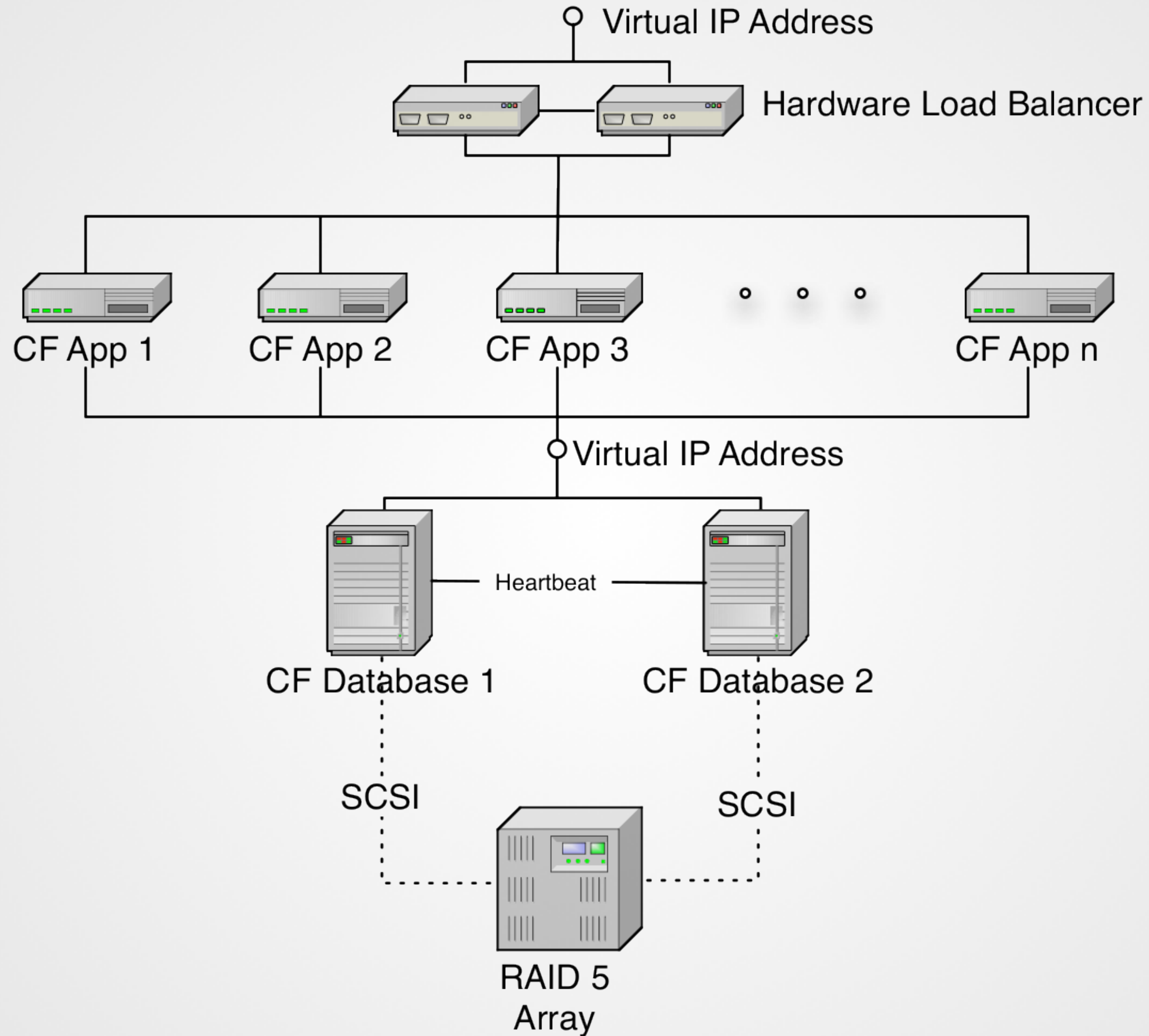
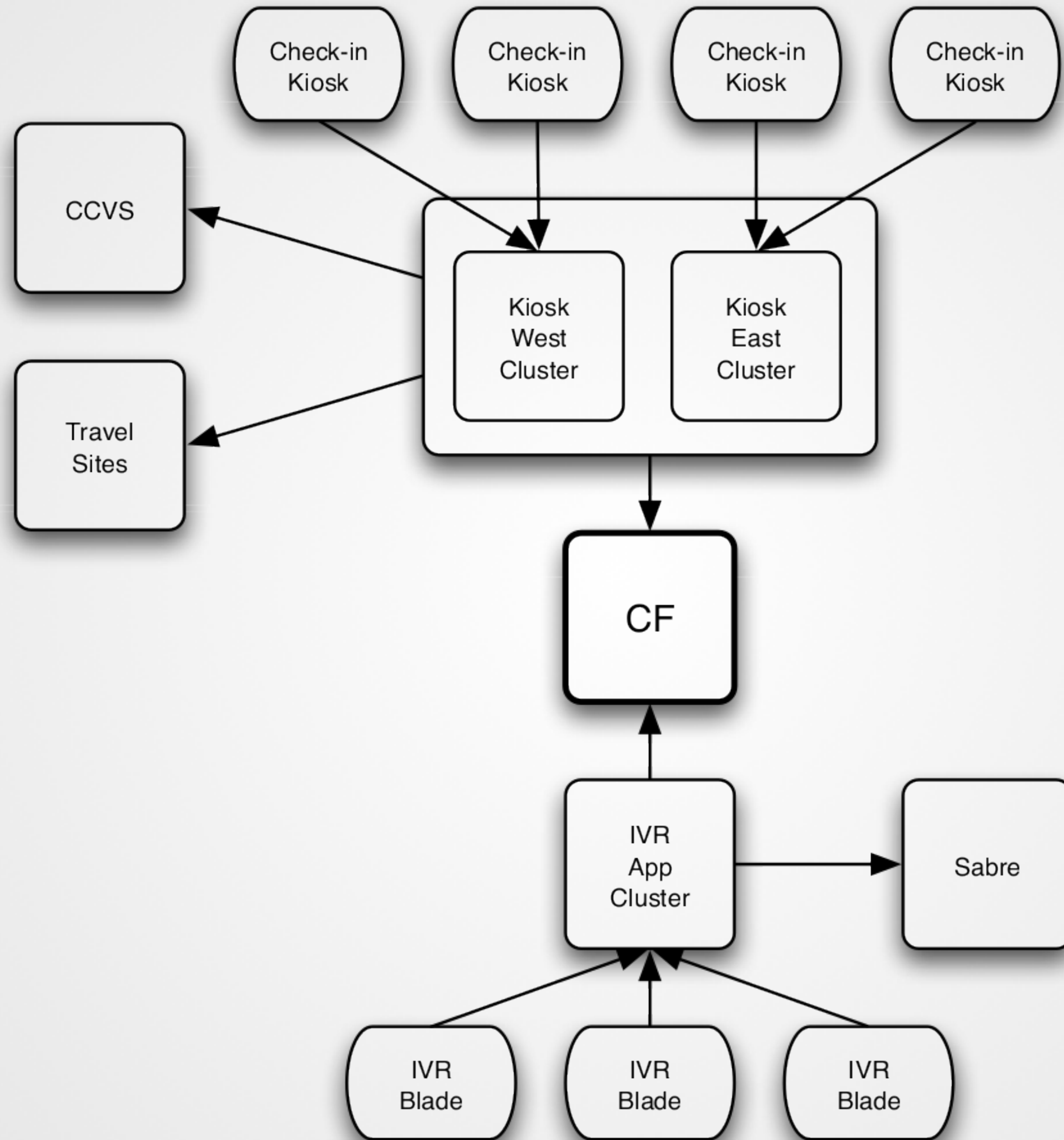
Fault Tolerance with Hystrix

