Using Hystrix to Build Resilient Distributed Systems

Matt Jacobs
Netflix

HYSTRIX
DEFEND YOUR APP

Who am I?

- Edge Platform team at Netflix
- In a 24/7 Support Role for Netflix production
  - (Not 365/24/7)
- Maintainer of Hystrix OSS
  - github.com/Netflix/Hystrix
  - I get a lot of internal questions too

NETFLIX

- Video on Demand
- 1000s of device types
- 1000s of microservices
- Control Plane completely on AWS
- Competing against products which have high availability (network / cable TV)

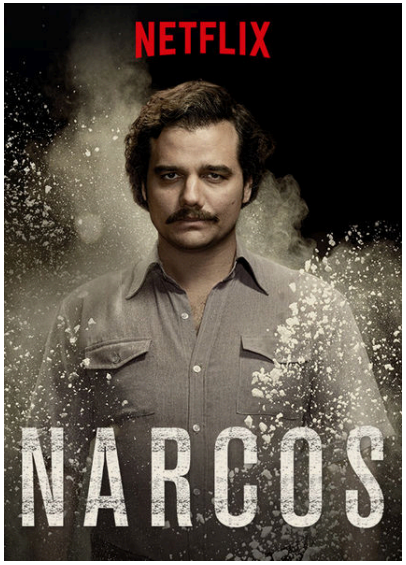
	Upstream		Downstream		Aggregate	
Rank	Application	Share	Application	Share	Application	Share
1	BitTorrent	28.56%	Netflix	37.05%	Netflix	34.70%
2	Netflix	6.78%	YouTube	17.85%	YouTube	16.88%
3	HTTP	5.93%	HTTP	6.06%	HTTP	6.05%
4	Google Cloud	5.30%	Amazon Video	3.11%	BitTorrent	4.35%
5	YouTube	5.21%	iTunes	2.79%	Amazon Video	2.94%
6	SSL - OTHER	5.10%	BitTorrent	2.67%	iTunes	2.62%
7	iCloud	3.08%	Hulu	2.58%	Facebook	2.51%
8	FaceTime	2.55%	Facebook	2.53%	Hulu	2.48%
9	Facebook	2.25%	MPEG - OTHER	2.30%	MPEG	2.16%
10	Dropbox	1.18%	SSL - OTHER	1.73%	SSL - OTHER	1.99%
		65.95%		78.69%		76.68%

sandvine

Source: http://www.sandvine.com/news/global_broadband_trends.asp

NETFLIX

- 2013 – now
  - from 40M to 75M+ paying customers
  - from 40 to 190+ countries (#netflixeverywhere)
  - In original programming, from House of Cards to hundreds of hours a year, in documentaries / movies / comedy specials/ TV series



Edge Platform @ Netflix

- Goal: Maximize availability of Netflix API
  - API used by all customer devices

Edge Platform @ Netflix

- Goal: Maximize availability of Netflix API
  - API used by all customer devices
- How?
  - Fault-tolerant by design
  - Operational visibility
  - Limit manual intervention
  - Understand failure modes before they happen

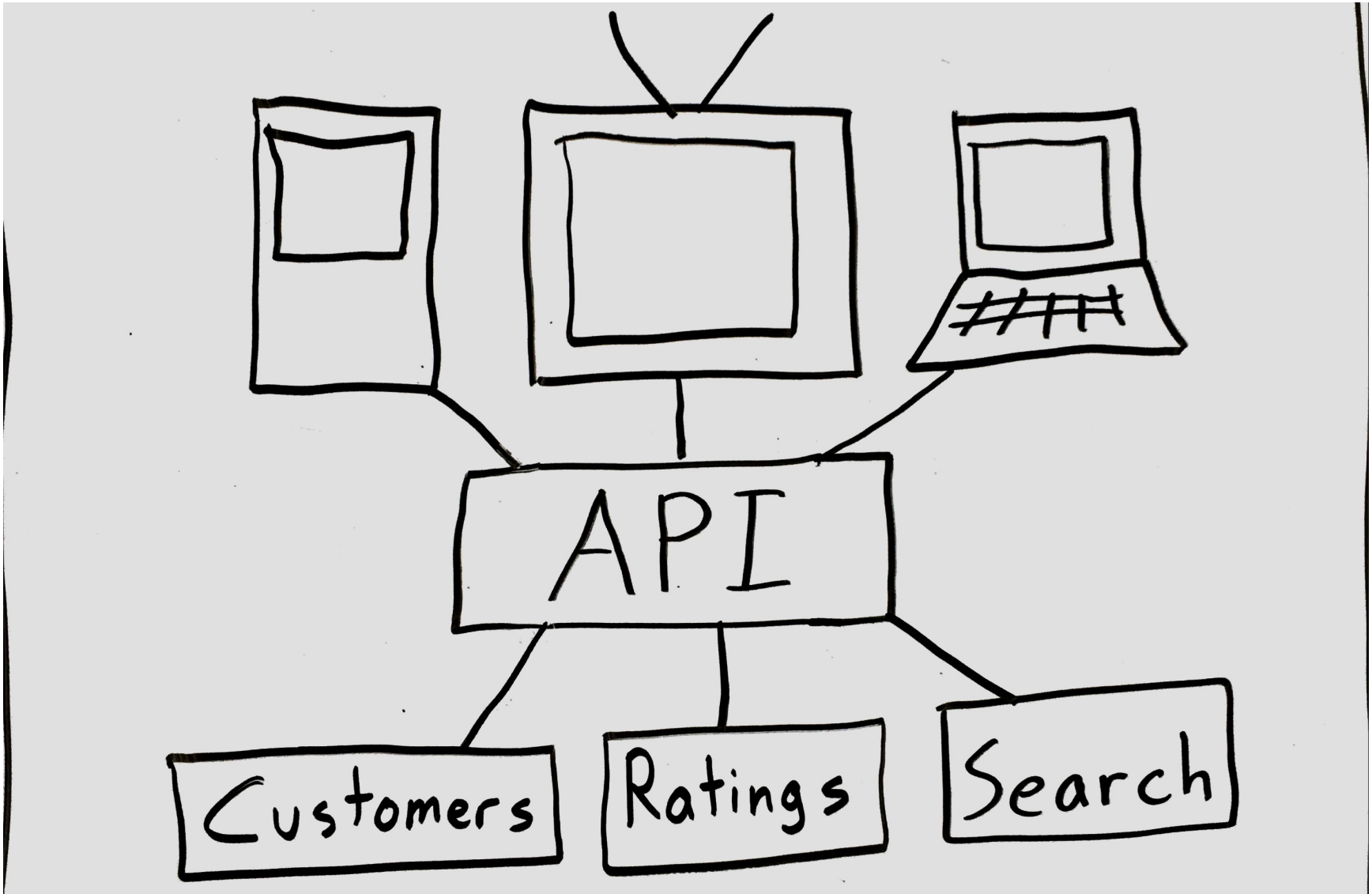
Hystrix @ Netflix

- Fault-tolerance pattern as a library
- Provides operational insights in real-time
- Automatic load-shedding under pressure
- Initial design/implementation by Ben Christensen

Scope of this talk

- User-facing request-response system that uses blocking I/O
- NOT batch system
- NOT nonblocking I/O
- (Hystrix can be used in those, though…)

Netflix API



Goals of this talk

- Philosophical Motivation
  - Why are distributed systems hard?
- Practical Motivation
  - Why do I keep getting paged?
- Solving those problems with Hystrix
  - How does it work?
  - How do I use it in my system?
  - How should a system behave if I use Hystrix?
  - What? Netflix was down for me – what happened there?

Goals of this talk

- Philosophical Motivation
  - Why are distributed systems hard?
- Practical Motivation
  - Why do I keep getting paged?
- Solving those problems with Hystrix
  - How does it work?
  - How do I use it in my system?
  - How should a system behave if I use Hystrix?
  - What? Netflix was down for me – what happened there?

Un-Distributed Systems

- Write an application and deploy!

Distributed Systems

- Write an application and deploy!
  - Add a redundant system for resiliency (no SPOF)
  - Break out the state
  - Break out the function (microservices)
  - Add more machines to scale horizontally

The Eight Fallacies of Distributed Computing

Peter Deutsch

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

For more details, read the article by Arnon Rotem-Gal-Oz

Source: https://blogs.oracle.com/jag/resource/Fallacies.html

Fallacies of Distributed Computing

- Network is reliable
- Latency is zero
- Bandwidth is infinite
- Network is secure
- Topology doesn’t change
- There is 1 administrator
- Transport cost is 0
- Network is homogenous

Fallacies of Distributed Computing

- Network is reliable
- Latency is zero
- Bandwidth is infinite
- Network is secure
- Topology doesn't change
- There is 1 administrator
- Transport cost is 0
- Network is homogenous

Fallacies of Distributed Computing

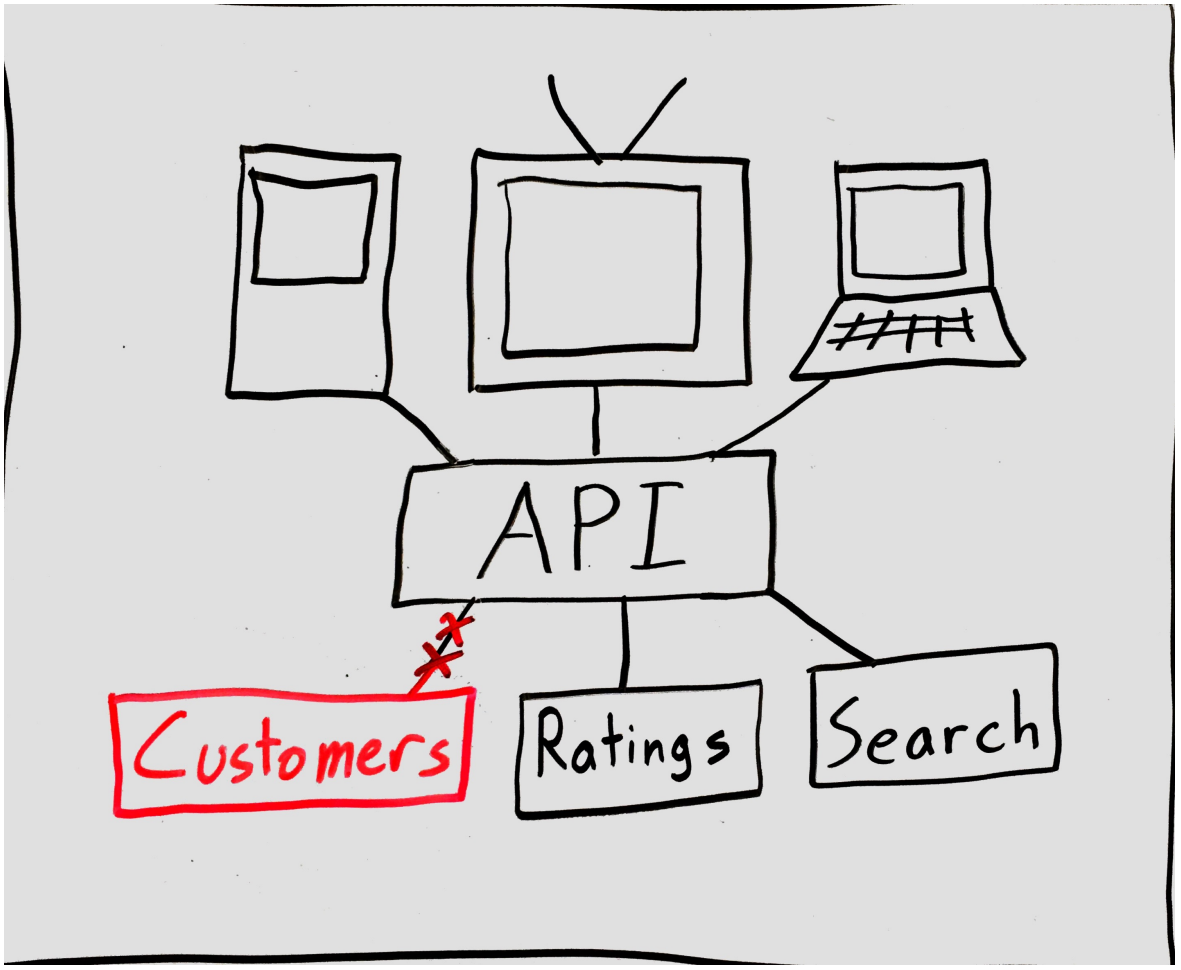
- In the cloud, other services are reliable
- In the cloud, latency is zero
- In the cloud, bandwidth is infinite
- Network is secure
- Topology doesn't change
- In the cloud, there is 1 administrator
- Transport cost is 0
- Network is homogenous

Other Services are Reliable

`getCustomer()` fails at 0% when on-box

~~Other Services are Reliable~~

`getCustomer()` used to fail at 0%

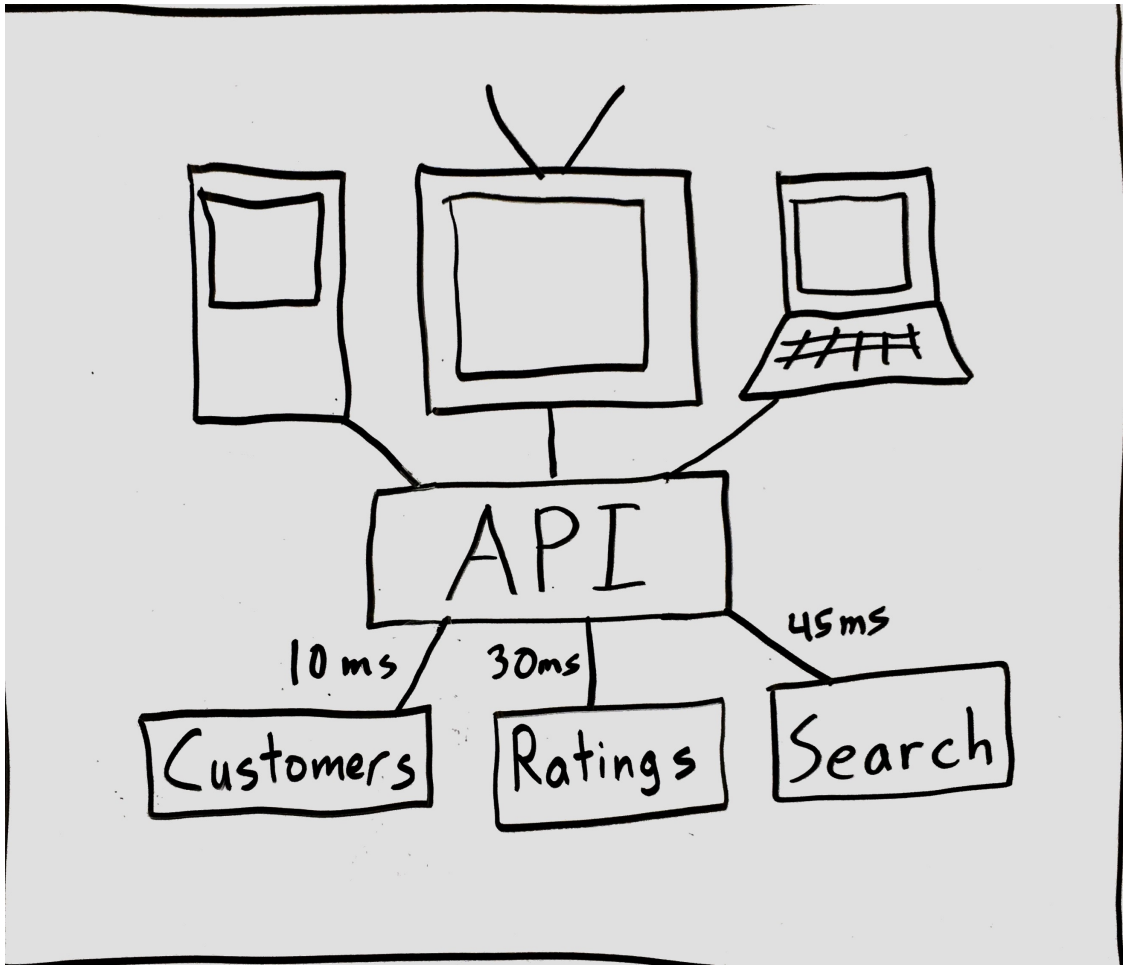


Latency is zero

- `search(term)` returns in µs or ns

~~Latency is zero~~

- `search(term)` used to return in µs or ns

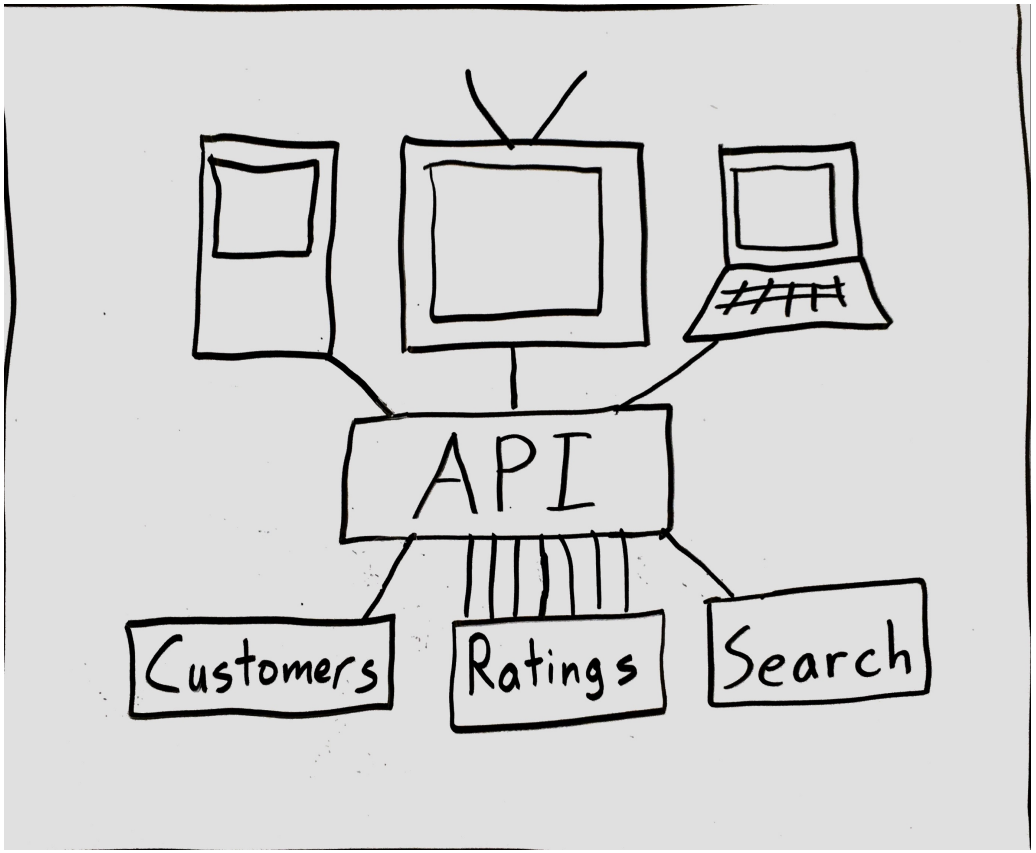


Bandwidth is infinite

- Making n calls to `getRating()` is fine

~~Bandwidth is infinite~~

- Making n calls to `getRating()` results in multiple network calls

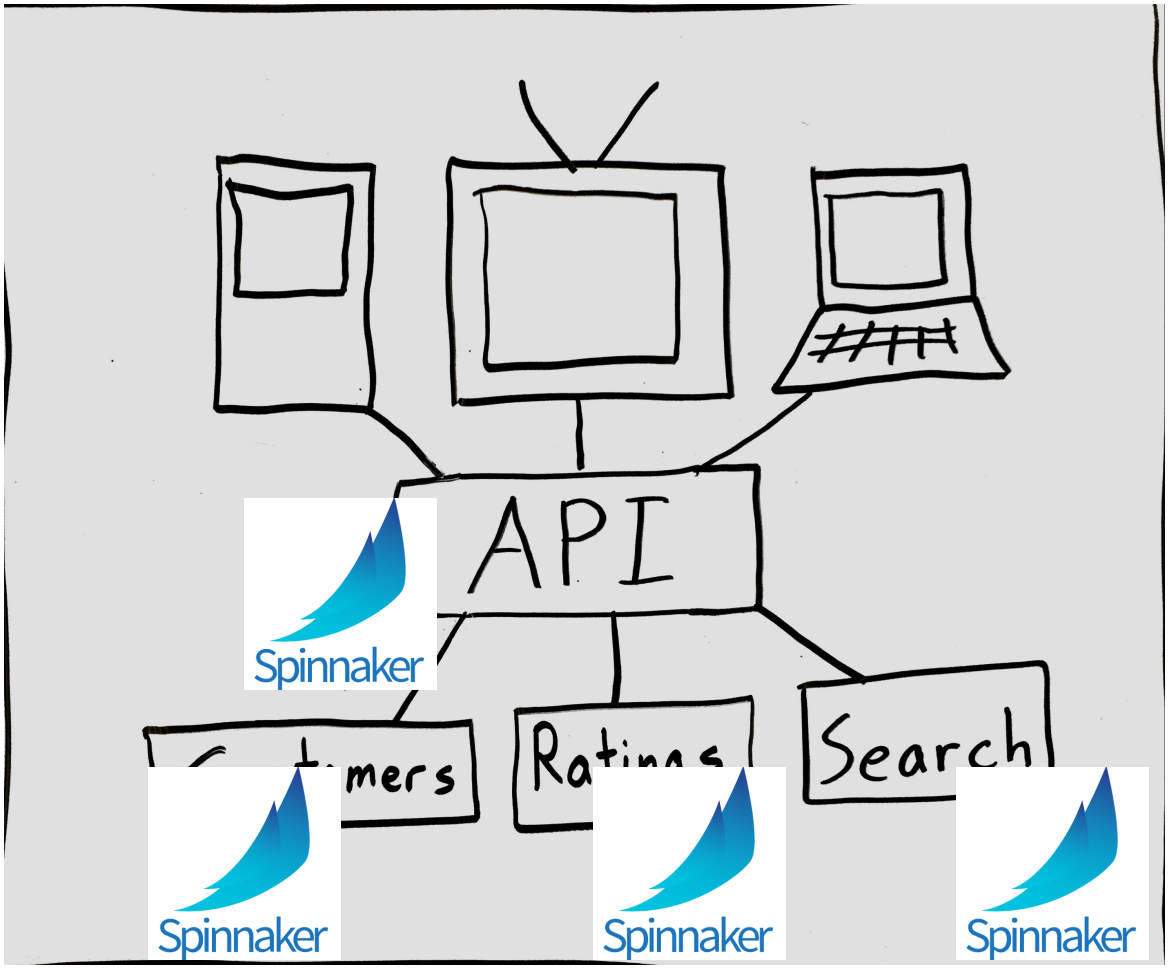


There is 1 administrator

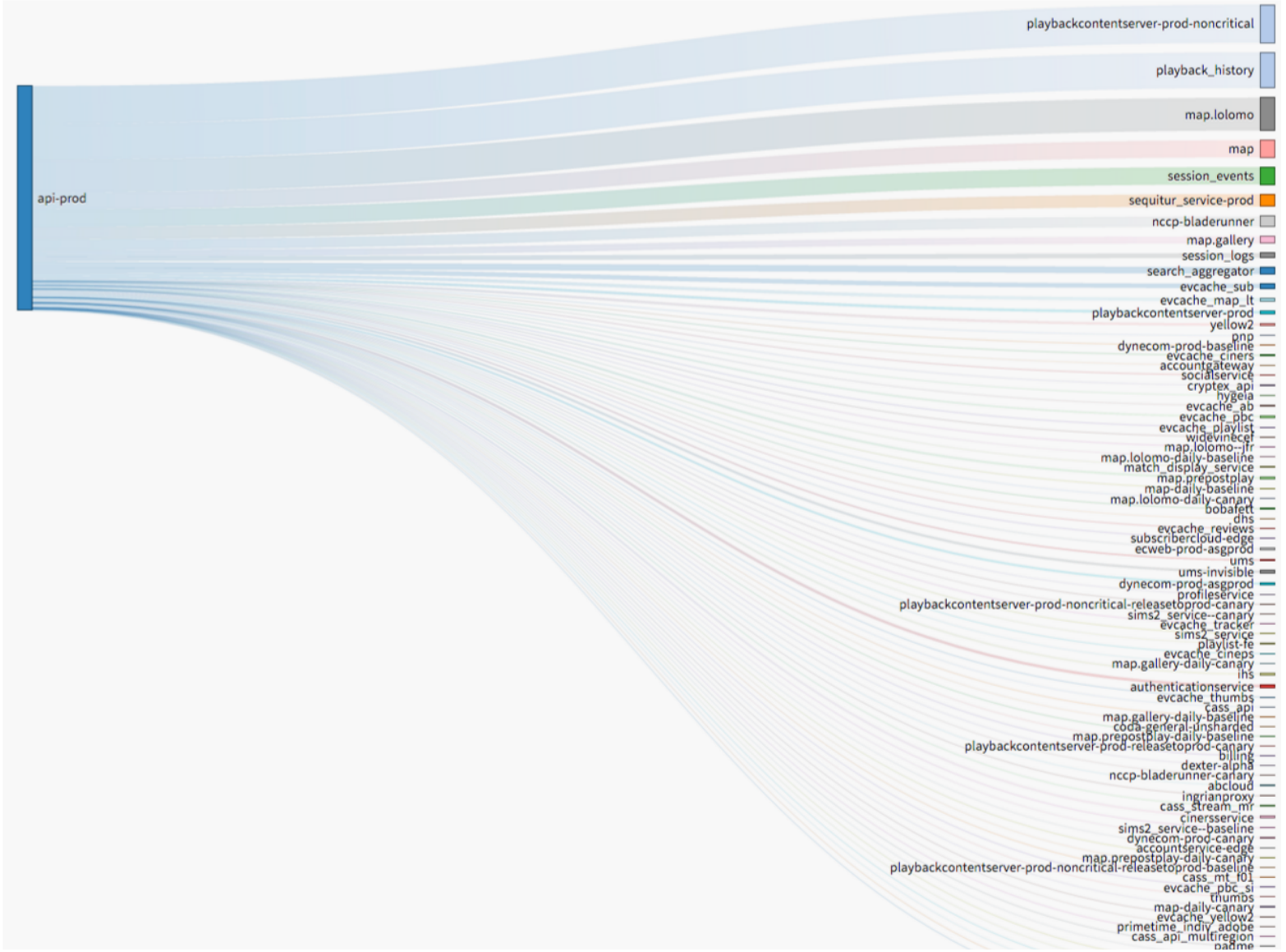
- Deployments are in 1 person's head

~~There is 1 administrator~~

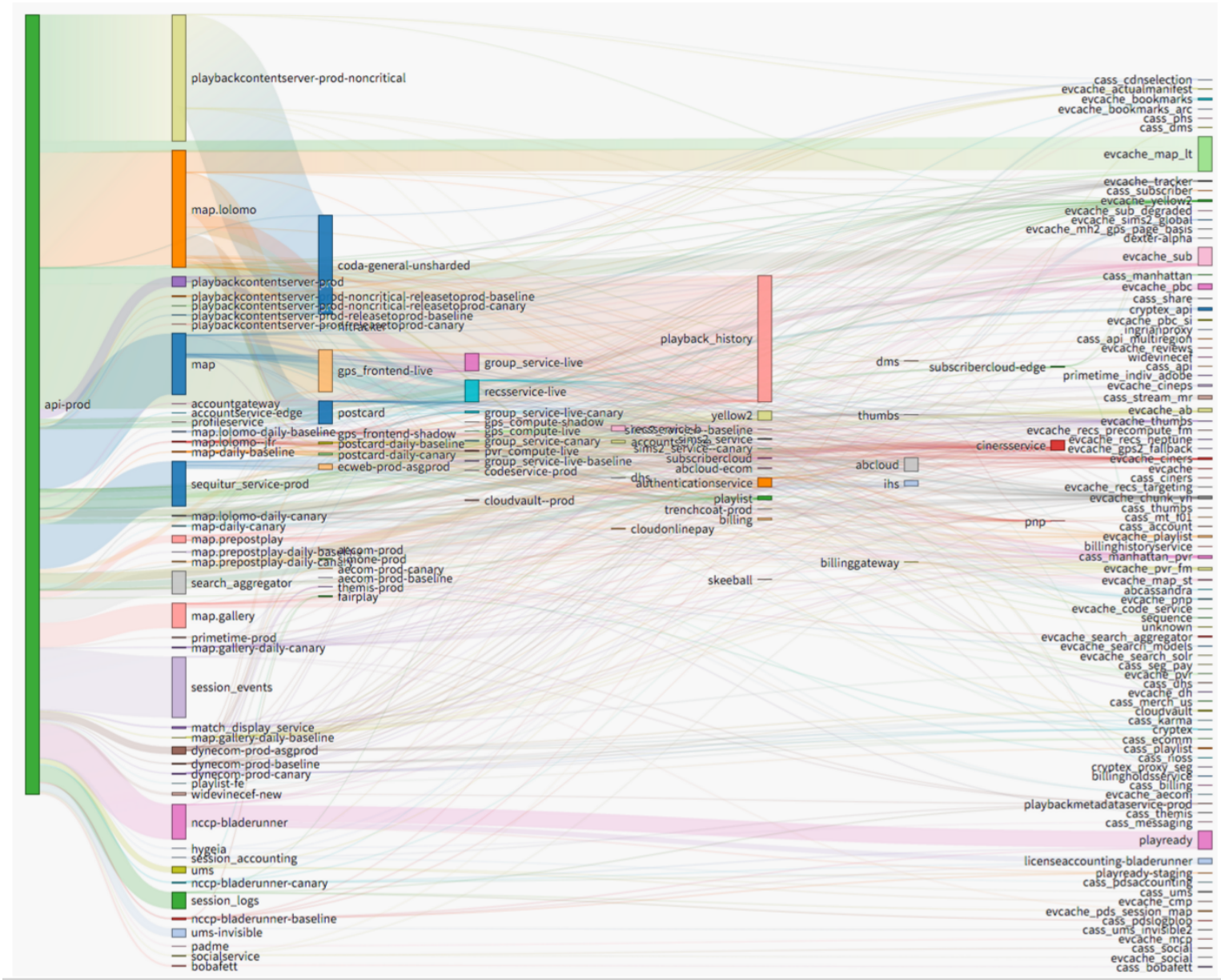
- Deployments are distributed



~~There is 1 administrator~~



~~There is 1 administrator~~



~~There is 1 administrator~~

```
Received: by jumbo.dec.com (5.54.3/4.7.34)
        id AA09105; Thu, 28 May 87 12:23:29 PDT
Date: Thu, 28 May 87 12:23:29 PDT
From: lamport (Leslie Lamport)
Message-Id: <8705281923.AA09105@jumbo.dec.com>
To: src-t
Subject: distribution
```

There has been considerable debate over the years about what constitutes a distributed system. It would appear that the following definition has been adopted at SRC:

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

Source: http://research.microsoft.com/en-us/um/people/lamport/pubs/distributed-system.txt

Goals of this talk

- Philosophical Motivation
  - Why are distributed systems hard?
- Practical Motivation
  - Why do I keep getting paged?
- Solving those problems with Hystrix
  - How does it work?
  - How do I use it in my system?
  - How should a system behave if I use Hystrix?
  - What? Netflix was down for me – what happened there?

