Keeping Movies Running Amid Thunderstorms

Fault-tolerant Systems @ Netflix

Sid Anand (@r39132)

QCon SF 2011

Backgrounder

Netflix Then and Now

NETFLIX

Netflix Then and Now

Netflix prior to circa 2009

Users watched DVDs at home

Peak days : **Friday, Saturday, Sunday**

Users returned DVDs & Updated their Qs

Peak days : **Sunday, Monday**

We shipped the next DVDs

Peak days : **Monday, Tuesday**

Scheduled Site Downtimes on alternate Wednesdays

Netflix post circa 2009

Users watch streaming at home

Peak days : **Friday, Saturday, Sunday**

Off-Peak days see many orders of magnitude more traffic than prior to 2009

User expectation is that streaming is always available

No Scheduled Site Downtimes

Fault Tolerance is a top design concern

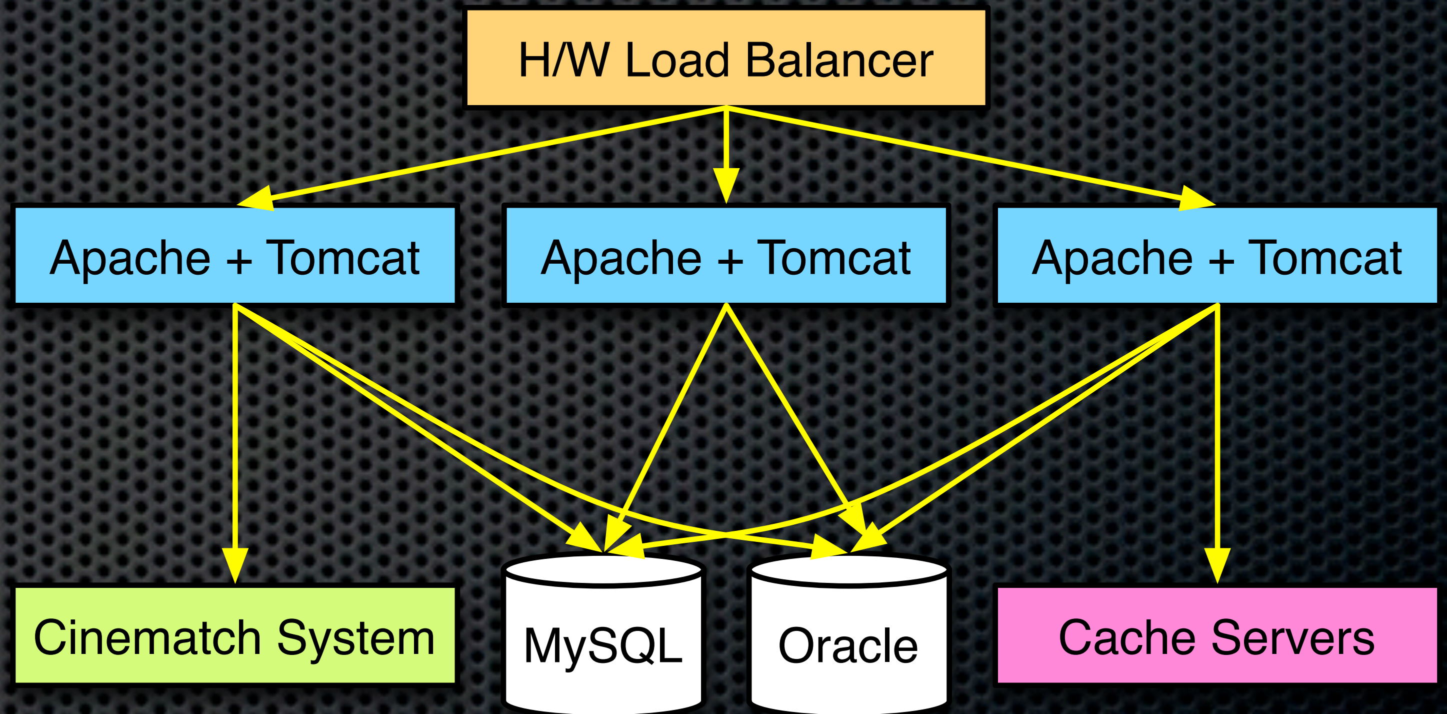
Netflix DC Architecture

A Simple System

Netflix's DC Architecture

Components

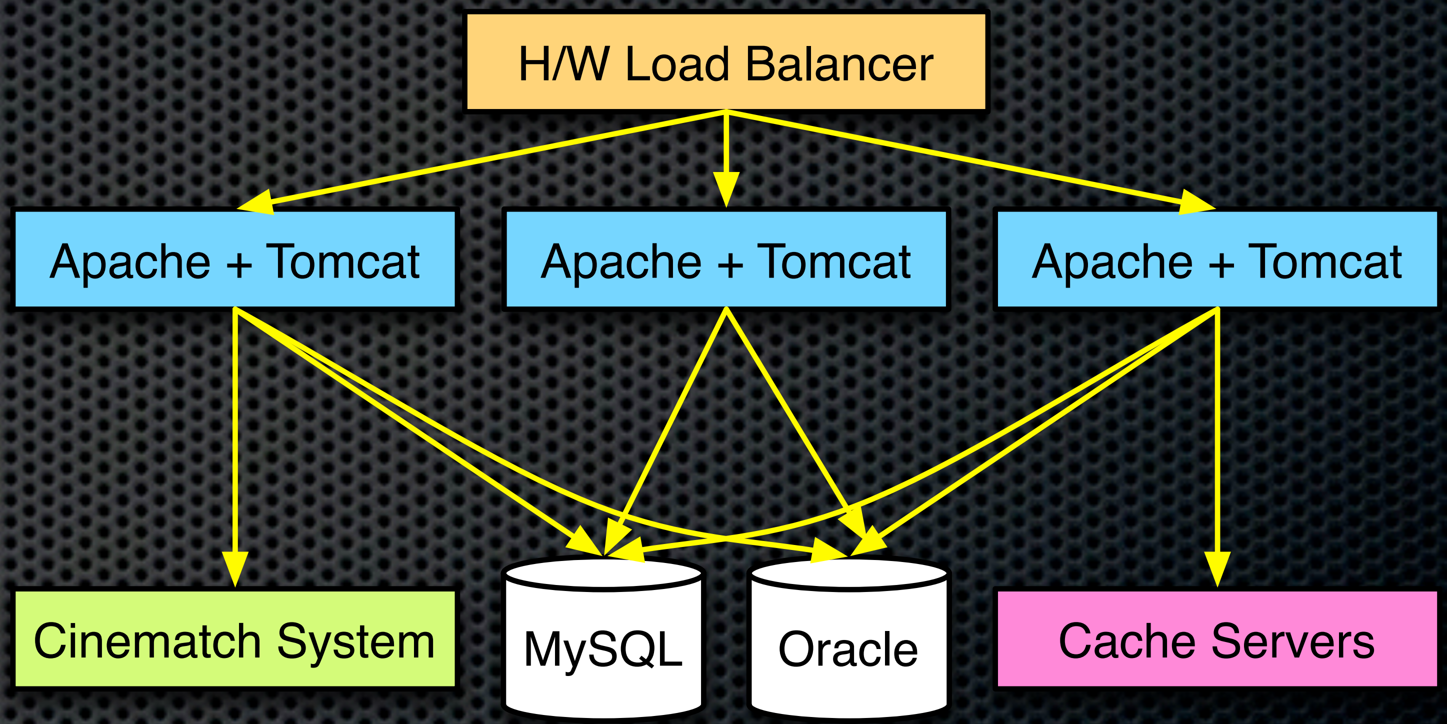
- 1 Netscaler H/W Load Balancer
- ~20 "**WWW**" Apache+Tomcat servers
- 3 Oracle DBs & 1 MySQL DB
- Cache Servers
- Cinematch Recommendation System



Netflix's DC Architecture

Types of Production Issues

- Java Garbage Collection problems, which would would result in slower WWW pages
- Deadlocks in our multi-threaded Java application would cause web page loading to timeout
- Transaction locking in the DB would result in the similar web page loading timeouts
- Under-optimized SQL or DB would cause slower web pages (**e.g. DB optimizer picks a sub-optimal the execution plan**)



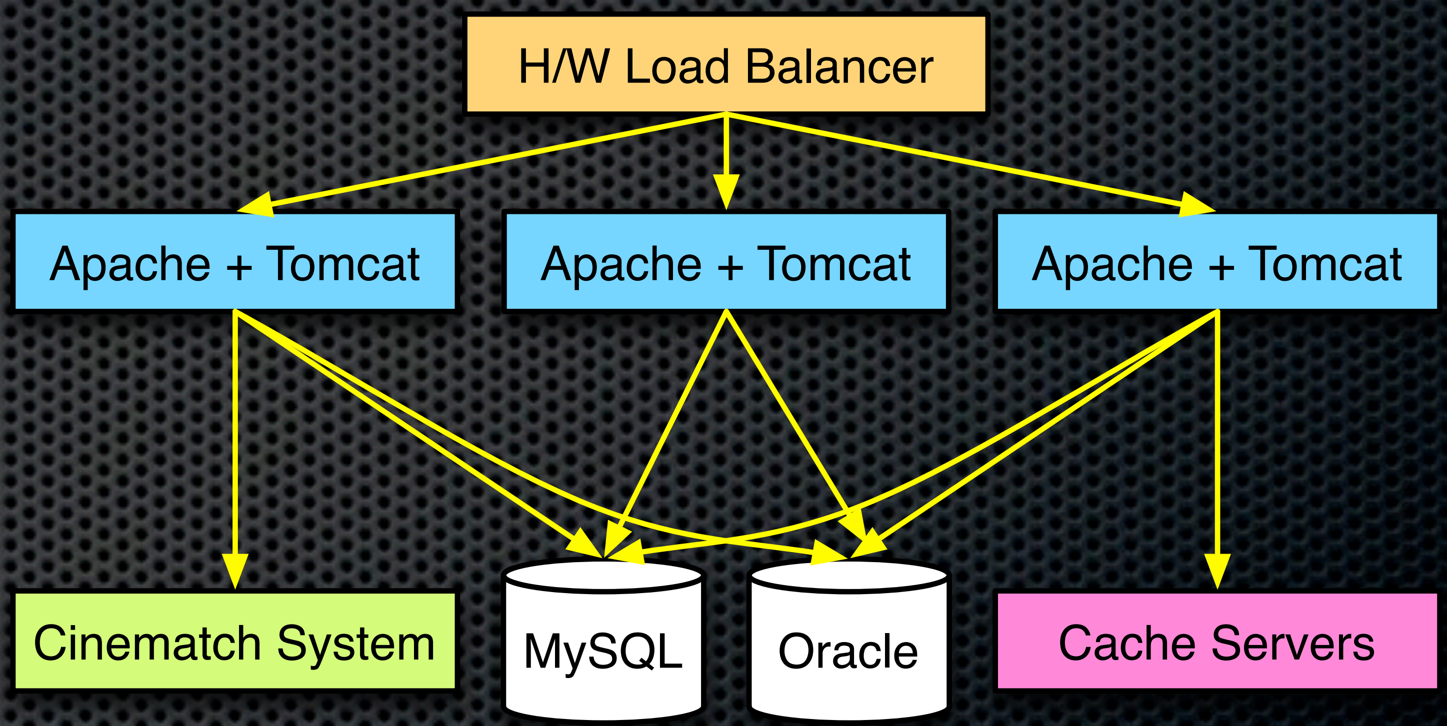
Netflix's DC Architecture

Architecture Pros

- As serious as these sound, they were typically single-system failure scenarios
- Single-system failures are relatively easy to resolve

Architecture Cons

- Not horizontally scalable
    - We're constrained by what can fit on a single box
- Not conducive to high-velocity development and deployment



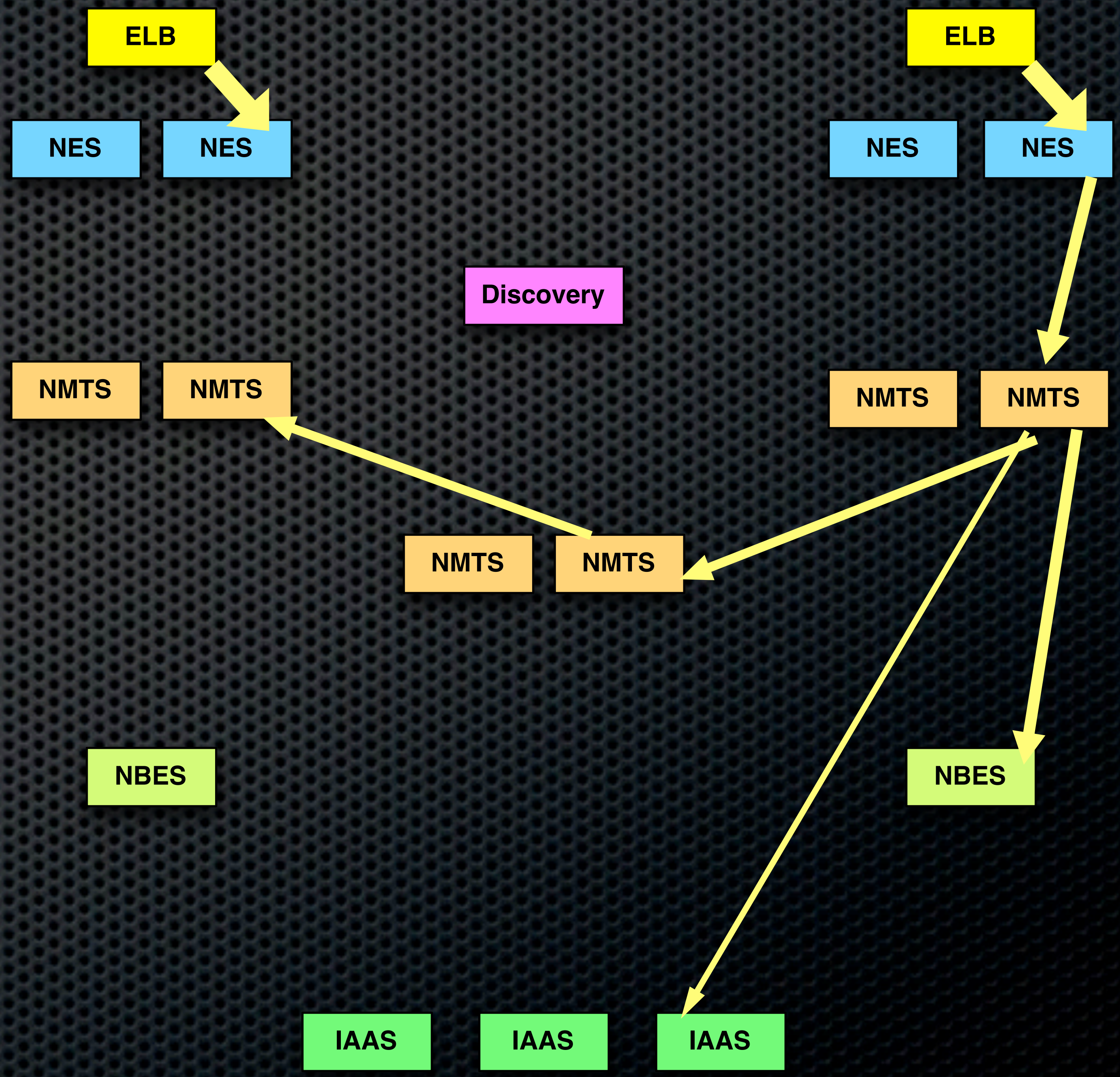
Netflix's Cloud Architecture

A Less Simple System

Netflix's Cloud Architecture

Components

- Many (~100) applications, organized in clusters
- Clusters can be at different levels in the call stack
- Clusters can call each other