Tomáš Livora

The Exception That Grounded an Airline


Virtual IP Address
Hardware Load Balancer
CF App 1
CF App 2
CF App 3
CF App n
Virtual IP Address
Heartbeat
CF Database 1
CF Database 2
SCSI
SCSI
RAID 5
Array



IVR = Interactive Voice Response

```java
public class FlightSearch implements SessionBean {

    private MonitoredDataSource connectionPool;

    public List lookupByCity(...) throws SQLException, RemoteException {
        Connection conn = null;
        Statement stmt = null;
        try {
            conn = connectionPool.getConnection();
            stmt = conn.createStatement();
            // do the lookup logic and return a list of results
        } finally {
            if (stmt != null) {
                stmt.close();
            }
            if (conn != null) {
                conn.close();
            }
        }
    }
}
```

What should they have done better?

Problems

- **denying the inevitability of failures**
- connections waiting forever
  - no timeouts
- tightly coupled components
  - direct dependencies between services
  - synchronous blocking calls
- cascading failures
  - error propagation throughout various systems
- no safe failure modes

Stability Patterns

Stability Patterns

- Timeouts
- Circuit Breaker
- Bulkheads
- Fail Fast
- Steady State
- Handshaking
- Test Harness
- Decoupling Middleware

The Pragmatic Programmers

Release It!

Design and Deploy Production-Ready Software

Michael T. Nygard

Timeouts

- avoid waiting for a response forever
- should always be set for remote calls
  - usually used at lower levels (operating systems)
  - ignored at higher levels (libraries, applications)
- need to be set carefully
  - waiting too long slows down the whole system
  - timing out too quickly may ignore some responses

Circuit Breaker

- similar to circuit breakers in electric circuits
  - detecting excess usage and failing first
- wraps dangerous calls and protects the system
- switching between different states
  - closed
  - open
  - half-open
- prevents cascading failures
- works closely with timeouts
- valuable place for monitoring

Bulkheads

- partitions that divide the inside of a ship into separate areas
  - a single penetration of the hull does not sink the ship
- similar technique used in software systems
  - keep a failure in one component from affecting other components
  - protect against bringing down the whole system
- using separate connection pools for different remote services
  - exhaustion of one pool do not affect other services

Fail Fast

- waiting for failure is a waste of time
- detect a potential failure in advance
  - improves stability by avoiding slow responses
  - helps to maintain capacity under heavy load
- check all necessary resources before the execution
  - check all connections
  - verify the states of circuit breakers
- check input parameters as soon as possible
- distinguish between system failures and application failures
  - trip or do not trip the circuit breaker

