Some value";
    }

    @Override
    protected String getFallback() {
        return "Some static fallback value"; // Static fallback
    }
}
```

```
@Test
public void shouldSucceed() {
    StaticFallbackCommand command = new StaticFallbackCommand(true);
    String s = command.execute();

    assertEquals("Some value", s);
}

@Test
public void shouldProvideStaticFallback() {
    StaticFallbackCommand command = new StaticFallbackCommand(false);
    String s = null;
    try {
        s = command.execute();
    } catch (Exception e) {
        fail("Did not fail silent");
    }
    assertEquals("Some static fallback value", s);
}
```

Cache fallback

```
public class CacheClient {  
    private final Map<String, String> map;  
  
    public CacheClient() {  
        map = new ConcurrentHashMap<>();  
    }  
  
    public void add(String key, String value) {  
        map.put(key, value);  
    }  
  
    public String get(String key) {  
        return map.get(key);  
    }  
}
```

```
public class CacheCommand extends HystrixCommand<String> {  
    private static final String COMMAND_GROUP = "primary";  
    private final CacheClient cache;  
    private final boolean failPrimary;  
    private final String req;  
  
    public CacheCommand(CacheClient cache, boolean failPrimary,  
                        String request) {  
        super(HystrixCommandGroupKey.Factory.asKey(COMMAND_GROUP));  
        this.cache = cache;  
        this.failPrimary = failPrimary;  
        this.req = request;  
    }  
  
    @Override  
    protected String run() throws Exception {  
        if (failPrimary)  
            throw new RuntimeException("Failure of primary");  
        String s = req + "-o";  
        cache.add(req, s);  
        return(s);  
    }  
...  
}
```

```
...

@Override
protected String getFallback() {
    return "Cached: " + new FBCommand(cache, req).execute();
}

private static class FBCommand extends HystrixCommand<String> {
    private static final String COMMAND_GROUP = "fallback";
    private final CacheClient cache;
    private final String request;

    public FBCommand(CacheClient cache, String request) {
        super(HystrixCommandGroupKey.Factory.asKey(COMMAND_GROUP));
        this.cache = cache;
        this.request = request;
    }

    @Override
    protected String run() throws Exception {
        return cache.get(request);
    }
}
```

```
@Test
public void shouldExecuteWithoutCache() {
    CacheClient cache = new CacheClient();
    String s = new CacheCommand(cache, false, "ping").execute();
    assertEquals("ping-o", s);
}

@Test
public void shouldRetrieveValueFromCache() {
    CacheClient cache = new CacheClient();
    new CacheCommand(cache, false, "ping").execute();
    String s = new CacheCommand(cache, true, "ping").execute();
    assertEquals("Cached: ping-o", s);
}