Operating a Distributed System

- How can I stop going to incident reviews?
  - Which occur every time we cause customer pain

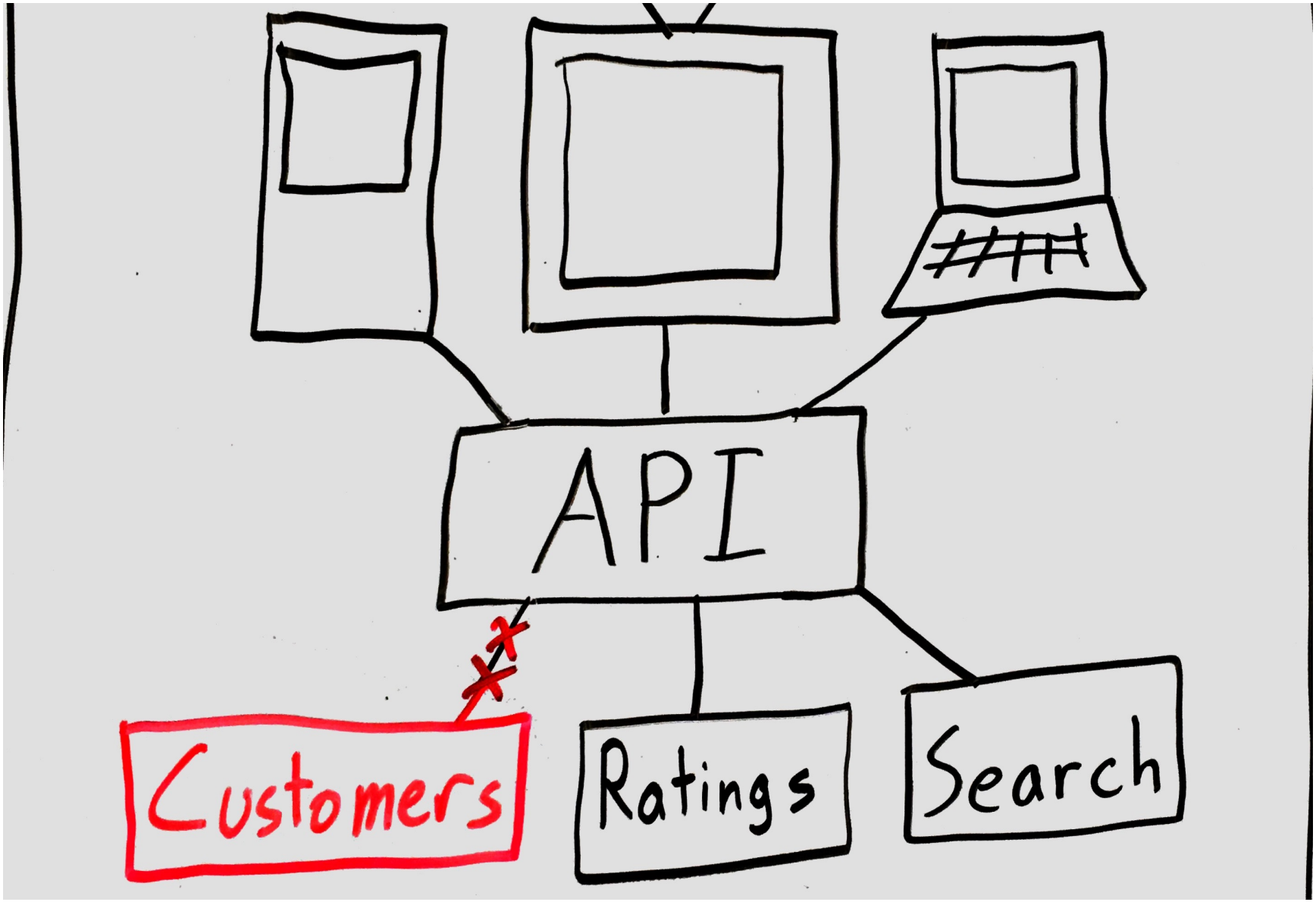
Operating a Distributed System

- How can I stop going to incident reviews?
  - Which occur every time we cause customer pain
- We must assume every other system can fail

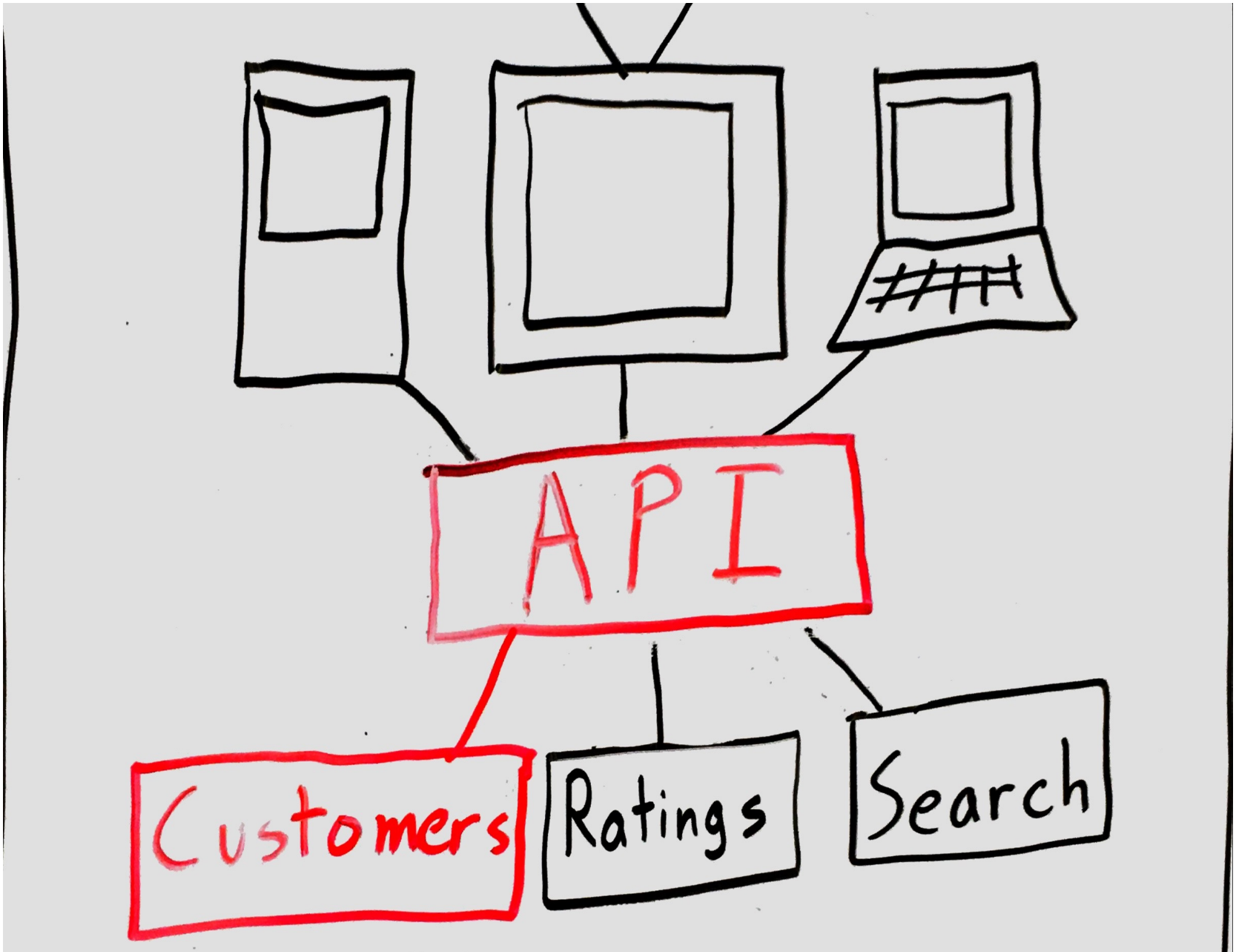
Operating a Distributed System

- How can I stop going to incident reviews?
  - Which occur every time we cause customer pain
- We must assume every other system can fail
- We must understand those failure modes and prevent them from cascading

Operating a Distributed System (bad)

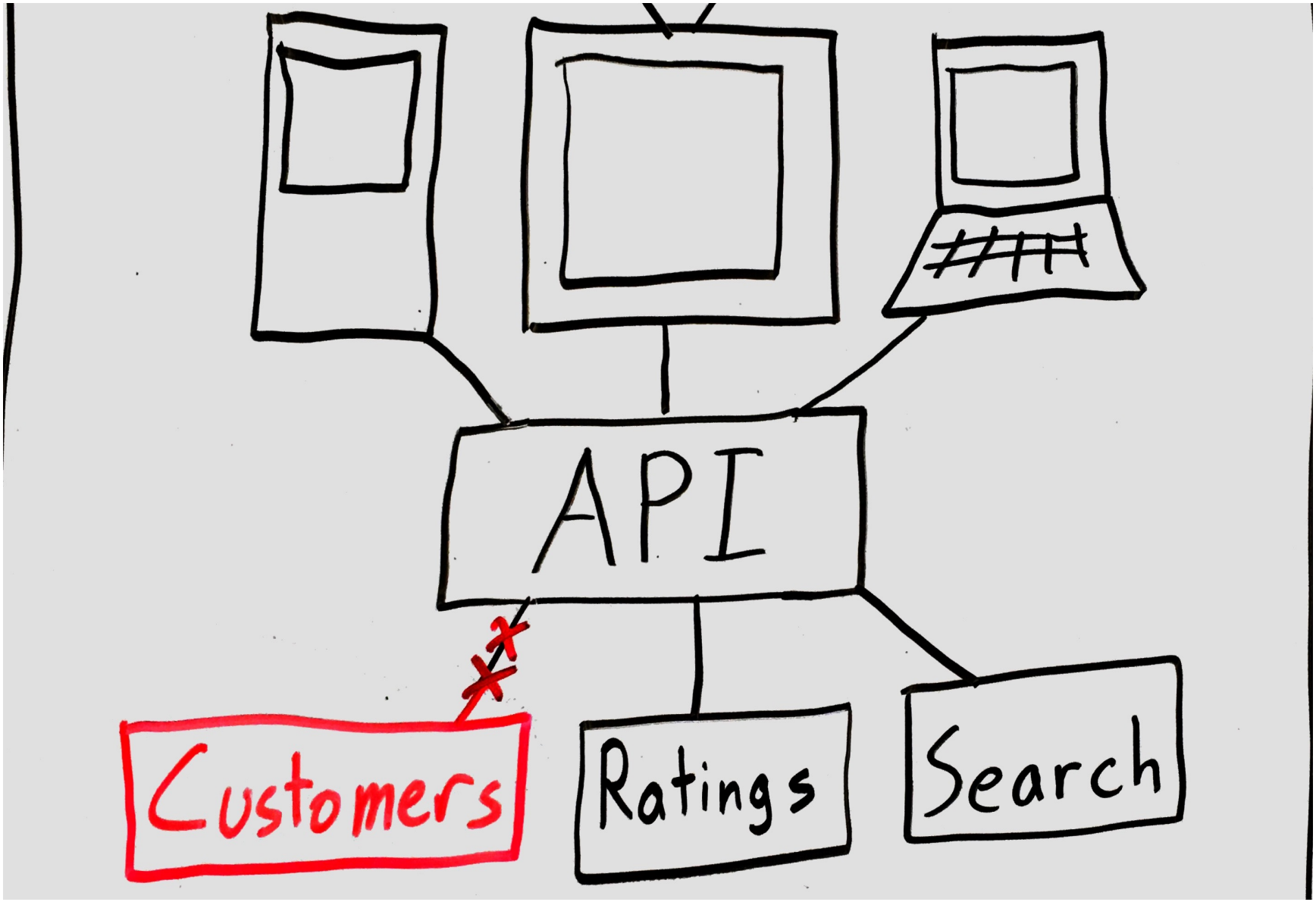


Operating a Distributed System (bad)



Operating a Distributed System (bad)

Operating a Distributed System (good)



Failure Modes (in the small)

- Errors
- Latency

If you don't plan for errors

How to handle Errors

```
return getCustomer(id);
```

How to handle Errors

```
try {
    return getCustomer(id);
} catch (Exception ex) {
    //TODO What to return here?
}
```

How to handle Errors

```
try {
    return getCustomer(id);
} catch (Exception ex) {
    //static value
    return null;
}
```

How to handle Errors

```
try {
    return getCustomer(id);
} catch (Exception ex) {
    //value from memory
    return anonymousCustomer;
}
```

How to handle Errors

```
try {
    return getCustomer(id);
} catch (Exception ex) {
    //value from cache
    return cache.getCustomer(id);
}
```

How to handle Errors

```
try {
    return getCustomer(id);
} catch (Exception ex) {
    //value from service
    return customerViaSecondSvc(id);
}
```

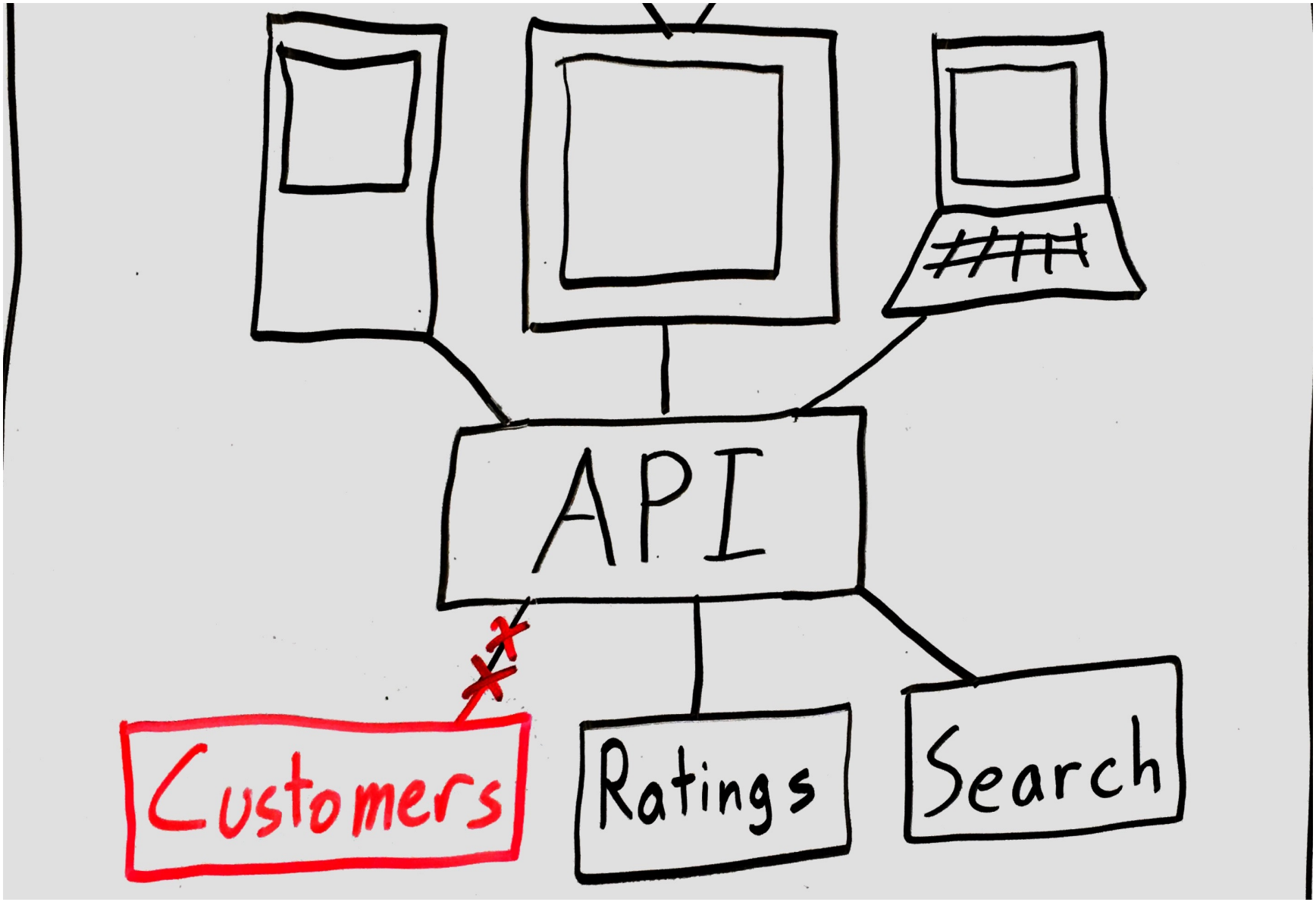
How to handle Errors

```
try {
    return getCustomer(id);
} catch (Exception ex) {
    //explicitly rethrow
    throw ex;
}
```

Handle Errors with Fallbacks

- Some options for fallbacks
  - Static value
  - Value from in-memory
  - Value from cache
  - Value from network
  - Throw
- Make error-handling explicit
- Applications have to work in the presence of either fallbacks or rethrown exceptions

Handle Errors with Fallbacks



Exposure to failures

- As your app grows, your set of dependencies is much more likely to get bigger, not smaller
- Overall uptime = (Dep uptime)^(num deps)

Dependency uptime

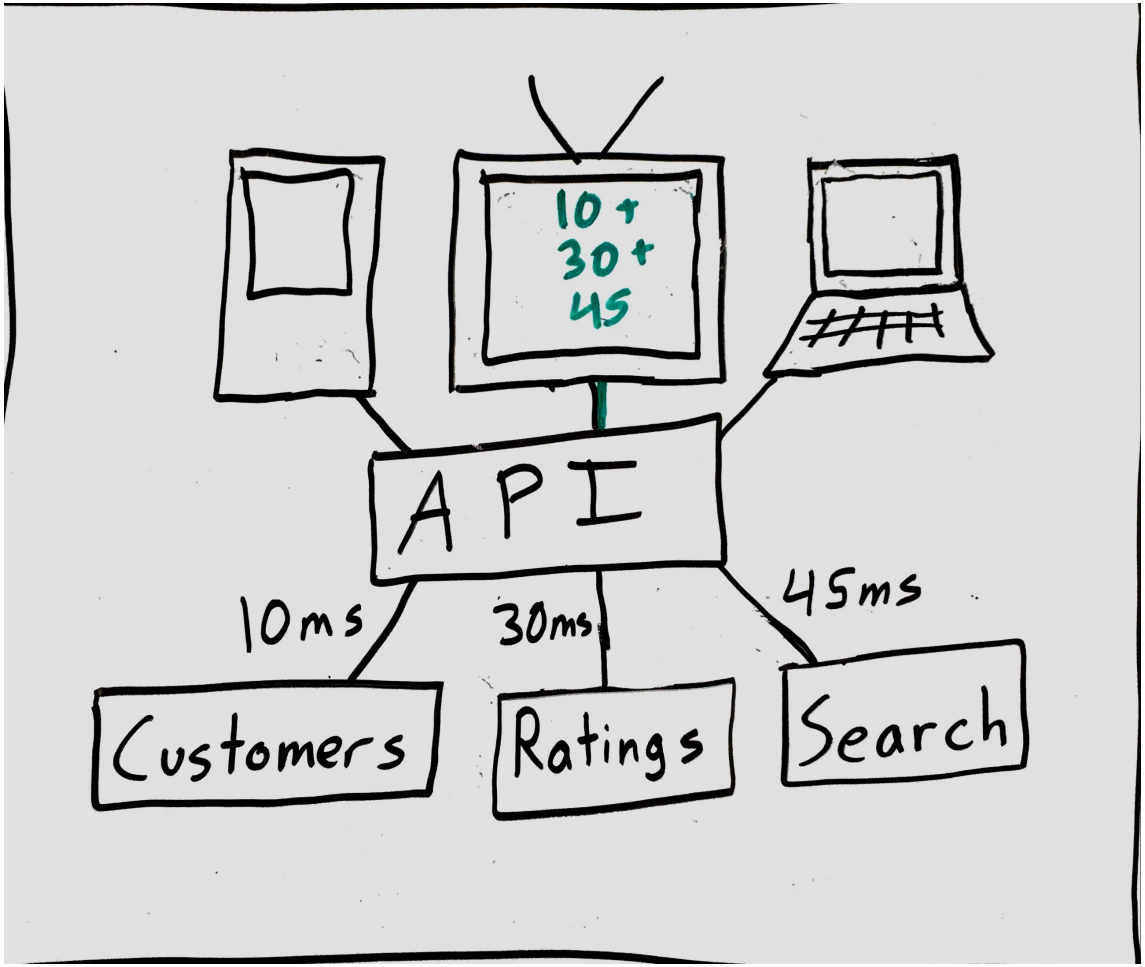
Dependencies	99.99%	99.9%	99%
5	99.95%	99.5%	95%
10	99.9%	99%	90.4%
25	99.75%	97.5%	77.8%

If you don’t plan for Latency

- First-order effect: outbound latency increase
- Second-order effect: increase resource usage

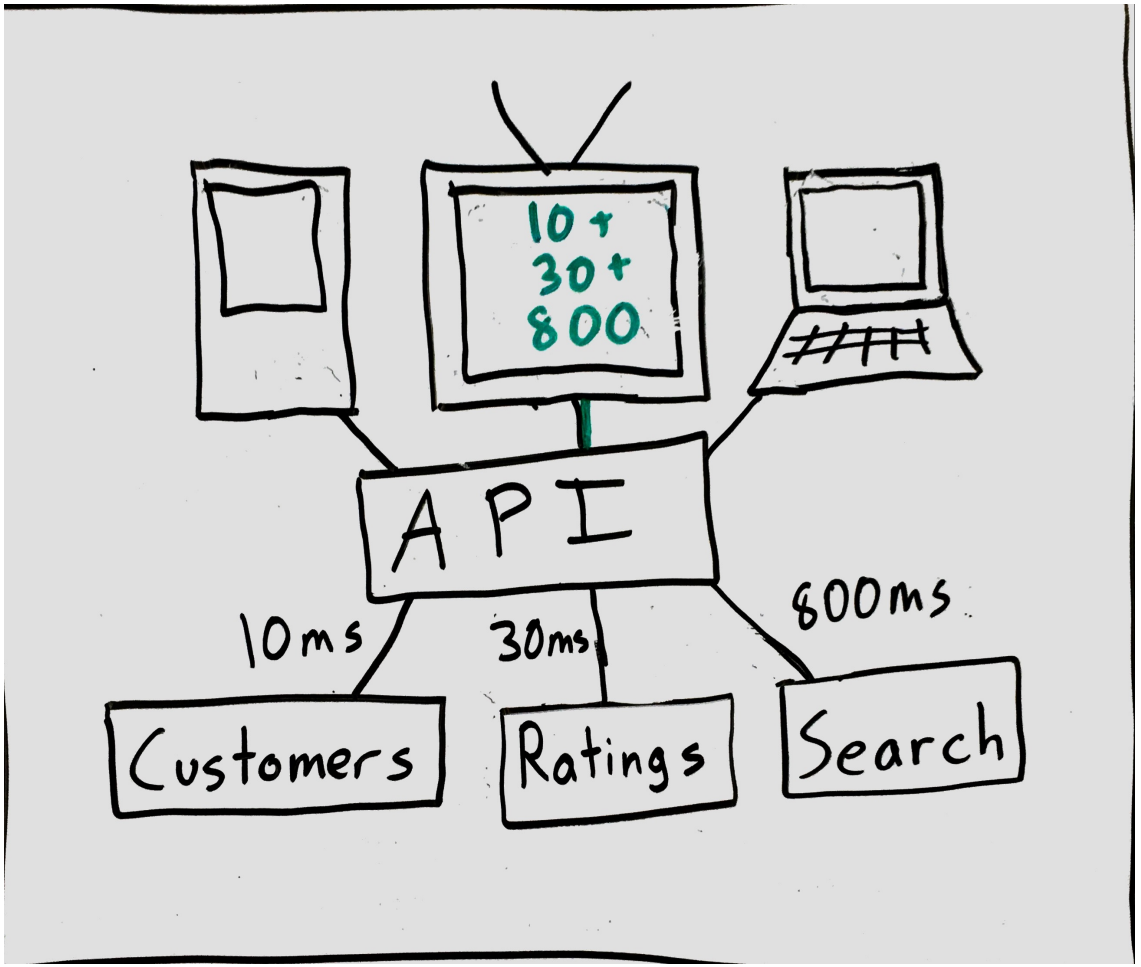
If you don't plan for Latency

- First-order effect: outbound latency increase



If you don't plan for Latency

- First-order effect: outbound latency increase



Queuing theory

- Say that your application is making 100RPS of a network call and mean latency is 10ms

Queuing theory

- Say that your application is making 100RPS of a network call and mean latency is 10ms
- The utilization of I/O threads has a distribution over time – according to Little's Law, the mean is 100 RPS * (1/100 s) = 1

Little's law

From Wikipedia, the free encyclopedia

In queueing theory, a discipline within the mathematical theory of probability, **Little's result**, **theorem**, **lemma**, **law** or **formula**[1][2] is a theorem by John Little which states:

> The long-term average number of customers in a stable system L is equal to the long-term average effective arrival rate, λ, multiplied by the (Palm-)average time a customer spends in the system, W; or expressed algebraically: $L = \lambda W$.

Queuing theory

- Say that your application is making 100RPS of a network call and mean latency is 10ms
- The utilization of I/O threads has a distribution over time – according to Little's Law, the mean is 100 RPS * (1/100 s) = 1
- If mean latency increased to 100ms, mean utilization increases to 10

Queuing theory

- Say that your application is making 100RPS of a network call and mean latency is 10ms
- The utilization of I/O threads has a distribution over time – according to Little's Law, the mean is 100 RPS * (1/100 s) = 1
- If mean latency increased to 100ms, mean utilization increases to 10
- If mean latency increased to 1000ms, mean utilization increases to 100

If you don't plan for Latency

```
"http-0.0.0.0-8443-102" daemon prio=3 tid=0x022a6400 nid=0x1bd runnable [0x78efb00
   java.lang.Thread.State: RUNNABLE
  at java.net.SocketInputStream.socketRead0(Native Method)
  at java.net.SocketInputStream.read(SocketInputStream.java:129)
  at com.sun.net.ssl.internal.ssl.InputRecord.readFully(InputRecord.java:293)
  at com.sun.net.ssl.internal.ssl.InputRecord.read(InputRecord.java:331)
  at com.sun.net.ssl.internal.ssl.SSLSocketImpl.readRecord(SSLSocketImpl.java:789)
  - locked <0xdd0ed968> (a java.lang.Object)
  at com.sun.net.ssl.internal.ssl.SSLSocketImpl.readD
  at com.sun.net.ssl.internal.ssl.AppInputStream.read
  - locked <0xdd0eda88> (a com.sun.net.ssl.internal.s
  at java.io.BufferedInputStream.fill(BufferedInputSt
  at java.io.BufferedInputStream.read1(BufferedInputS
  at java.io.BufferedInputStream.read(BufferedInputSt
  - locked <0xddb1f6d0> (a java.io.BufferedInputStrea
  at sun.net.www.http.HttpClient.parseHTTPHeader(HttpClient.java:687)
  at sun.net.www.http.HttpClient.parseHTTP(HttpClient.java:632)
  at sun.net.www.http.HttpClient.parseHTTP(HttpClient.java:652)
  ..................
```

STUCK thread due to lack of HTTPS timeout with a remote Web Service provider (blocking IO)

Handle Latency with Timeouts

- Bound the worst-case latency

Handle Latency with Timeouts

- Bound the worst-case latency
- What should we return upon timeout?
  - We've already designed a fallback

Bound resource utilization

- Don’t take on too much work globally

Bound resource utilization

- Don’t take on too much work globally
- Don’t let any single dependency take too many resources

Bound resource utilization

- Don’t take on too much work globally
- Don’t let any single dependency take too many resources
- Bound the concurrency of each dependency

Bound resource utilization

- Don’t take on too much work globally
- Don’t let any single dependency take too many resources
- Bound the concurrency of each dependency
- What should we do if we’re at that threshold?
  - We’ve already designed a fallback.

Goals of this talk

- Philosophical Motivation
  - Why are distributed systems hard?
- Practical Motivation
  - Why do I keep getting paged?
- Solving those problems with Hystrix
  - How does it work?
  - How do I use it in my system?
  - How should a system behave if I use Hystrix?
  - What? Netflix was down for me – what happened there?

Hystrix Goals

- Handle Errors with Fallbacks
- Handle Latency with Timeouts
- Bound Resource Utilization

Handle Errors with Fallback

- We need something to execute
- We need a fallback

Handle Errors with Fallback

```
class CustLookup extends HystrixCommand<Customer> {
    @Override
    public Customer run() {
        return svc.getOverHttp(customerId);
    }
}
```

Handle Errors with Fallback

```
class CustLookup extends HystrixCommand<Customer> {
    @Override
    public Customer run() {
        return svc.getOverHttp(customerId);
    }

    @Override
    public Customer getFallback() {
        return Customer.anonymousCustomer();
    }
}
```

Handle Errors with Fallback

```
class CustLookup extends HystrixCommand<Customer> {
    private final CustomersService svc;
    private final long customerId;

    public CustLookup(CustomersService svc, long id)
     {
         this.svc = svc;
         this.customerId = id;
     }


     //run() : has references to svc and customerId
     //getFallback()
}
```

Handle Latency with Timeouts

- We need a timeout value
- And a timeout mechanism

Handle Latency with Timeouts

```
class CustLookup extends HystrixCommand<Customer> {
   private final CustomersService svc;
   private final long customerId;

   public CustLookup(CustomersService svc, long id)
   {
      super(timeout = 100);
      this.svc = svc;
      this.customerId = id;
   }


   //run(): has references to svc and customerId
   //getFallback()
}
```

Handle Latency with Timeouts

- What if run() does not respect your timeout?