Fault Tolerance Libraries

Fault Tolerance Libraries

- JRugged
- Failsafe
- Resilience4j
- **Hystrix**

JRugged

- a Java library of robustness design patterns
  - https://github.com/Comcast/jrugged
- provides three mechanisms
  - initializers
  - circuit breakers
  - performance monitors

```
CircuitBreaker circuitBreaker = new CircuitBreaker();
circuitBreaker.invoke(() -> service.call());
```

Failsafe

- a lightweight, zero-dependency library for handling failures
  - https://github.com/jhalterman/failsafe
- fault tolerance mechanisms
  - timeouts
  - circuit breakers
  - fallbacks
- other features
  - retries
  - event listeners

```java
CircuitBreaker circuitBreaker = new CircuitBreaker()
    .withFailureThreshold(3, 10)
    .withSuccessThreshold(5)
    .withDelay(1, TimeUnit.MINUTES);
Failsafe.with(circuitBreaker).run(() -> remoteService.call());
```

```java
RetryPolicy retryPolicy = new RetryPolicy()
    .retryOn(ConnectException.class)
    .withDelay(1, TimeUnit.SECONDS)
    .withMaxRetries(3);
Failsafe.with(retryPolicy).run(() -> remoteService.call());
```

```java
Failsafe.with(retryPolicy)
    .withFallback(this::callFallback)
    .get(() -> remoteService.call());
```

Resilience4j

- a lightweight fault tolerance library for Java 8 and functional programming
  - https://github.com/resilience4j/resilience4j
- based on Vavr (formerly Javaslang) and RxJava
- many different mechanisms
  - circuit breaker, fallback, bulkheads
  - rate limiter, automatic retrying, response caching
  - metrics monitoring
- annotation-based configuration possible (AOP)

```java
// Create a CircuitBreaker with a default configuration
CircuitBreaker circuitBreaker = CircuitBreaker.ofDefaults("backendName");

// Create a Retry with 3 retries and 500ms interval between retries
Retry retryContext = Retry.ofDefaults("backendName");

// Decorate your call to BackendService.doSomething()
Try.CheckedSupplier<String> decoratedSupplier = Decorators
    .ofCheckedSupplier(() -> backendService.doSomething())
    .withCircuitBreaker(circuitBreaker)
    .withRetry(retryContext)
    .decorate();

// Invoke the decorated function and recover from any exception
Try<String> result = Try.of(decoratedSupplier)
    .recover(throwable -> "Hello from Recovery");
```

Hystrix

Hystrix

- the most popular fault tolerance library
- developed by Netflix
- provides various mechanisms
  - timeouts
  - circuit breakers, fallbacks
  - isolation by thread pools
  - request caching and collapsing
- annotation-based configuration possible (AOP)
- provides monitoring capabilities (Hystrix Dashboard)