@Test
public void shouldNotRetrieveAnyValue() {
    CacheClient cache = new CacheClient();
    String s = new CacheCommand(cache, true, "ping").execute();
    assertEquals("Cached: null", s);
}
```

... and so on

Advanced features



Advanced Features

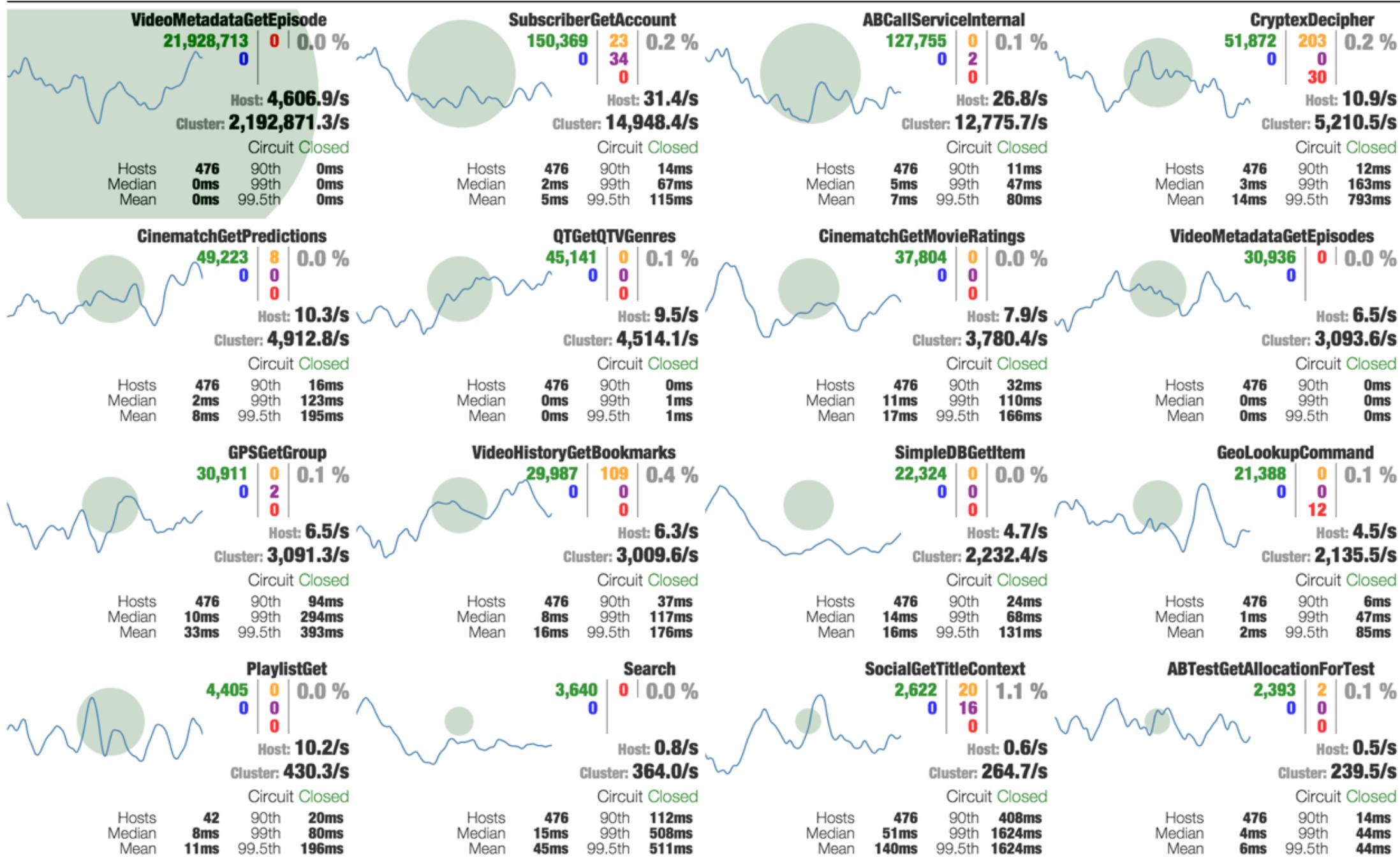
- Request Caching
- Request Collapsing
- Plugins
 - Event Notifier, Metrics Publisher, Properties Strategy, Concurrency Strategy, Command Execution Hook
- Contributions