Handle Latency with Timeouts

```
class CustLookup extends HystrixCommand<Customer> {

    @Override
    public Customer run() {
        return svc.getOverHttp(customerId);
    }

    @Override
    public Customer getFallback() {
        return Customer.anonymousCustomer();
    }
}
```

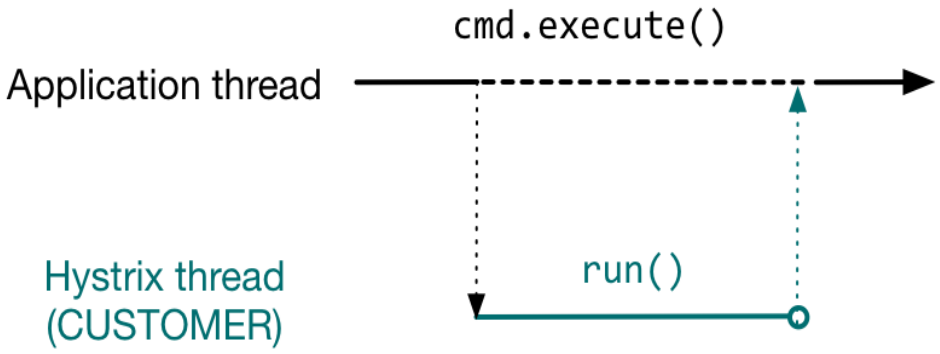
Handle Latency with Timeouts

```
class CustLookup extends HystrixCommand<Customer> {

    @Override
    public Customer run() {
        return svc.nowThisThreadIsMine(customerId);
    }

    @Override
    public Customer getFallback() {
        return Customer.anonymousCustomer();
    }
}
```

Handle Latency with Timeouts

- What if run() does not respect your timeout?
- We can’t control the I/O library
- We can put it on a different thread to control its impact

Hystrix execution model



Bound Resource Utilization

- Since we're using separate thread pool, we should give it a fixed size

Bound Resource Utilization

```
class CustLookup extends HystrixCommand<Customer> {

    public CustLookup(CustomersService svc, long id)
    {
        super(timeout = 100,
              threadPool = “CUSTOMER”,
              threadPoolSize = 8);
        this.svc = svc;
        this.customerId = id;
    }

    //run() : has references to svc and customerId
    //getFallback()
}
```

Thread pool as a bulkhead

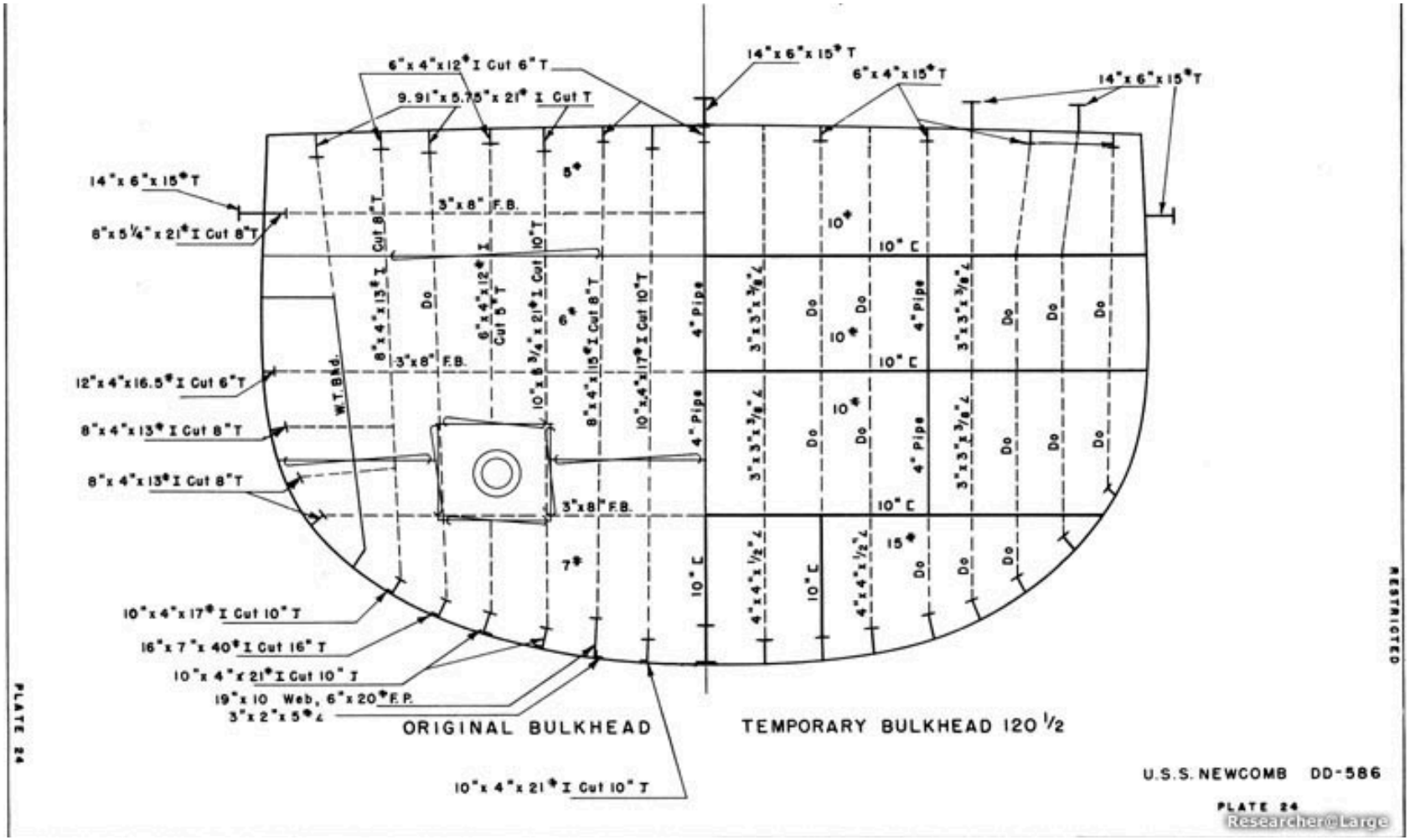
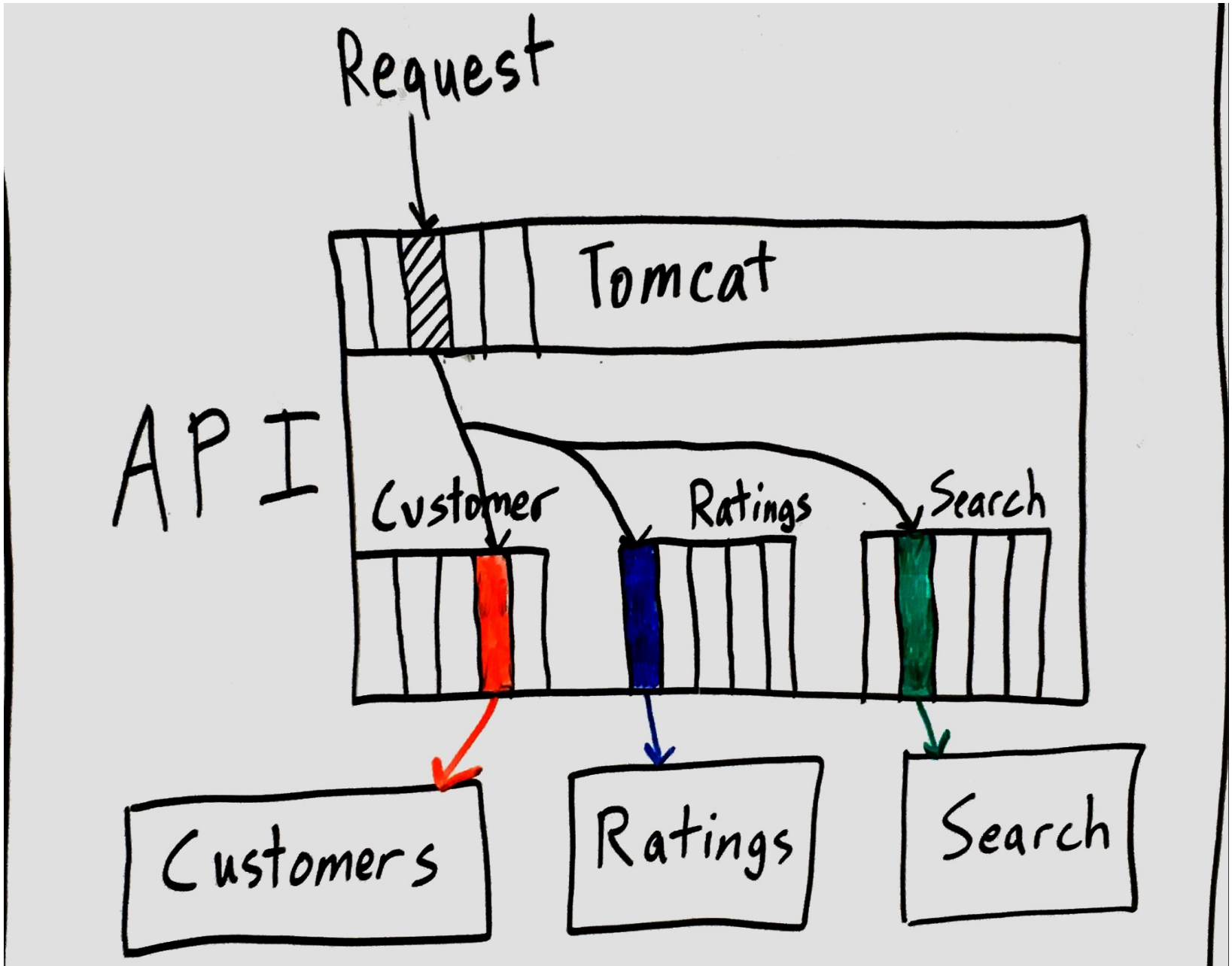


Image Credit: http://www.researcheratlarge.com/Ships/DD586/DD586ForwardRepair.html

Thread pool as a bulkhead



Circuit-Breaker

- Popularized by Michael Nygard in “Release It!”

Circuit-Breaker

- We can build feedback loop

Circuit-Breaker

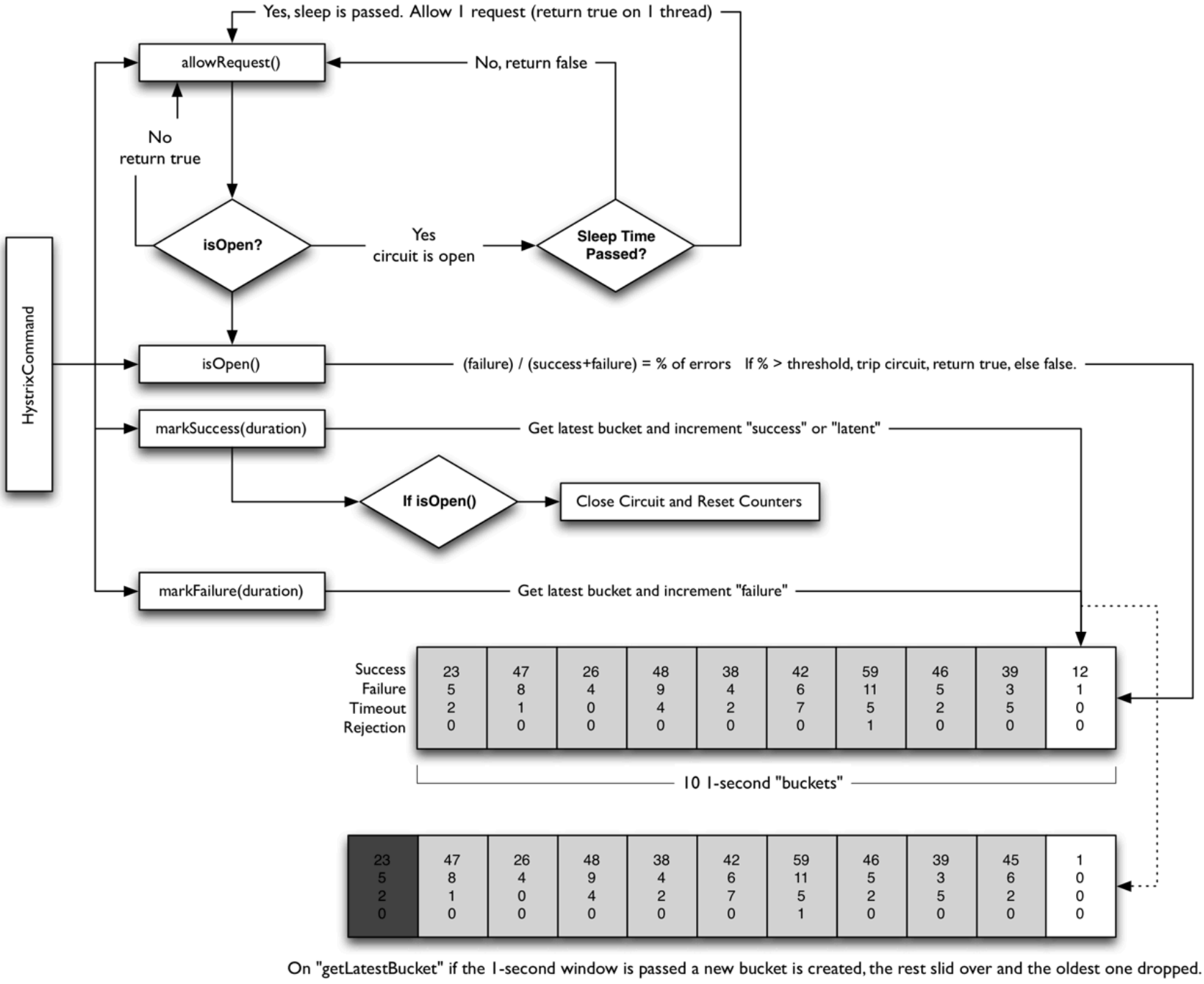
- We can build feedback loop
- If command has high error rate in last 10 seconds, it's more likely to fail right now

Circuit-Breaker

- We can build feedback loop
- If command has high error rate in last 10 seconds, it’s more likely to fail right now
- Fail fast now – don’t spend my resources

Circuit-Breaker

- We can build feedback loop
- If command has high error rate in last 10 seconds, it's more likely to fail right now
- Fail fast now – don't spend my resources
- Give downstream system some breathing room


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 23 47 26 48 38 42 59 46 39 12
Failure 5 8 4 9 4 6 11 5 3 1
Timeout 2 1 0 4 2 7 5 2 5 0
Rejection 0 0 0 0 0 0 1 0 0 0
10 1-second "buckets"
23 47 26 48 38 42 59 46 39 45 1
5 8 4 9 4 6 11 5 3 6 0
2 1 0 4 2 7 5 2 5 2 0
0 0 0 0 0 0 1 0 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.

Hystrix as a Decision Tree

- Compose all of the above behaviors

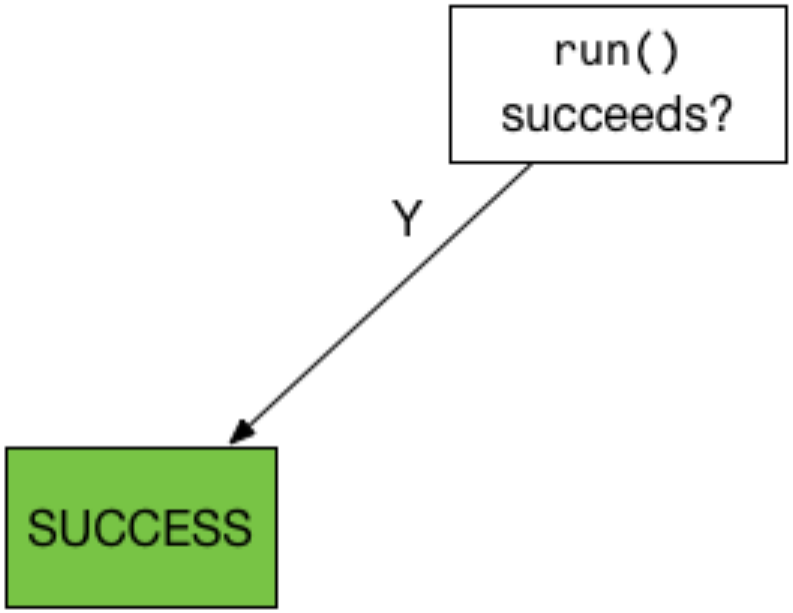
Hystrix as a Decision Tree

- Compose all of the above behaviors
- Now you've got a library!

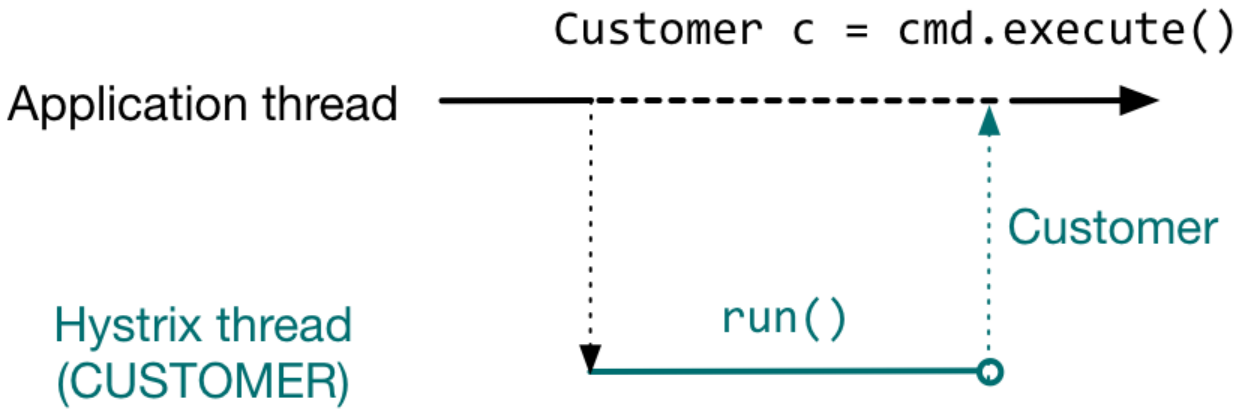
SUCCESS

```
class CustLookup extends HystrixCommand<Customer> {
    @Override
    public Customer run() {
        return svc.getOverHttp(customerId); //succeeds
    }

    @Override
    public Customer getFallback() {
        return Customer.anonymousCustomer();
    }
}
```

SUCCESS



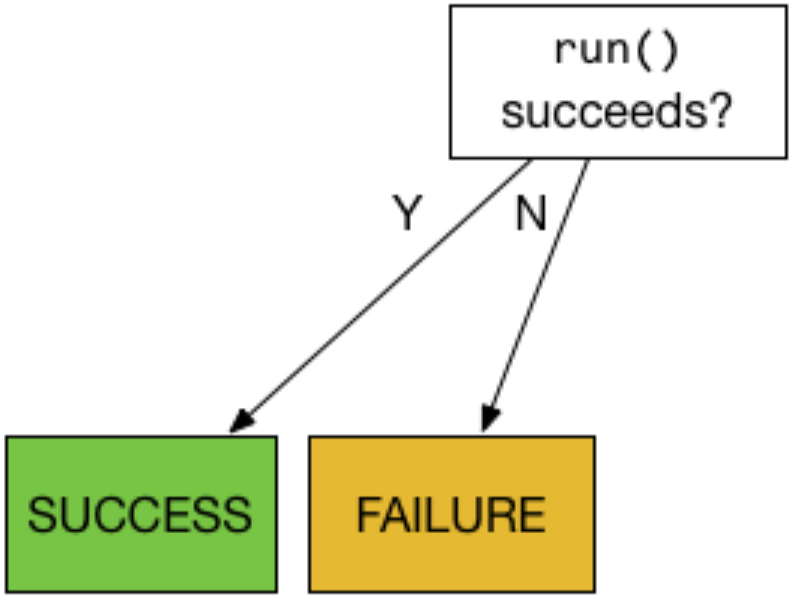
SUCCESS


Customer c = cmd.execute()
Application thread
Customer
Hystrix thread
(CUSTOMER)
run()

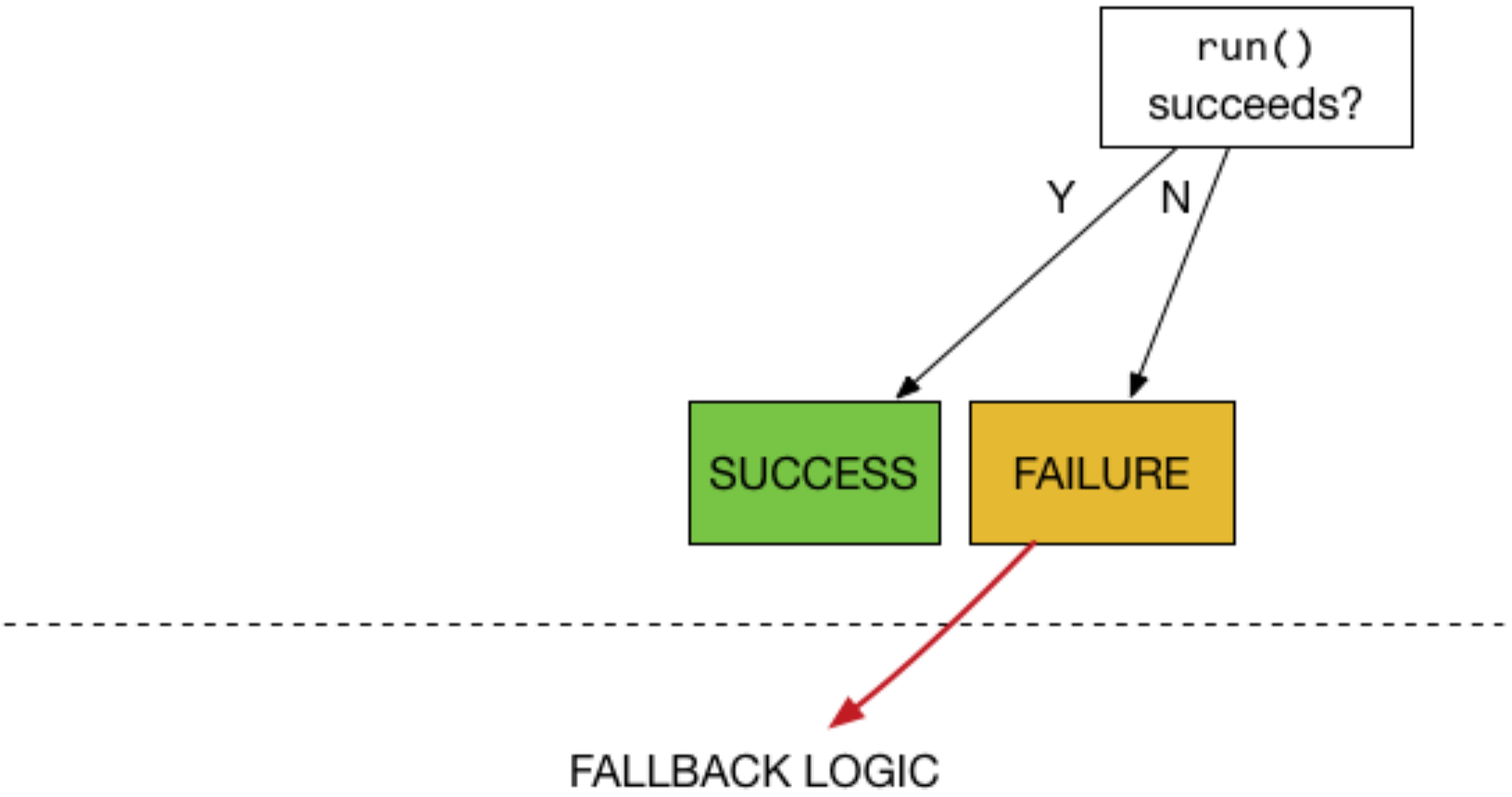
FAILURE

```
class CustLookup extends HystrixCommand<Customer> {
    @Override
    public Customer run() {
        return svc.getOverHttp(customerId); //throws
    }

    @Override
    public Customer getFallback() {
        return Customer.anonymousCustomer();
    }
}
```

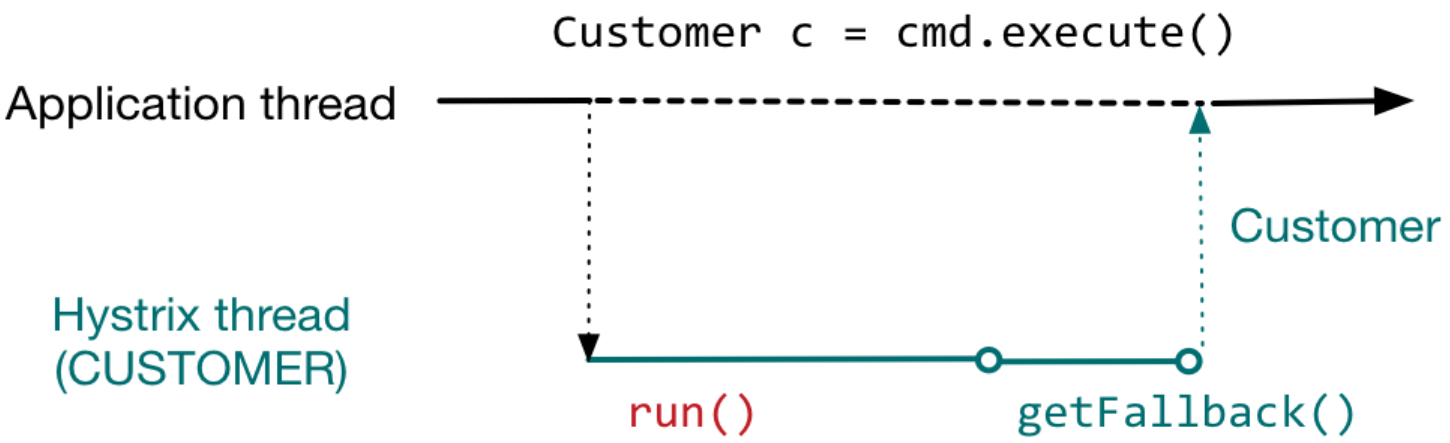
FAILURE



FAILURE



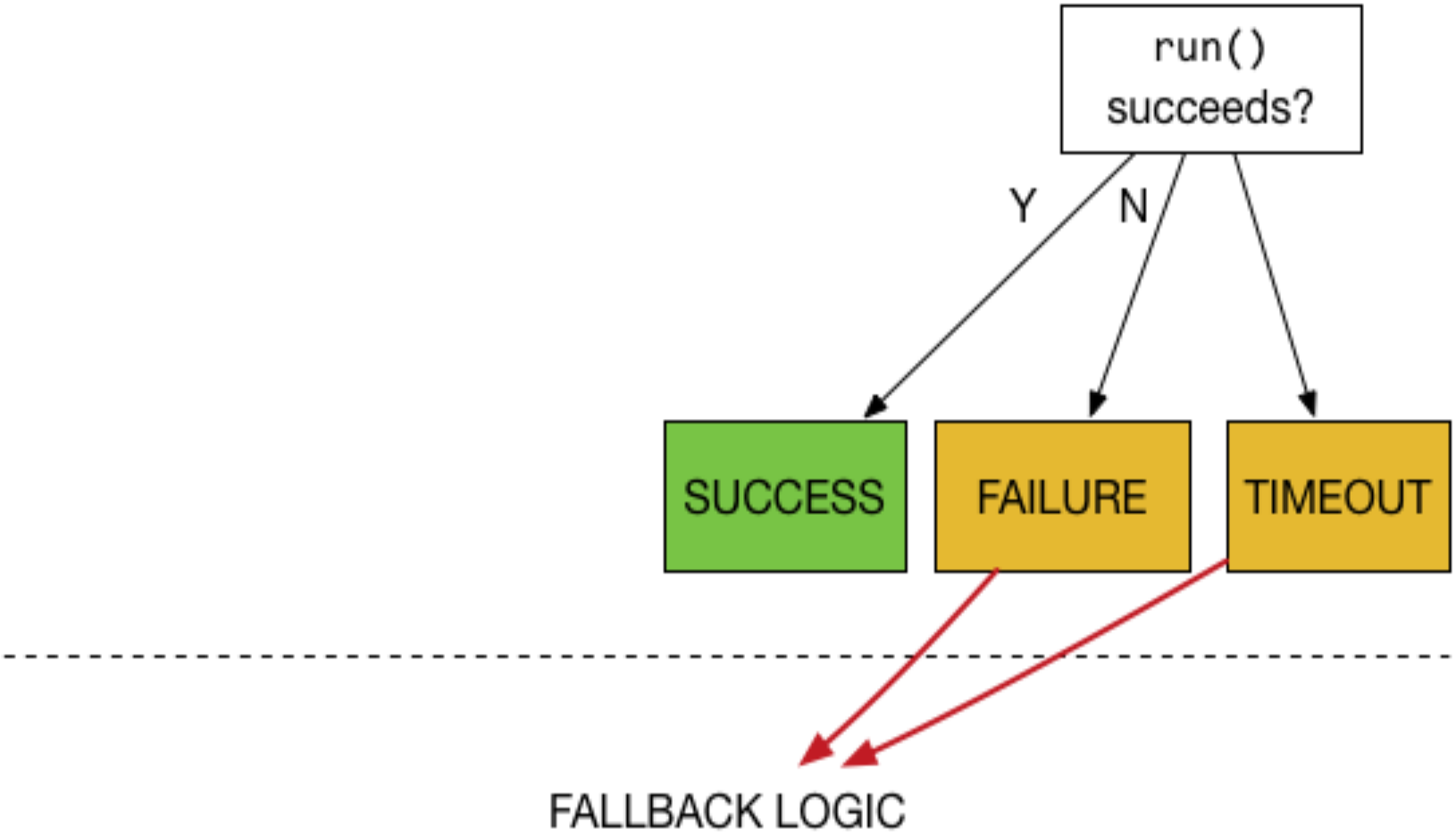
FAILURE


Customer c = cmd.execute()
Application thread
Customer
Hystrix thread
(CUSTOMER)
run()
getFallback()

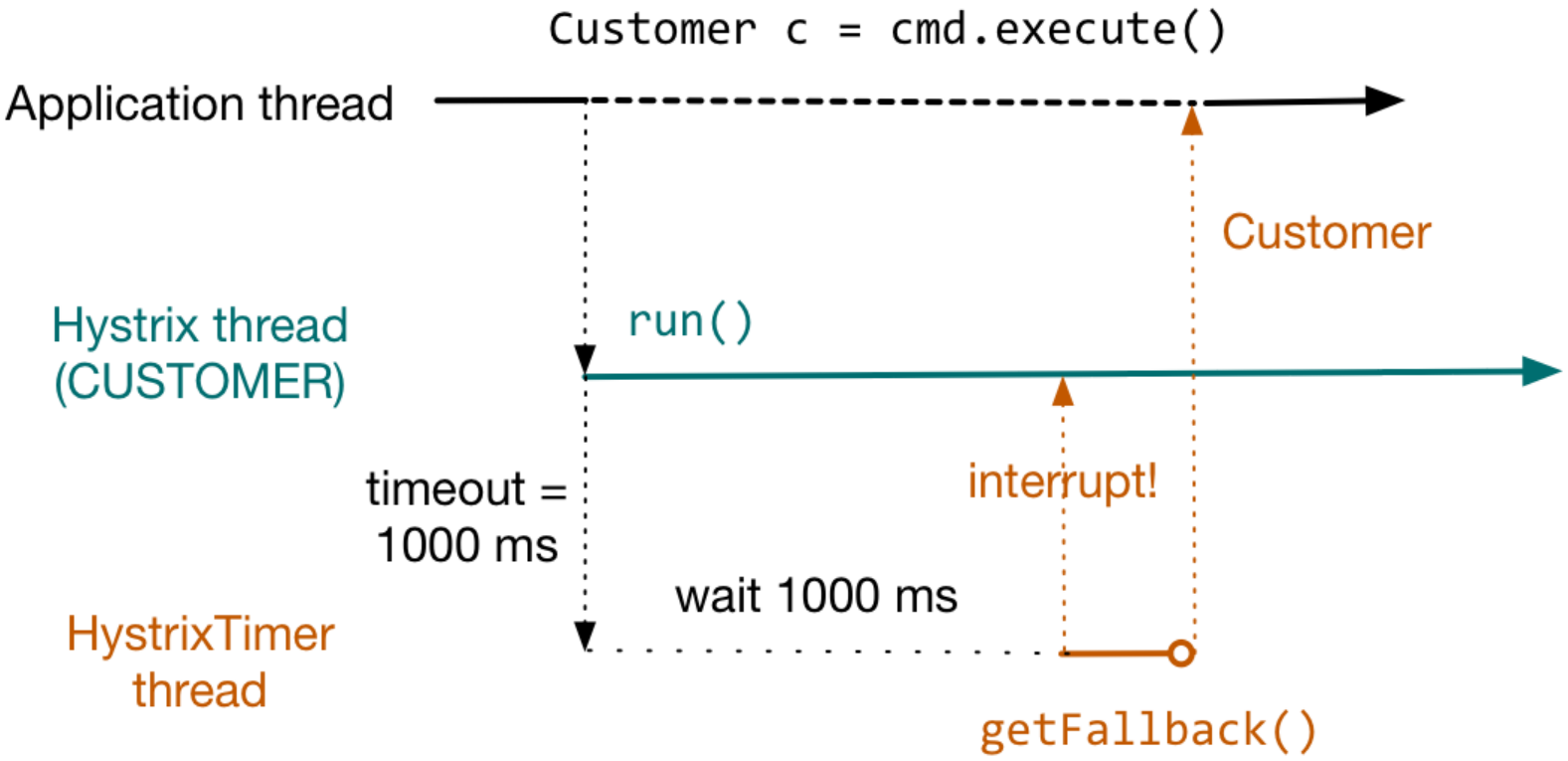
TIMEOUT

```
class CustLookup extends HystrixCommand<Customer> {
    @Override
    public Customer run() {
        //doesn't return in time
        return svc.getOverHttp(customerId);
    }

    @Override
    public Customer getFallback() {
        return Customer.anonymousCustomer();
    }
}
```

TIMEOUT



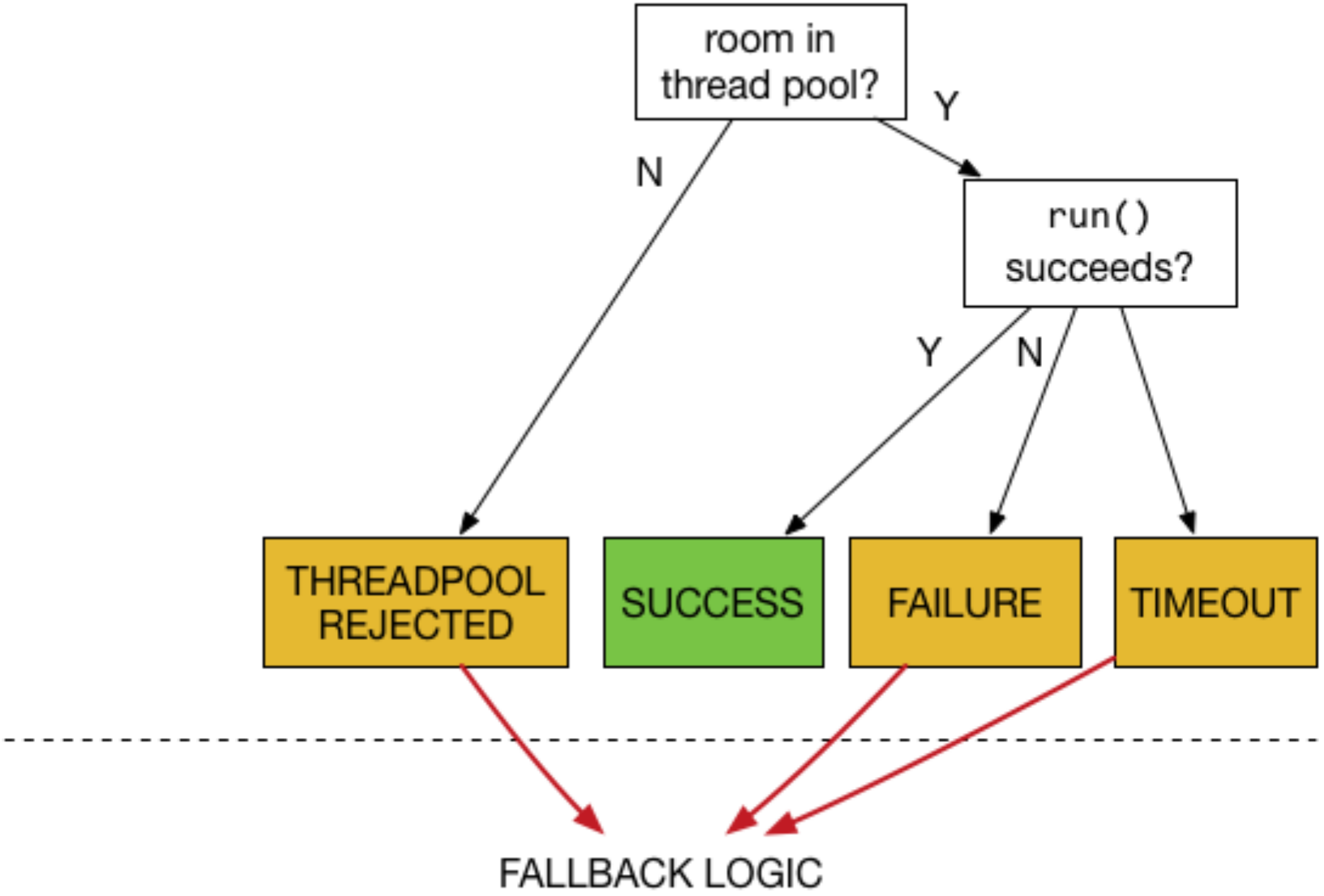
TIMEOUT


Customer c = cmd.execute()
Application thread
Customer
Hystrix thread
(CUSTOMER)
run()
timeout =
1000 ms
interrupt!
wait 1000 ms
HystrixTimer
thread
getFallback()