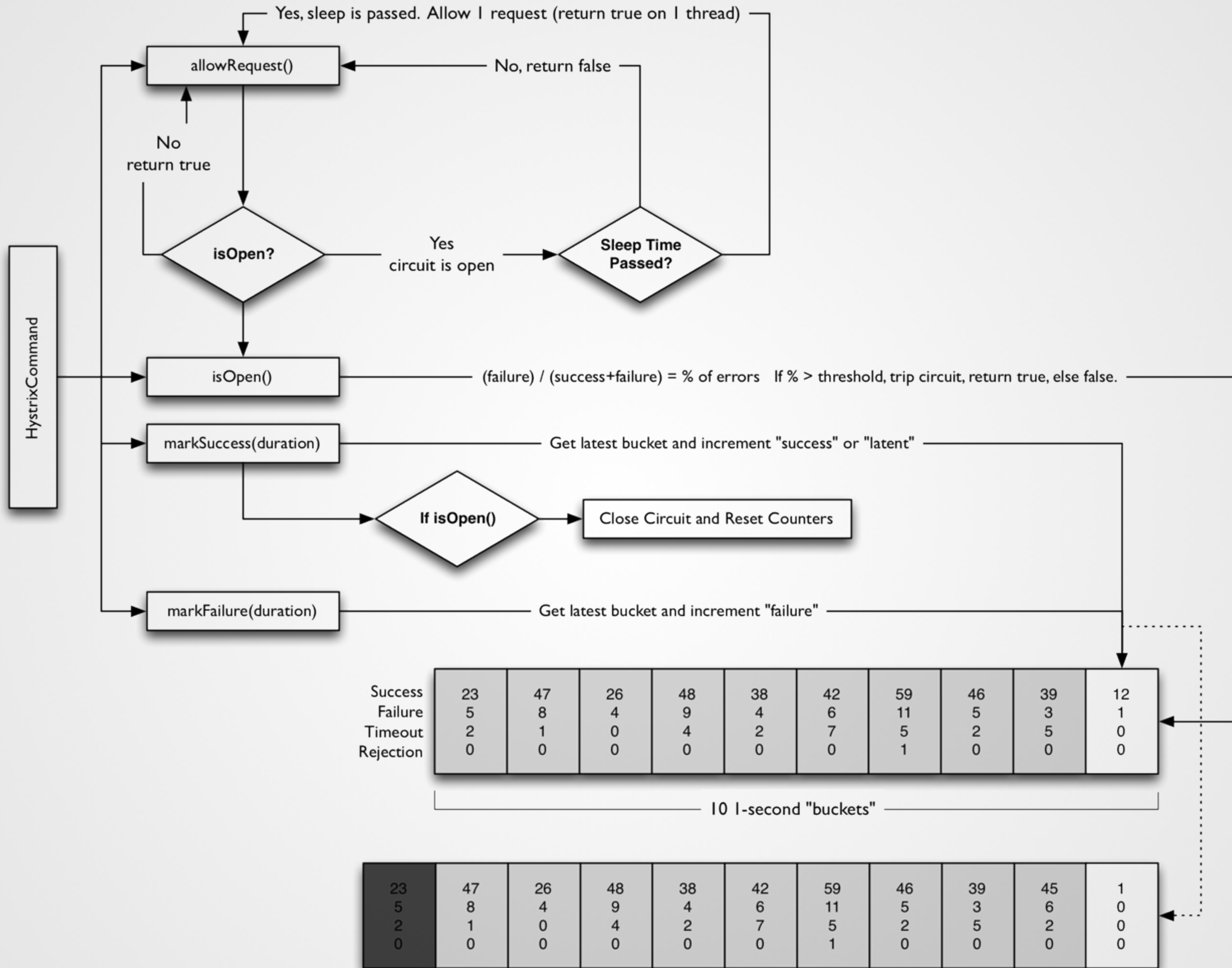
Hystrix Command

- wraps a single remote service method
  - need to provide different implementation for each method
- based on command design pattern
  - extend *HystrixCommand* abstract class
  - perform remote service call in *run()* method
  - execute by calling *execute()* method on an instance
- provides a large set of configuration options
  - command group, command name...
- allows fallback method implementation

```java
public class GetUserCommand extends HystrixCommand<User> {

    private static final UserServiceClient userServiceClient =
        new UserServiceClient();

    private final String userName;

    public GetUserCommand(String userName) {
        super(HystrixCommandGroupKey.Factory.asKey("UserService"));
        this.userName = userName;
    }

    @Override
    protected User run() {
        return userServiceClient.getUser(userName);
    }
}

...

User john = new GetUserCommand("john").execute();
```

Circuit Breaker

- starts in ***closed*** state and makes remote calls as usual
- when an error occurs
  - record a failure
  - execute a fallback method (if provided)
- when an error rate exceeds the defined threshold
  - move to ***open*** state and stop executing remote calls
  - wait in this state during the specified sleep window
- after the sleep window elapses
  - switch to ***half-open*** state and a single request is tried
  - if it succeeds, move to *closed* state, otherwise move to *open* state


Yes, sleep is passed. Allow 1 request (return true on 1 thread)
allowRequest()
No, return false
No
return true
isOpen?
Yes
circuit is open
Sleep Time
Passed?
HystrixCommand
isOpen()
(failure) / (success+failure) = % of errors If % > threshold, trip circuit, return true, else false.
markSuccess(duration)
Get latest bucket and increment "success" or "latent"
If isOpen()
Close Circuit and Reset Counters
markFailure(duration)
Get latest bucket and increment "failure"
Success
Failure
Timeout
Rejection
23 5 2 0
47 8 1 0
26 4 0 0
48 9 4 0
38 4 2 0
42 6 7 0
59 11 5 1
46 5 2 0
39 3 5 0
12 1 0 0
10 1-second "buckets"
23 5 2 0
47 8 1 0
26 4 0 0
48 9 4 0
38 4 2 0
42 6 7 0
59 11 5 1
46 5 2 0
39 3 5 0
45 6 2 0
1 0 0 0
On "getLatestBucket" if the 1-second window is passed a new bucket is created, the rest slid over and the oldest one dropped.