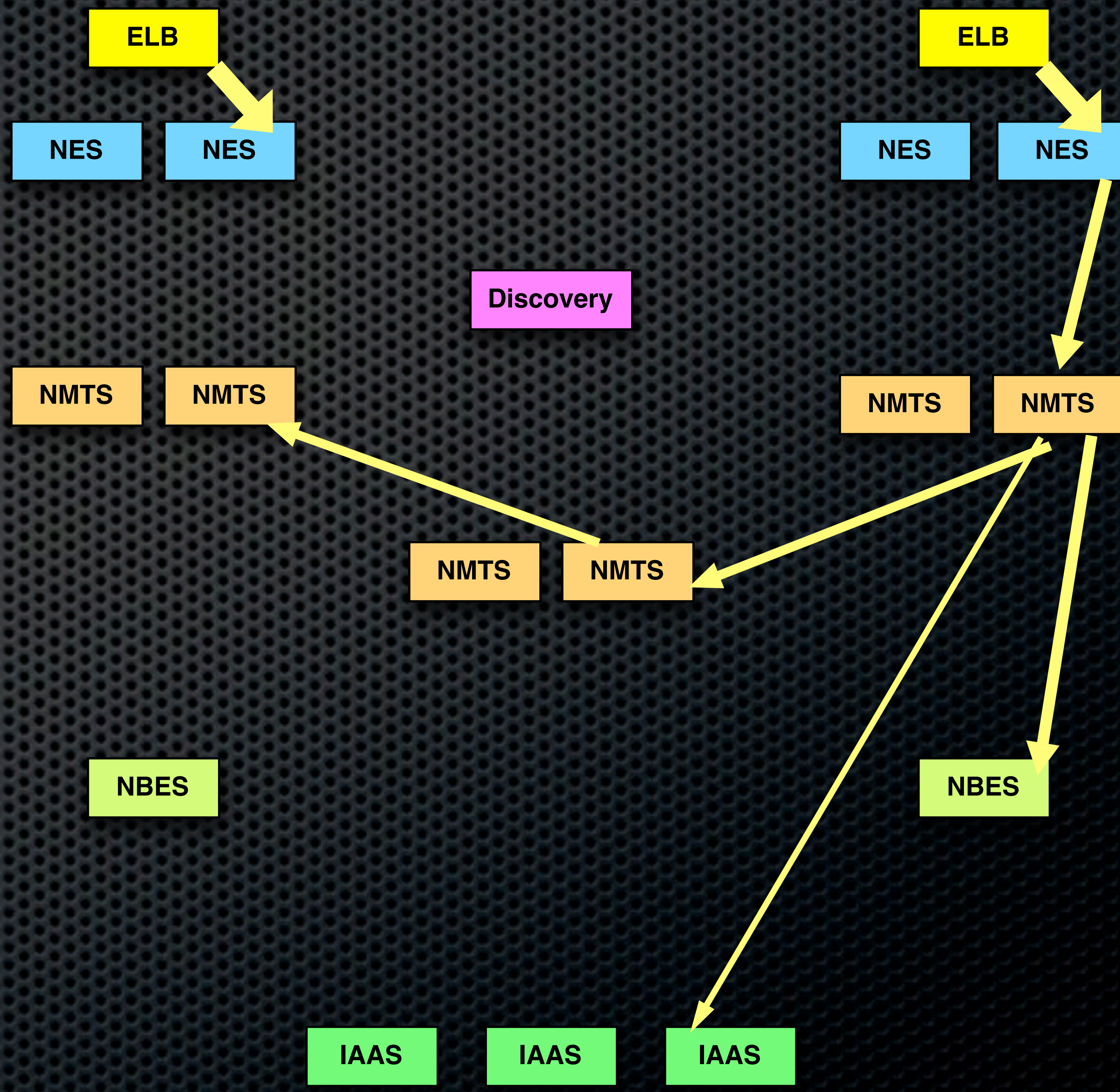


Thursday, November 17, 2011

Netflix's Cloud Architecture

Levels

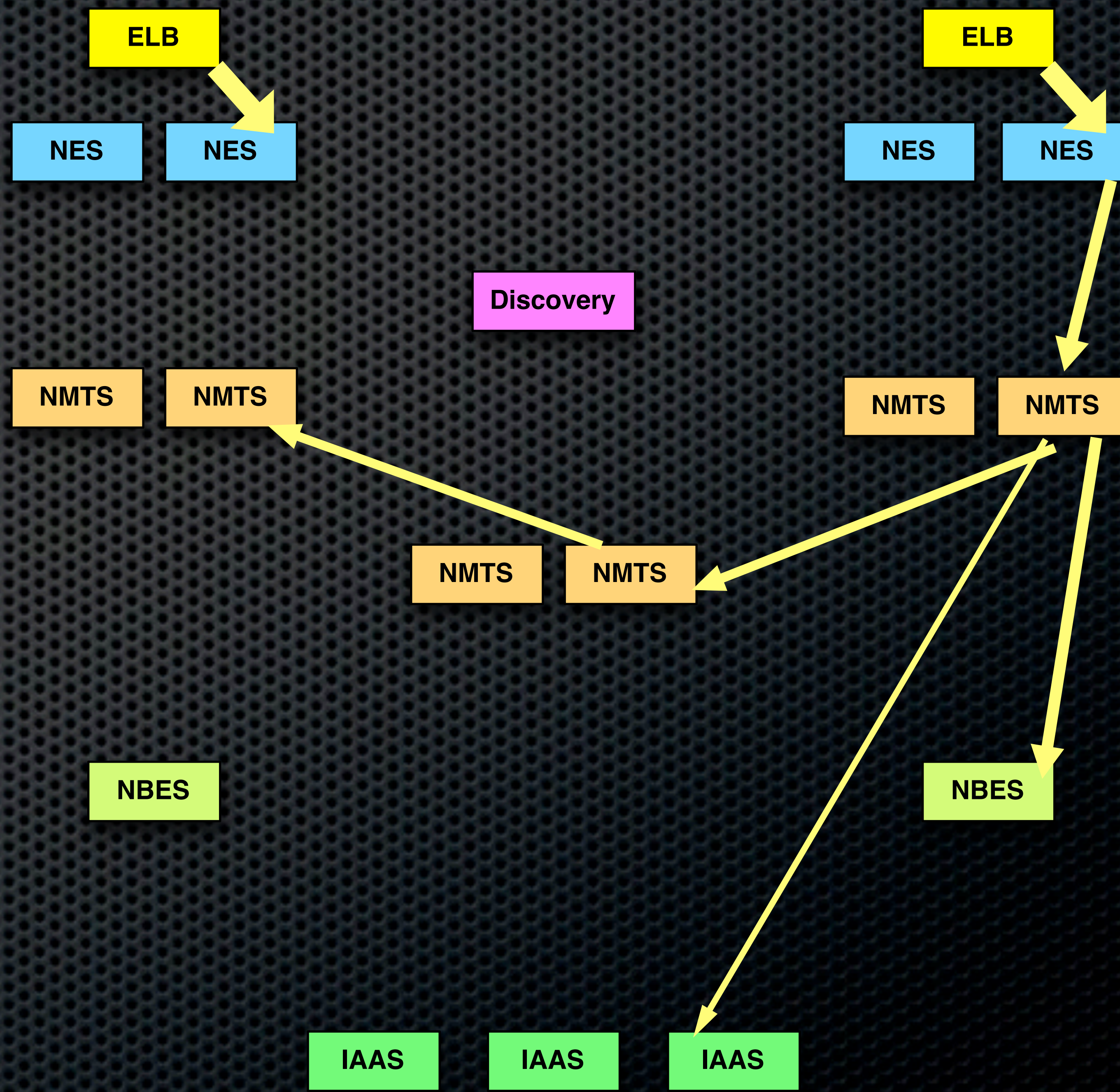
- NES : Netflix Edge Services
- NMTS : Netflix Mid-tier Services
- NBES : Netflix Back-end Services
- IAAS : AWS IAAS Services
- Discovery : Help services discover NMTS and NBES services



Netflix's Cloud Architecture

Components (NES)

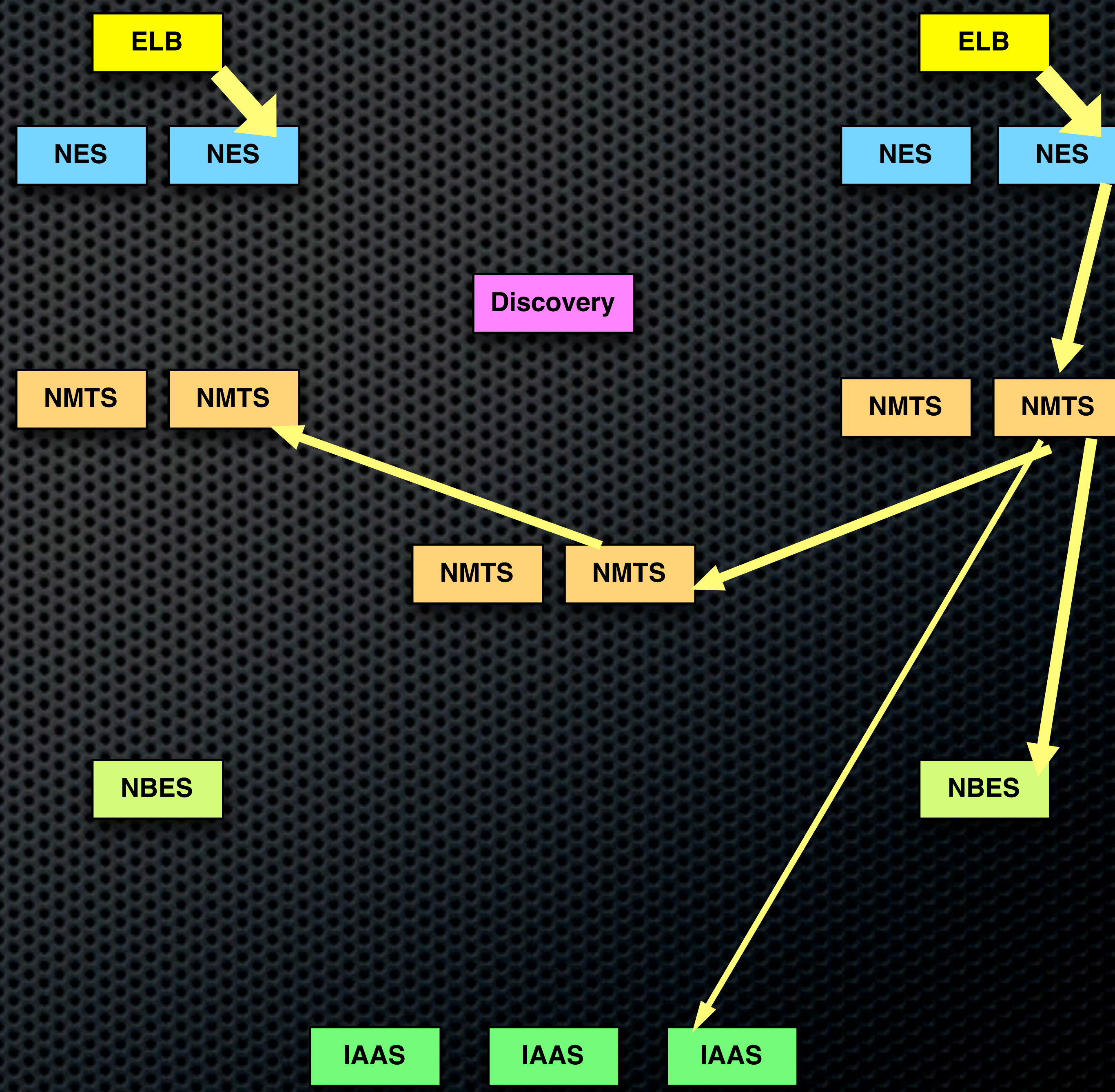
- Overview
  - Any service that browsers and streaming devices connect to over the internet
  - They sit behind AWS Elastic Load Balancers (**a.k.a. ELB**)
  - They call clusters at lower levels



Netflix's Cloud Architecture

Components (NES)

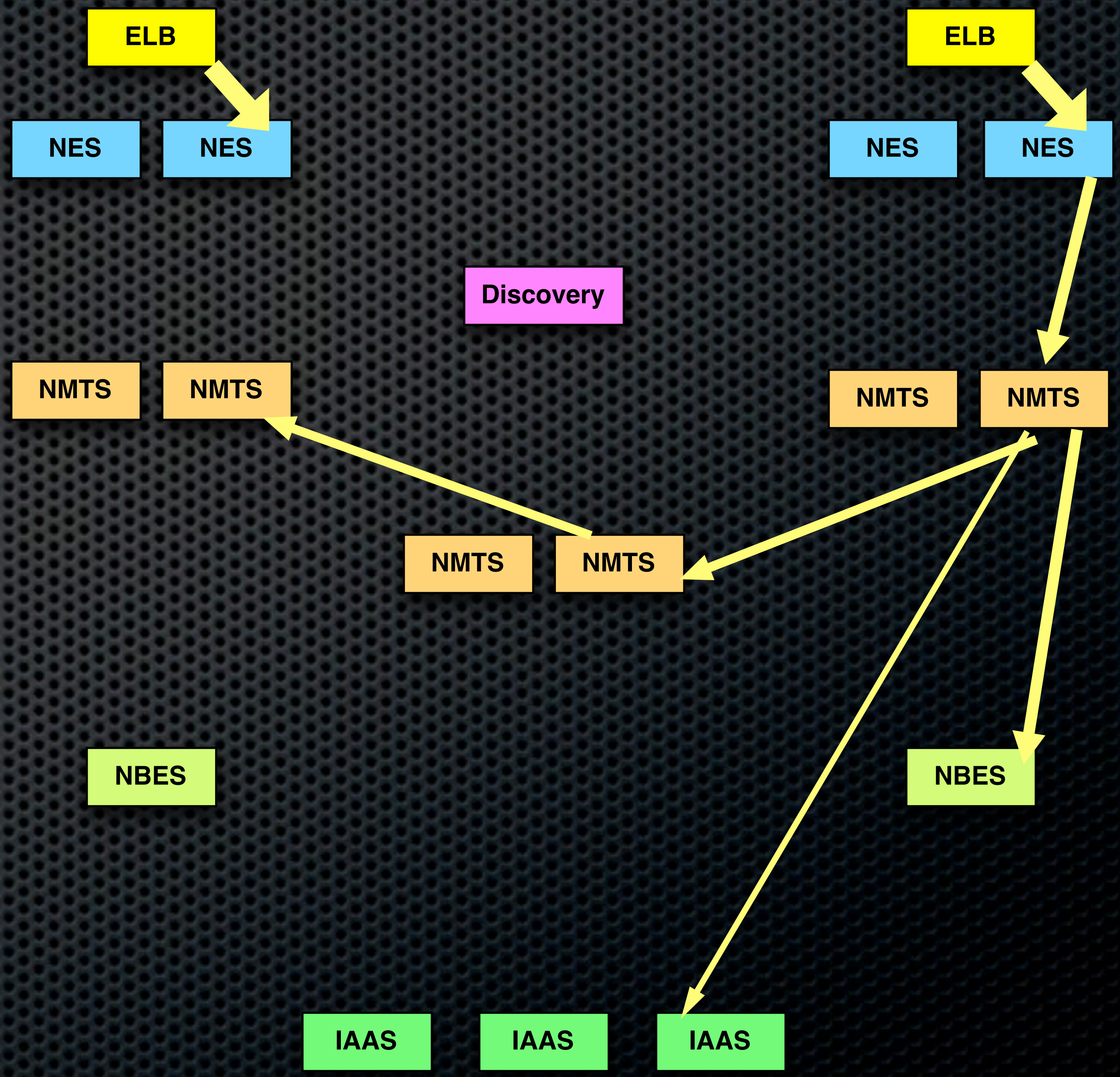
- Examples
  - API Servers
    - Support the video browsing experience
    - Also allows users to modify their Q
  - Streaming Control Servers
    - Support streaming video playback
    - Authenticate your Wii, PS3, etc...
    - Download DRM to the Wii, PS3, etc...
    - Return a list of CDN urls to the Wii, PS3, etc...