THREADPOOL-REJECTED

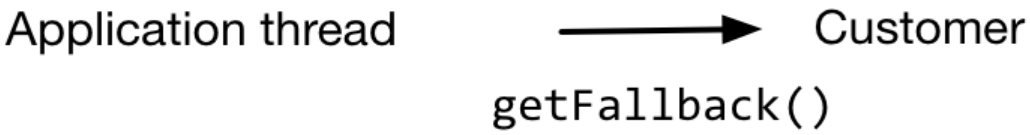
```
class CustLookup extends HystrixCommand<Customer> {
    @Override
    public Customer run() {
        //never runs – threadpool is full
        return svc.getOverHttp(customerId);
    }

    @Override
    public Customer getFallback() {
        return Customer.anonymousCustomer();
    }
}
```

THREADPOOL-REJECTED



THREADPOOL-REJECTED

```
Customer c = cmd.execute()
```

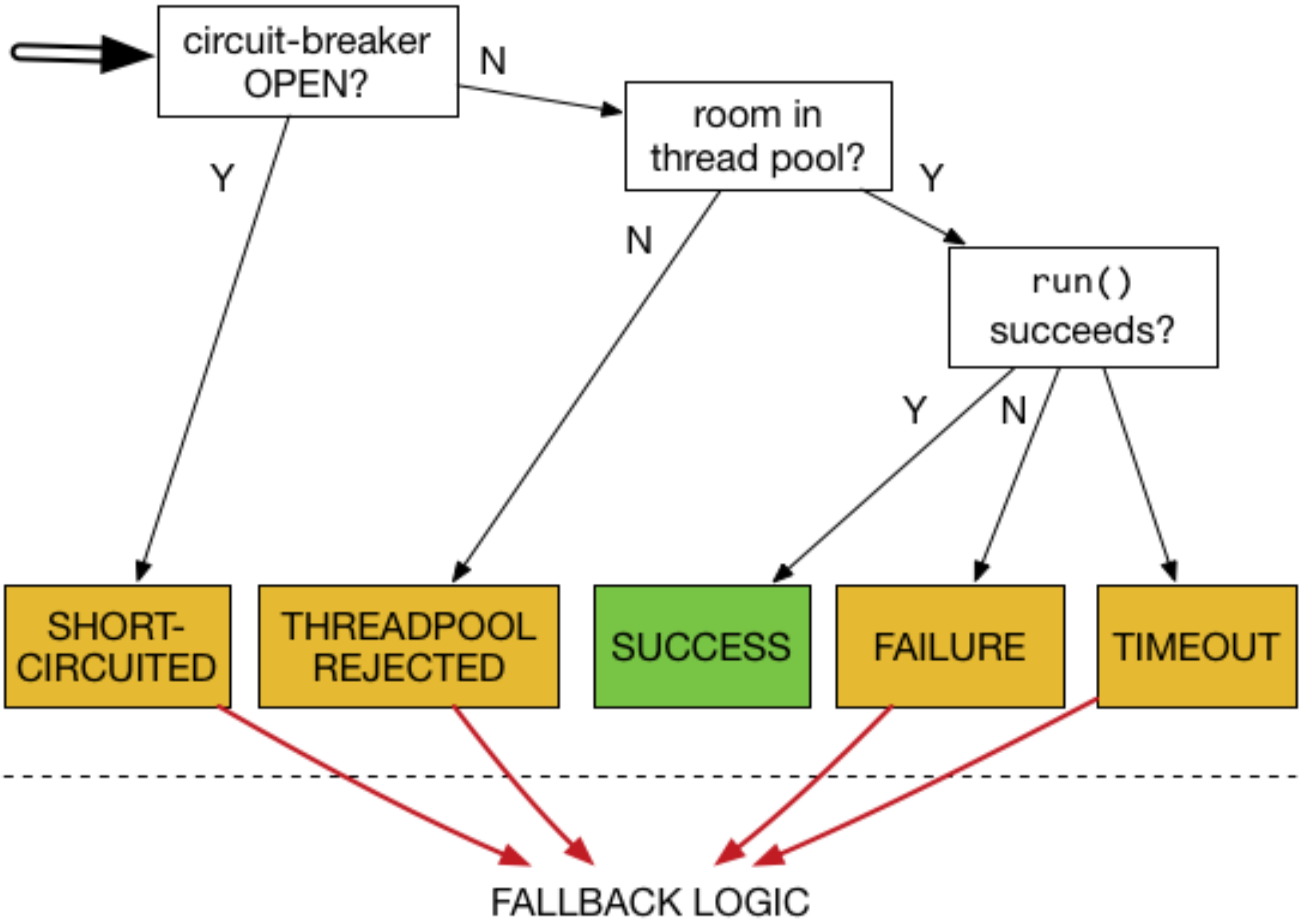


CUSTOMER
thread pool full!

SHORT-CIRCUITED

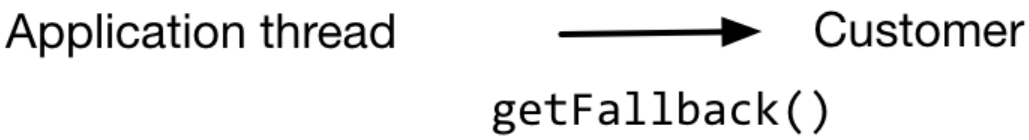
```
class CustLookup extends HystrixCommand<Customer> {
    @Override
    public Customer run() {
        //never runs – circuit is open
        return svc.getOverHttp(customerId);
    }

    @Override
    public Customer getFallback() {
        return Customer.anonymousCustomer();
    }
}
```

SHORT-CIRCUITED



SHORT-CIRCUITED

```
Customer c = cmd.execute()
```



CUSTOMER
threads available

Fallback Handling

- What are the basic error modes?
  - Failure
  - Latency

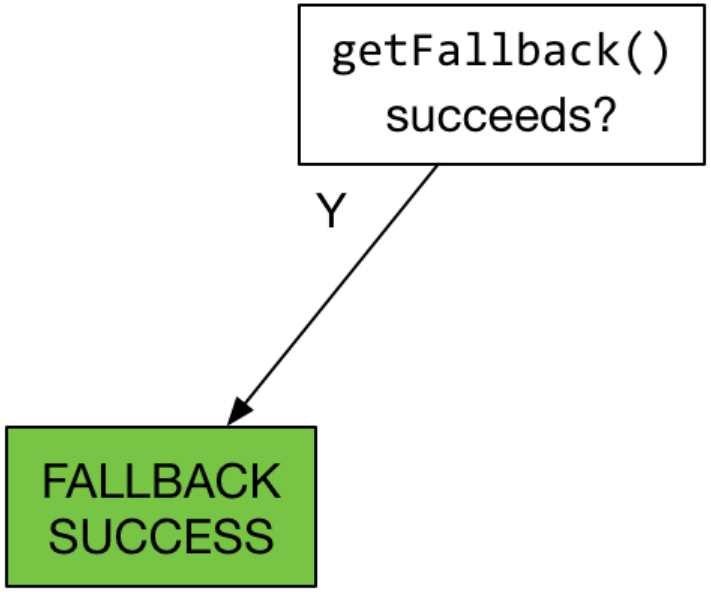
Fallback Handling

- What are the basic error modes?
  - Failure
  - Latency
- We need to be aware of failures in fallback
  - No automatic recovery provided – single fallback is enough

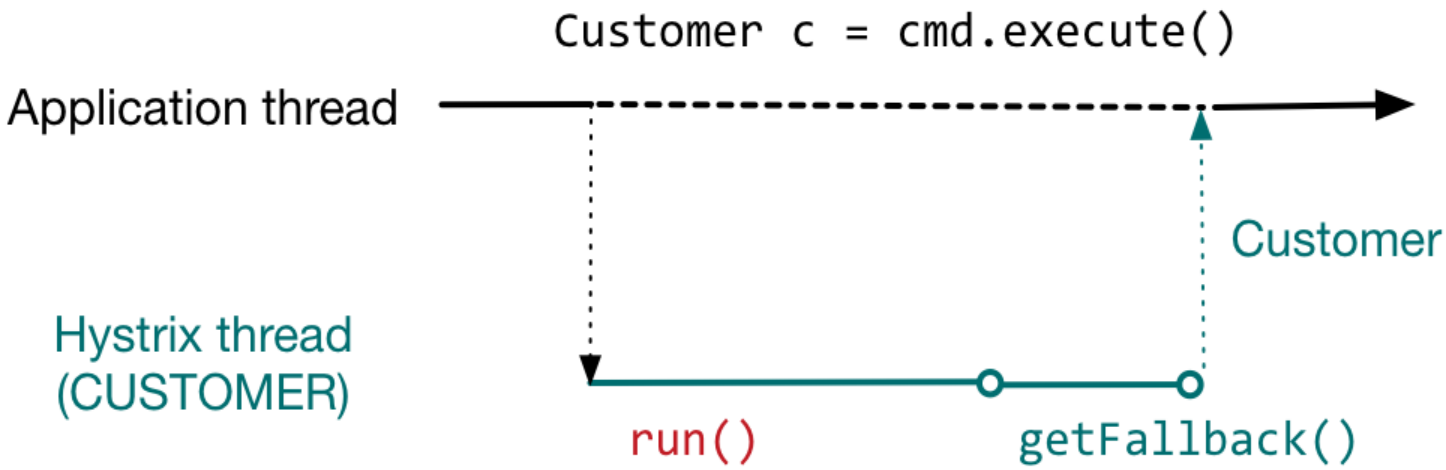
Fallback Handling

- What are the basic error modes?
  - Failure
  - Latency
- We need to be aware of failures in fallback
- We need to protect ourselves from latency in fallback
  - Add a semaphore to fallback to bound concurrency

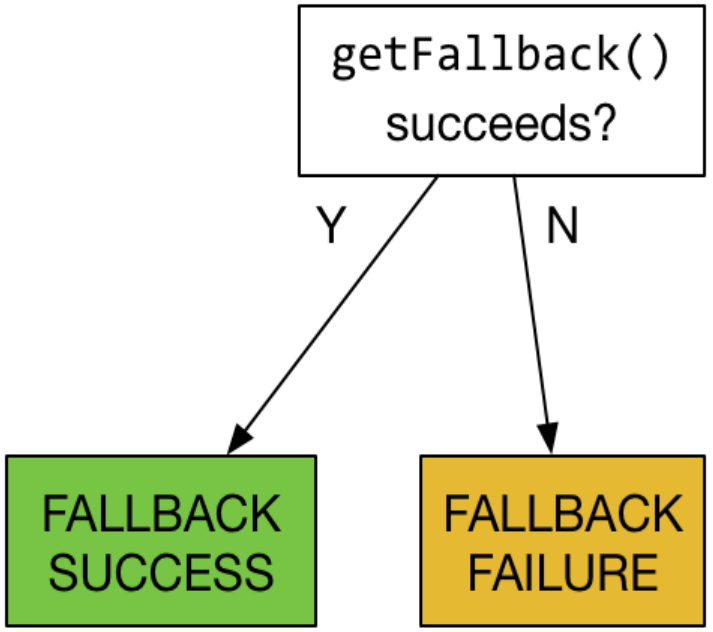
FALLBACK SUCCESS



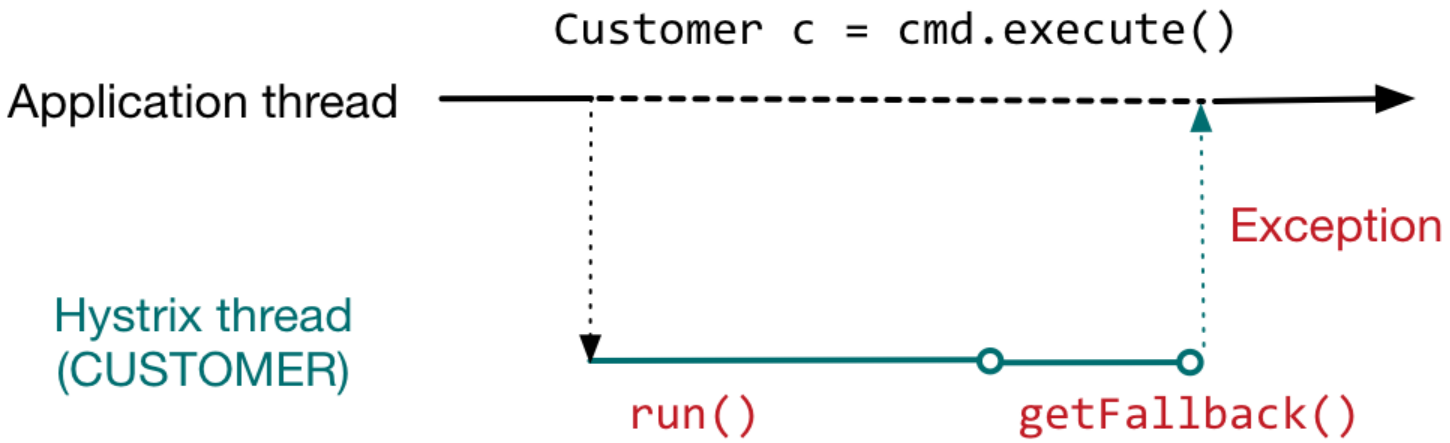
FALLBACK SUCCESS



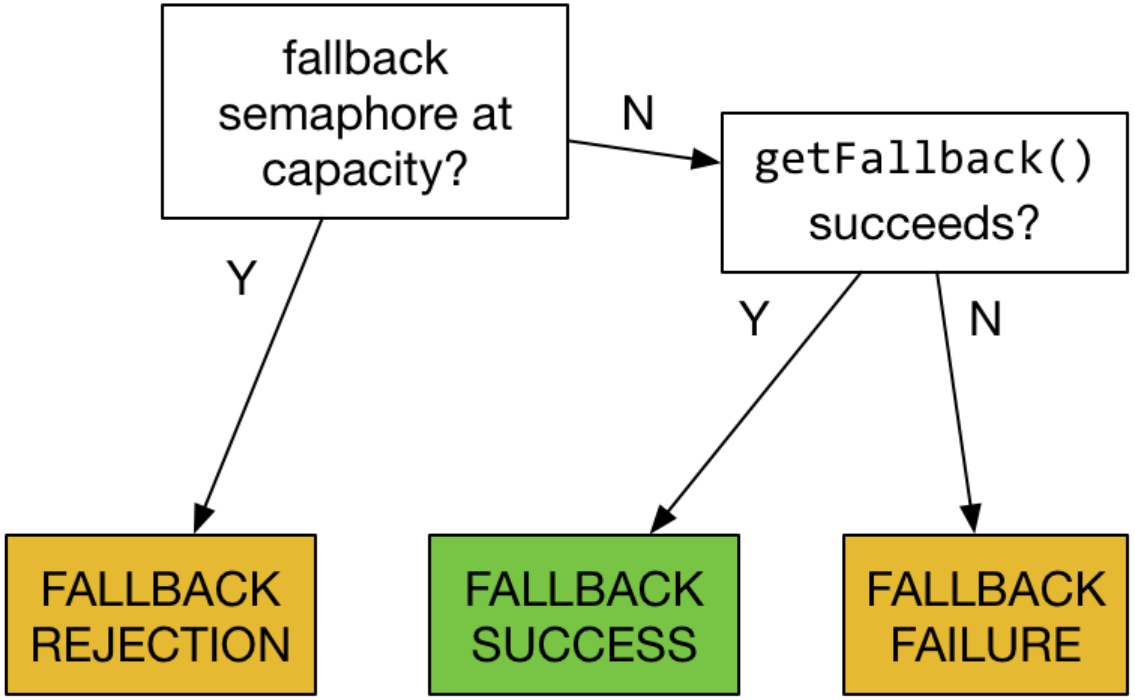
FALLBACK FAILURE



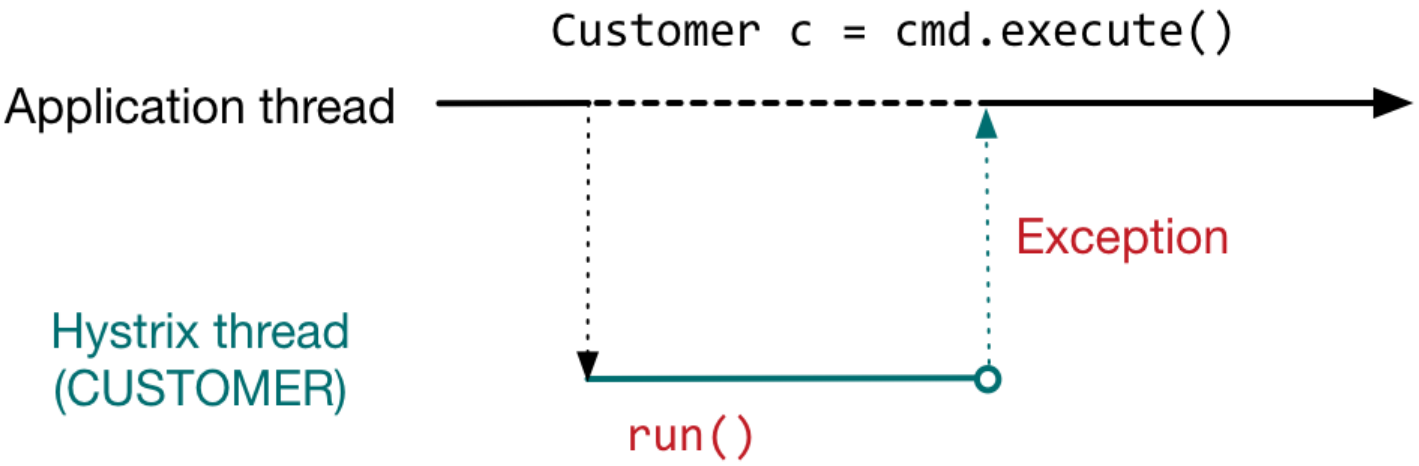
FALLBACK FAILURE



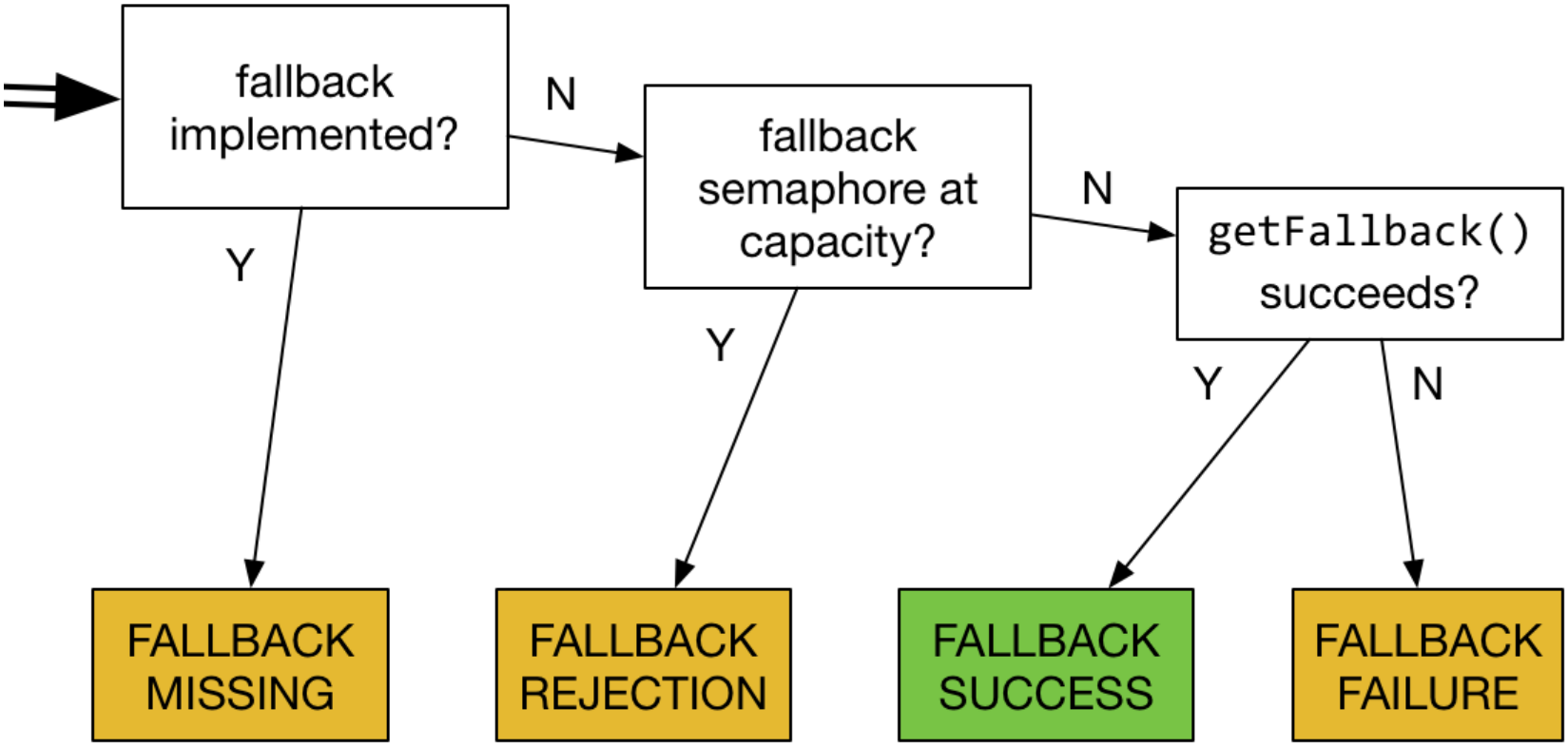
FALLBACK REJECTION



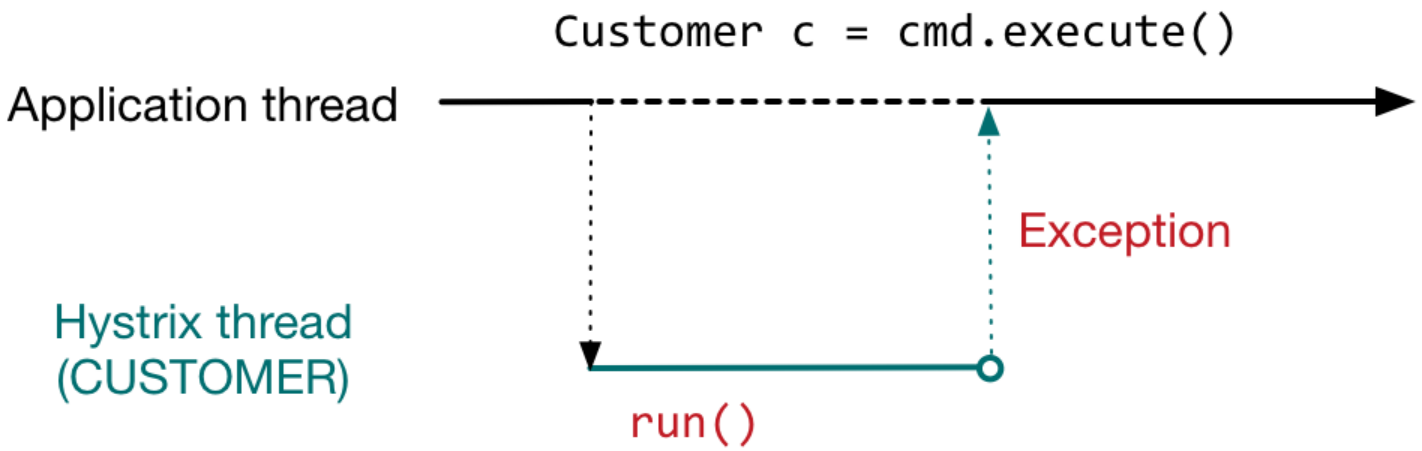
FALLBACK REJECTION

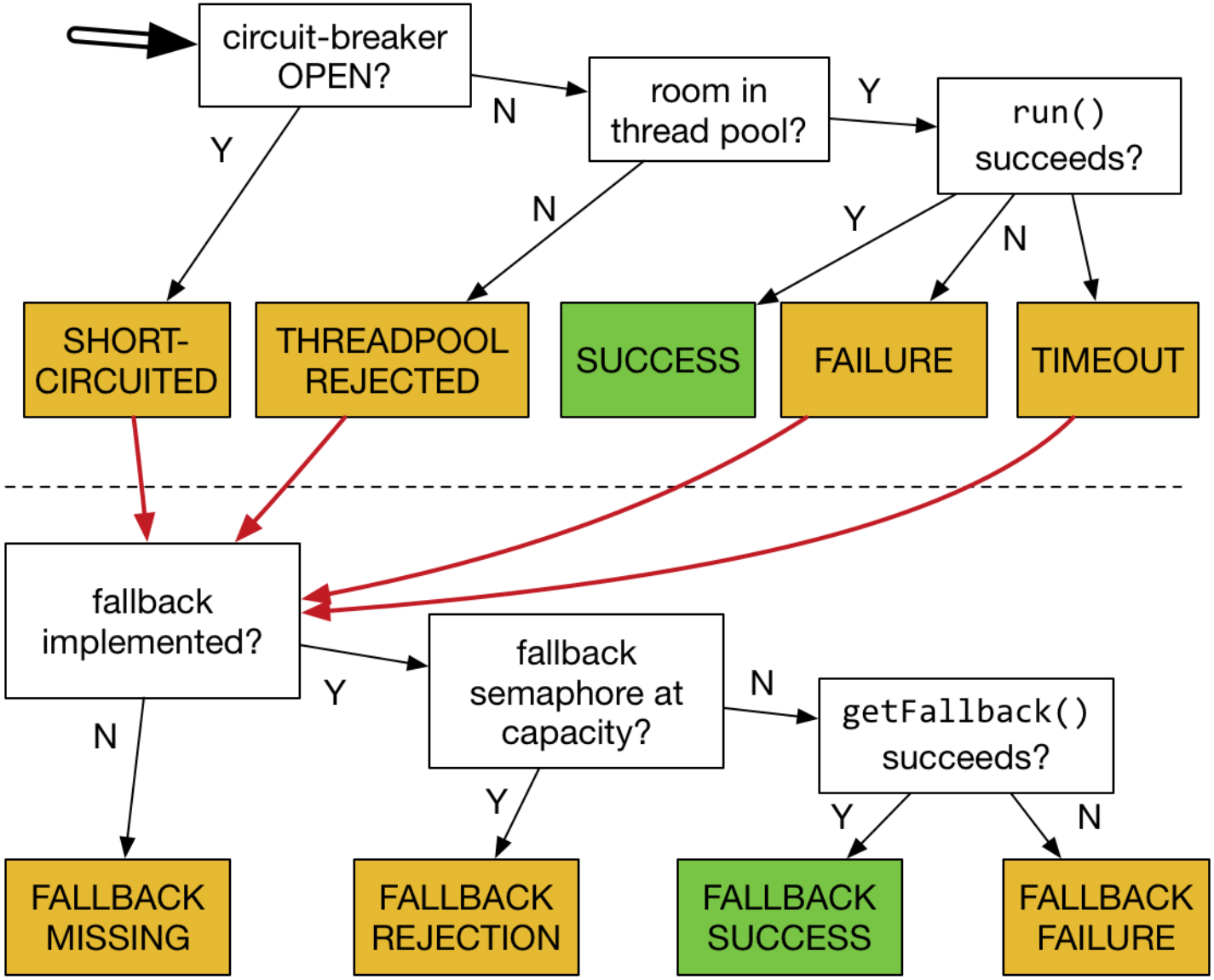


FALLBACK MISSING



FALLBACK MISSING




circuit-breaker OPEN?
Y
N
room in thread pool?
Y
N
run() succeeds?
Y
N
SHORT-CIRCUITED
THREADPOOL REJECTED
SUCCESS
FAILURE
TIMEOUT
fallback implemented?
N
Y
fallback semaphore at capacity?
Y
N
getFallback() succeeds?
Y
N
FALLBACK MISSING
FALLBACK REJECTION
FALLBACK SUCCESS
FALLBACK FAILURE

Hystrix as a building block

- Now we have bounded latency and concurrency, along with a consistent fallback mechanism

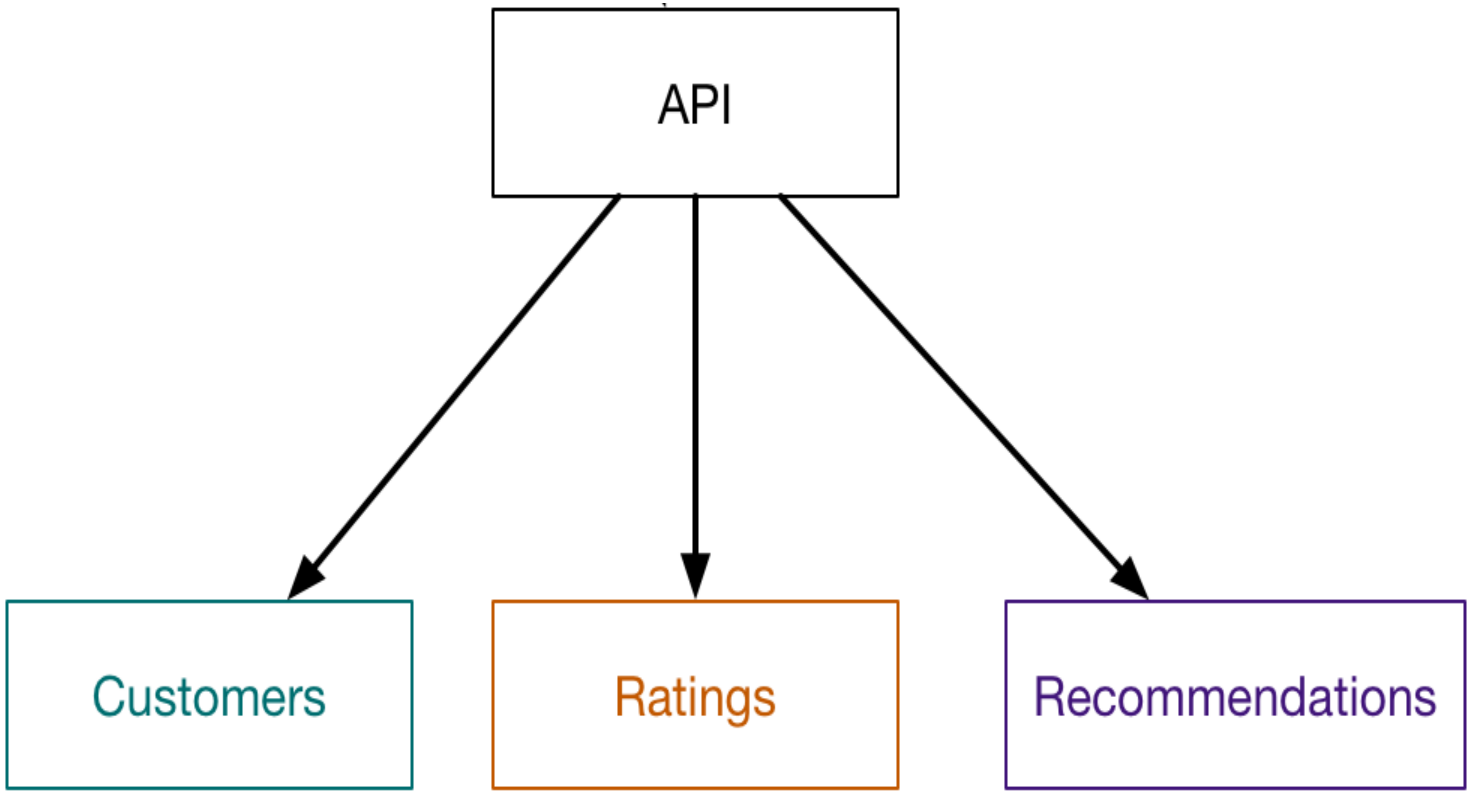
Hystrix as a building block

- Now we have bounded latency and concurrency, along with a consistent fallback mechanism
- In practice within the Netflix API, we have:
  - ~250 commands
  - ~90 thread pools
  - 10s of billions of command executions / day

Goals of this talk

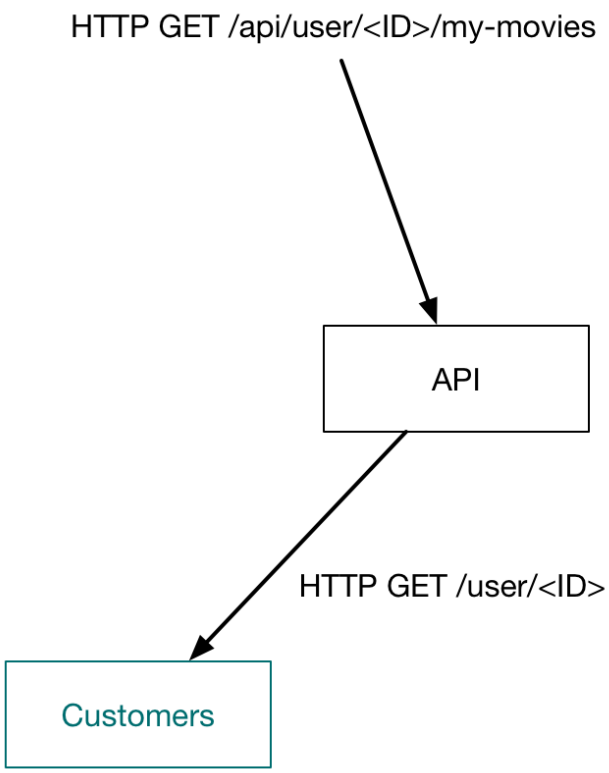
- Philosophical Motivation
  - Why are distributed systems hard?
- Practical Motivation
  - Why do I keep getting paged?
- Solving those problems with Hystrix
  - How does it work?
  - How do I use it in my system?
  - How should a system behave if I use Hystrix?
  - What? Netflix was down for me – what happened there?