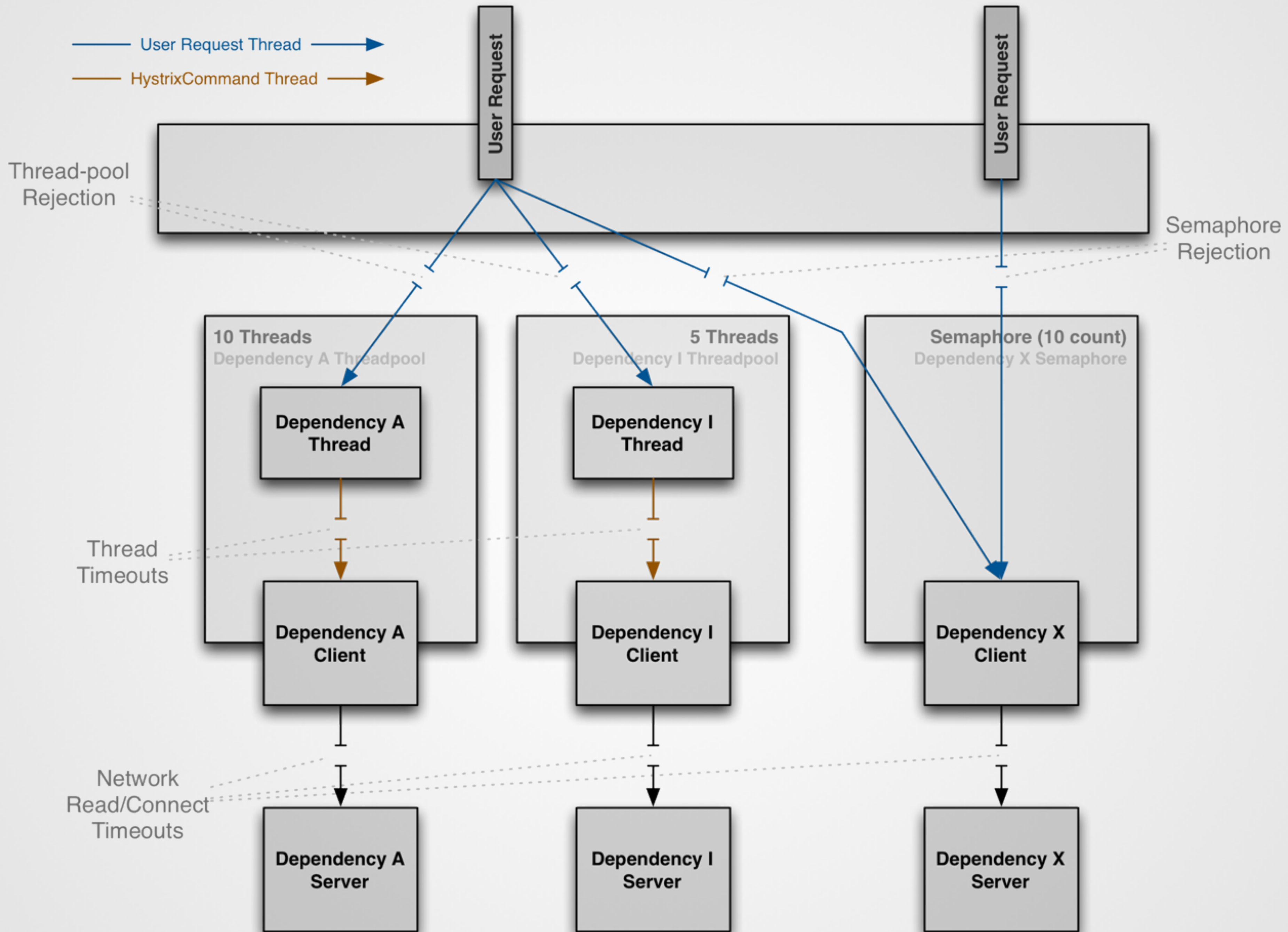
Fallbacks

- support graceful degradation
  - return a default value in case the main command fails
  - circuit breakers still count this as a failure
- should not call any remote service directly
  - another Hystrix command need to be used
- not suitable in some cases
  - a command that performs a write operation
  - batch systems/offline computation
- need to override *getFallback()* method from *HystrixCommand* class

```java
public class CommandHelloFailure extends HystrixCommand<String> {

    private final String name;

    public CommandHelloFailure(String name) {
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        this.name = name;
    }

    @Override
    protected String run() {
        throw new RuntimeException("this command always fails");
    }

    @Override
    protected String getFallback() {
        return "Hello Failure " + name + "!";
    }
}
```

Isolation

- bulkheads pattern implementation
- semaphores
  - limit the number of concurrent calls to any given dependency
  - no timing out options
- thread pools (default)
  - isolate dependencies from each other
    - thread-pool per command group by default
    - configurable for each command
  - additional computational overhead



User Request Thread
HystrixCommand Thread
User Request
User Request
Thread-pool Rejection
Semaphore Rejection
10 Threads
Dependency A Threadpool
5 Threads
Dependency I Threadpool
Semaphore (10 count)
Dependency X Semaphore
Dependency A Thread
Dependency I Thread
Thread Timeouts
Dependency A Client
Dependency I Client
Dependency X Client
Network Read/Connect Timeouts
Dependency A Server
Dependency I Server
Dependency X Server

Request Collapsing

- collapsing multiple requests within a short time window
  - a single back-end dependency call
- reduction of the number of threads and network connections
  - suitable in case of high number of concurrent requests
- global or user context collapsing
- latency before the actual command is executed

Without Collapsing: Request == Thread == Network Connection

API Request
API Request
API Request
API Request
API Request

new Command().execute()
new Command().queue()
new Command().execute()
new Command().execute()
new Command().queue()

Hystrix Isolation Threadpool
10 Threads

Dependency
REST API

With Collapsing: Requests within 'window' == 1 Thread == 1 Network Connection

API Request
API Request
API Request
API Request
API Request

new Command().execute()
new Command().queue()
new Command().execute()
new Command().execute()
new Command().queue()