Netflix's Cloud Architecture

Components (NMTS)

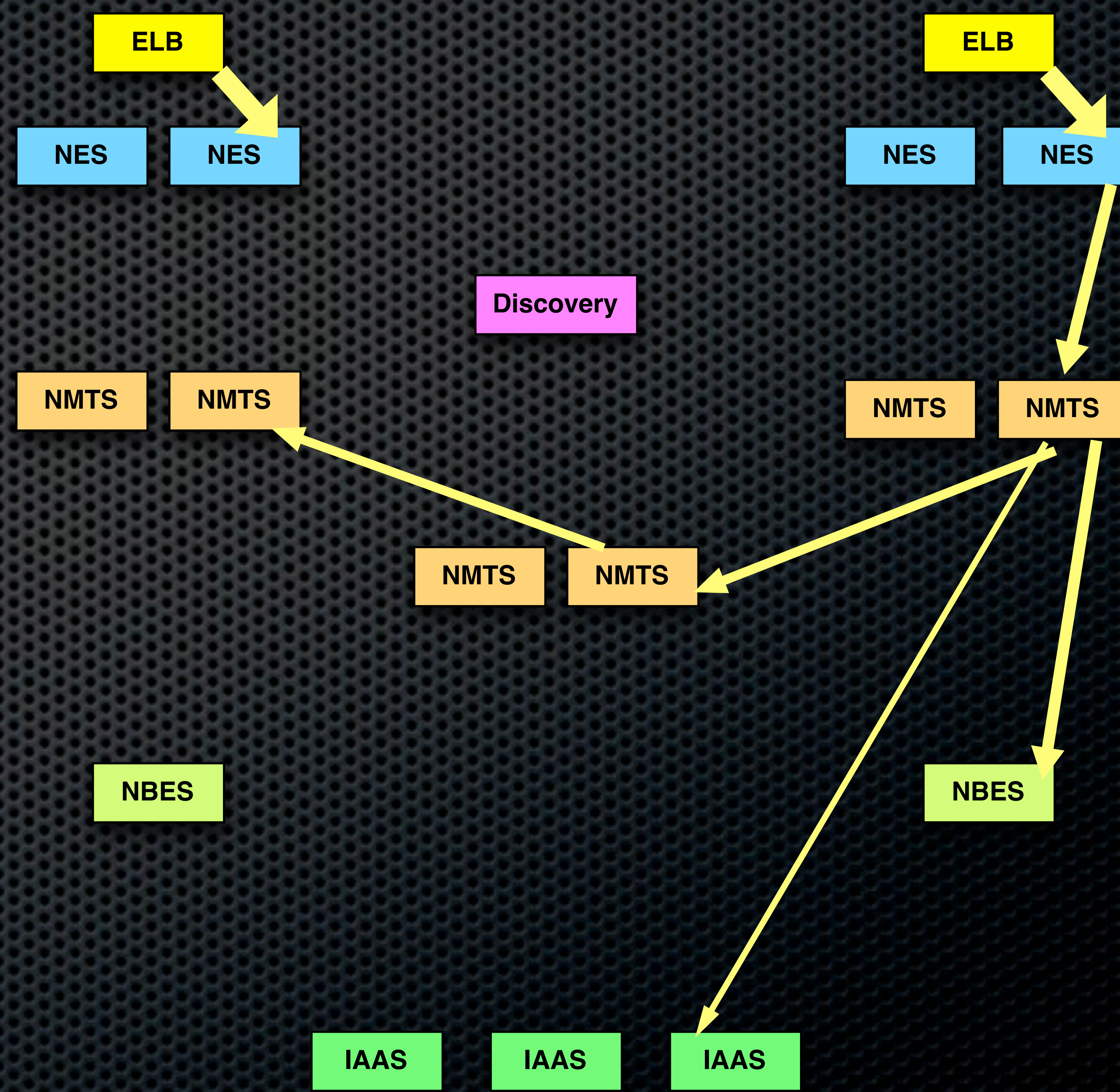
- Overview
  - Can call services at the same or lower levels
    - Other NMTS
    - NBES, IAAS
    - Not NES
  - Exposed through our Discovery service



Netflix's Cloud Architecture

Components (NMTS)

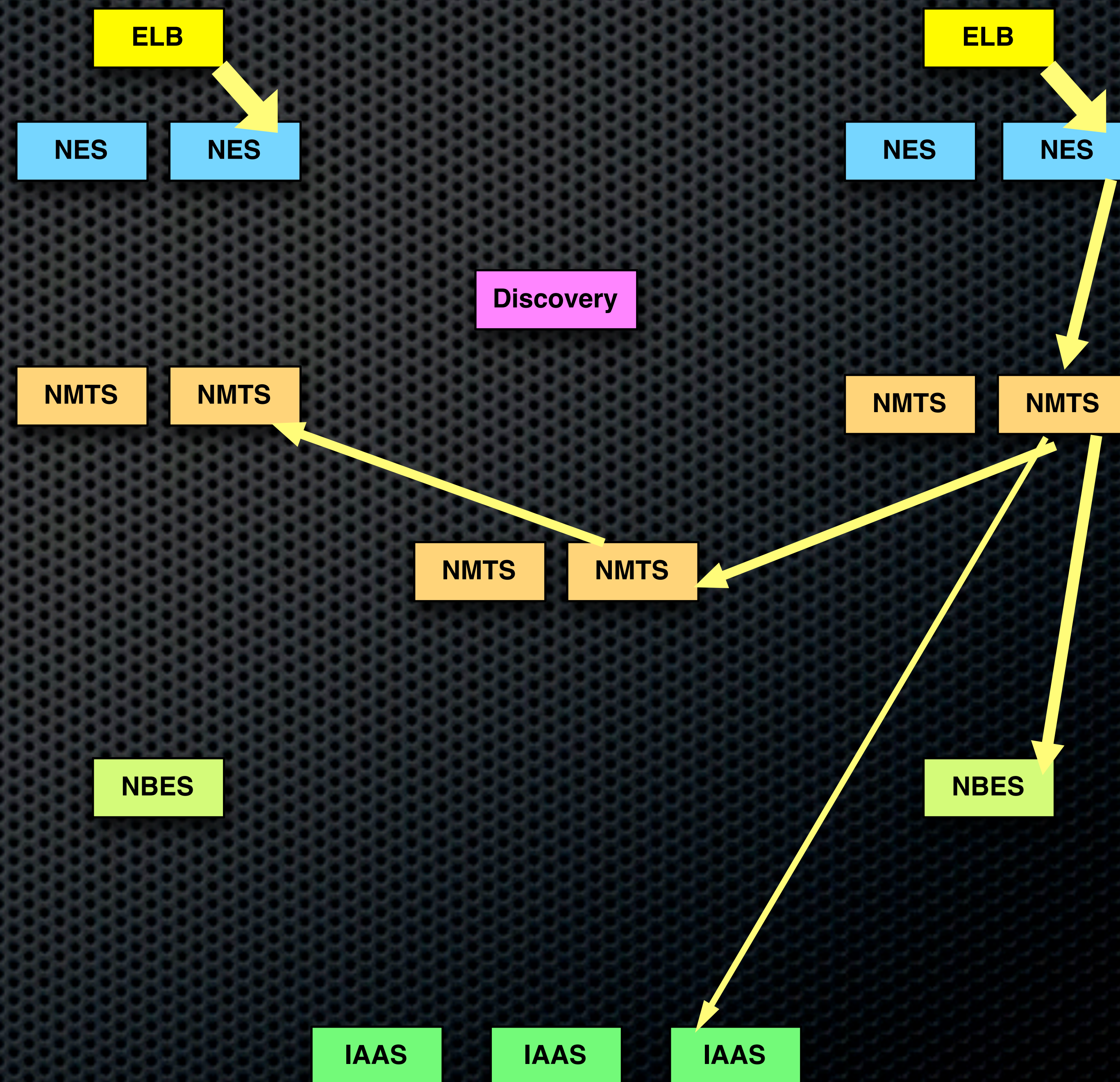
- Examples
  - Netflix Queue Servers
    - Modify items in the users' movie queue
  - Viewing History Servers
    - Record and track all streaming movie watching
  - SIMS Servers
    - Compute and serve user-to-user and movie-to-movie similarities



Netflix's Cloud Architecture

Components (NBES)

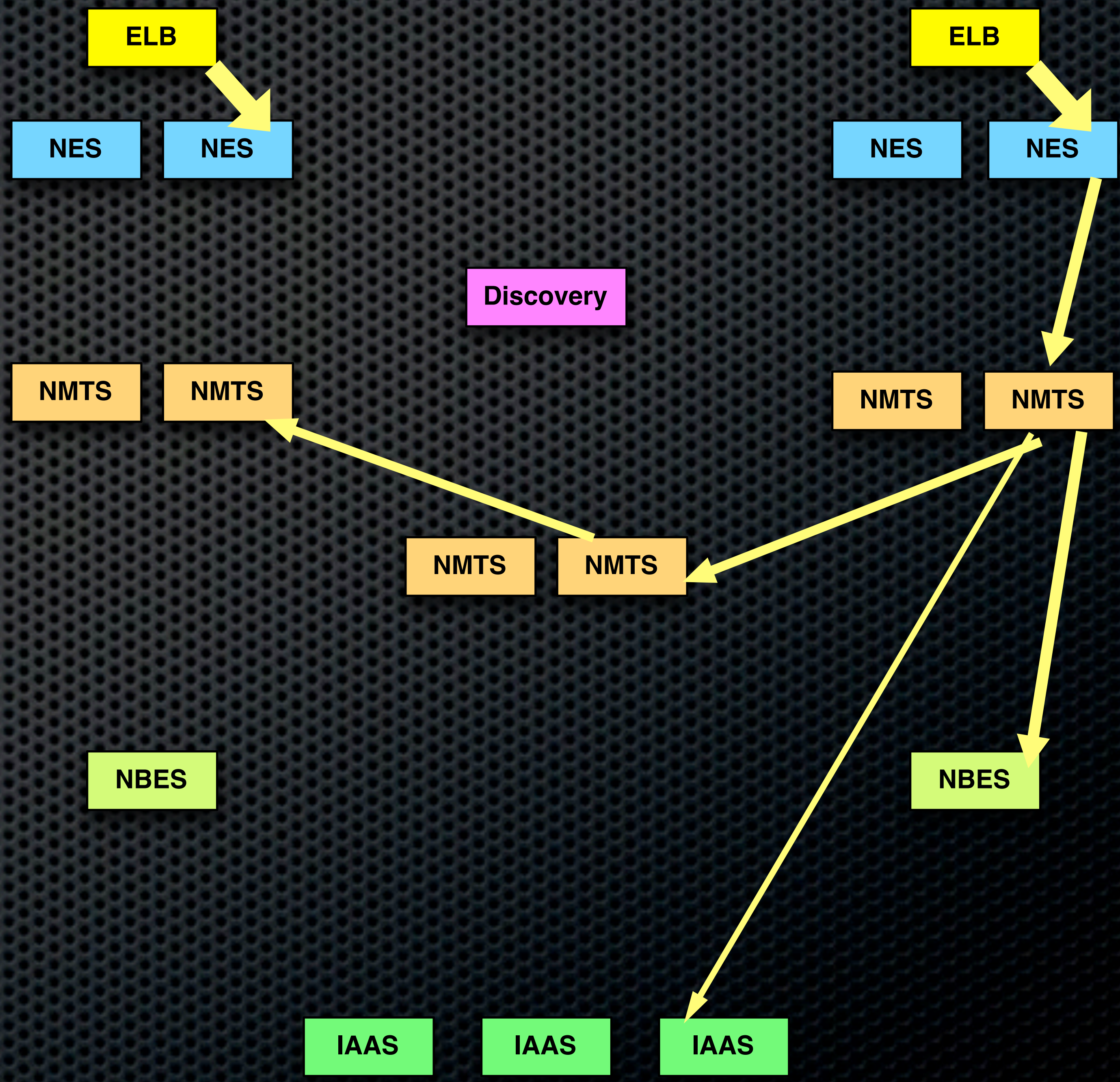
- Overview
  - A back-end, usually 3rd party, open-source service
  - Leaf in the call tree. Cannot call anything else



Netflix's Cloud Architecture

Components (NBES)

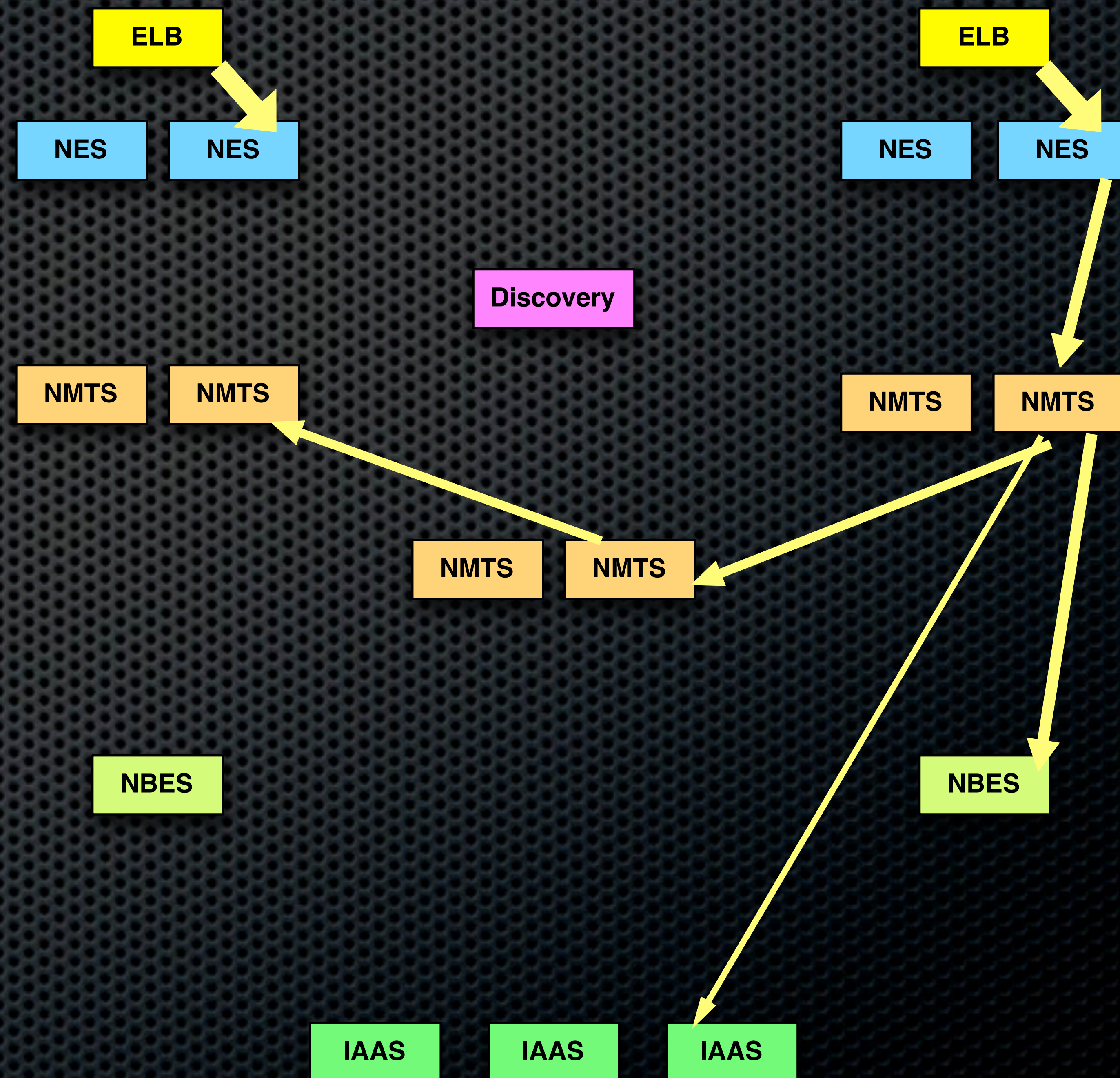
- Examples
  - Cassandra Clusters
    - Our new cloud database is Cassandra and stores all sorts of data to support application needs
  - Zookeeper Clusters
    - Our distributed lock service and sequence generator
  - Memcached Clusters
    - Typically caches things that we store in S3 but need to access quickly or often