Example



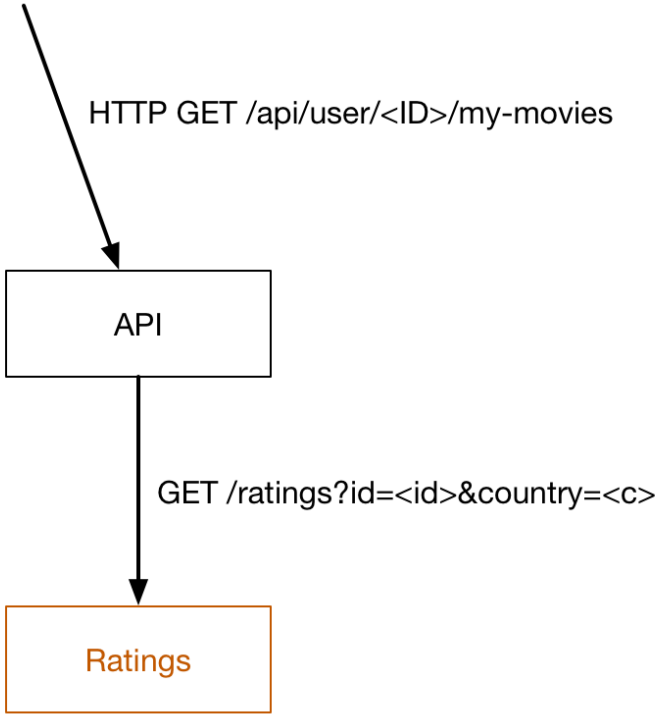
CustomerCommand

- Takes id as arg
- Makes HTTP call to service
- Uses anonymous user as fallback
  - Customer has no personalization



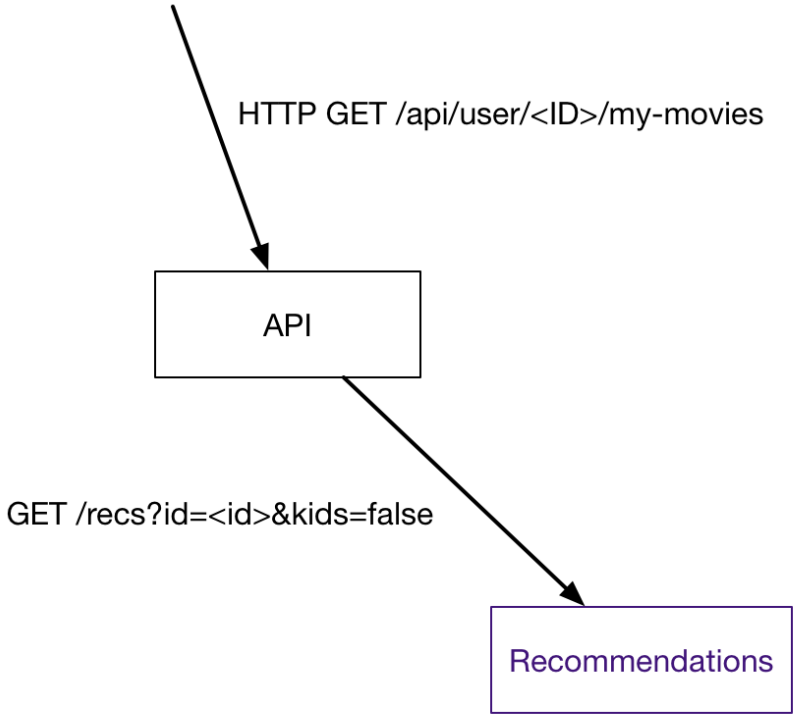
RatingsCommand

- Takes Customer as argument
- Makes HTTP call to service
- Uses empty list for fallback
  - Customer has no ratings



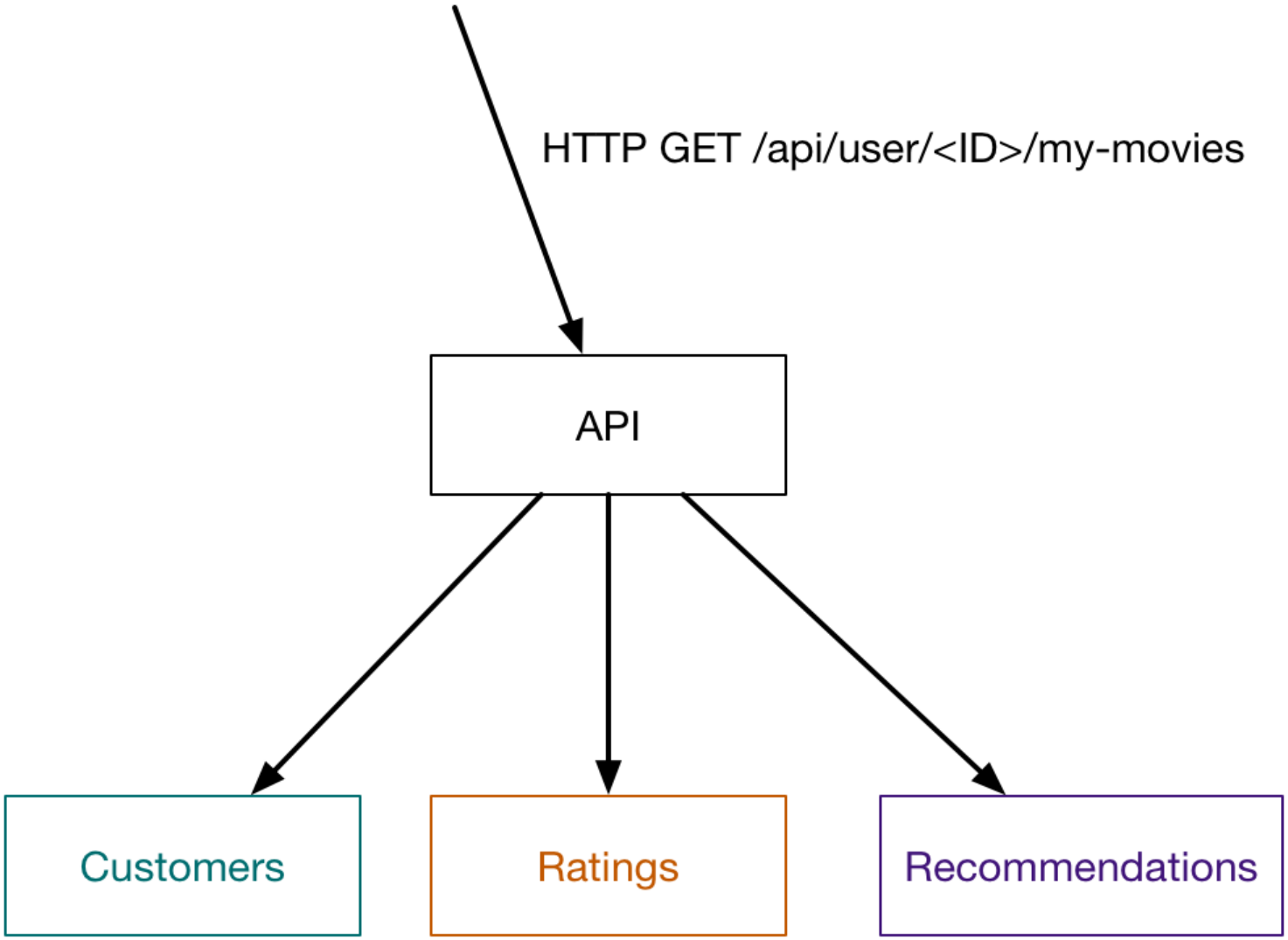
RecommendationsCommand

- Takes Customer as argument
- Makes HTTP call to service
- Uses precomputed list for fallback
  - Customer gets unpersonalized recommendations



Fallbacks in this example

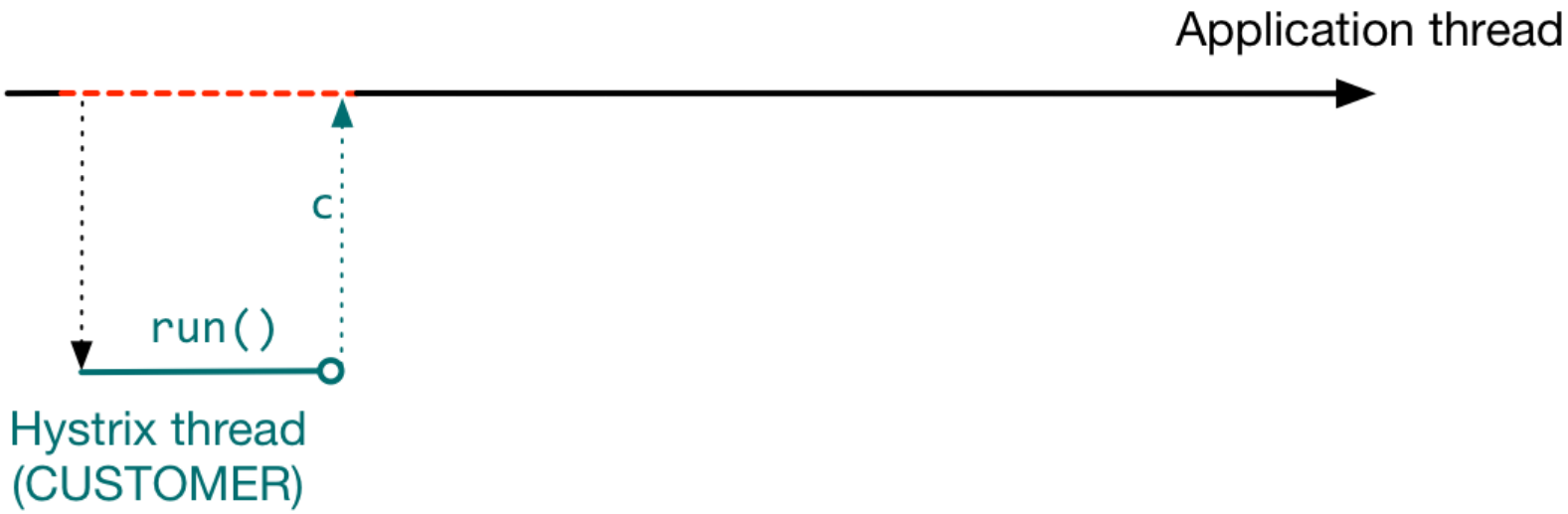
- Fallbacks vary in importance
  - Customer has no personalization
  - Customer gets unpersonalized recommendations
  - Customer has no ratings


HTTP GET /api/user/<ID>/my-movies
API
Customers
Ratings
Recommendations

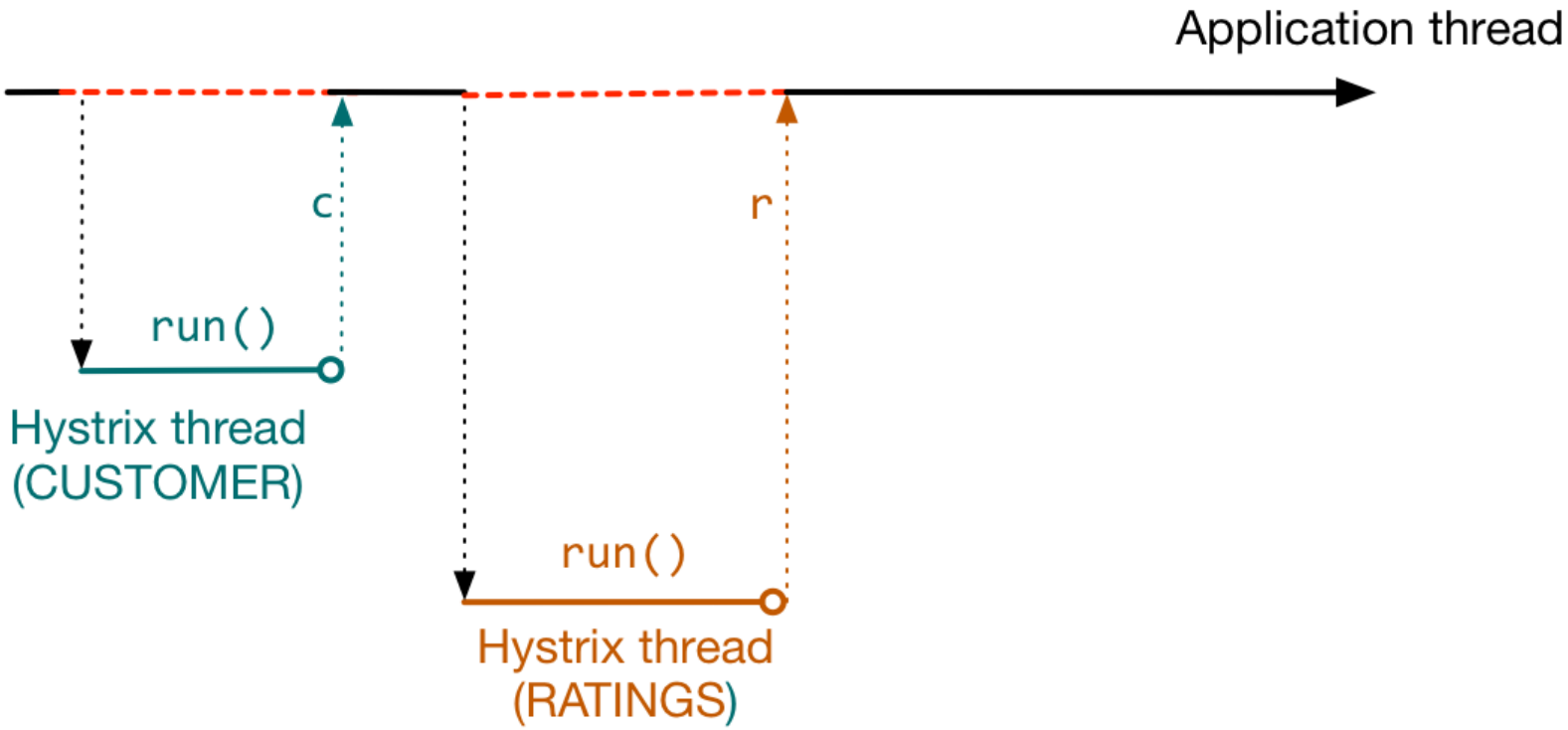
HystrixCommand.execute()

- Blocks application thread on Hystrix thread
- Returns T

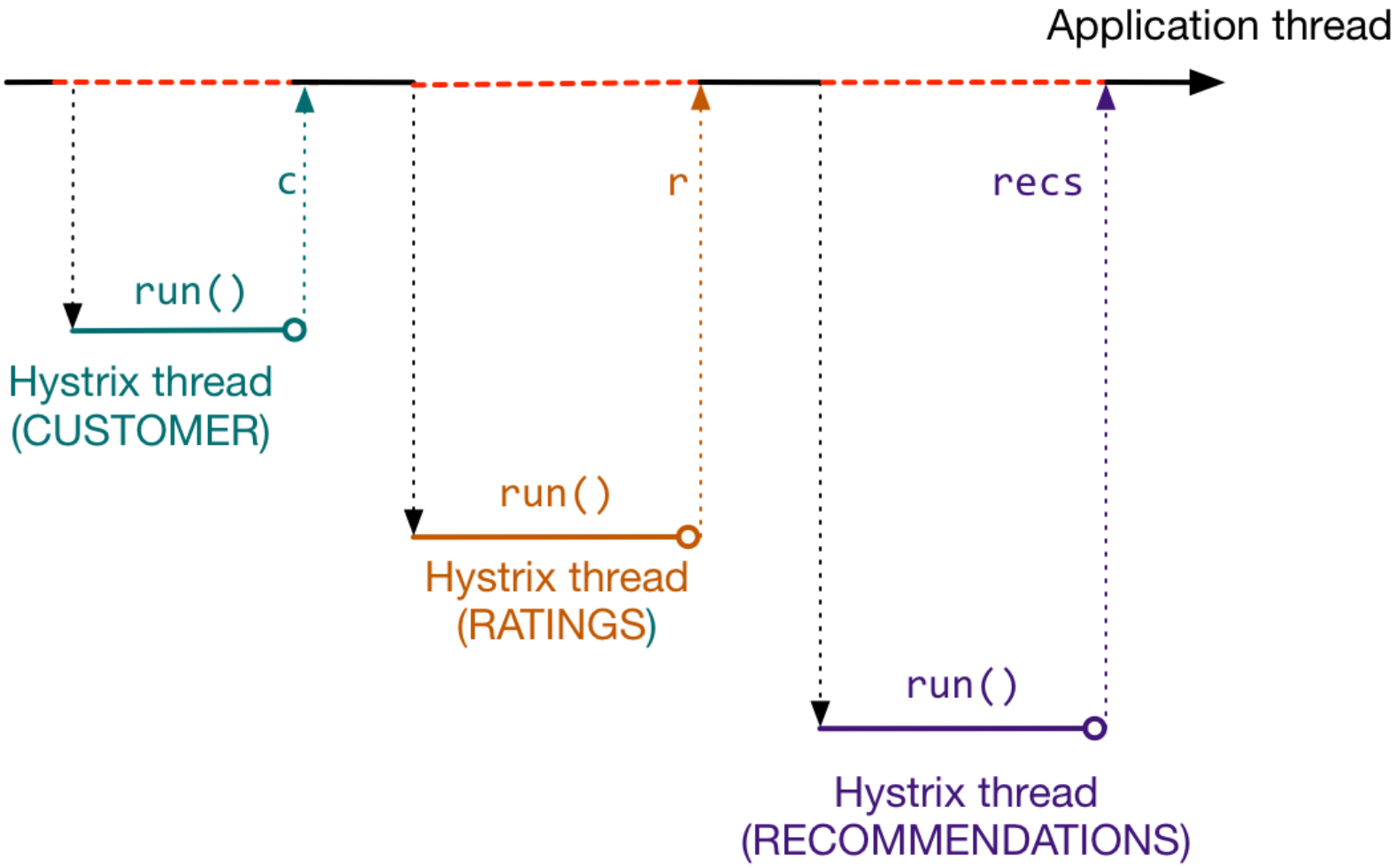
```
1 Customer c = new CustomerCommand(id).execute();
```



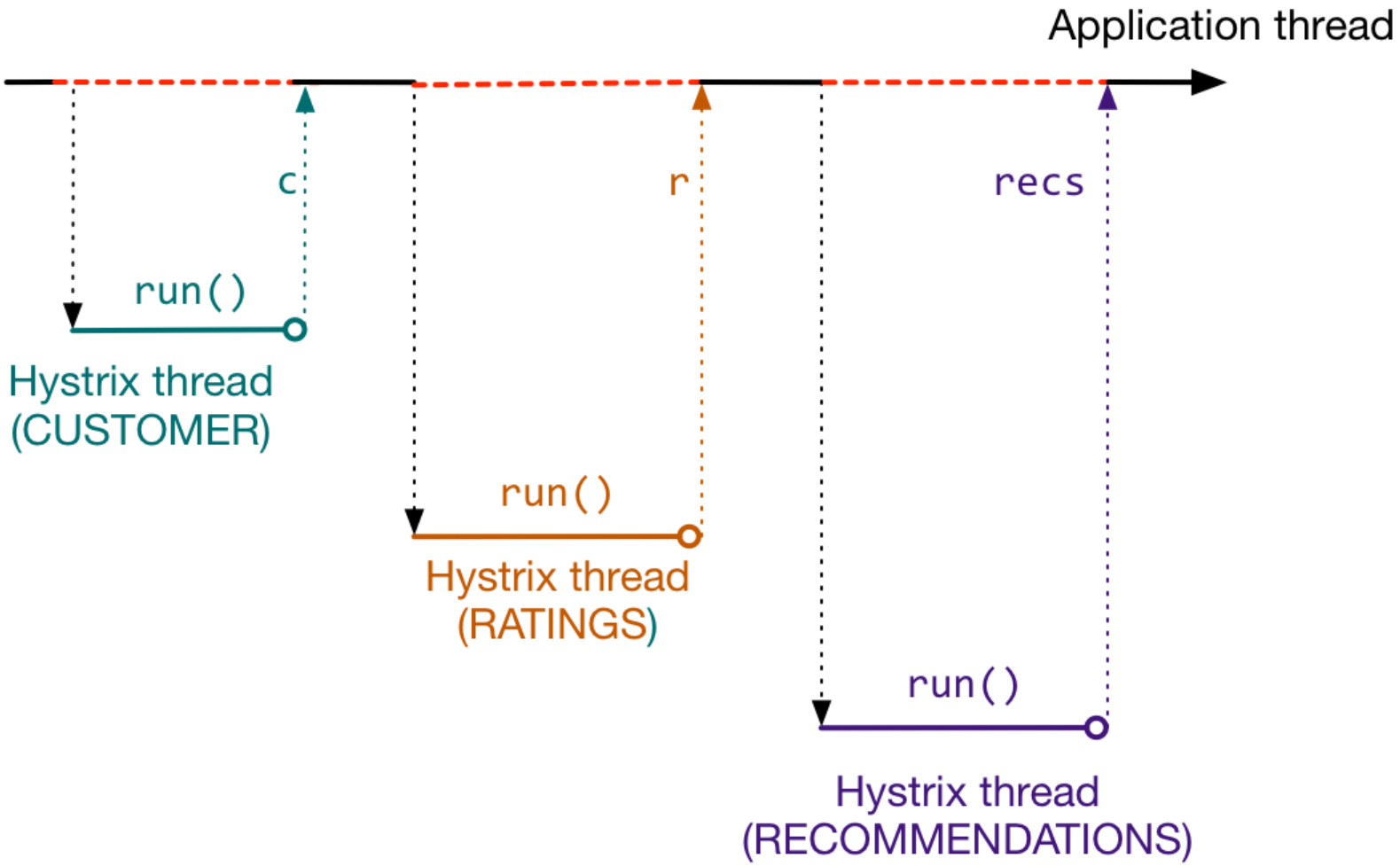
```
1 Customer c = new CustomerCommand(id).execute();
2 Ratings r = new RatingsCommand(c).execute();
```



```
1 Customer c = new CustomerCommand(id).execute();
2 Ratings r = new RatingsCommand(c).execute();
3 Recs recs = new RecsCommand(c).execute();
```



```
1 Customer c = new CustomerCommand(id).execute();
2 Ratings r = new RatingsCommand(c).execute();
3 Recs recs = new RecsCommand(c).execute();
4 return resp(c, r, recs);
```



HystrixCommand.queue()

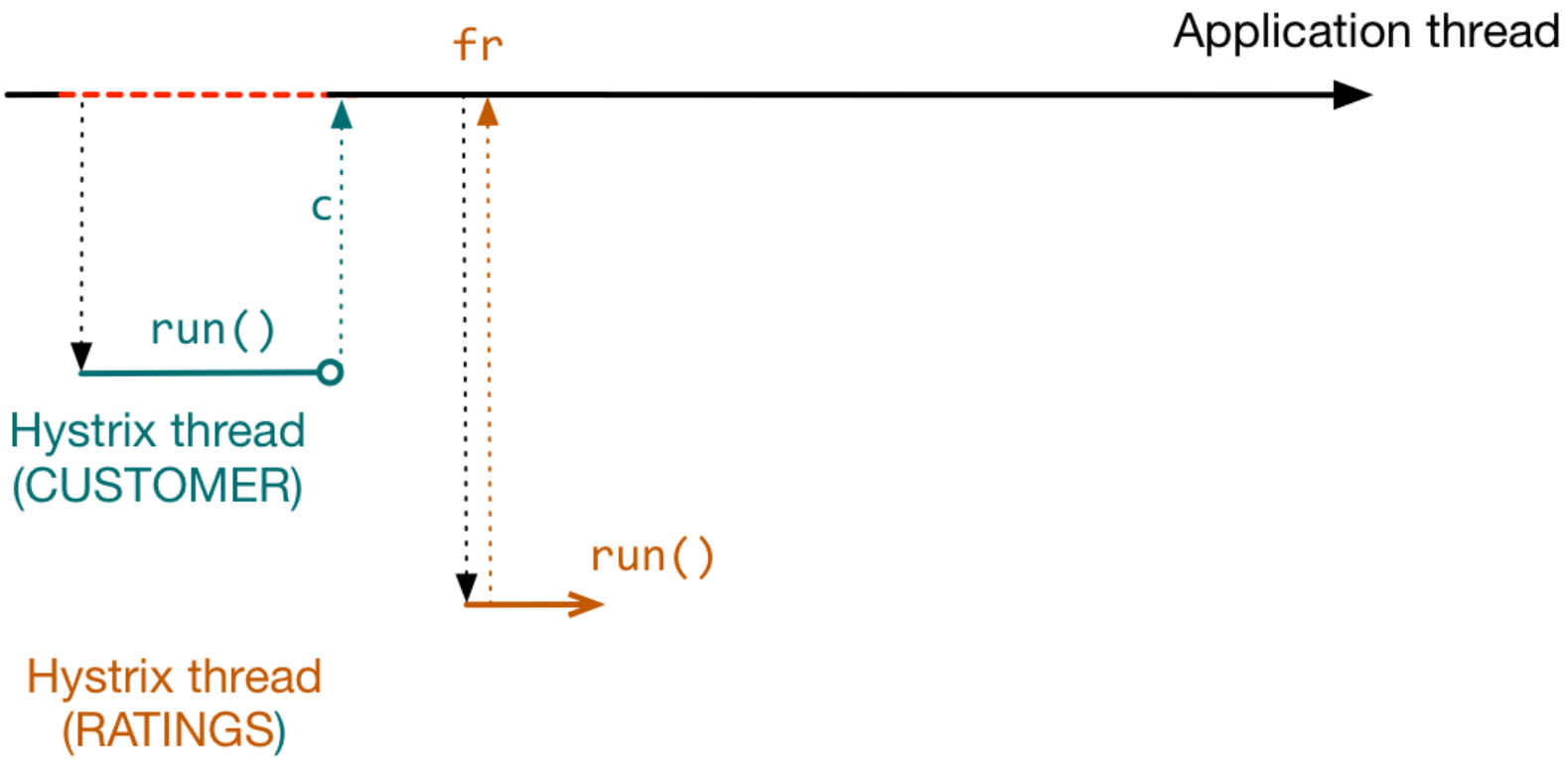
- Does not block application thread
  - Application thread can launch multiple HystrixCommands
- Returns java.util.concurrent.Future to application thread

```
1 Customer c = new CustomerCommand(id).execute();
```

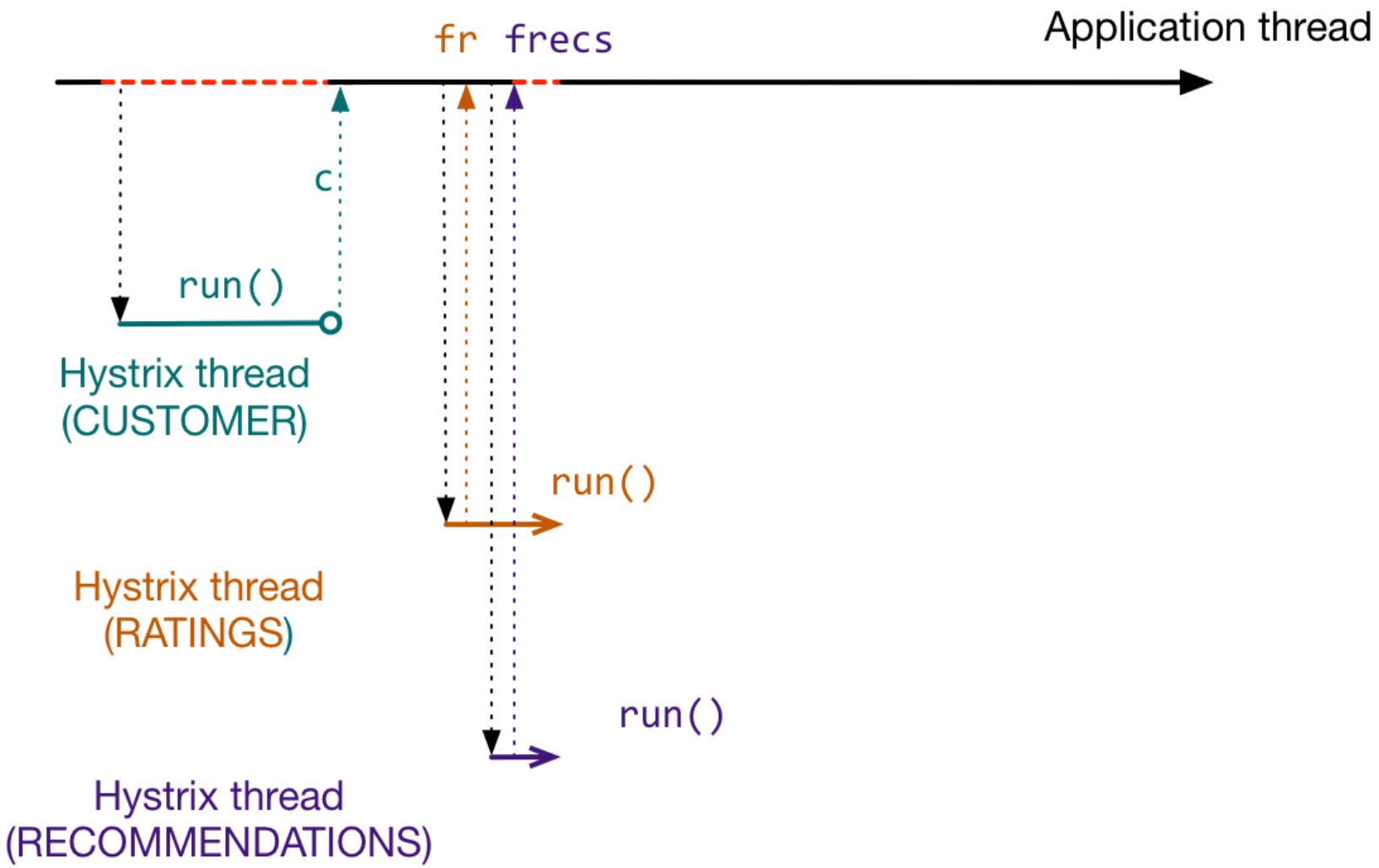
Hystrix thread
(RATINGS)

Hystrix thread
(RECOMMENDATIONS)