Collapser

Hystrix Isolation Threadpool
10 Threads

Dependency
REST API

Request Caching

- eliminates duplicate thread executions
  - within a single request context
- data retrieval is consistent throughout the request
  - underlying *run()* method executed only once
  - all executing threads will received the same data
- executions matched based on a cache key
  - need to implement *getCacheKey()* method
  - returned *null* means "do not cache" (default)

```java
public class CommandUsingRequestCache extends HystrixCommand<Boolean> {

    private final int value;

    protected CommandUsingRequestCache(int value) {
        super(HystrixCommandGroupKey.Factory.asKey("ExampleGroup"));
        this.value = value;
    }

    @Override
    protected Boolean run() {
        return value == 0 || value % 2 == 0;
    }

    @Override
    protected String getCacheKey() {
        return String.valueOf(value);
    }
}
```

Hystrix Javanica

- Hystrix configuration using Java annotations
  - *@HystrixCommand*
  - *@HystrixProperty*
  - *@DefaultProperties*
  - *@HystrixCollapser*
  - *@CacheResult*
  - *@CacheRemove*
  - *@CacheKey*
- need to use AspectJ (AOP) in your project

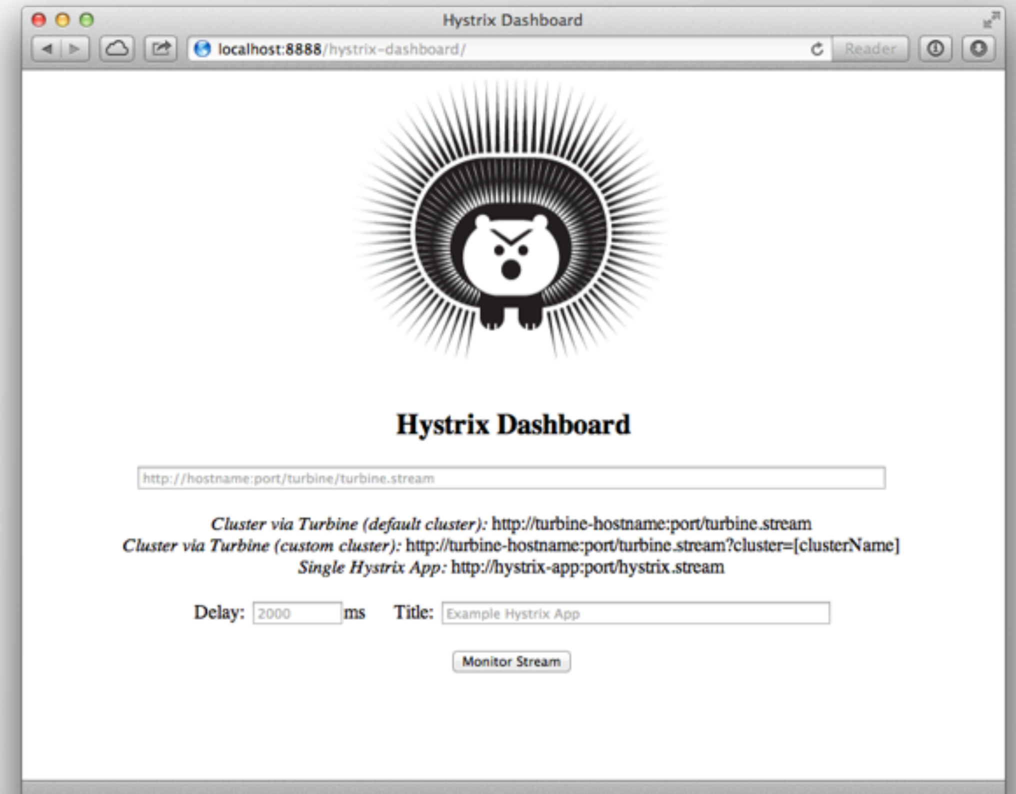
```java
@DefaultProperties(groupKey = "UserService")
class UserServiceClient {

    @HystrixCommand(
        commandProperties = {
            @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "40")
            @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value = "3500")
        },
        threadPoolProperties = {
            @HystrixProperty(name = "coreSize", value = "30")
        }
    )
    public User getUserById(Integer id) {
        return target.path("users/{id}").resolveTemplate("id", id).request().get(User.class);
    }

    @HystrixCommand(fallbackMethod = "getActiveUsersFallback")
    public List<User> getActiveUsers() {
        return target.path("users/active").request().get(List.class);
    }

    public List<User> getActiveUsersFallback() {
        return Collections.emptyList();
    }
}
```

Metrics and Monitoring

- commands generate metrics on execution outcomes and latency
  - modeled as a first-class stream
  - written to in-memory data structures
- published using REST API
  - need to deploy *HystrixMetricsStreamServlet*
  - can be consumed by Hystrix Dashboard

Hystrix Dashboard

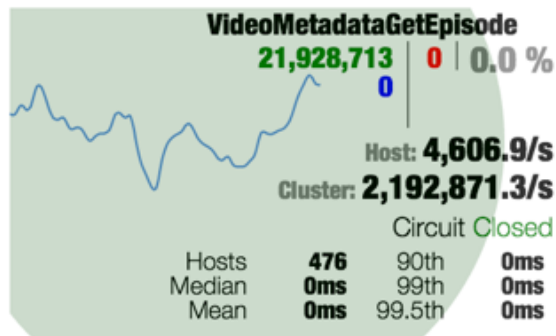
- metrics monitoring in real time
  - single server
  - multiple servers (Turbine)
- finding the cause of problems quickly
- web application
  - WAR file deployable in servlet containers