Netflix's Cloud Architecture

Components (IAAS)

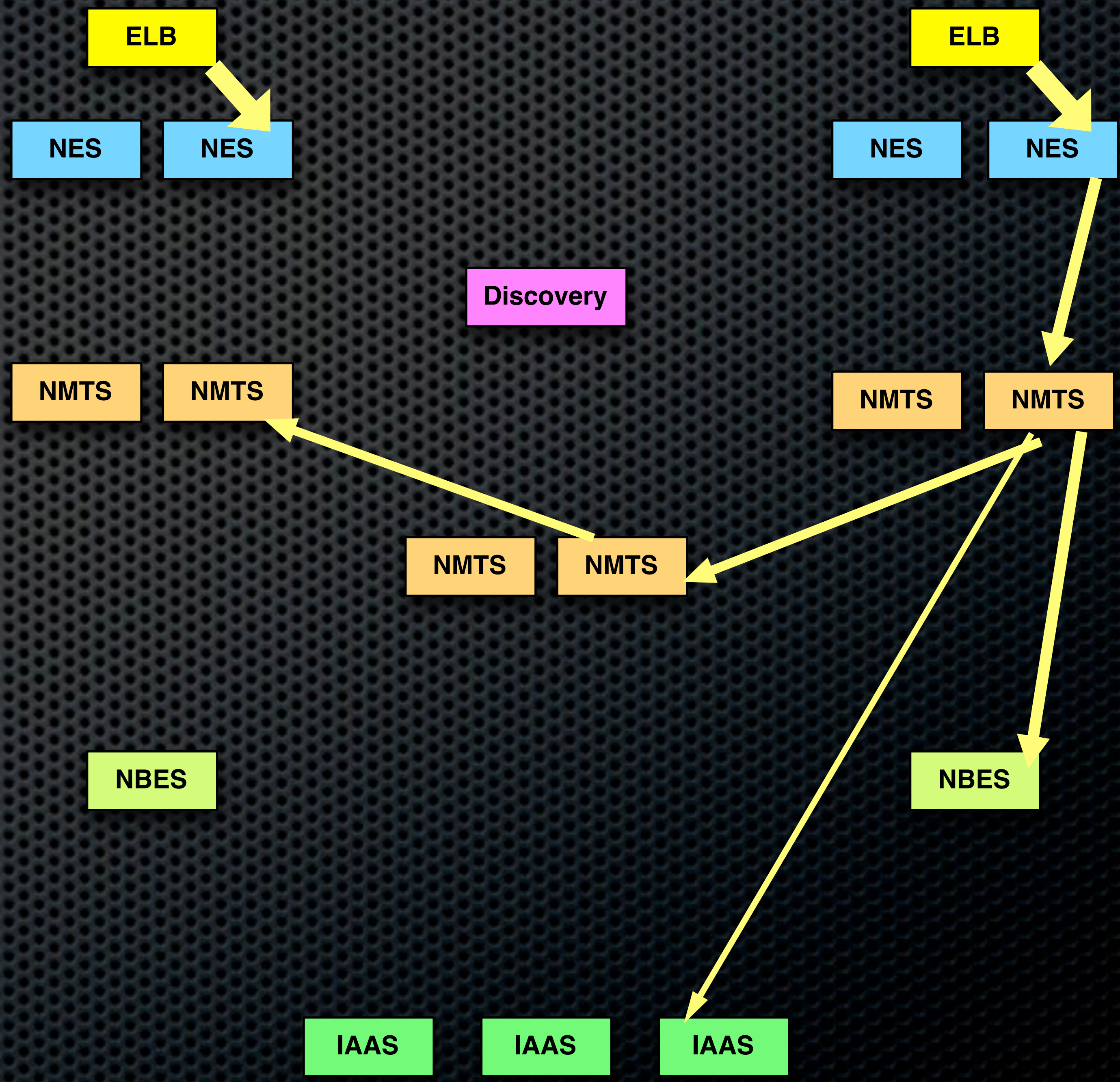
- Examples
  - AWS S3
    - Large-sized data (**e.g. video encodes, application logs, etc...**) is stored here, not Cassandra
  - AWS SQS
    - Amazon's message queue to send events (**e.g. Facebook network updates are processed asynchronously over SQS**)



Netflix's Cloud Architecture

Types of Production Issues

- A user-issued call will pass through multiple levels during normal operation
- We are now exposed to multi-system coincident failures, a.k.a. coordinated failures



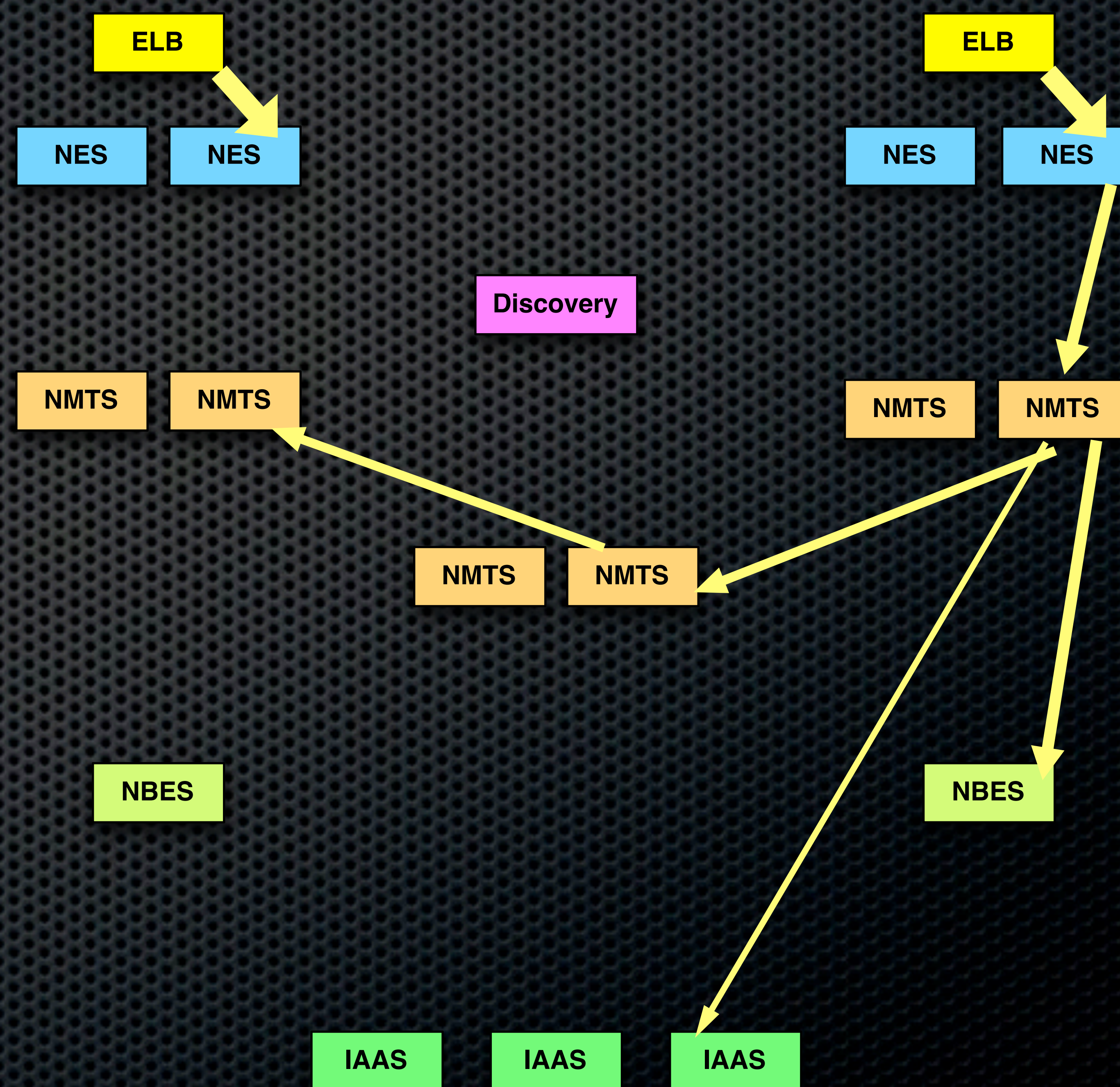
Netflix’s Cloud Architecture

Architecture Pros

- Horizontally scalable at every level
    - Should give us maximum availability
- Supports high-velocity development and deployment

Architecture Cons

- A user-issued call will pass through multiple levels (**a.k.a. hops**) during normal operation
    - Latency can be a concern
- We are now exposed to multi-system coincident failures, a.k.a. coordinated failures
- A lot of moving parts

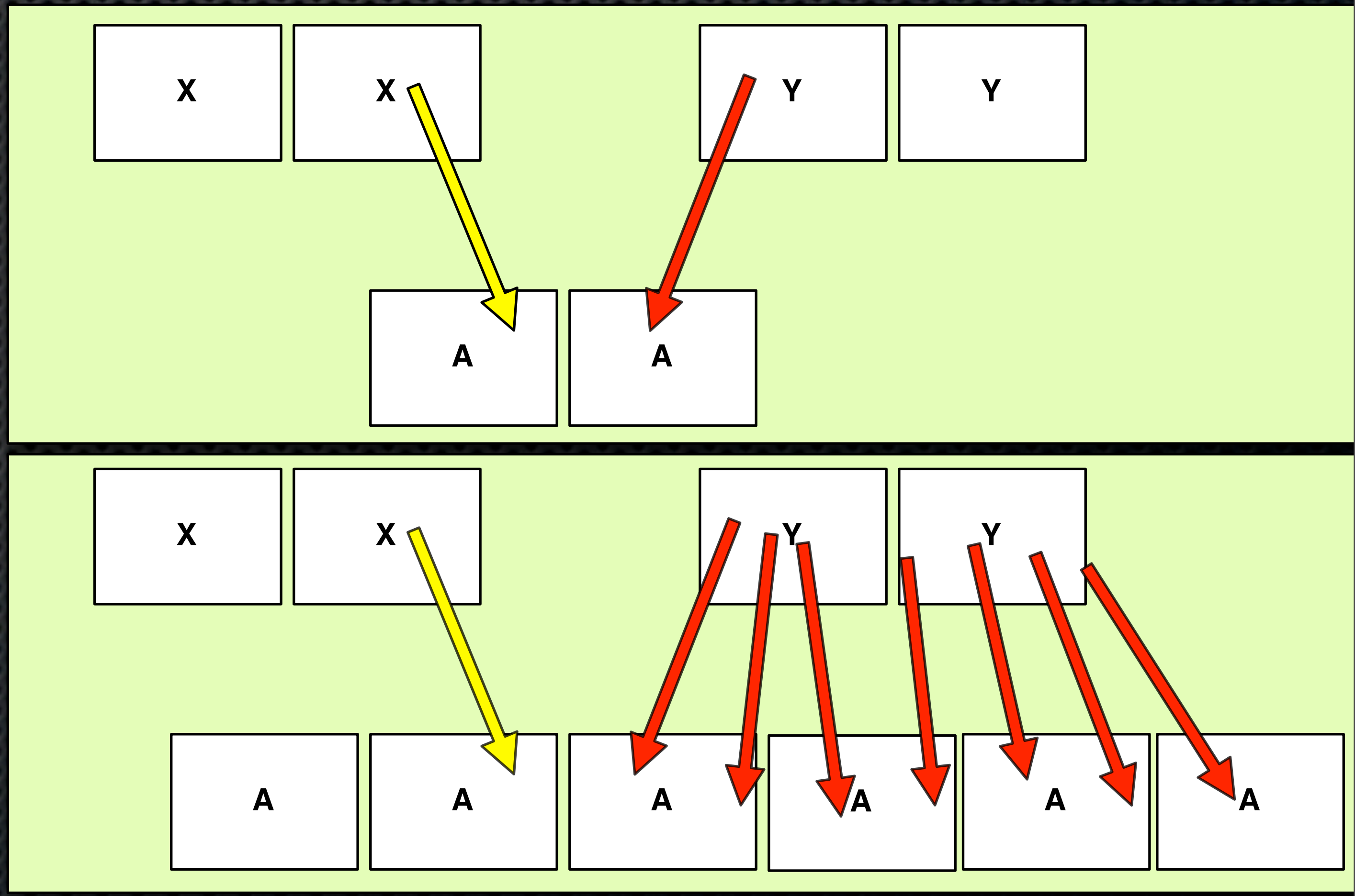


Issue 1

Capacity Planning

Issue 1

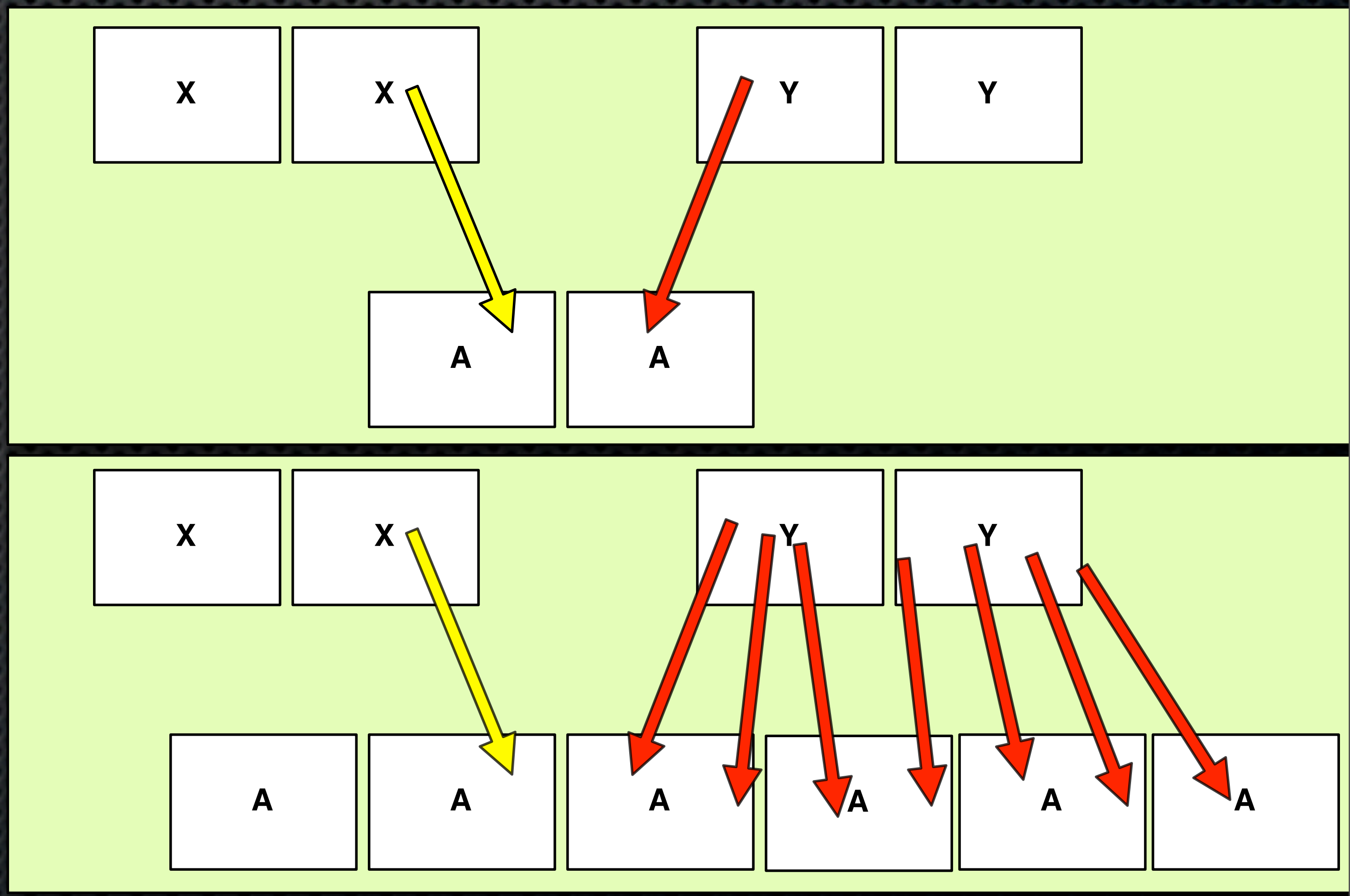
- Service **X** and Service **Y**, each made up of 2 instances, call Service **A**, also made up of 2 instance
- If either of these services expect a large increase in traffic, they need to let the owner of Service **A** know
- Service **A** can then scale up ahead of the traffic increase



Disaster Avoided ??