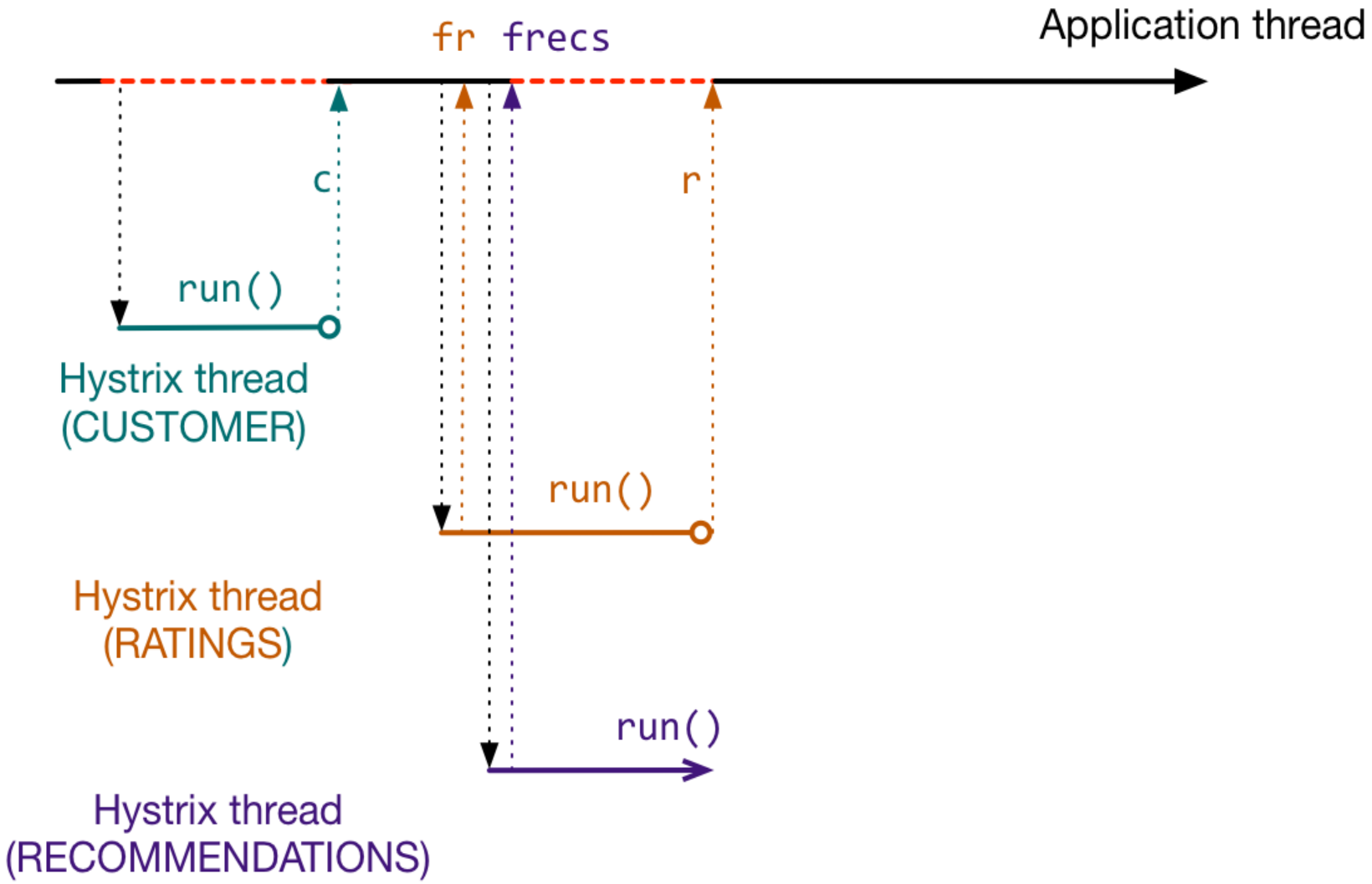
```
1 Customer c = new CustomerCommand(id).execute();
2 Future<Ratings> fr = new RatingsCommand(c).queue();
```



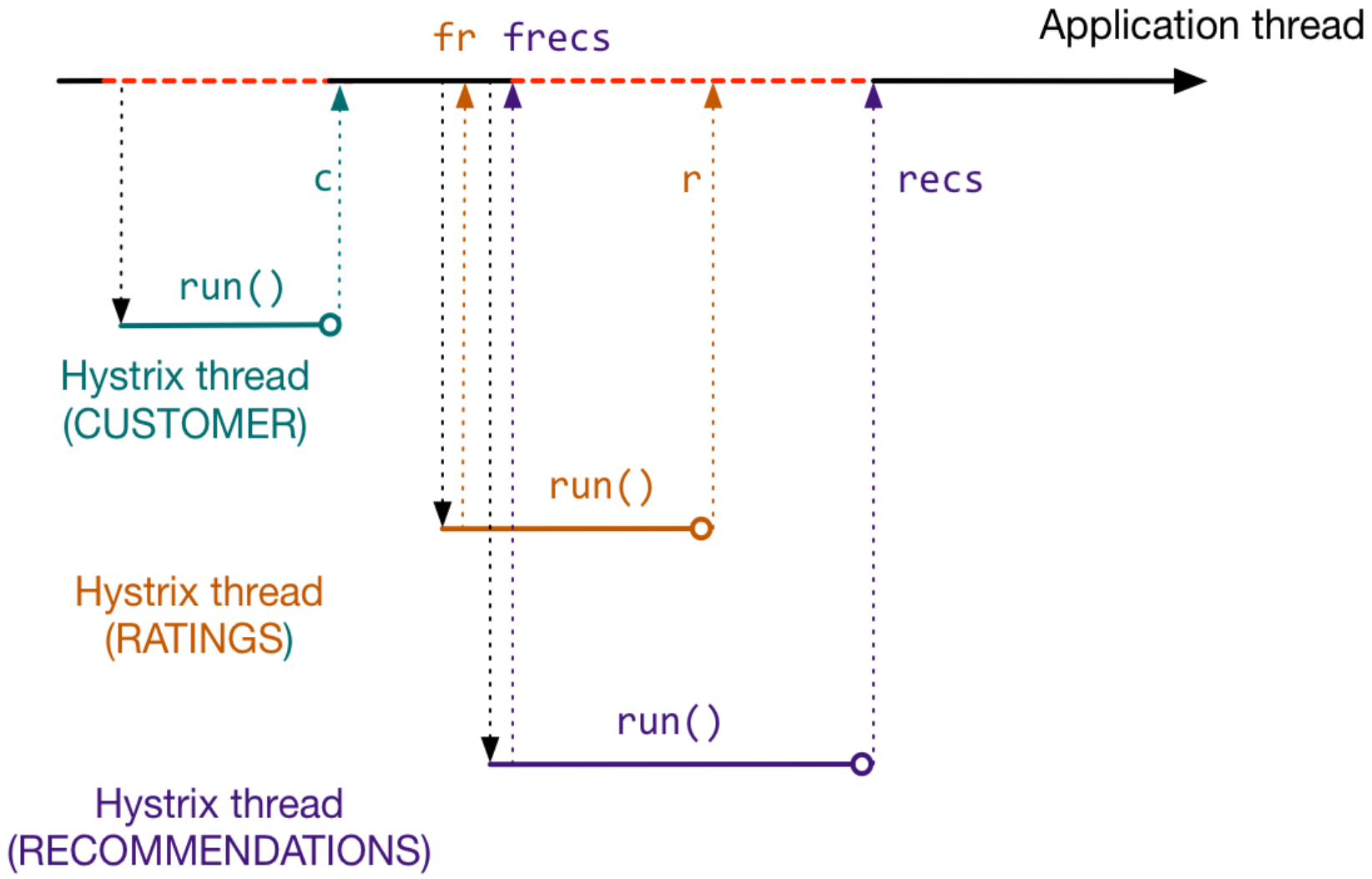
```
1 Customer c = new CustomerCommand(id).execute();
2 Future<Ratings> fr = new RatingsCommand(c).queue();
3 Future<Recs> frecs = new RecsCommand(c).queue();
```



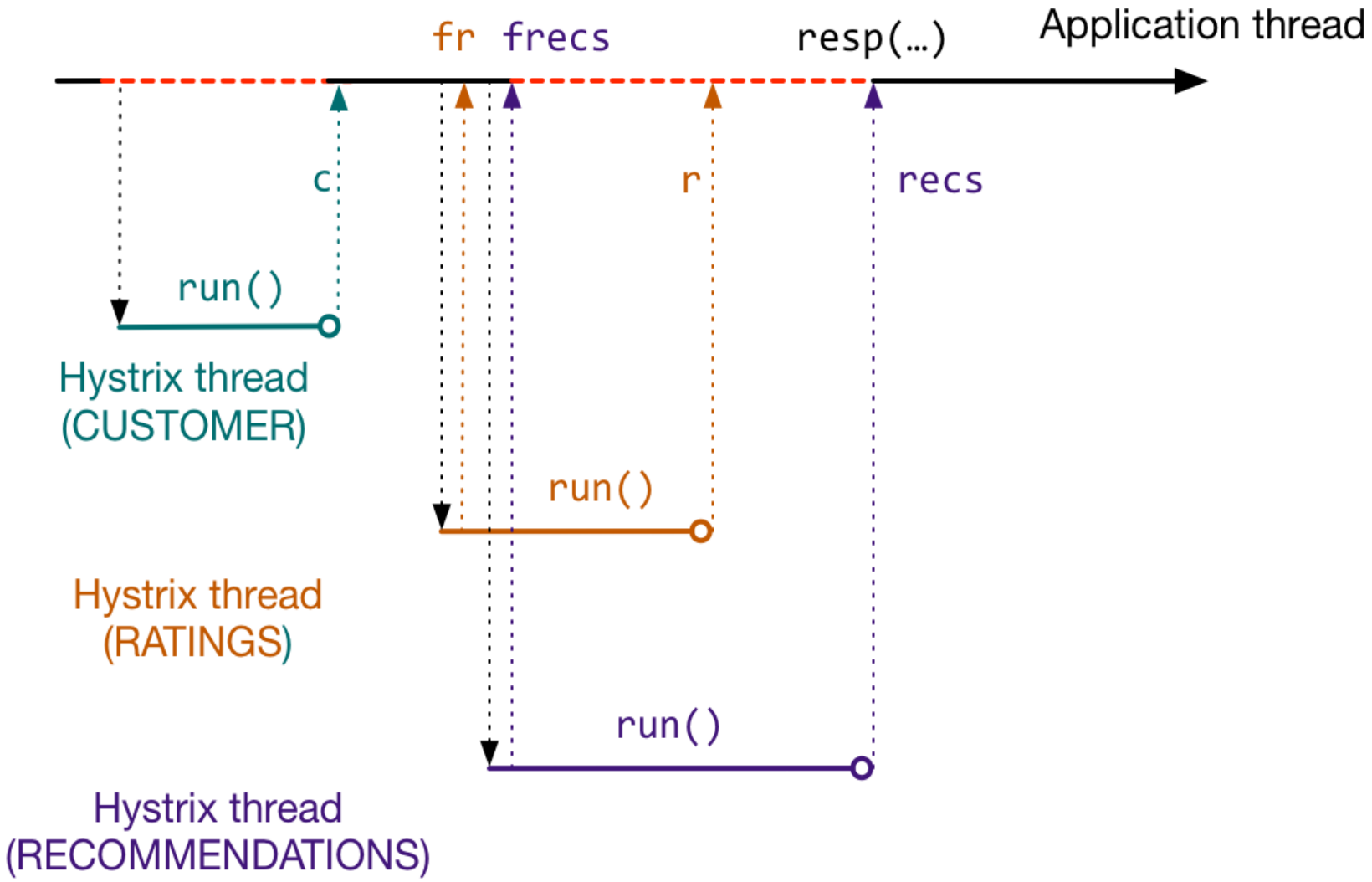
```
1 Customer c = new CustomerCommand(id).execute();
2 Future<Ratings> fr = new RatingsCommand(c).queue();
3 Future<Recs> frecs = new RecsCommand(c).queue();
4 Ratings r = fr.get();
```



```
1 Customer c = new CustomerCommand(id).execute();
2 Future<Ratings> fr = new RatingsCommand(c).queue();
3 Future<Recs> frecs = new RecsCommand(c).queue();
4 Ratings r = fr.get();
5 Recs recs = frecs.get();
```



```
1 Customer c = new CustomerCommand(id).execute();
2 Future<Ratings> fr = new RatingsCommand(c).queue();
3 Future<Recs> frecs = new RecsCommand(c).queue();
4 Ratings r = fr.get();
5 Recs recs = frecs.get();
6 return resp(c, r, recs);
```



HystrixCommand.observe()

- Does not block application thread
- Returns rx.Observable to application thread
- RxJava – reactive stream
  - reactivex.io

```
1 Observable<Customer> o_c = new CustomerCommand(id).observe();
```

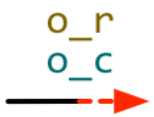
Application thread

Hystrix thread
(CUSTOMER)

Hystrix thread
(RATINGS)

Hystrix thread
(RECOMMENDATIONS)

```
1 Observable<Customer> o_c = new CustomerCommand(id).observe();
2 Observable<Ratings> o_r = o_c.flatMap(customer ->
      new RatingsCommand(customer).observe());
```



Application thread

Hystrix thread
(CUSTOMER)

Hystrix thread
(RATINGS)

Hystrix thread
(RECOMMENDATIONS)

```
1 Observable<Customer> o_c = new CustomerCommand(id).observe();
2 Observable<Ratings> o_r = o_c.flatMap(customer ->
      new RatingsCommand(customer).observe());
3 Observable<Recs> o_recs = o_c.flatMap(customer ->
      new RecsCommand(customer).observe());
```

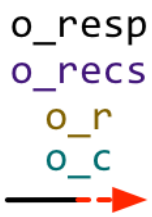
Application thread

Hystrix thread
(CUSTOMER)

Hystrix thread
(RATINGS)

Hystrix thread
(RECOMMENDATIONS)

```
1 Observable<Customer> o_c = new CustomerCommand(id).observe();
2 Observable<Ratings> o_r = o_c.flatMap(customer ->
      new RatingsCommand(customer).observe());
3 Observable<Recs> o_recs = o_c.flatMap(customer ->
      new RecsCommand(customer).observe());
4 Observable<Response> o_resp =
      Observable.zip(o_c, o_r, o_recs, this::resp);
```



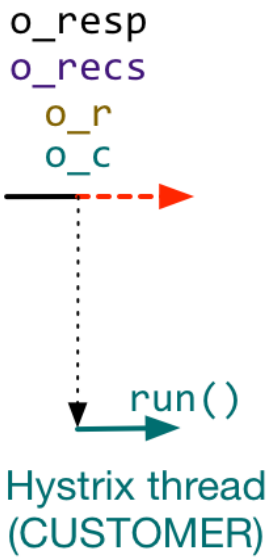
Application thread

Hystrix thread
(CUSTOMER)

Hystrix thread
(RATINGS)

Hystrix thread
(RECOMMENDATIONS)

```
1 Observable<Customer> o_c = new CustomerCommand(id).observe();
2 Observable<Ratings> o_r = o_c.flatMap(customer ->
      new RatingsCommand(customer).observe());
3 Observable<Recs> o_recs = o_c.flatMap(customer ->
      new RecsCommand(customer).observe());
4 Observable<Response> o_resp =
      Observable.zip(o_c, o_r, o_recs, this::resp);
5 return o_resp.toBlocking().single();
```

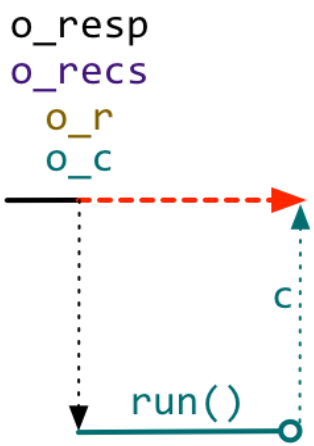


Application thread

Hystrix thread
(RATINGS)

Hystrix thread
(RECOMMENDATIONS)

```
1 Observable<Customer> o_c = new CustomerCommand(id).observe();
2 Observable<Ratings> o_r = o_c.flatMap(customer ->
      new RatingsCommand(customer).observe());
3 Observable<Recs> o_recs = o_c.flatMap(customer ->
      new RecsCommand(customer).observe());
4 Observable<Response> o_resp =
      Observable.zip(o_c, o_r, o_recs, this::resp);
5 return o_resp.toBlocking().single();
```



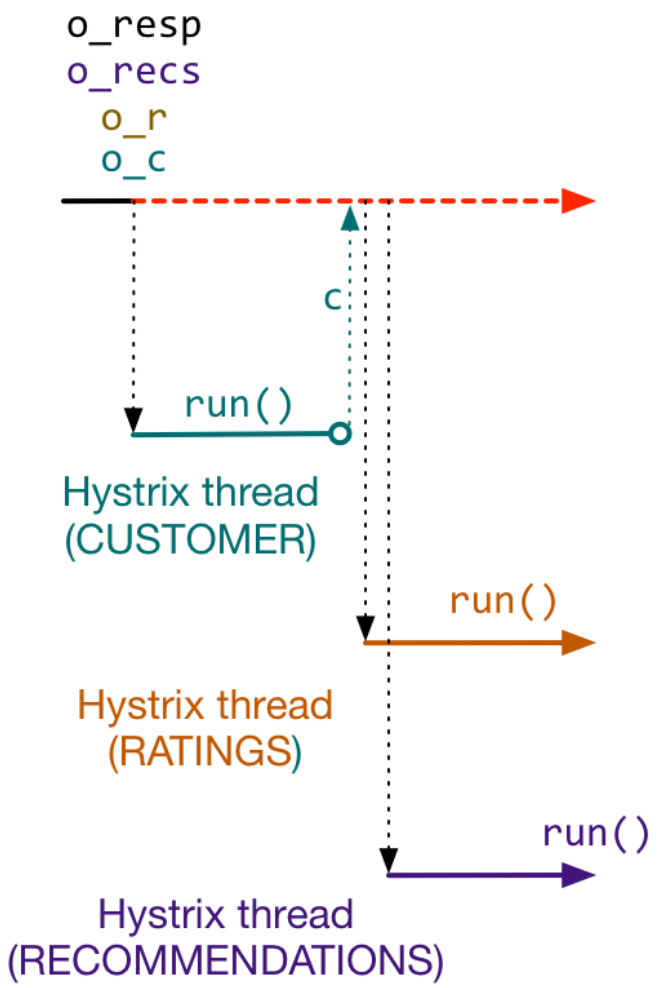
Application thread

Hystrix thread
(CUSTOMER)

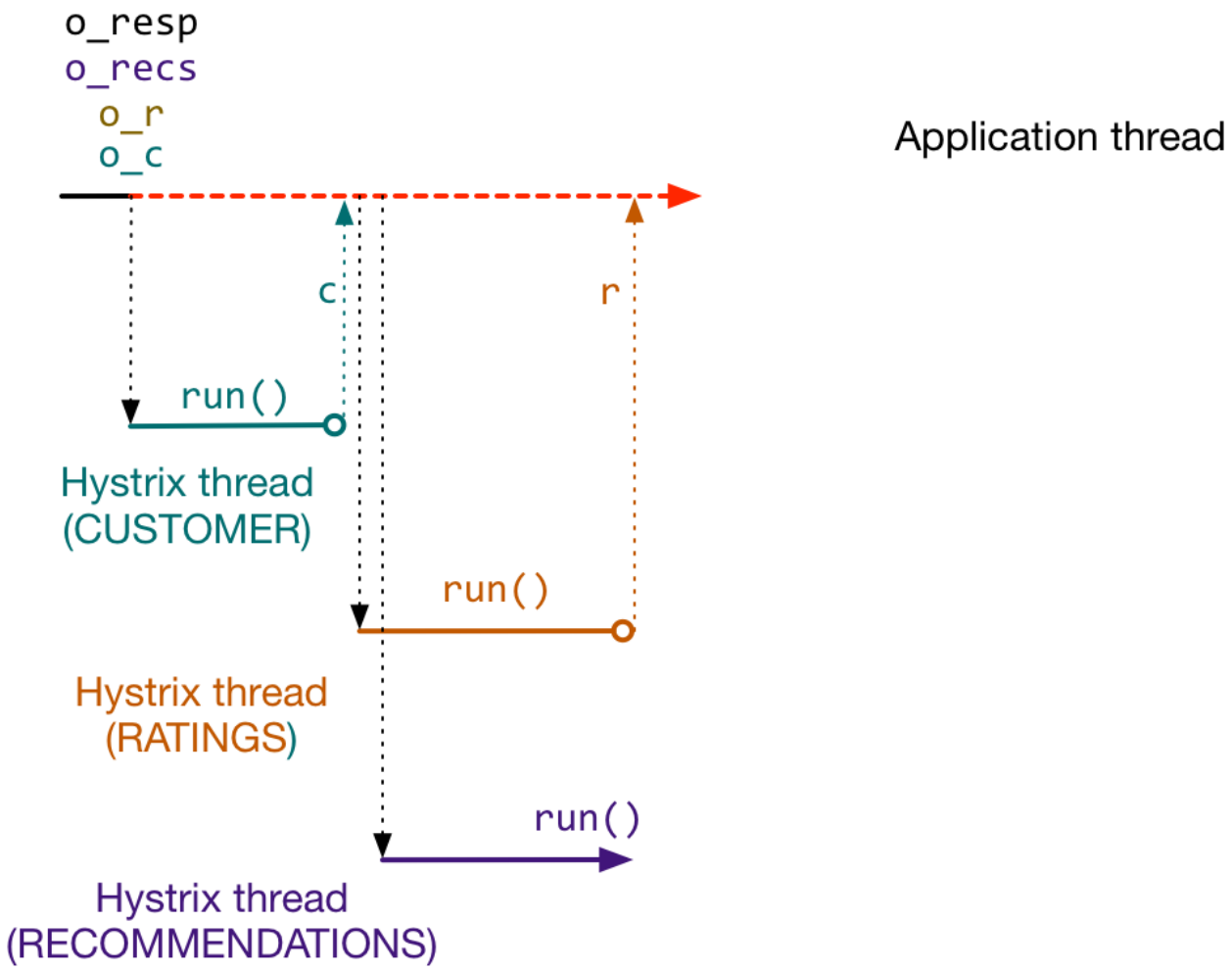
Hystrix thread
(RATINGS)

Hystrix thread
(RECOMMENDATIONS)

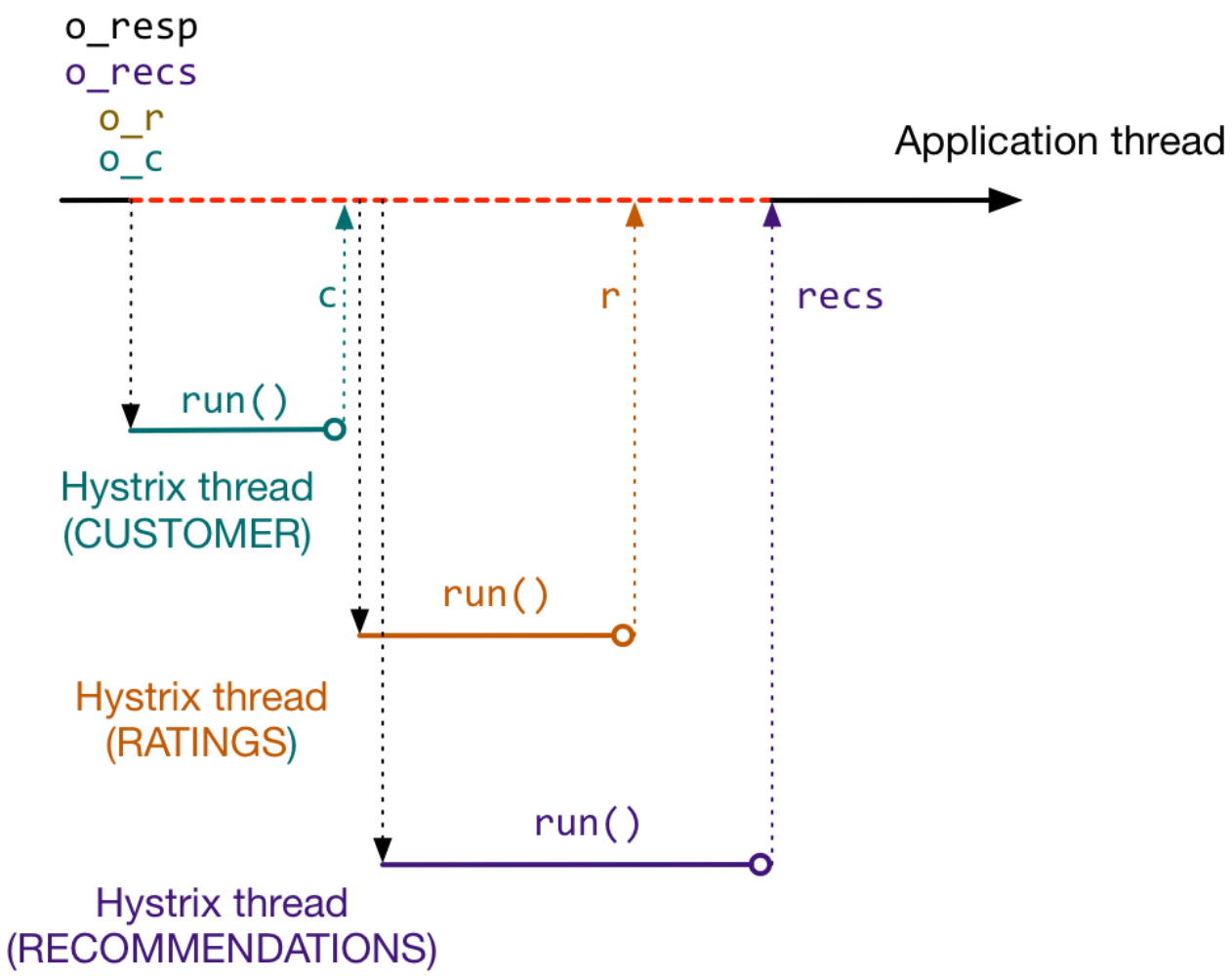
```
1 Observable<Customer> o_c = new CustomerCommand(id).observe();
2 Observable<Ratings> o_r = o_c.flatMap(customer ->
      new RatingsCommand(customer).observe());
3 Observable<Recs> o_recs = o_c.flatMap(customer ->
      new RecsCommand(customer).observe());
4 Observable<Response> o_resp =
      Observable.zip(o_c, o_r, o_recs, this::resp);
5 return o_resp.toBlocking().single();
```



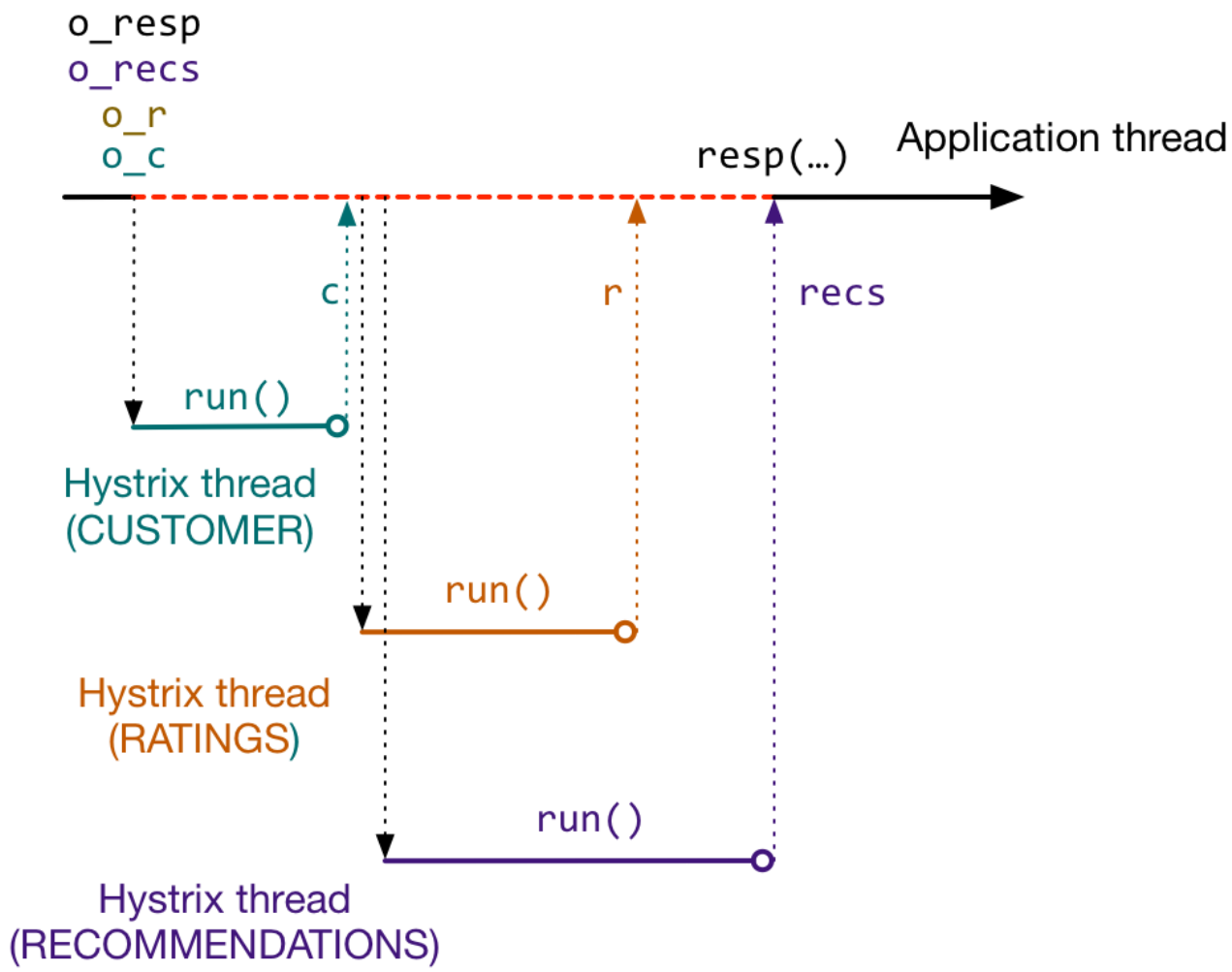
```
1 Observable<Customer> o_c = new CustomerCommand(id).observe();
2 Observable<Ratings> o_r = o_c.flatMap(customer ->
      new RatingsCommand(customer).observe());
3 Observable<Recs> o_recs = o_c.flatMap(customer ->
      new RecsCommand(customer).observe());
4 Observable<Response> o_resp =
      Observable.zip(o_c, o_r, o_recs, this::resp);
5 return o_resp.toBlocking().single();
```



```
1 Observable<Customer> o_c = new CustomerCommand(id).observe();
2 Observable<Ratings> o_r = o_c.flatMap(customer ->
      new RatingsCommand(customer).observe());
3 Observable<Recs> o_recs = o_c.flatMap(customer ->
      new RecsCommand(customer).observe());
4 Observable<Response> o_resp =
      Observable.zip(o_c, o_r, o_recs, this::resp);
5 return o_resp.toBlocking().single();
```



```
1 Observable<Customer> o_c = new CustomerCommand(id).observe();
2 Observable<Ratings> o_r = o_c.flatMap(customer ->
      new RatingsCommand(customer).observe());
3 Observable<Recs> o_recs = o_c.flatMap(customer ->
      new RecsCommand(customer).observe());
4 Observable<Response> o_resp =
      Observable.zip(o_c, o_r, o_recs, this::resp);
5 return o_resp.toBlocking().single();
```



Goals of this talk

- Philosophical Motivation
  - Why are distributed systems hard?
- Practical Motivation
  - Why do I keep getting paged?
- Solving those problems with Hystrix
  - How does it work?
  - How do I use it in my system?
  - How should a system behave if I use Hystrix?
  - What? Netflix was down for me – what happened there?