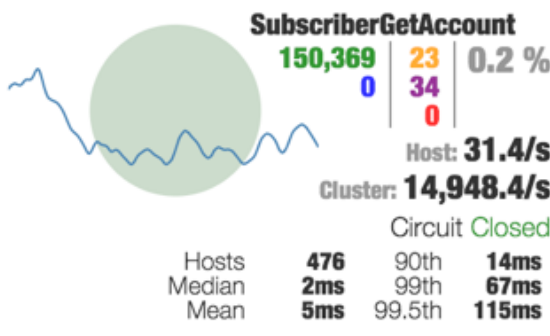


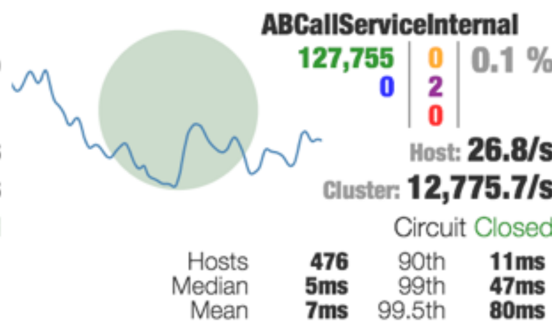
Circuit Breakers

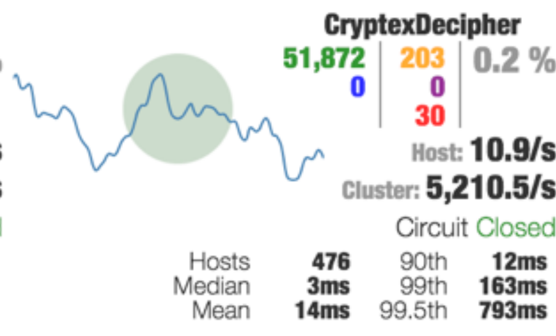
Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5

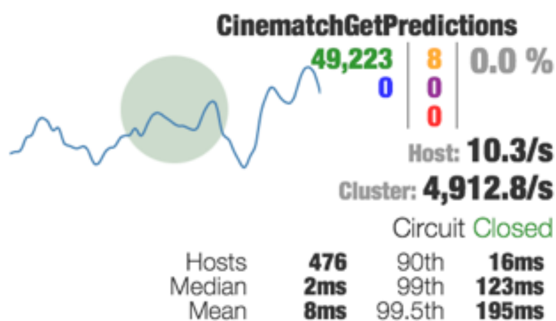
Success | Latent | Short-Circuited | Timeout | Rejected | Failure | Error %