 - Metrics Event Stream, Metrics Publisher, Java Annotations, Clojure Bindings, ...
- Dashboard



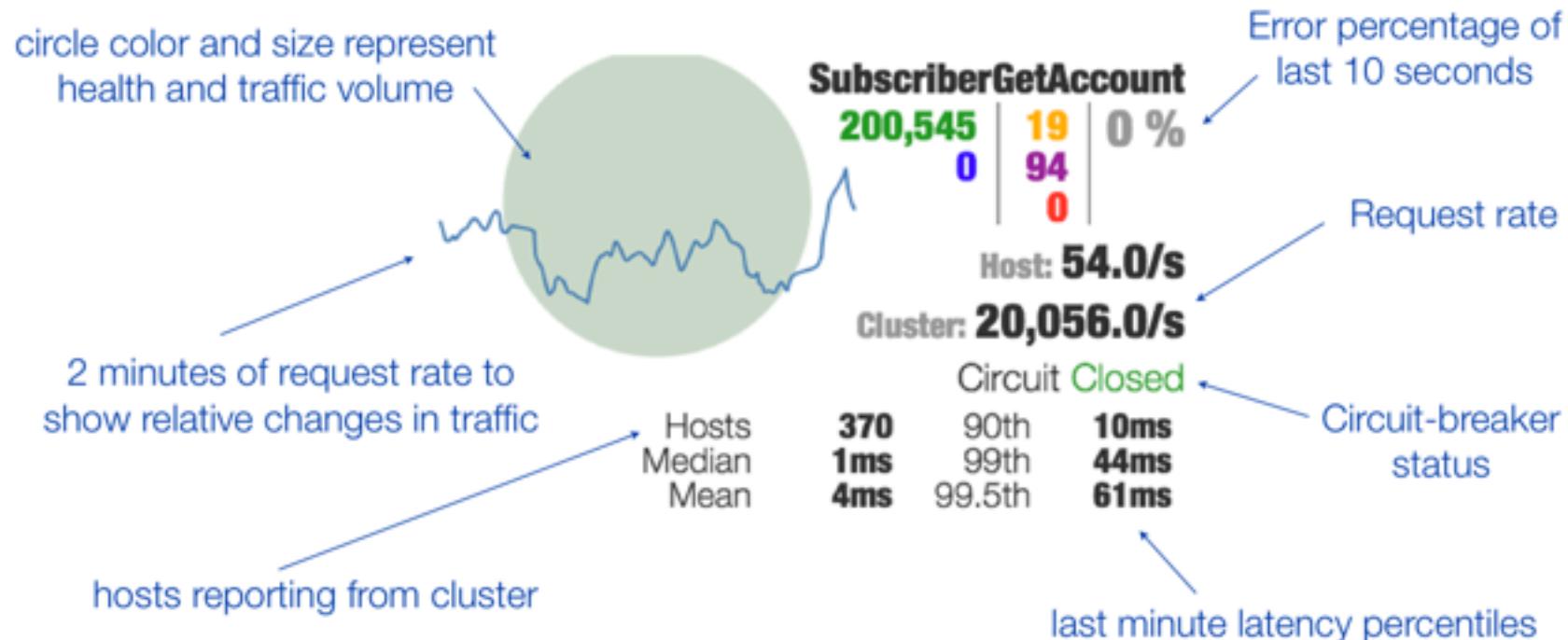
Circuit Breakers

Sort: [Error then Volume](#) | [Alphabetical](#) | [Volume](#) | [Error](#) | [Mean](#) | [Median](#) | [90](#) | [99](#) | [99.5](#)

[Success](#) | [Latent](#) | [Short-Circuited](#) | [Timeout](#) | [Rejected](#) | [Failure](#) | Error %



Source: <https://github.com/Netflix/Hystrix/wiki/Dashboard>