Hystrix Metrics

- Hystrix provides a nice semantic layer to model
  - Abstracts details of I/O mechanism
    - HTTP?
    - UDP?
    - Postgres?
    - Cassandra?
    - Memcached?
    - …

Hystrix Metrics

- Hystrix provides a nice semantic layer to model
  - Abstracts details of I/O mechanism
- Can capture event counts and event latencies

Hystrix Metrics

- Can be aggregated for historical time-series
  - hystrix-servo-metrics-publisher

Friday, December 12, 2014

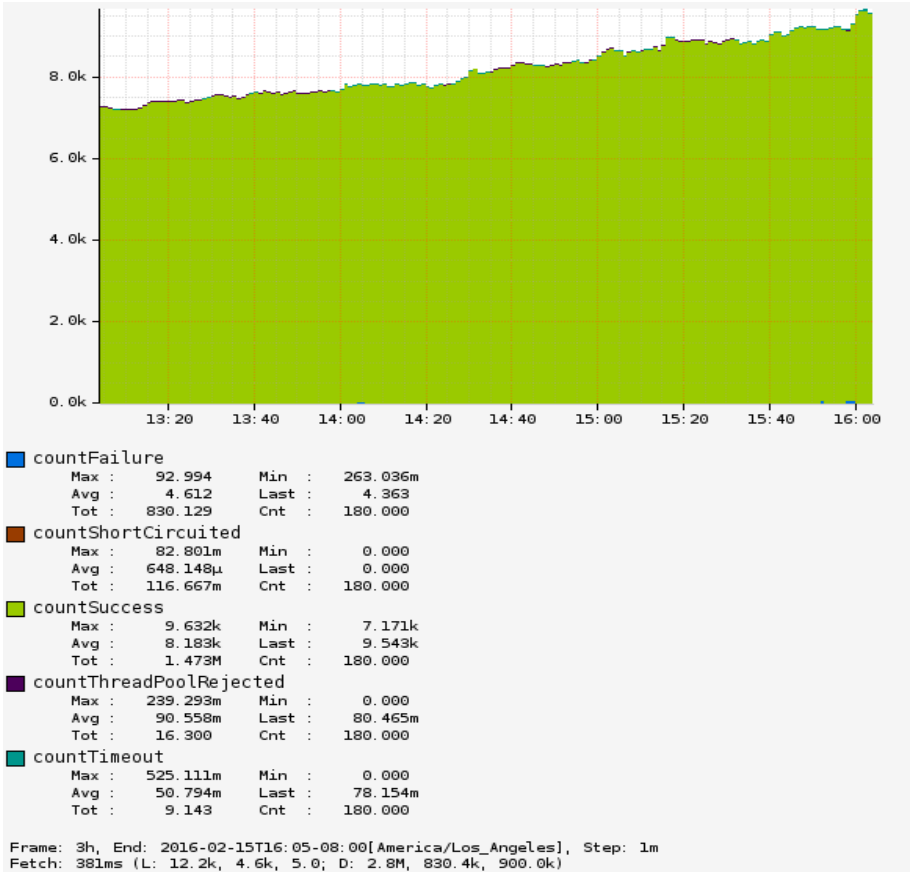
Introducing Atlas: Netflix's Primary Telemetry Platform

Various previous Tech Blog posts have referred to our centralized monitoring system, and we've presented at least one talk about it previously. Today, we want to both discuss the platform and ecosystem we built for time-series telemetry and its capabilities and announce the open-sourcing of its underlying foundation.

Source: http://techblog.netflix.com/2014/12/introducing-atlas-netflixs-primary.html

Hystrix Metrics

- Can be aggregated for historical time-series
  - hystrix-servo-metrics-publisher



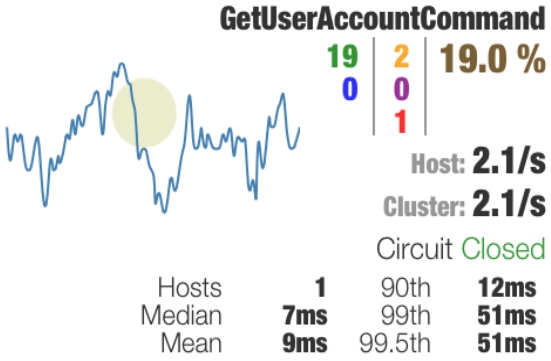
Hystrix Metrics

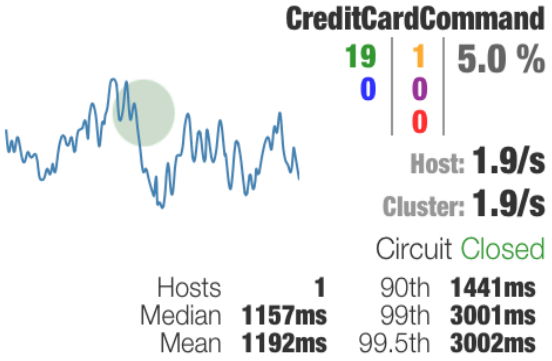
- Can be streamed off-box
  - hystrix-metrics-event-stream

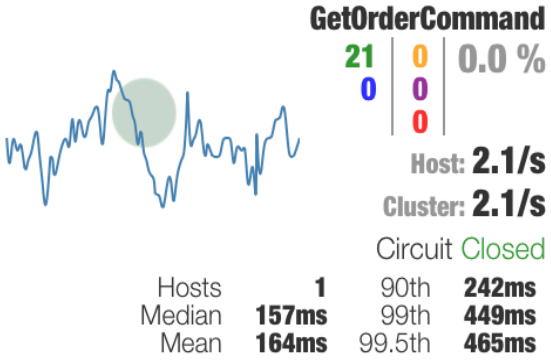

```
data: {"type":"HystrixCommand","name":"CreditCardCommand","group":"CreditCard","currentTime":1455581661051,"isCircuitBreakerOpen":f
alse,"errorPercentage":0,"errorCount":0,"requestCount":1,"rollingCountBadRequests":0,"rollingCountCollapsedRequests":0,"rollingCoun
tEmit":0,"rollingCountExceptionsThrown":0,"rollingCountFailure":0,"rollingCountFallbackEmit":0,"rollingCountFallbackFailure":0,"rol
lingCountFallbackMissing":0,"rollingCountFallbackRejection":0,"rollingCountFallbackSuccess":0,"rollingCountResponsesFromCache":0,"r
ollingCountSemaphoreRejected":0,"rollingCountShortCircuited":0,"rollingCountSuccess":1,"rollingCountThreadPoolRejected":0,"rollingC
ountTimeout":0,"currentConcurrentExecutionCount":0,"rollingMaxConcurrentExecutionCount":1,"latencyExecute_mean":1486,"latencyExecut
e":{"0":1486,"25":1486,"50":1486,"75":1486,"90":1486,"95":1486,"99":1486,"99.5":1486,"100":1486},"propertyValue_circuitBreakerReque
stVolumeThreshold":20,"propertyValue_circuitBreakerSleepWindowInMilliseconds":5000,"propertyValue_circuitBreakerErrorThresholdPerce
ntage":50,"propertyValue_circuitBreakerForceOpen":false,"propertyValue_circuitBreakerForceClosed":false,"propertyValue_circuitBreak
erEnabled":true,"propertyValue_executionIsolationStrategy":"THREAD","propertyValue_executionIsolationThreadTimeoutInMilliseconds":3
000,"propertyValue_executionTimeoutInMilliseconds":3000,"propertyValue_executionIsolationThreadInterruptOnTimeout":true,"propertyVa
lue_executionIsolationThreadPoolKeyOverride":null,"propertyValue_executionIsolationSemaphoreMaxConcurrentRequests":10,"propertyValu
e_fallbackIsolationSemaphoreMaxConcurrentRequests":10,"propertyValue_metricsRollingStatisticalWindowInMilliseconds":10000,"property
Value_requestCacheEnabled":true,"propertyValue_requestLogEnabled":true,"reportingHosts":1,"threadPool":"CreditCard"}
```

Hystrix Metrics

- Stream can be consumed by UI
  - Hystrix-dashboard







Hystrix metrics

- Can be grouped by request
  - hystrix-core:HystrixRequestLog

GetUserAccountCommand[SUCCESS][27ms], GetPaymentInformationCommand[SUCCESS][29ms], GetOrderCommand[SUCCESS][238ms], GetUserAccountCommand[SUCCESS, RESPONSE_FROM_CACHE][0ms]x2, CreditCardCommand[SUCCESS][1125ms]

Hystrix metrics

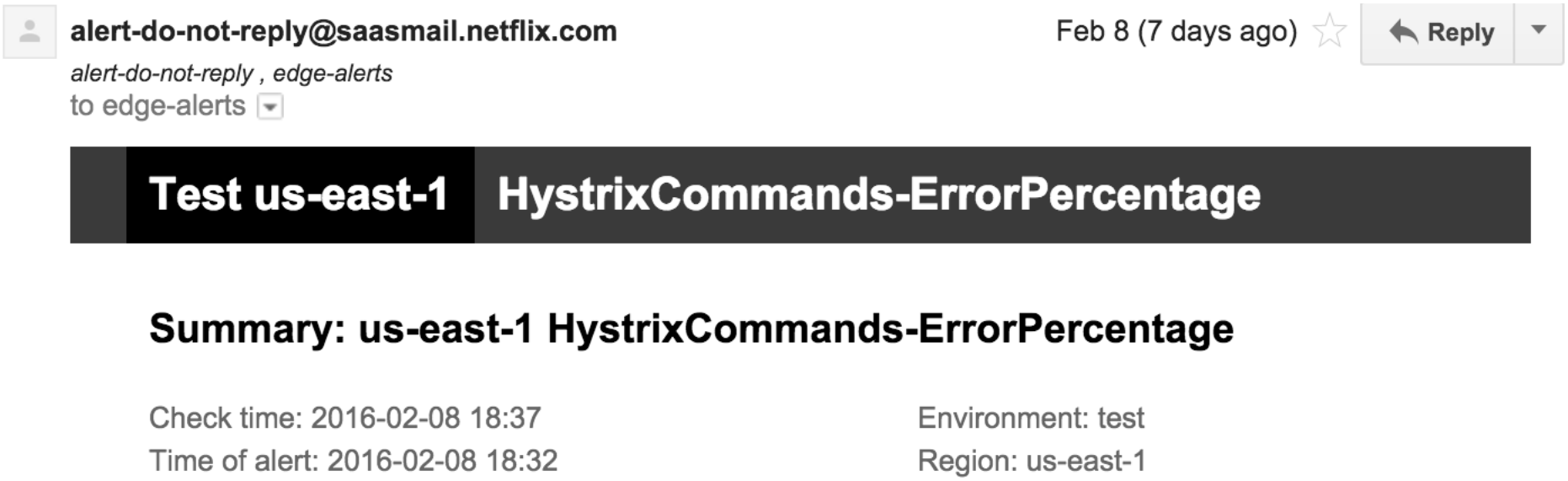
- <DEMO>

Taking Action on Hystrix Metrics

- Circuit Breaker opens/closes

Taking Action on Hystrix Metrics

- Circuit Breaker opens/closes
- Alert/Page on Error %



Taking Action on Hystrix Metrics

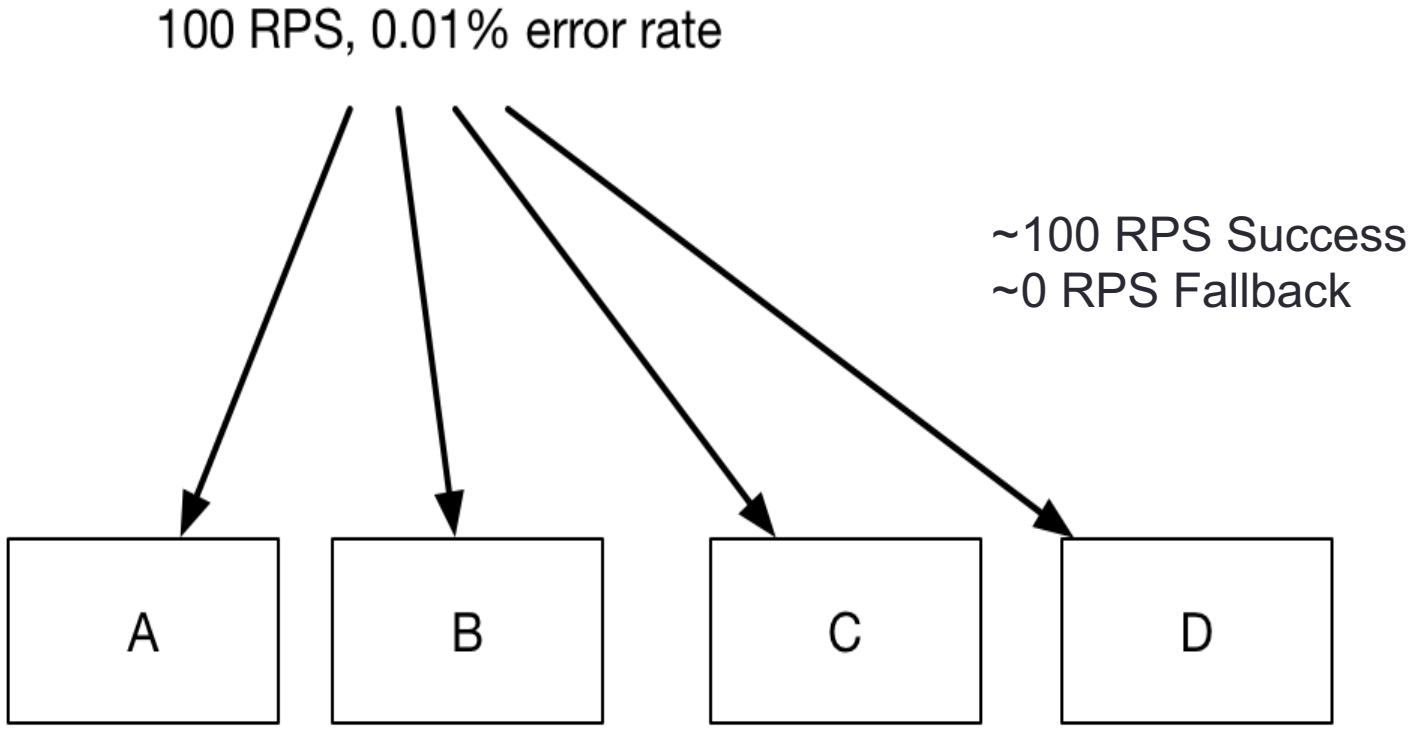
- Circuit Breaker opens/closes
- Alert/Page on Error %
- Alert/Page on Latency

Taking Action on Hystrix Metrics

- Circuit Breaker opens/closes
- Alert/Page on Error %
- Alert/Page on Latency
- Use in canary analysis

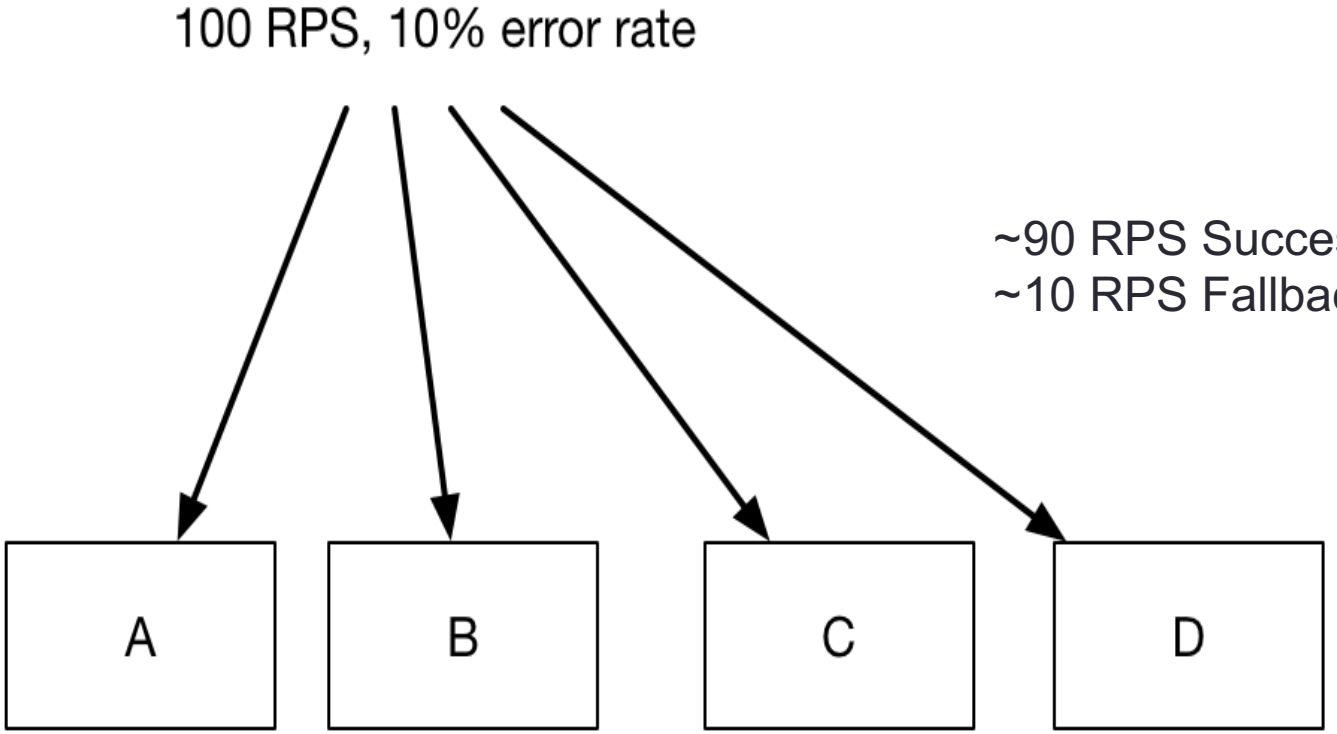
AccountVerifyPhone_countSuccess	0%	PASS
SocialGetTitleContext_countSuccess	-2%	PASS
IdentitySwitchProfile_countSuccess	+8.1%	PASS

How the system should behave



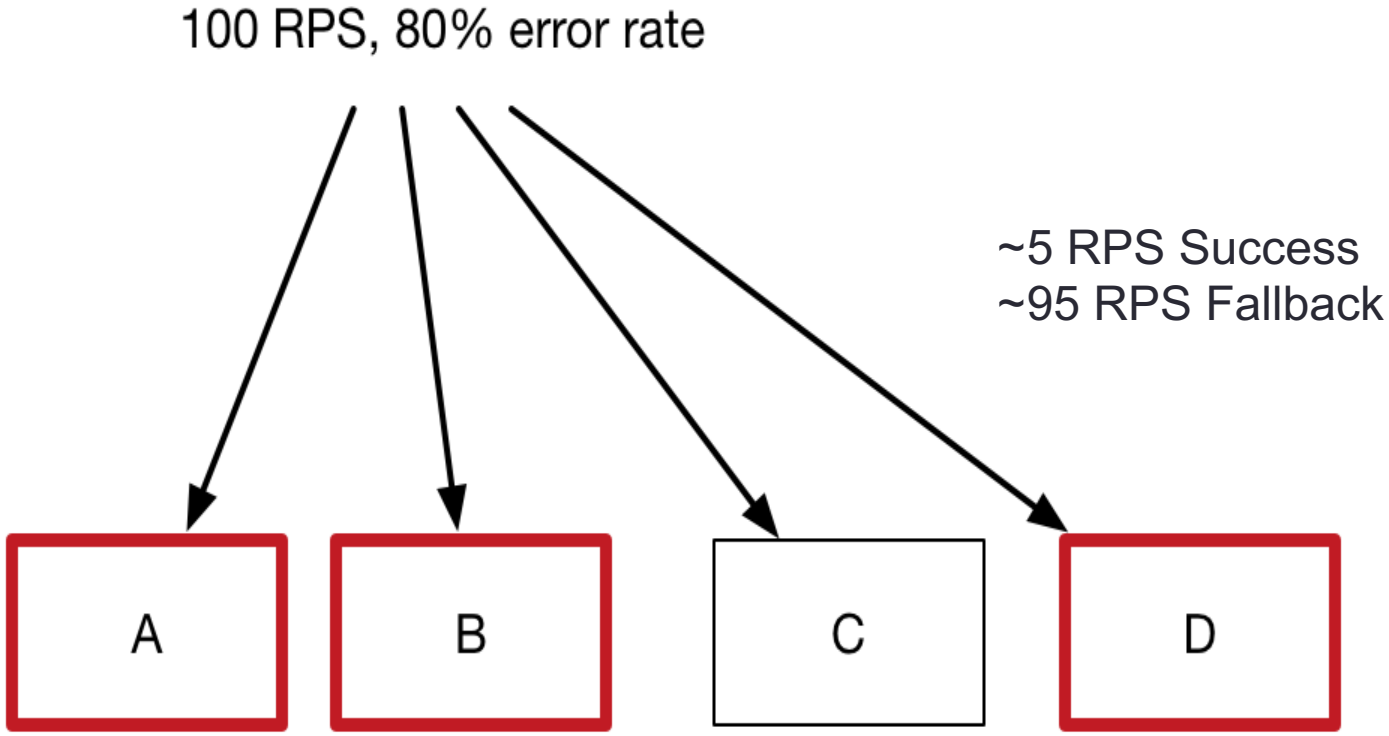
	A	B	C	D
SUCCESS	~24.99	~24.99	~24.99	~24.99
FAILURE	~0.01	~0.01	~0.01	~0.01
TIMEOUT	~0	~0	~0	~0
REJECTED	~0	~0	~0	~0
SHORT-CIRCUITED	~0	~0	~0	~0

How the system should behave



	A	B	C	D
SUCCESS	~22.5	~22.5	~22.5	~22.5
FAILURE	~2.5	~2.5	~2.5	~2.5
TIMEOUT	~0	~0	~0	~0
REJECTED	~0	~0	~0	~0
SHORT-CIRCUITED	~0	~0	~0	~0

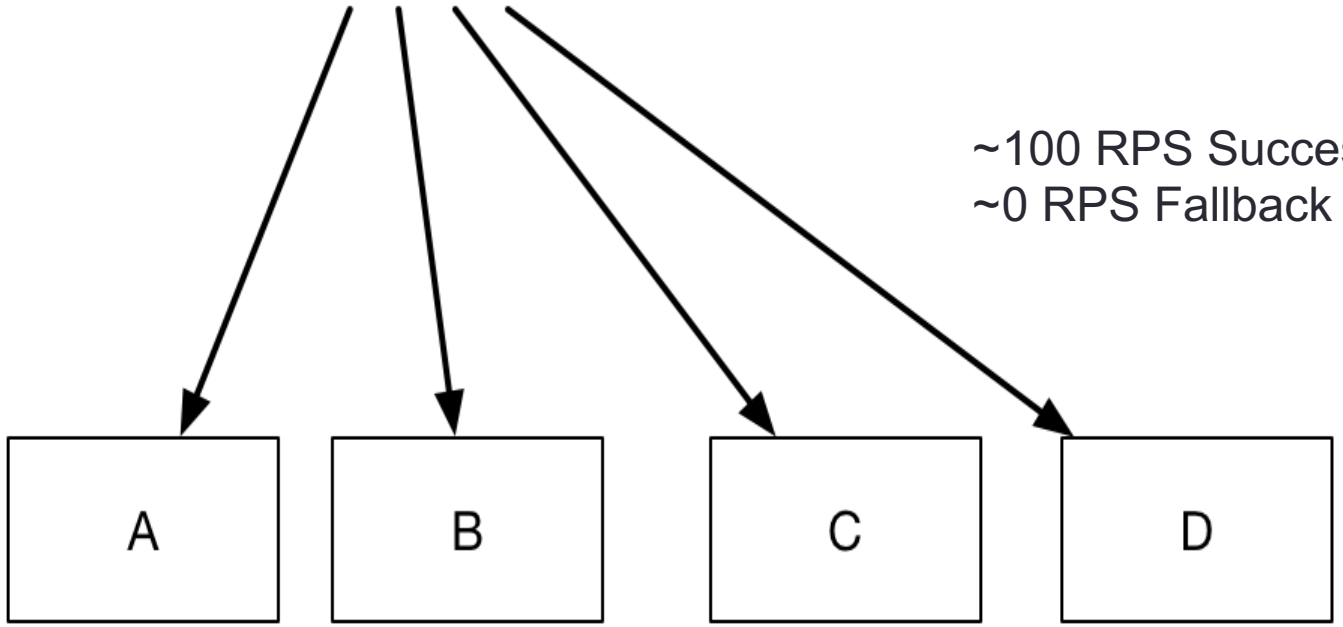
How the system should behave



	A	B	C	D
SUCCESS	~0	~0	~5	~0
FAILURE	~0	~0	~20	~0
TIMEOUT	~0	~0	~0	~0
REJECTED	~0	~0	~0	~0
SHORT-CIRCUITED	~25	~25	~0	~25

How the system should behave

100 RPS, 0.01% error rate, 10ms mean latency

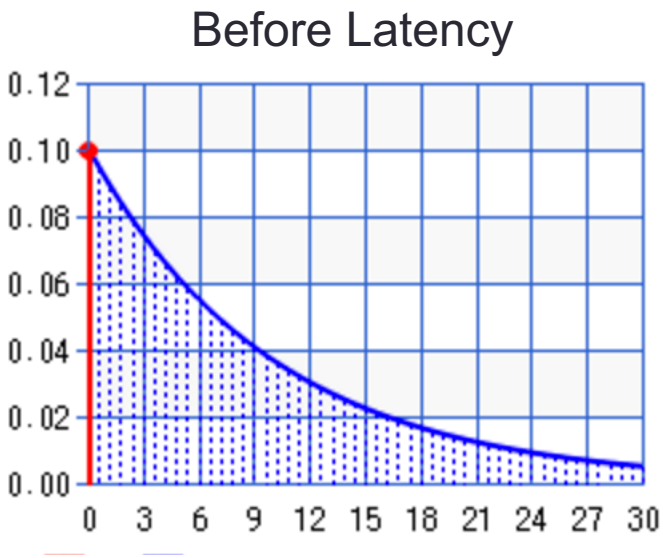


~100 RPS Success
~0 RPS Fallback

	A	B	C	D
SUCCESS	~24.99	~24.99	~24.99	~24.99
FAILURE	~0.01	~0.01	~0.01	~0.01
TIMEOUT	~0	~0	~0	~0
REJECTED	~0	~0	~0	~0
SHORT-CIRCUITED	~0	~0	~0	~0

How the system should behave

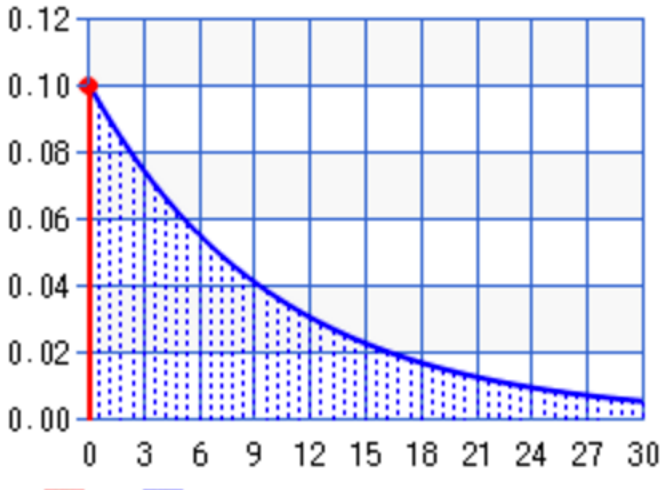
- What happens if mean latency goes from 10ms -> 30ms?



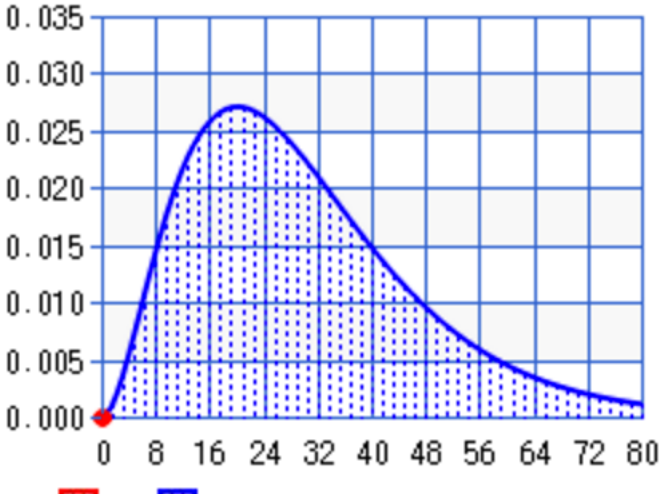
How the system should behave

- What happens if mean latency goes from 10ms -> 30ms?

Before Latency



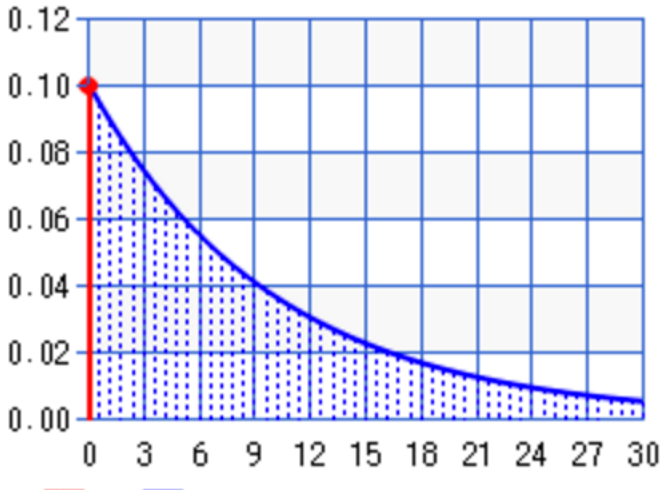
After Latency



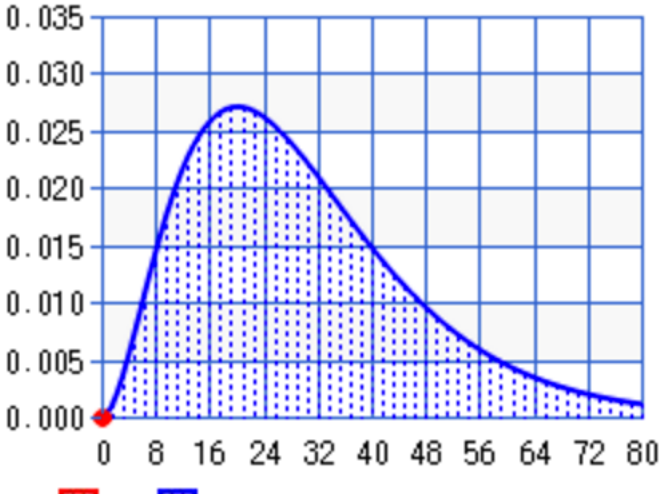
How the system should behave

- What happens if mean latency goes from 10ms -> 30ms?
  - Increased timeouts

Before Latency



After Latency

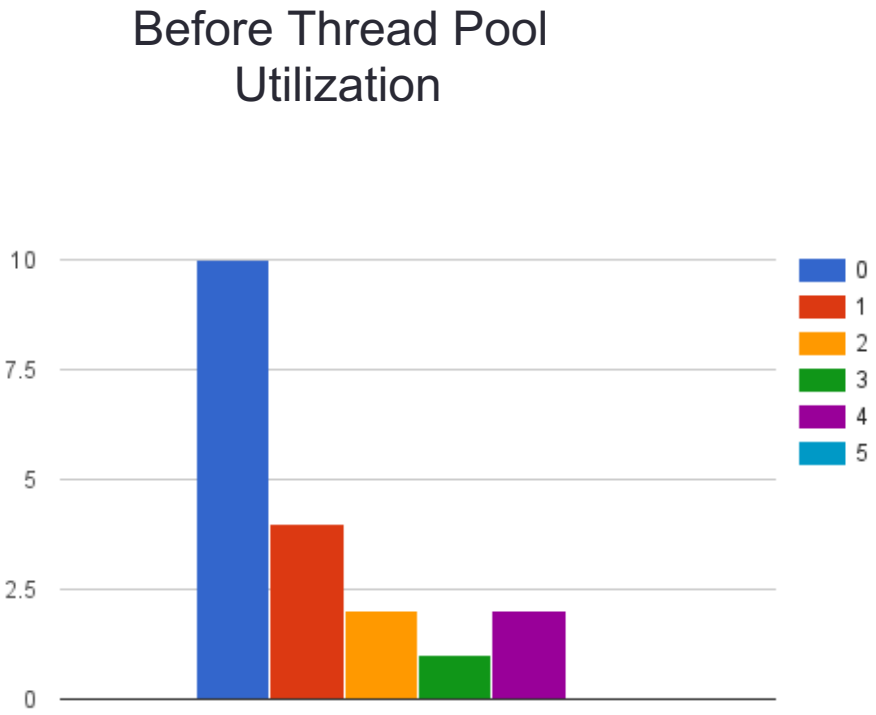


How the system should behave

- What happens if mean latency goes from 10ms -> 30ms at 100 RPS?