Issue 1

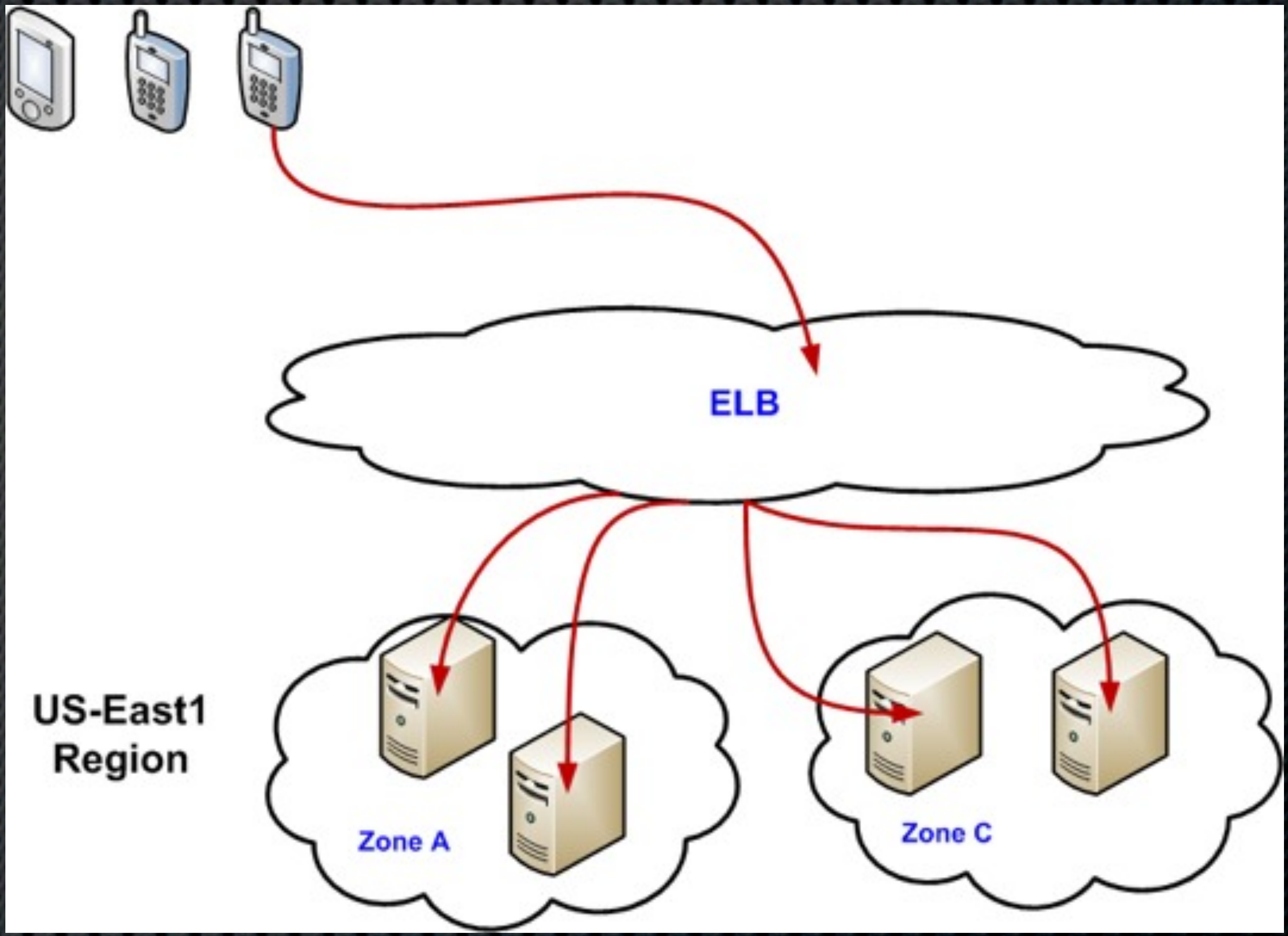
- A given application owner may need to contact 20 other application owners each time he expects to get a large increase in traffic
    - Too much human coordination
- A few options
    - Some service owners vastly over-provision for their application
        - Not cost effective
    - Auto-scaling
        - We want to generalize the model first proved by our Streaming Control Server (a.k.a. NCCP) team



ELB AutoScaling Interlude

How to use an ELB

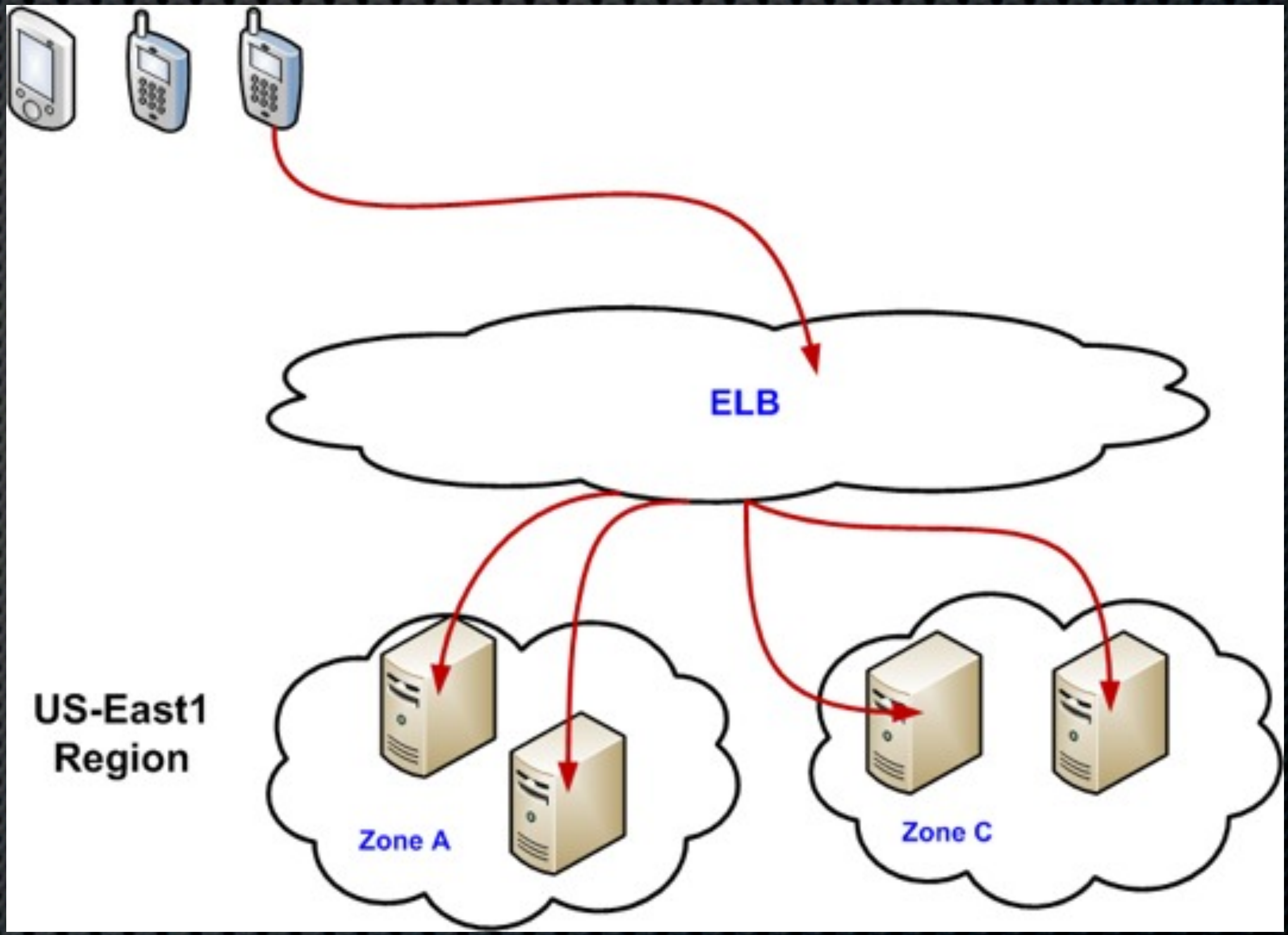
- An elastic-load balancer (ELB) routes traffic to your EC2 instances
  - e.g. of an ELB : nccp-wii-11111111.us-east-1.elb.amazonaws.com
- Netflix maps a CNAME to this ELB
  - e.g. : nccp.wii.netflix.com
- Netflix then registers the API Service's EC2 instances with this ELB
- The ELB periodically polls attached EC2 instances to ensure the instances are healthy



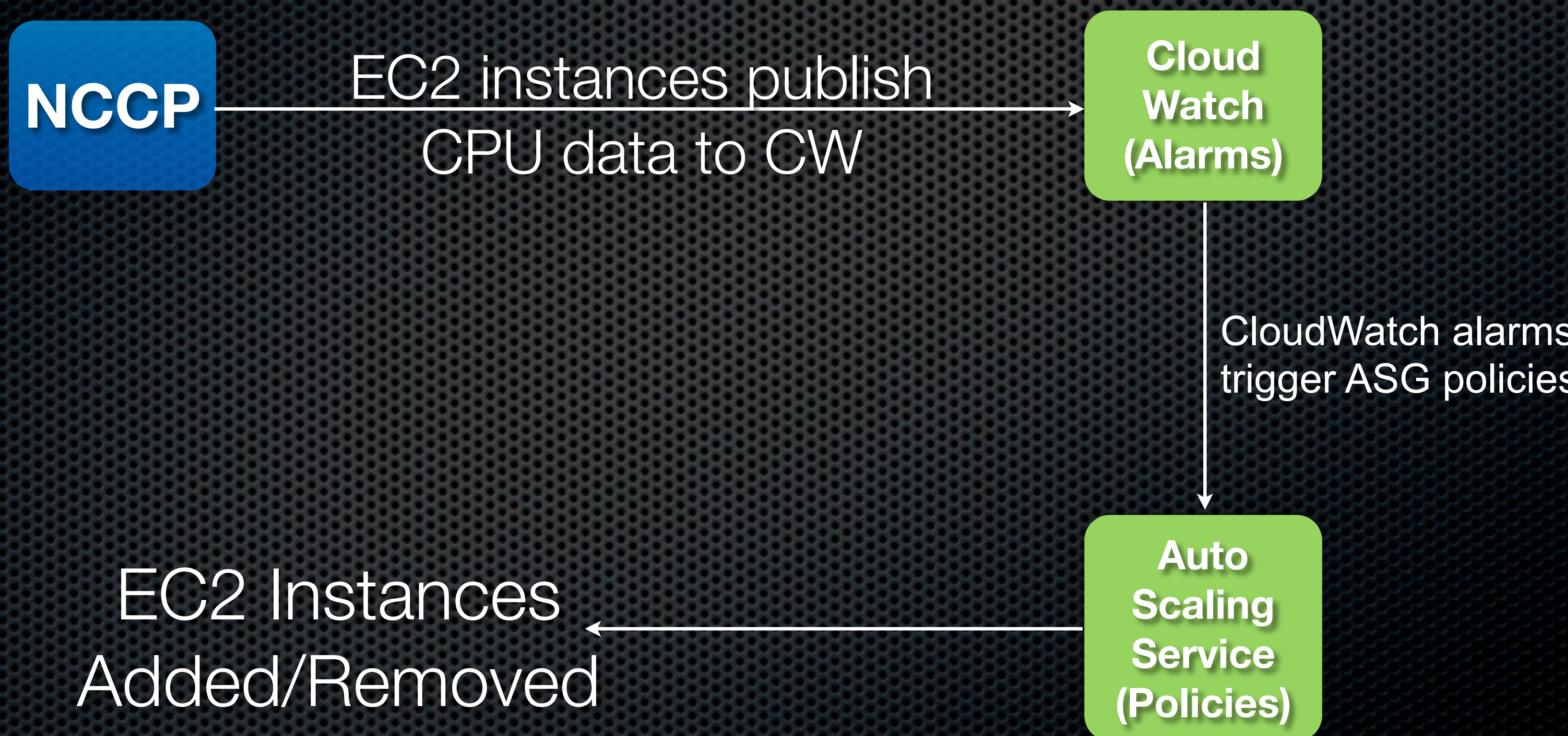
ELB AutoScaling Interlude

Taking this a bit further

- The NCCP servers can publish metrics to AWS CloudWatch
- We can set up an alarm in Cloud Watch on a metric (**e.g. CPU**)
- We can associate an auto scale policy with that alarm (**e.g. if CPU > 60%, add 3 more instances**)
- When a metric goes above a limit, an alarm is triggered, causing auto-scaling, which grows our pool