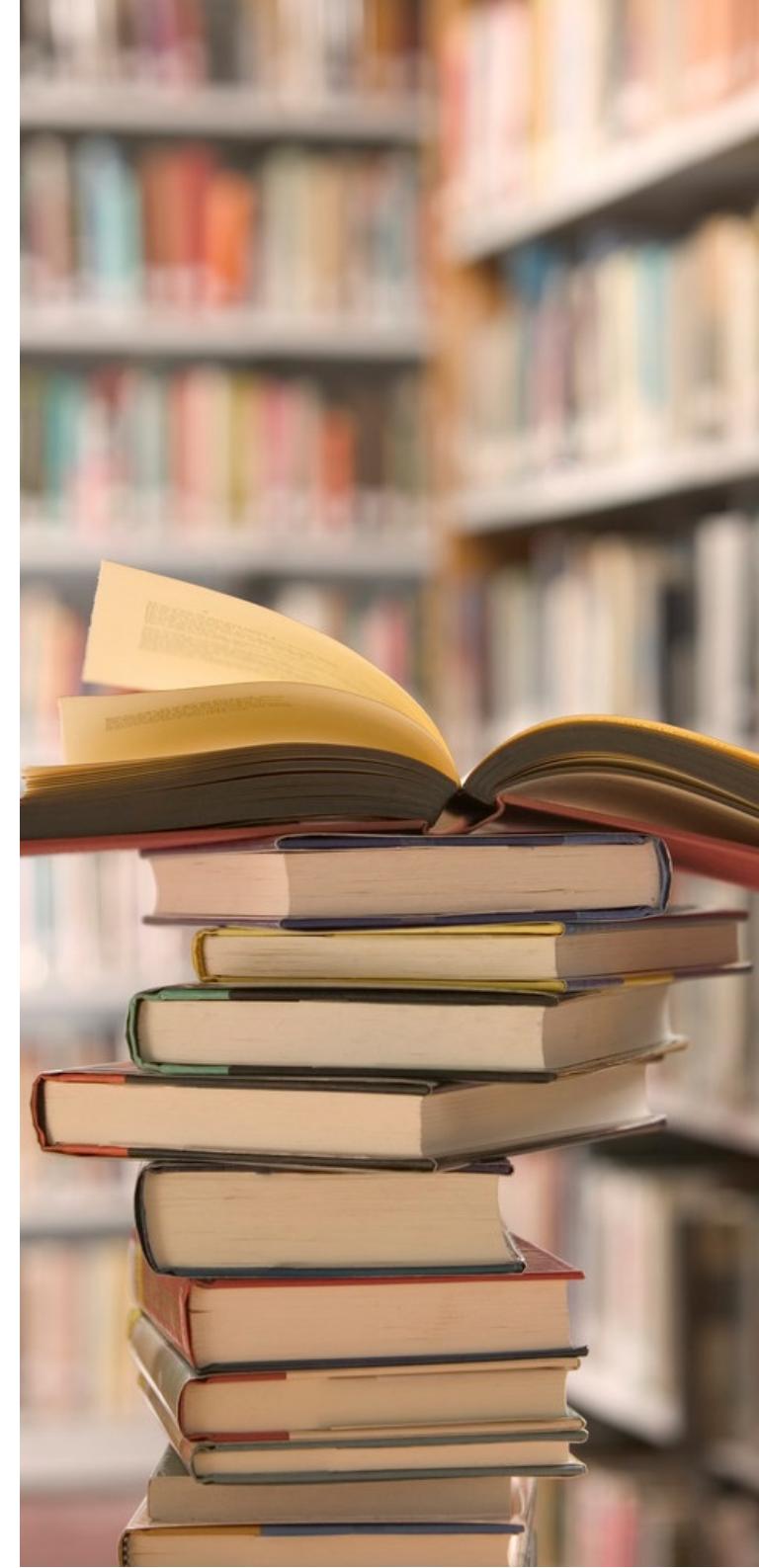


Rolling 10 second counters
with 1 second granularity

Successes	200,545	Thread timeouts
Short-circuited (rejected)	0	Thread-pool Rejections
		Failures/Exceptions

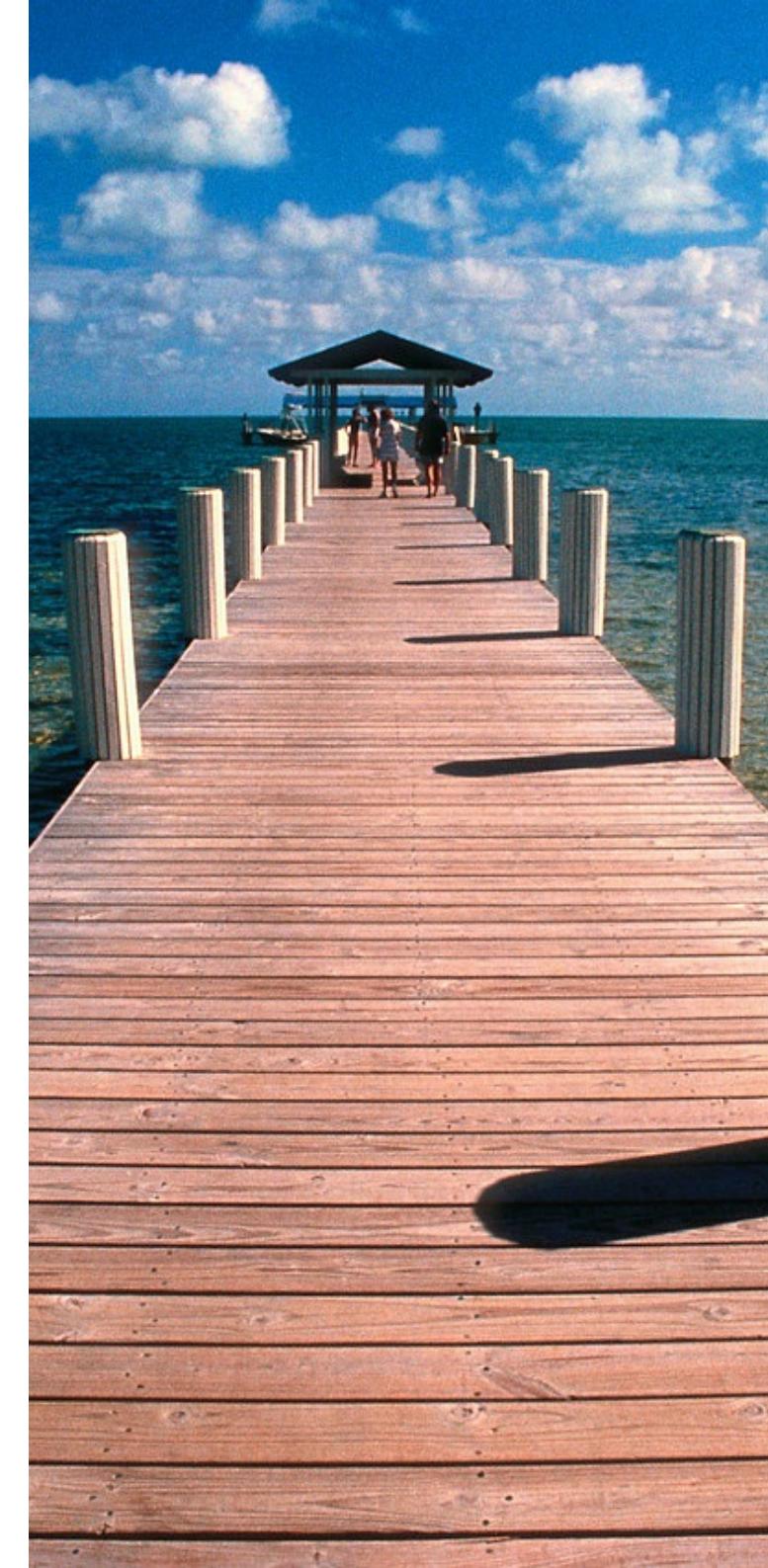
Further reading

1. Hystrix Wiki,
<https://github.com/Netflix/Hystrix/wiki>
2. Michael T. Nygard, Release It!,
Pragmatic Bookshelf, 2007
3. Robert S. Hanmer,
Patterns for Fault Tolerant Software,
Wiley, 2007
4. Andrew Tanenbaum, Marten van Steen,
Distributed Systems – Principles and
Paradigms,
Prentice Hall, 2nd Edition, 2006



Wrap-up

- Resilient software design becomes a must
- Hystrix is a resilience library
 - Provides isolation and latency control
 - Easy to start with
 - Yet some learning curve
 - Many fault-tolerance patterns implementable
 - Many configuration options
 - Customizable
 - Operations proven



Become a resilient software developer!



It's all about production!

@ufried