How the system should behave

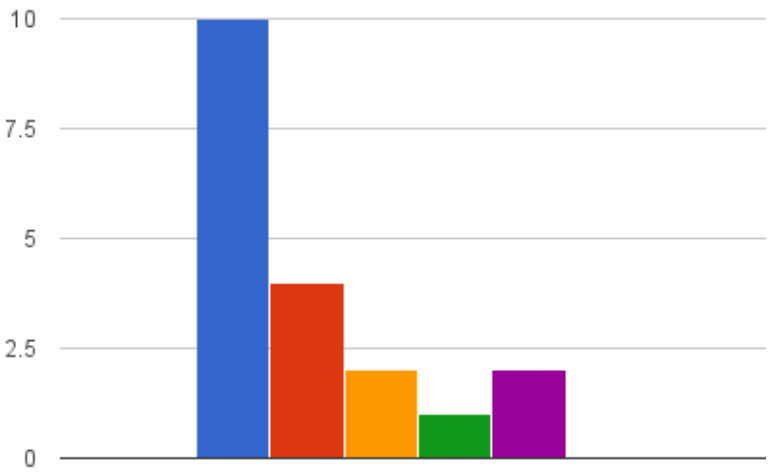
- What happens if mean latency goes from 10ms -> 30ms at 100 RPS?
  - By Little's Law, mean utilization: 1 -> 3



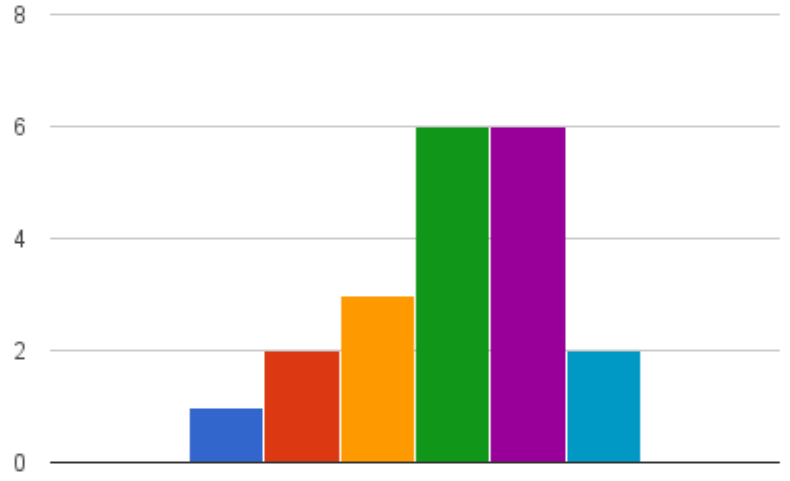
How the system should behave

- What happens if mean latency goes from 10ms -> 30ms at 100 RPS?
  - By Little's Law, mean utilization: 1 -> 3

Before Thread Pool Utilization



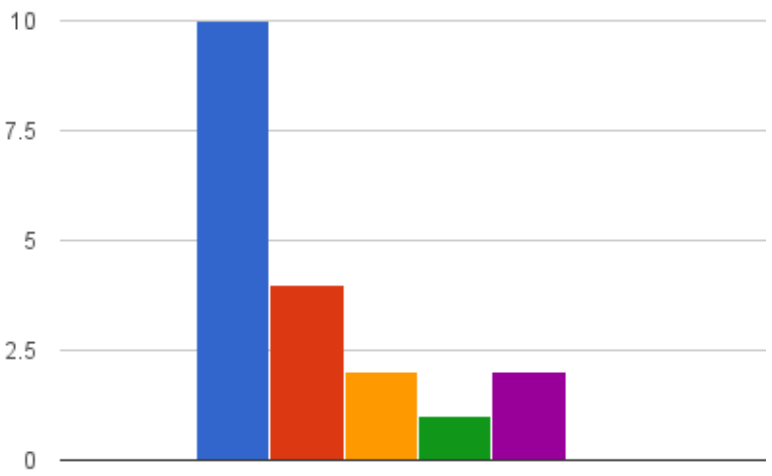
After Thread Pool Utilization



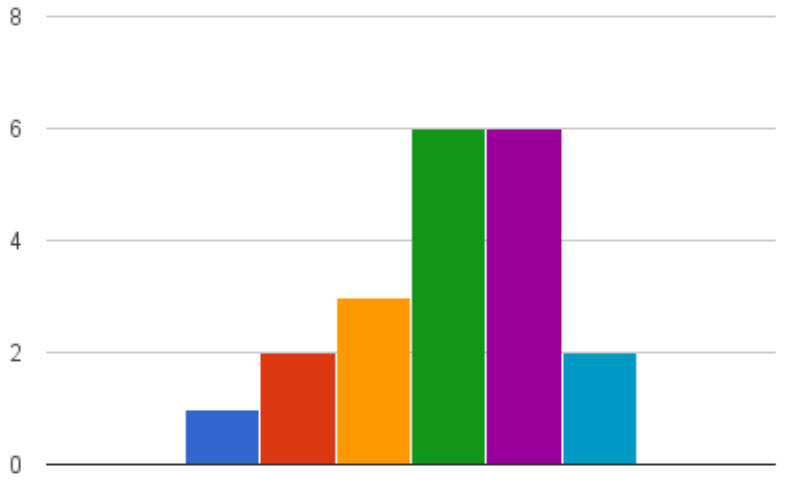
How the system should behave

- What happens if mean latency goes from 10ms -> 30ms at 100 RPS?
  - By Little's Law, mean utilization: 1 -> 3
  - Increased thread pool rejections

Before Thread Pool Utilization

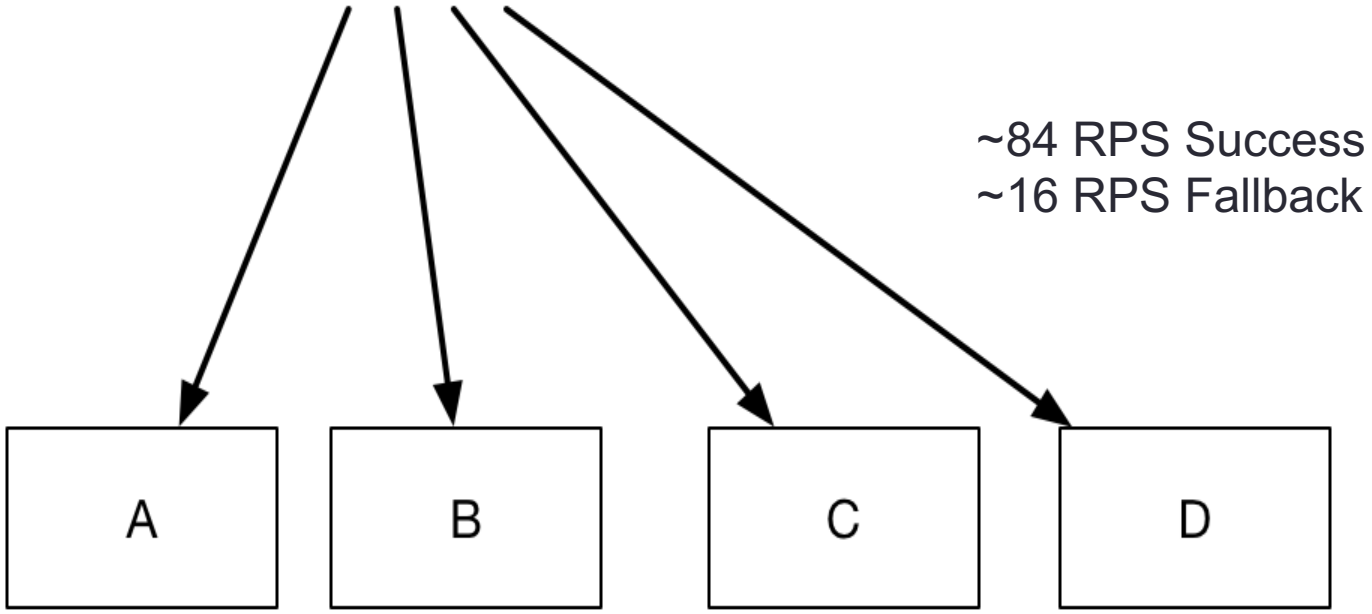


After Thread Pool Utilization



How the system should behave

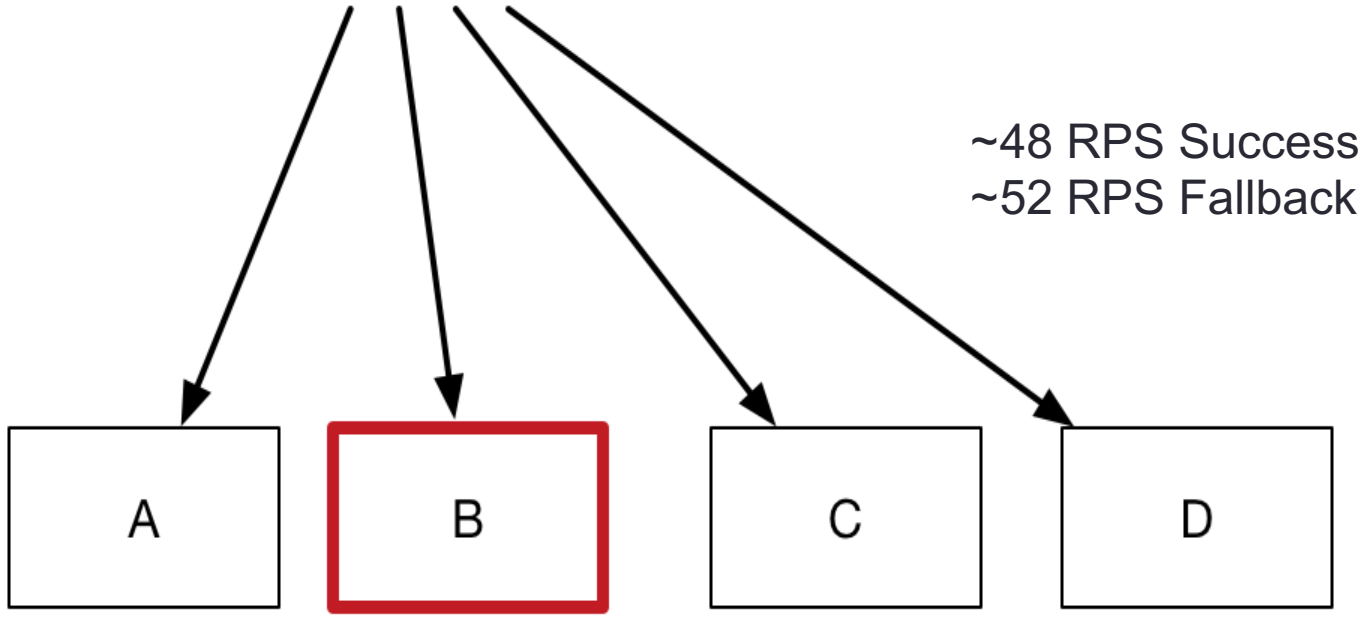
100 RPS, 0.01% error rate, 30ms mean latency



	A	B	C	D
SUCCESS	~21	~21	~21	~21
FAILURE	~0.01	~0.01	~0.01	~0.01
TIMEOUT	~1	~1	~1	~1
REJECTED	~3	~3	~3	~3
SHORT-CIRCUITED	~0	~0	~0	~0

How the system should behave

100 RPS, 40% error rate, 30ms mean latency



	A	B	C	D
SUCCESS	~16	~0	~16	~16
FAILURE	~5	~0	~5	~5
TIMEOUT	~1	~0	~1	~1
REJECTED	~3	~0	~3	~3
SHORT-CIRCUITED	~0	~25	~0	~0

Lessons learned

- Everything we measure has a distribution (once you add time)
  - Queuing theory can teach us a lot

Lessons learned

- Everything we measure has a distribution (once you add time)
  - Queuing theory can teach us a lot
- Errors are more frequent, but latency is more consequential

Lessons learned

- Everything we measure has a distribution (once you add time)
  - Queuing theory can teach us a lot
- Errors are more frequent, but latency is more consequential
- 100% success rates are not what you want!

Lessons learned

- Everything we measure has a distribution (once you add time)
  - Queuing theory can teach us a lot
- Errors are more frequent, but latency is more consequential
- 100% success rates are not what you want!
  - Don’t know if your fallbacks work
  - You have outliers, and they’re not being failed

Lessons learned

- Everything we measure has a distribution (once you add time)
  - Queueing theory can teach us a lot
- Errors are more frequent, but latency is more consequential
- 100% success rates are not what you want!
  - Don’t know if your fallbacks work
  - You have outliers, and they’re not being failed
- Visualization (especially realtime) goes a long way

Goals of this talk

- Philosophical Motivation
  - Why are distributed systems hard?
- Practical Motivation
  - Why do I keep getting paged?
- Solving those problems with Hystrix
  - How does it work?
  - How do I use it in my system?
  - How should a system behave if I use Hystrix?
  - What? Netflix was down for me – what happened there?

Areas Hystrix doesn't cover

- Traffic to the system
- Functional bugs
- Data issues
- AWS
- Service Discovery
- Etc…

Hystrix doesn’t help if...

- Your fallbacks fail

Hystrix doesn’t help if...

- Your fallbacks fail
- Upstream systems do not tolerate fallbacks

Hystrix doesn’t help if...

- Your fallbacks fail
- Upstream systems do not tolerate fallbacks
- Resource bounds are too loose

Hystrix doesn’t help if...

- Your fallbacks fail
- Upstream systems do not tolerate fallbacks
- Resource bounds are too loose
- I/O calls don’t use Hystrix

Failing fallbacks

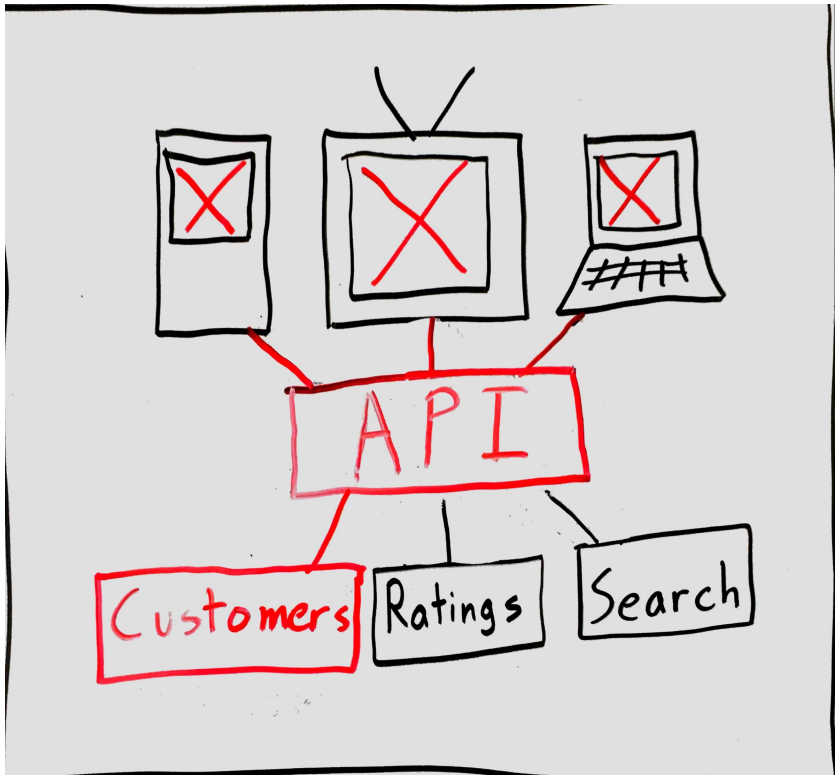
- By definition, fallbacks are exercised less

Failing fallbacks

- By definition, fallbacks are exercised less
- Likely less tested

Failing fallbacks

- By definition, fallbacks are exercised less
- Likely less tested
- If fallback fails, then you have cascaded failure



Unusable fallbacks

- Even if fallback succeeds, upstream systems/UIs need to handle them

Unusable fallbacks

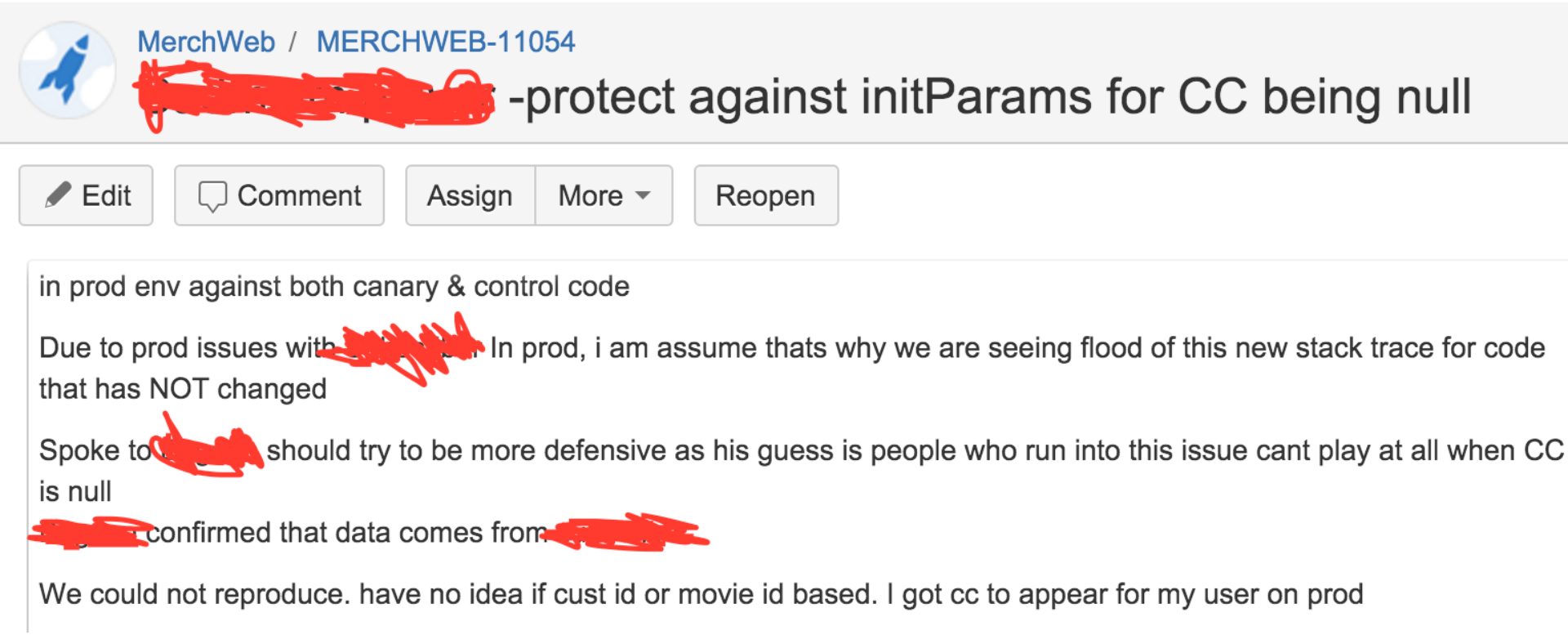
- Even if fallback succeeds, upstream systems/UIs need to handle them
- What happens when UI receives anonymous customer?

Unusable fallbacks

- Even if fallback succeeds, upstream systems/UIs need to handle them
- What happens when UI receives anonymous customer?
  - Need integration tests that expect fallback data

Unusable fallbacks

- Even if fallback succeeds, upstream systems/UIs need to handle them
- What happens when UI receives anonymous customer?



Loose Resource Bounding

- “Kids, get ready for school!”

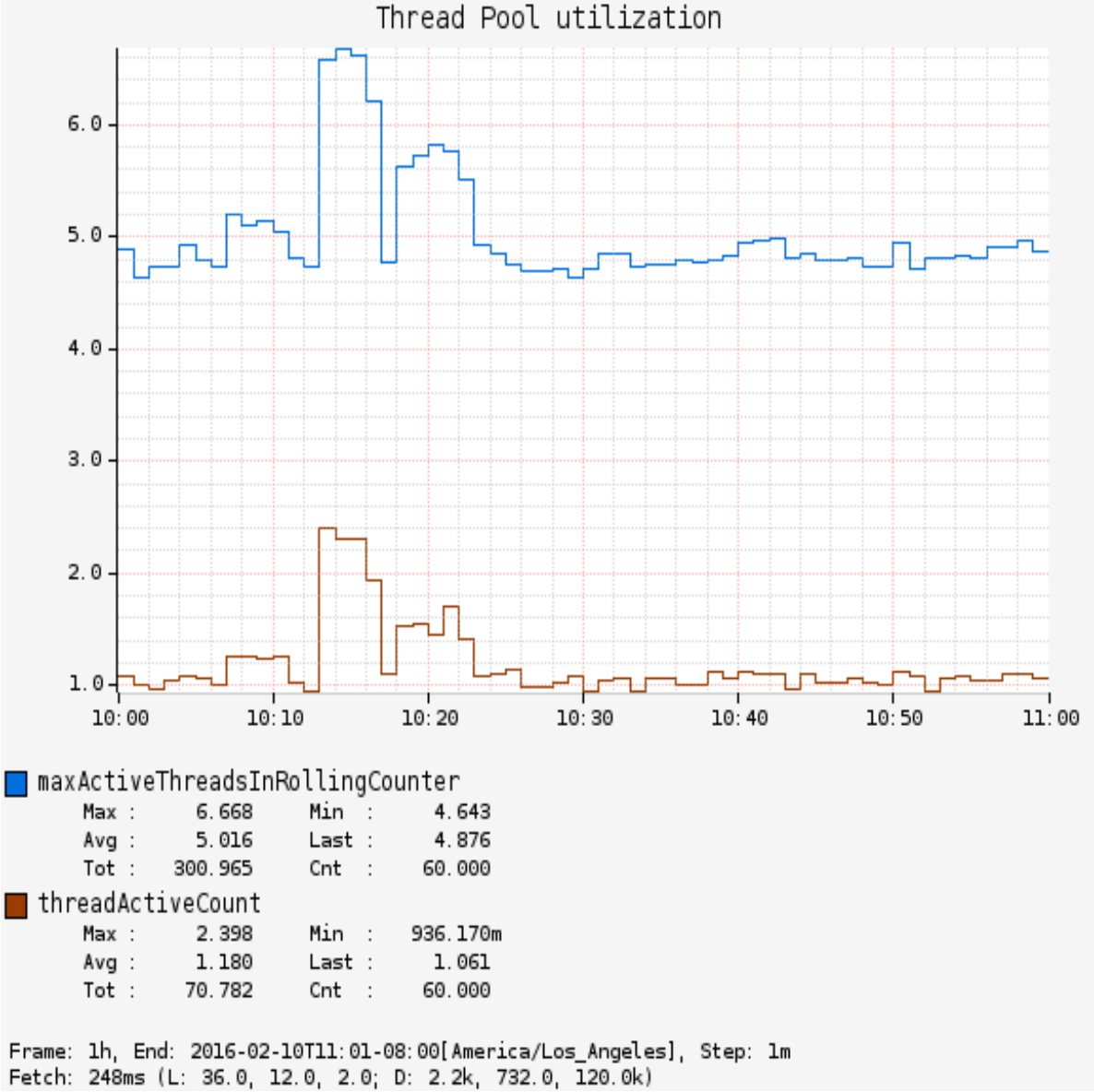
Loose Resource Bounding

- “Kids, get ready for school!”
  - “You’ve got 8 hours to get ready”

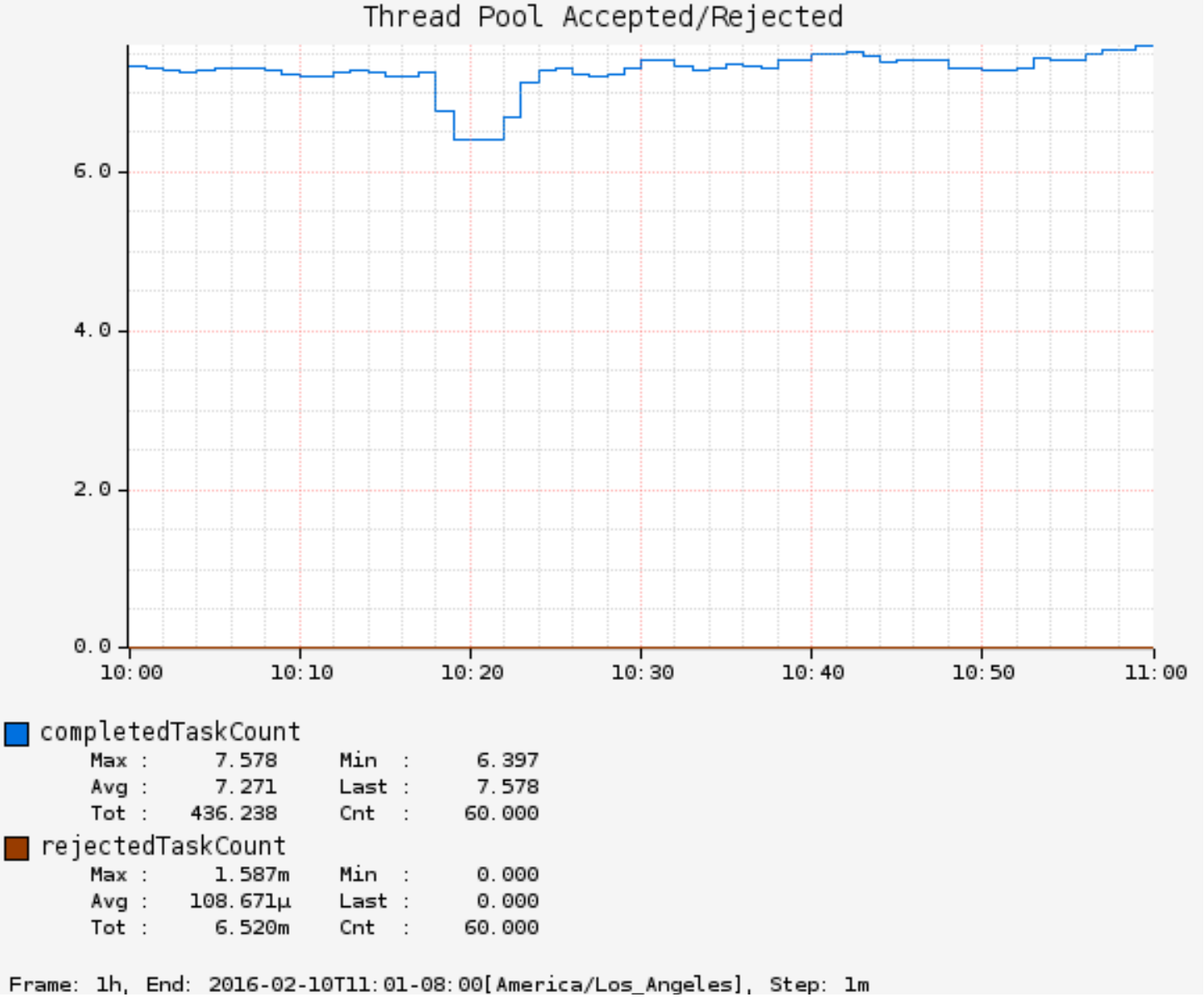
Loose Resource Bounding

- “Kids, get ready for school!”
  - “You’ve got 8 hours to get ready”
- Speed limit is 1000mph

Loose Resource Bounding



Loose Resource Bounding



Unwrapped I/O calls

- Any place where user traffic triggers I/O is dangerous

Unwrapped I/O calls

- Any place where user traffic triggers I/O is dangerous
- Hystrix-network-auditor-agent finds all stack traces that:
  - Are on Tomcat thread
  - Do Network work
  - Don’t use Hystrix

How to Prove our Resilience

“Trust, but verify”

How to Prove our Resilience

- Get lucky and have a real outage

How to Prove our Resilience

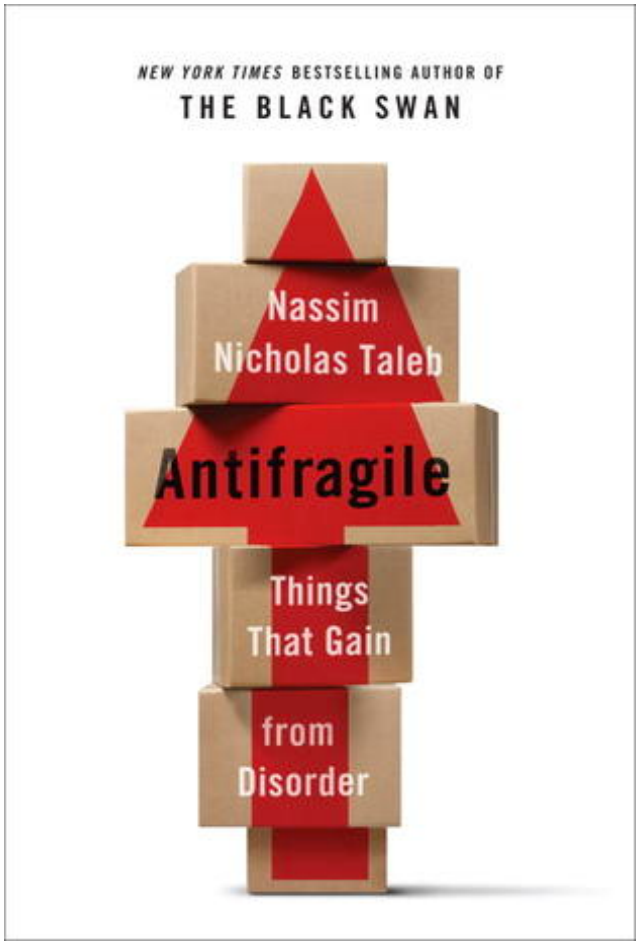
- Get lucky and have a real outage
  - Hopefully our system reacts the way we expect

How to Prove our Resilience

- Get lucky and have a real outage
  - Hopefully our system reacts the way we expect
- Cause the outage we want to protect against

How to Prove our Resilience

- Get lucky and have a real outage
  - Hopefully our system reacts the way we expect
- Cause the outage we want to protect against



Failure Injection Testing

NETFLIX

The Netflix Tech Blog

Thursday, October 23, 2014

FIT : Failure Injection Testing

by Kolton Andrus, Naresh Gopalani, Ben Schmaus

It's no secret that at Netflix we enjoy deliberately breaking things to test our production systems. Doing so lets us validate our assumptions and prove that our mechanisms for handling failure will work when called upon. Netflix has a tradition of implementing a range of tools that create failure, and it is our pleasure to introduce you to the latest of these solutions, FIT or Failure Injection Testing.

FIT is a platform that simplifies creation of failure within our ecosystem with a greater degree of precision for what we fail and who we will impact. FIT also allows us to propagate our failures across the entirety of Netflix in a consistent and controlled manner.

Why We Built FIT
While breaking things is fun, we do not enjoy causing our customers pain. Some of our Monkeys, by design, can go a little too wild when let out of their cages. Latency Monkey in

Links

- Netflix US & Canada Blog
- Netflix America Latina Blog
- Netflix Brasil Blog
- Netflix Benelux Blog
- Netflix DACH Blog
- Netflix France Blog
- Netflix Nordics Blog
- Netflix UK & Ireland Blog
- Netflix ISP Speed Index
- Open positions at Netflix
- Netflix Website

Failure Injection Testing

- Inject failures at any layer
  - Hystrix is a good one

Failure Injection Testing

- Inject failures at any layer
  - Hystrix is a good one
- Scope (region / % traffic / user / device)

Failure Injection Testing

- Inject failures at any layer
  - Hystrix is a good one
- Scope (region / % traffic / user / device)
- Scenario (fail?, add latency?)

Failure Injection Testing

- Inject error and observe fallback successes

Failure Injection Testing

- Inject error and observe fallback successes
- Inject error and observe device handling fallbacks

Failure Injection Testing

- Inject error and observe fallback successes
- Inject error and observe device handling fallbacks
- Inject latency and observe thread-pool rejections and timeouts

Failure Injection Testing

- Inject error and observe fallback successes
- Inject error and observe device handling fallbacks
- Inject latency and observe thread-pool rejections and timeouts
- We have an off-switch in case this goes poorly!

FIT in Action

- <DEMO>

Operational Summary

- Downstream transient errors cause fallbacks and a degraded but not broken customer experience

Operational Summary

- Downstream transient errors cause fallbacks and a degraded but not broken customer experience
- Fallbacks go away once errors subside

Operational Summary

- Downstream transient errors cause fallbacks and a degraded but not broken customer experience
- Fallbacks go away once errors subside
- Downstream latency increases load/latency, but not to a dangerous level

Operational Summary

- Downstream transient errors cause fallbacks and a degraded but not broken customer experience
- Fallbacks go away once errors subside
- Downstream latency increases load/latency, but not to a dangerous level
- We have near realtime visibility into our system and all of our downstream systems

References

- Sandvine internet usage report
- Fallacies of Distributed Computing
- Little's Law
- Release It!
- Netflix Techblog: Atlas
- Netflix Techblog: FIT
- Netflix Techblog: Hystrix
- Netflix Techblog: How API became reactive
- Spinnaker OSS
- Antifragile

Coordinates

- github.com/Netflix/Hystrix
- @HystrixOSS
- @NetflixOSS
- @mattrjacobs
- mjacobs@netflix.com
- jobs.netflix.com (We're hiring!)

Bonus: Hystrix Config

Bonus: Hystrix 1.5 Metrics

Bonus: Horror Stories