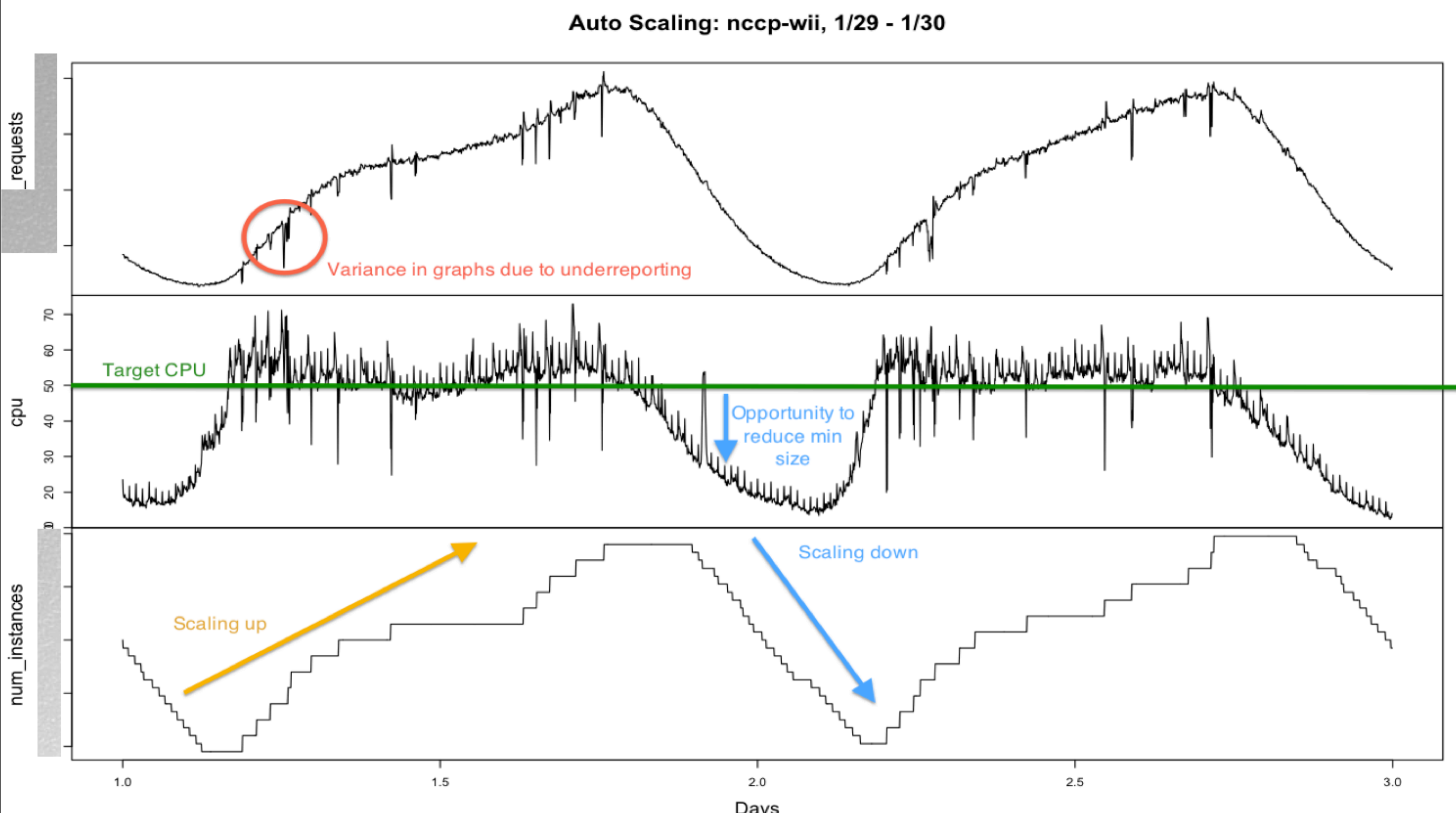


ELB AutoScaling Interlude



ELB AutoScaling Interlude

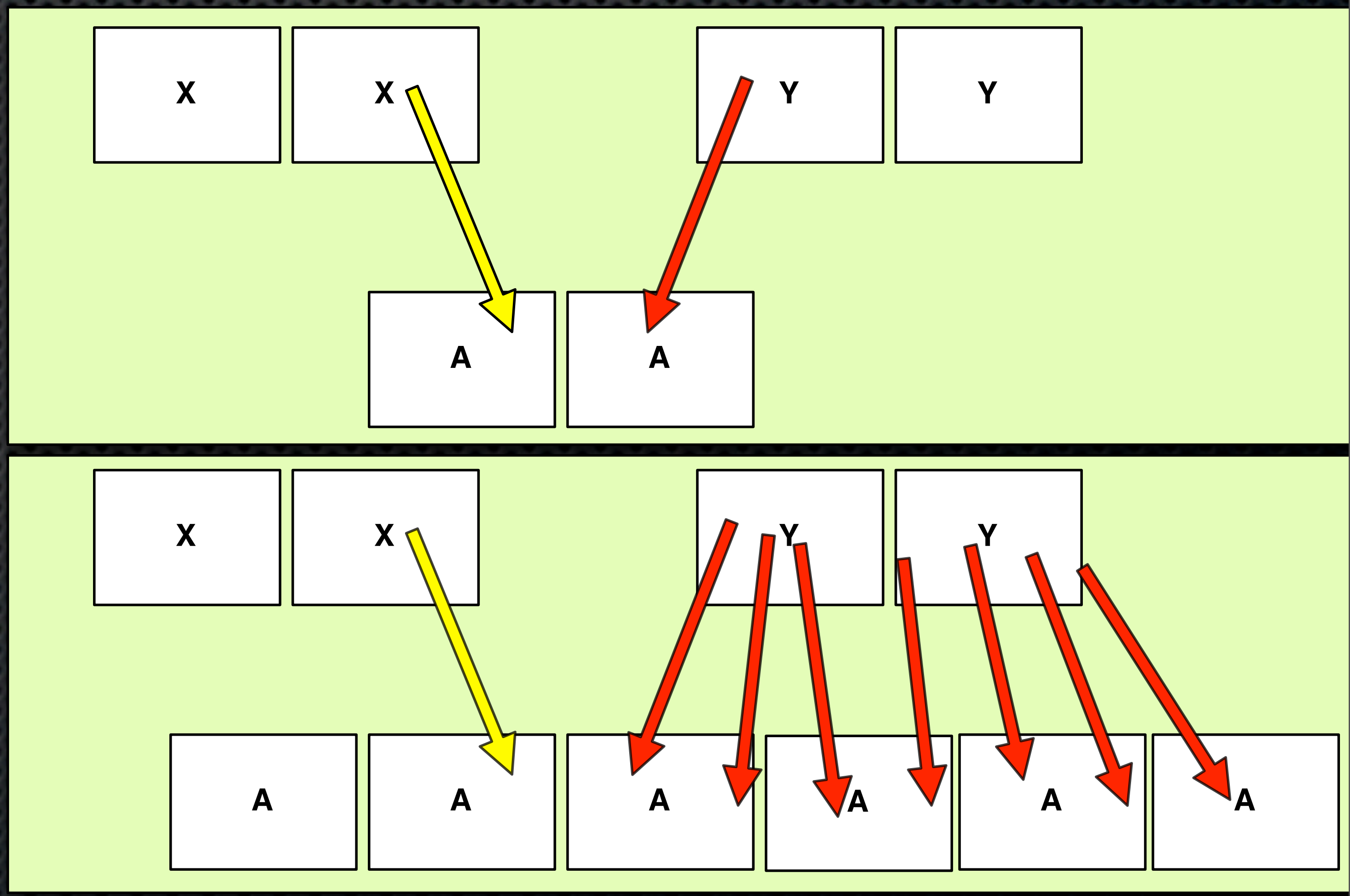
Scale Out Event	Average CPU > 60% for 5 minutes
Scale In Event	Average CPU < 30% FOR 5 minutes
Cool Down Period	10 minutes
Auto-Scale Alerts	DLAutoScaleEvents


Auto Scaling: nccp-wii, 1/29 - 1/30
_requests
Variance in graphs due to underreporting
cpu
70
60
50
40
30
20
Target CPU
Opportunity to reduce min size
num_instances
Scaling up
Scaling down
1.0
1.5
2.0
2.5
3.0
Days

Issue 1

Summary

We would like to have auto-scaling at all levels.



Issue 2

Thundering herds to NMTS

Issue 2

Step 1

Service **X** and Service **Y**, each made up of 2 instances, call Service **A**, also made up of 2 instance

Step 2a

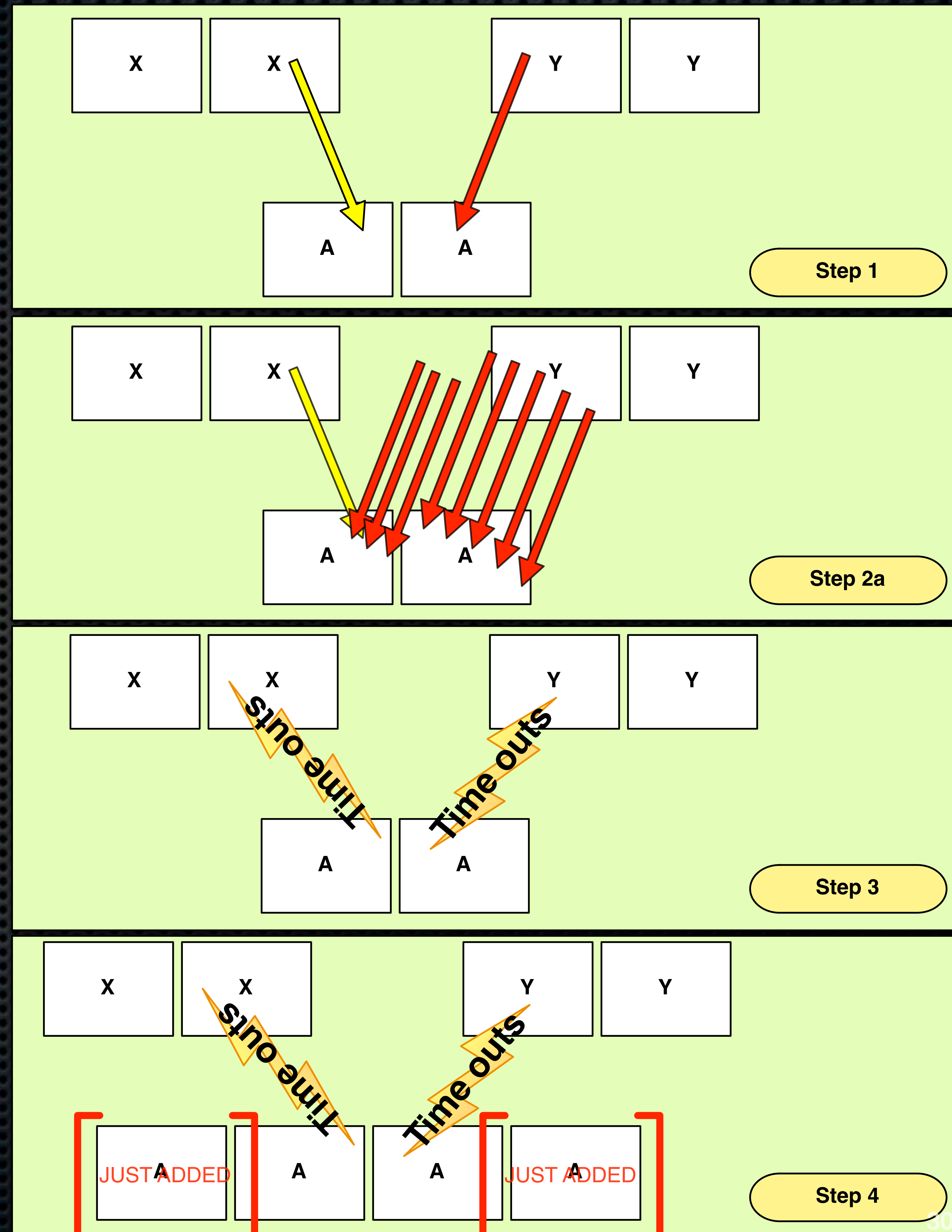
Service **Y** overwhelms Service **A**

Step 3

Services **X** & **Y** experience read and connection timeouts against an overwhelmed Service **A**

Step 4

Service **A**'s tier get 2 more machines



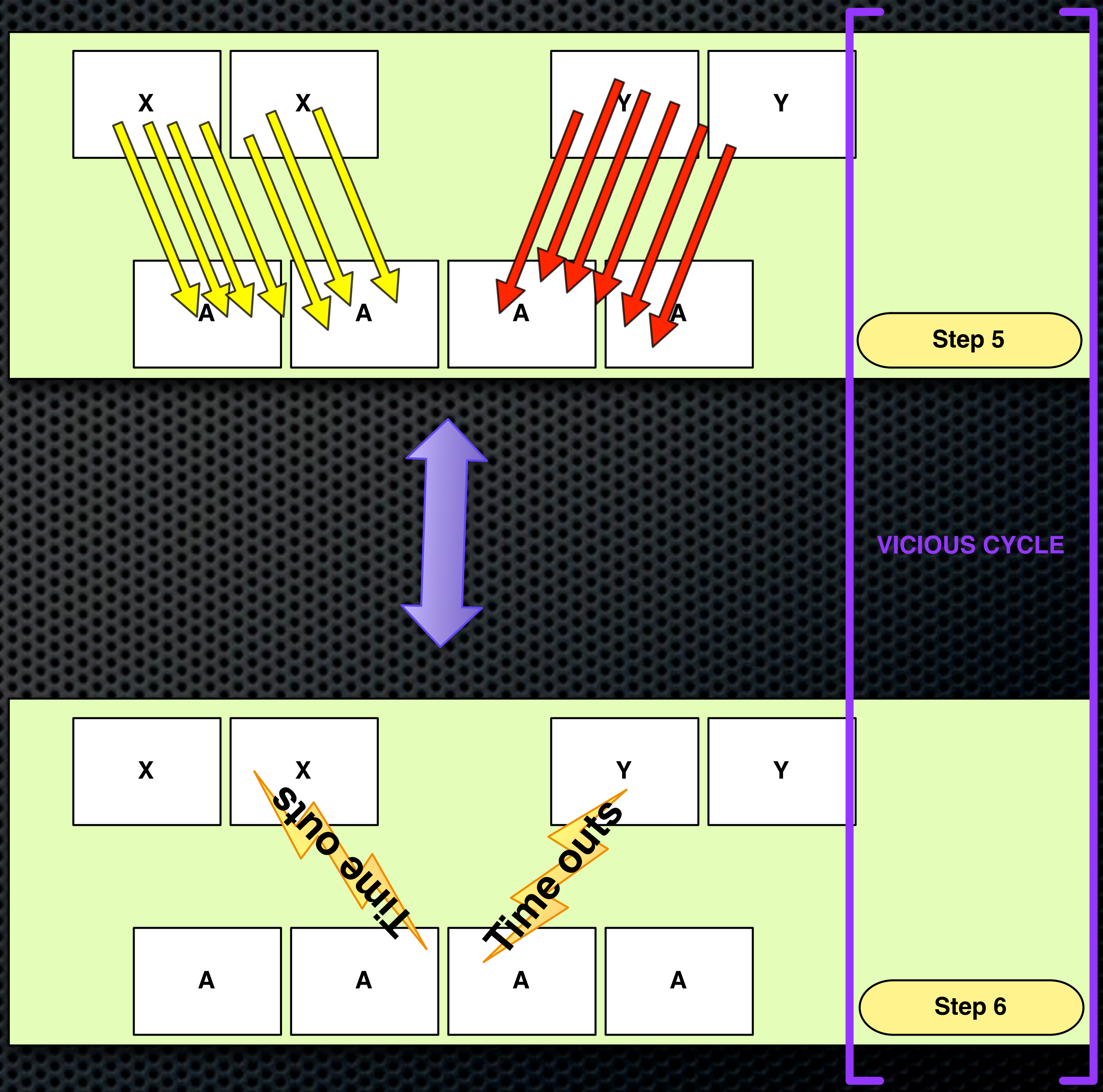
Issue 2

Step 5

- New requests + Retries cause request storms (**a.k.a. thundering herds**)
- If Service **A** can be grown to exceed retry storm steady-state traffic volume, we can exit this vicious cycle

Step 6

- Else, more timeouts, and VC continues



Issue 2

Step 1

Service **X** and Service **Y**, each made up of 2 instances, call Service **A**, also made up of 2 instance