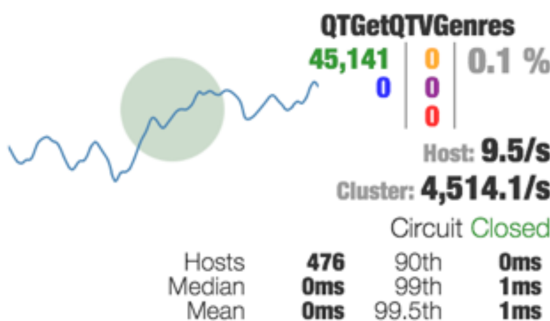


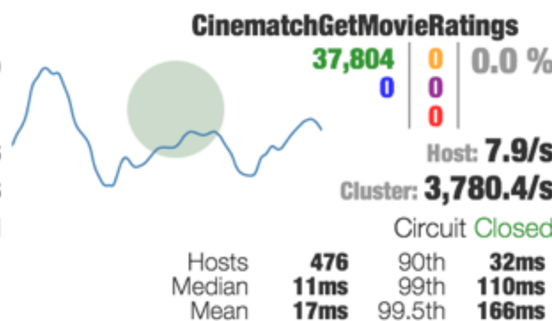


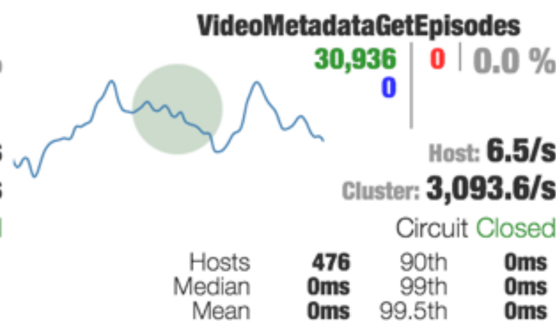


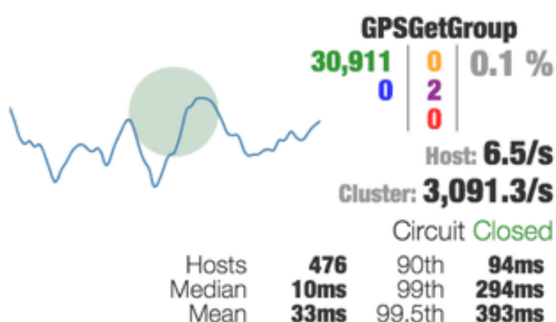


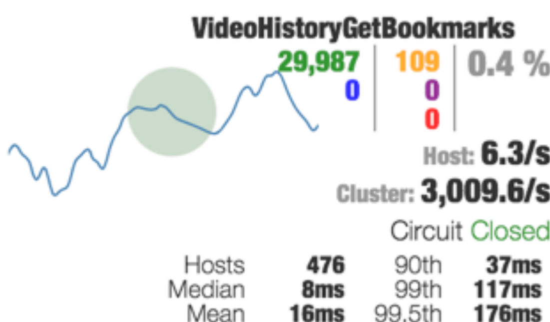


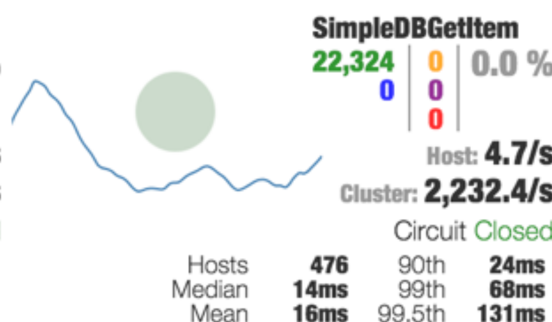


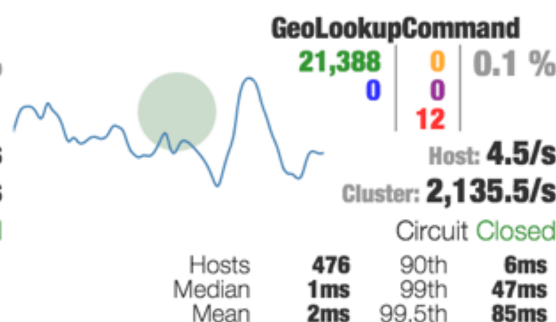


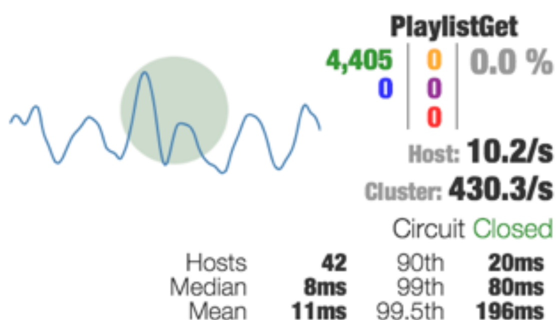


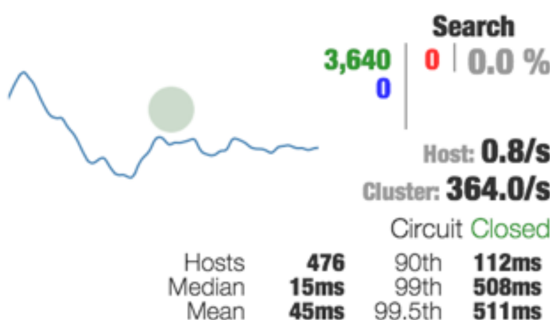


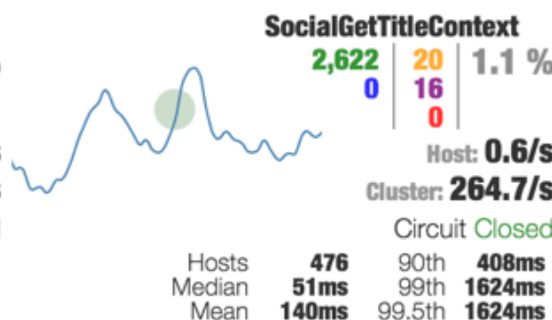


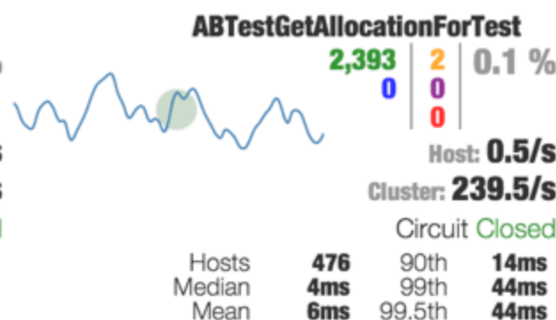












Follow-up

- ***Hystrix workshop***
  - simple assignments teaching the basics of Hystrix
  - https://github.com/livthomas/hystrix-workshop
- ***Six principles for building fault tolerant microservices on the JVM***
  - Devoxx presentation by Christopher Batey
  - https://youtu.be/dKWNZnuZhd0

Sources

- Fallacies of Distributed Computing Explained
  - http://www.rgoarchitects.com/Files/fallacies.pdf
- Release It!: Design and Deploy Production-Ready Software
  - https://pragprog.com/book/mnee/release-it
- Netflix Hystrix GitHub Wiki
  - https://github.com/Netflix/Hystrix/wiki
- Fault Tolerance in Microservices
  - https://is.muni.cz/th/396542/fi_m/?lang=en