Step 2b

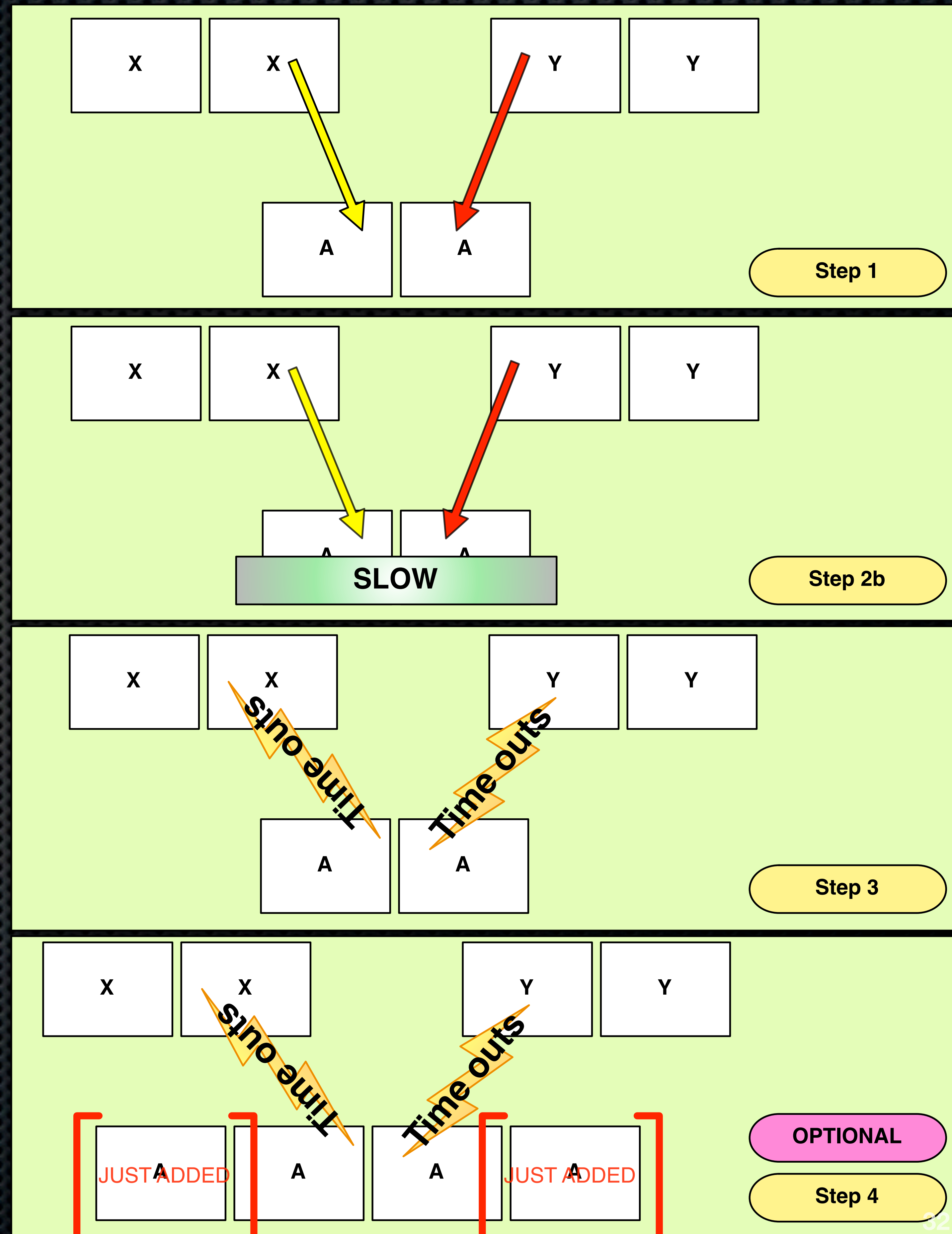
Service **A** experiences slowness

Step 3

Services **X** & **Y** experience read and connection timeouts against a slower Service **A**

Step 4

If the slowness can be fixed by adding more machines to Service **A**'s tier, then do so



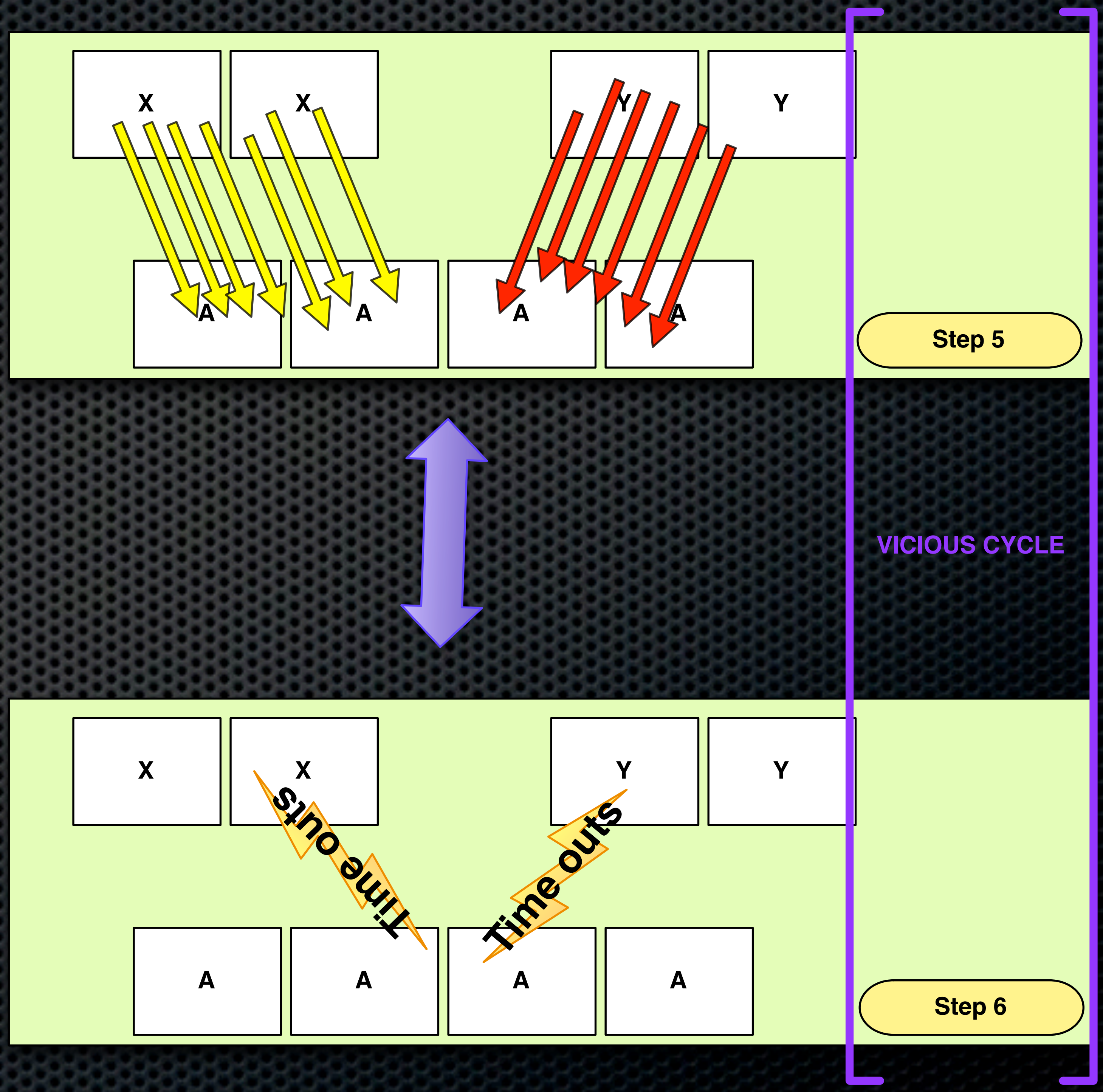
Issue 2

Step 5

- New requests + Retries cause request storms (**a.k.a. thundering herds**)
- If Service **A** can be grown to exceed retry storm steady-state traffic volume, we can exit this vicious cycle

Step 6

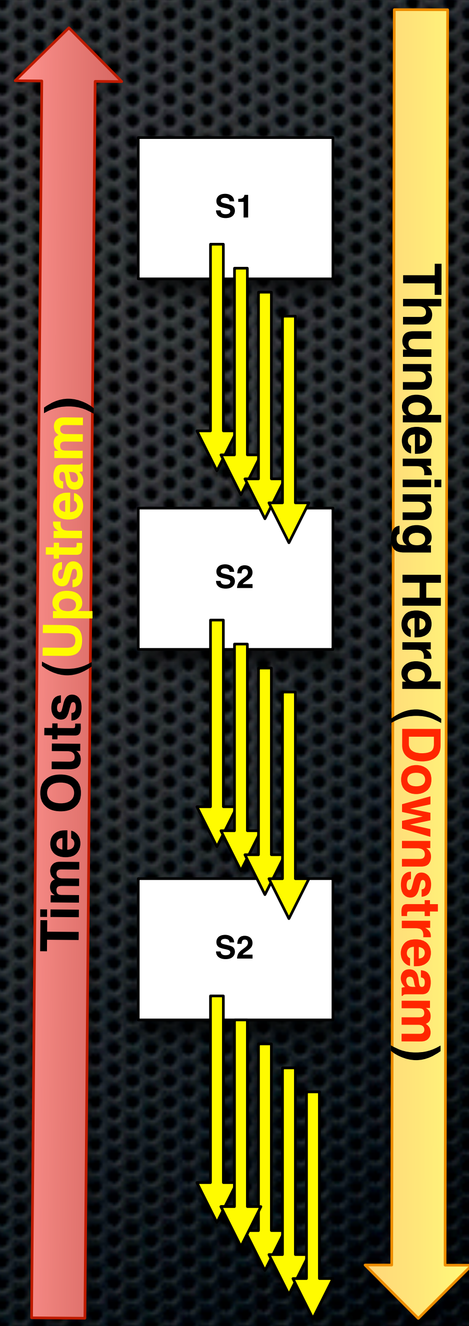
- Else, more timeouts, and VC continues



Issue 2

Potential Causes of Thundering Herd

- Service **Y** sends more traffic to Service **A**, without checking if Service **A** has available capacity
- Service **A** slows down
- Service **Y**'s time outs against Service **A** are set too low
- Service **Y**'s retries against Service **A** are too aggressive
- Natural organic growth in traffic hit a tipping point in the system -- in Service **A** in this case



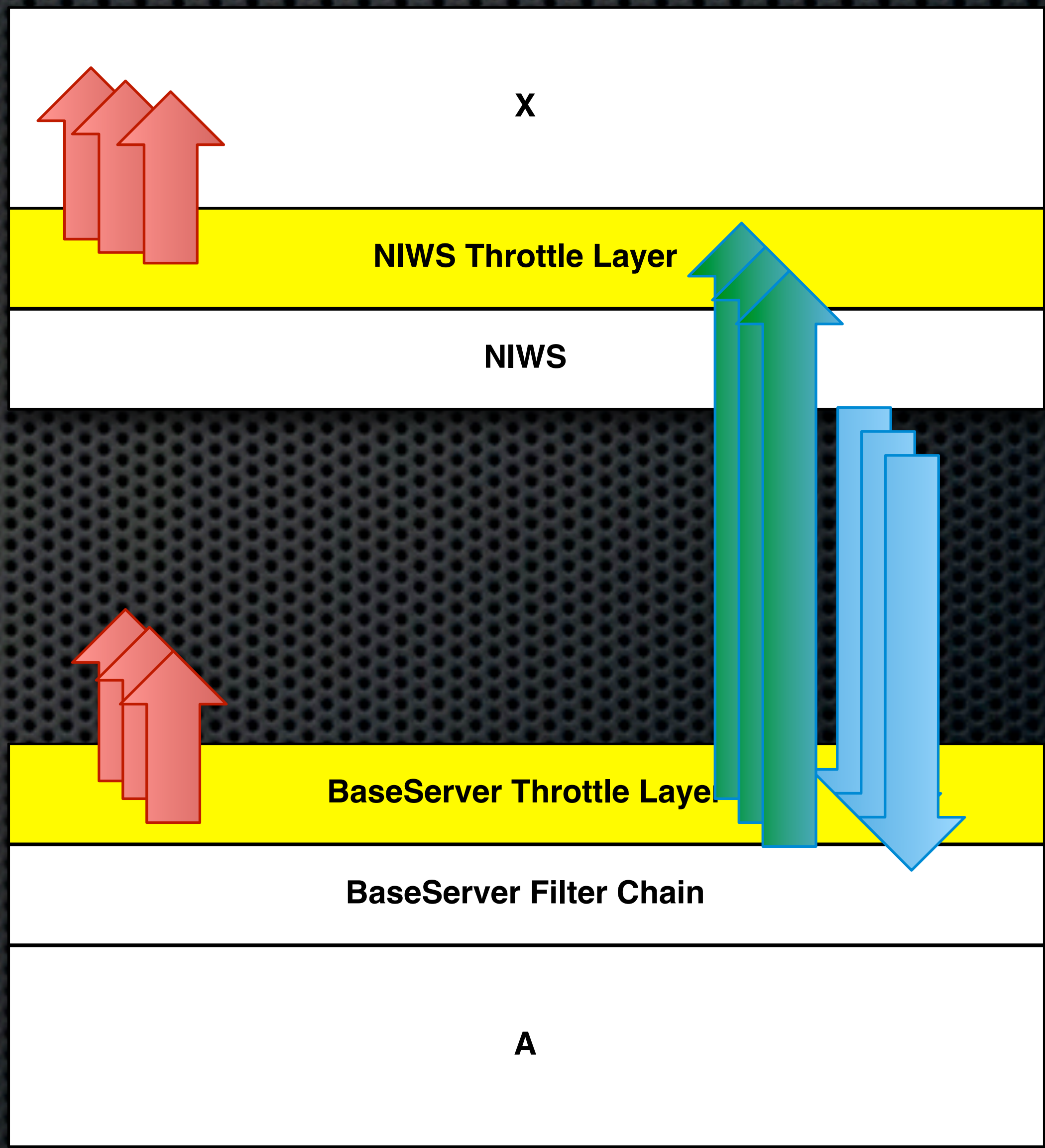
Solutions to Issue 2

Thundering herds to NMTS

Solutions to Issue 2

The Platform Solution

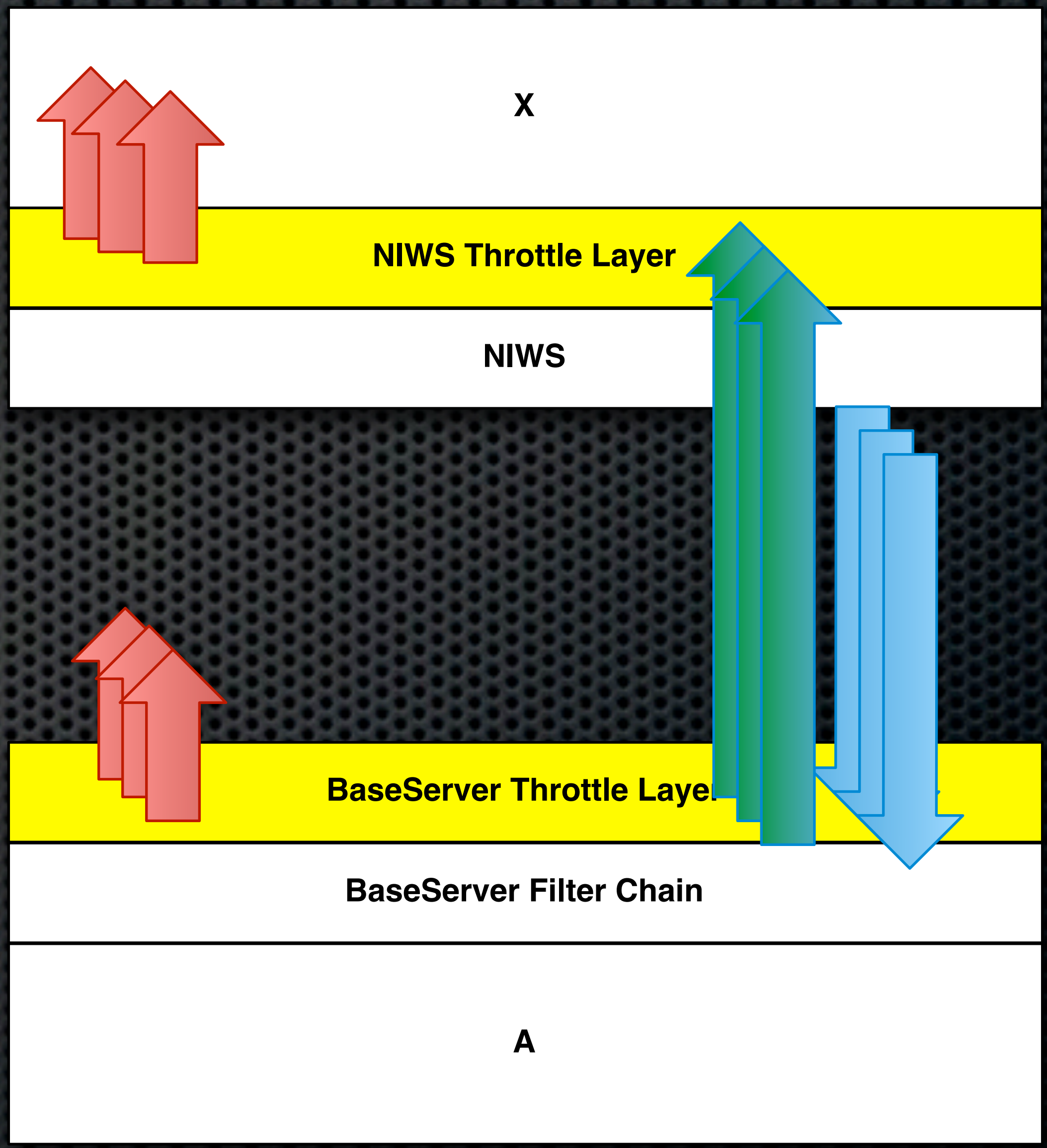
- Every service at Netflix sits on the platform.jar
- The platform.jar offers 2 components of interest here:
    - **NIWS library** : the client-side of Netflix Inter-Web Service calls. Handles retry, failover, thundering-herd prevention, & fast failure
    - **BaseServer library** : a set of Tomcat servlet filters that protect the underlying application servlet stack. In this context, it throttles traffic



Solutions to Issue 2

The Platform Solution

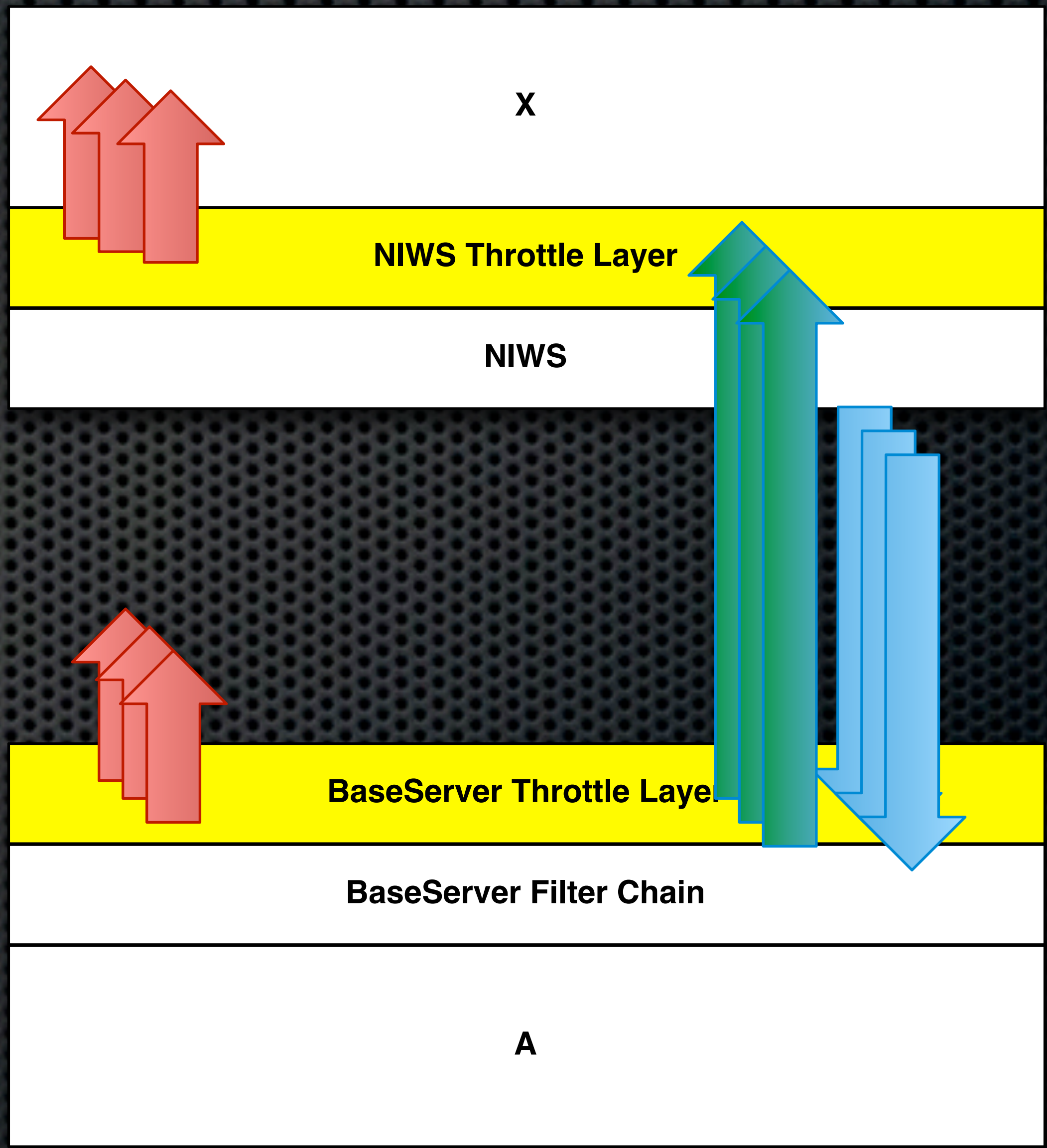
- **NIWS library**
  - Fair Retry Logic : e.g. exponential bounded backoff
  - Takes 2 configuration params per client:
    - Max_Num_of_Requests (**a.k.a. MNR**)
    - Sample_Interval_in_seconds (**a.k.a. SI**)
  - Ensures that a client does not send more than **MNR/SI** requests/s, else throttles requests at the client



Solutions to Issue 2

The Platform Solution

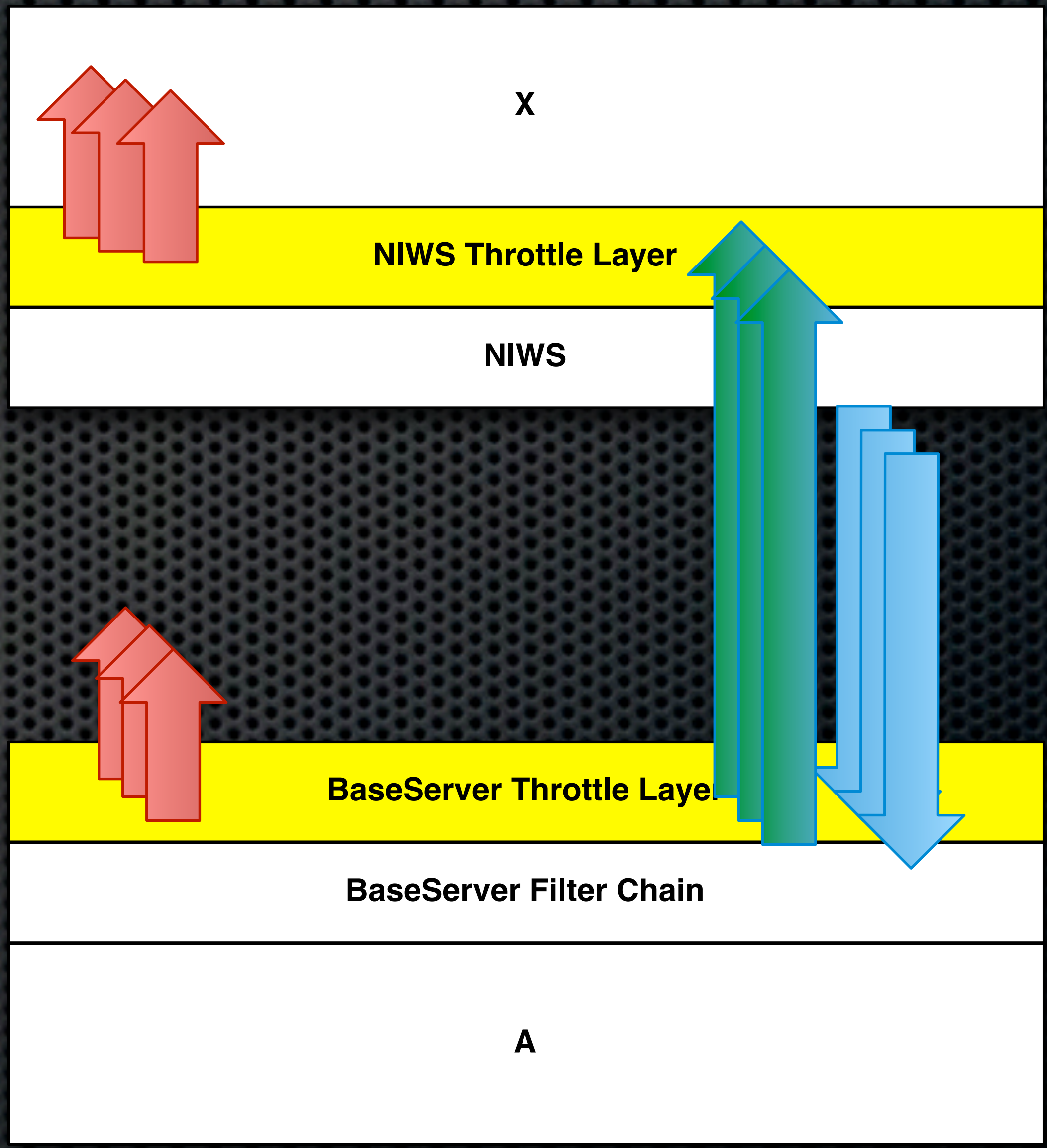
- **BaseServer**
  - As an additional fail-safe, the server can set throttles that are not client specific (**i.e. the limits apply to total inbound traffic, regardless of client**)
  - Takes 1 configuration parameter:
    - Max_Num_of_Concurrent_Requests (**a.k.a. MNCR**)
  - Ensures that a server does not handle more than **MNCR** requests at any instant
  - If the traffic exceeds the limits, reject excess calls at the server (**i.e. 503s**)



Solutions to Issue 2

The Platform Solution

- **Graceful Degradation**
  - Any client that is throttled at either the **NIWS Throttle Layer** or the **BaseServer Throttle Layer** need to implement graceful degradation
  - Netflix's Web Scale Traffic falls in 2 categories:
    - Users get a personalized set of movies to pick from (**i.e. via API Edge Server path**)
      - **GD** : Show popular movies, not personalized movies
    - Users can start watching a movie (**i.e. via NCCP Edge Server path**)
      - **GD** : tougher problem to solve
        - When device leases expire, we honor them if we are unable to generate a new one for them



Solutions to Issue 2

This all sounds great!

- But, what if developers do not use these built-in features of the platform or neglect to set their configuration appropriately?
    - (**i.e. the default RPS limit in the NIWS client is Integer.MAX_VALUE**)

Solutions to Issue 2

We have a little help

Simian Army

Prevention is the best medicine

Simian Army

- **Chaos Monkey**
  - Simulates hard failures in AWS by killing a few instances per ASG (**e.g. Auto Scale Group**)
    - Similar to how EC2 instances can be killed by AWS with little warning
  - Tests clients' ability to gracefully deal with broken connections, interrupted calls, etc...
  - Verifies that all services are running within the protection of AWS Auto Scale Groups, which reincarnates killed instances
    - If not, the Chaos monkey will win!

Simian Army

- **Latency Monkey**
  - Simulates soft failures -- i.e. a service gets slower
  - Injects random delays in NIWS (**client-side**) or BaseServer (**server-side**) of a client-server interaction in production
  - Tests the ability of applications to detect and recover (i.e. Graceful Degradation) from the harder problem of delays, that leads to thundering herd and timeouts

Simian Army

Does this solve all of our issues?

Simian Army

The infinite cloud is infinite when your needs are moderate!

To ensure fairness among tenants, AWS meters or limits every resource

Hence, we hit limits quite often. Our “velocity” is limited by how long it takes for AWS to turn around and raise the limit -- a few hours!

Simian Army

- **Limits Monkey**
  - Checks once a day whether we are approaching one of our limits and triggers alerts for us to proactively reach out to AWS!

- **Conformity & Janitor Monkeys**
  - Finds and clean up orphaned resources (**e.g. EC2 instances that are not in an ASG, unreferenced security groups, ELBs, ASGs, etc...**) to increase head-room
    - Buys us more time before we run out of resources and also saves us **$$$$**

NETFLIX

Simian Army

The Simian Army fills the gap created by an absence of process and a need to ensure fault-tolerance and efficient operation of our systems

NETFLIX

Fast Rollback

Fault-tolerant deployment

Fast Rollback

What is the point of having Fault-Tolerant layers if deployments of a bug can take them down?

Fast Rollback

Fast Rollback

Optimism causes outages

Fast Rollback

Optimism causes outages

Production traffic is unique

Fast Rollback

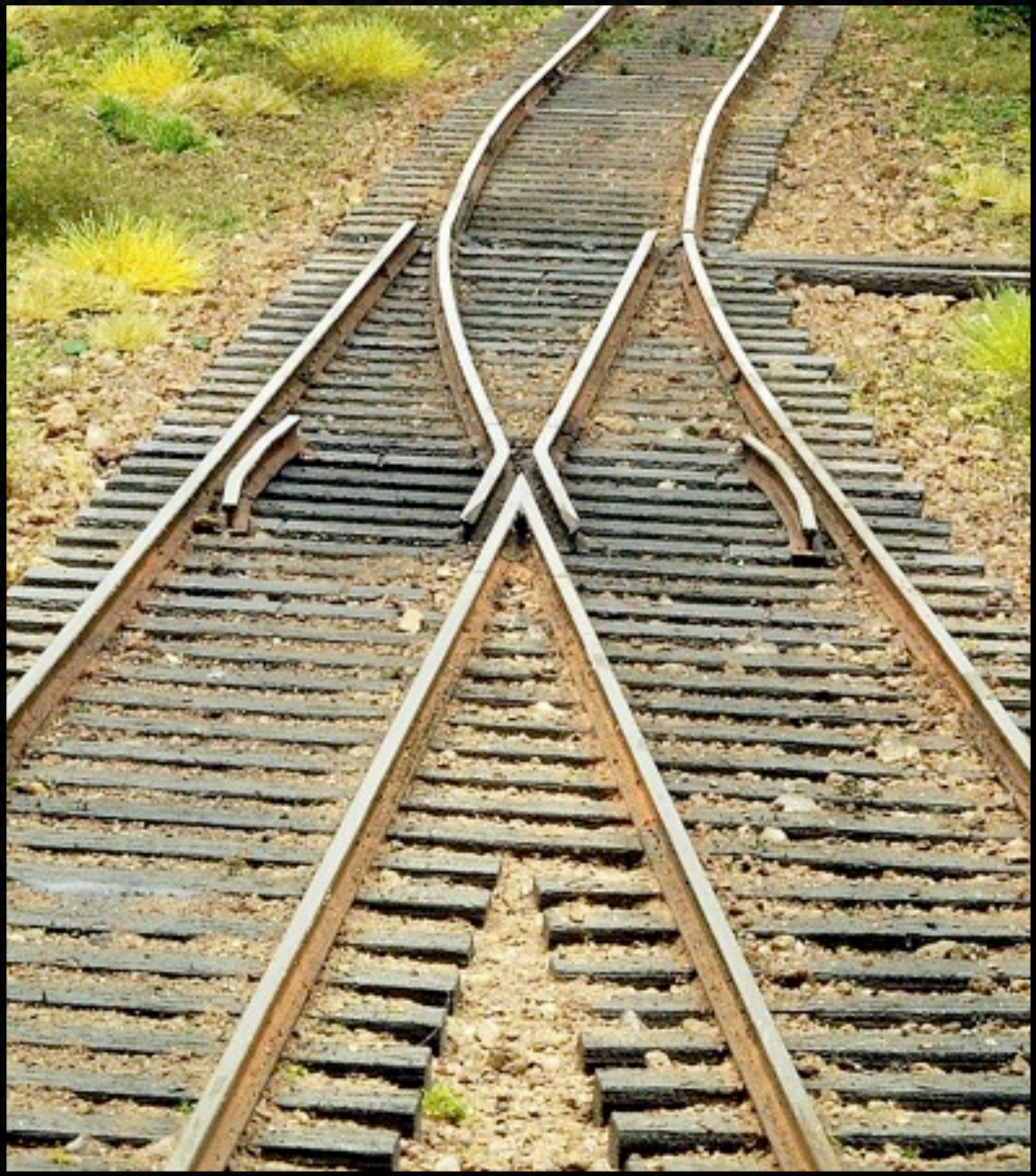
Optimism causes outages

Production traffic is unique

Keep old version running

Fast Rollback

- Optimism causes outages
- Production traffic is unique
- Keep old version running
- Switch traffic to new version

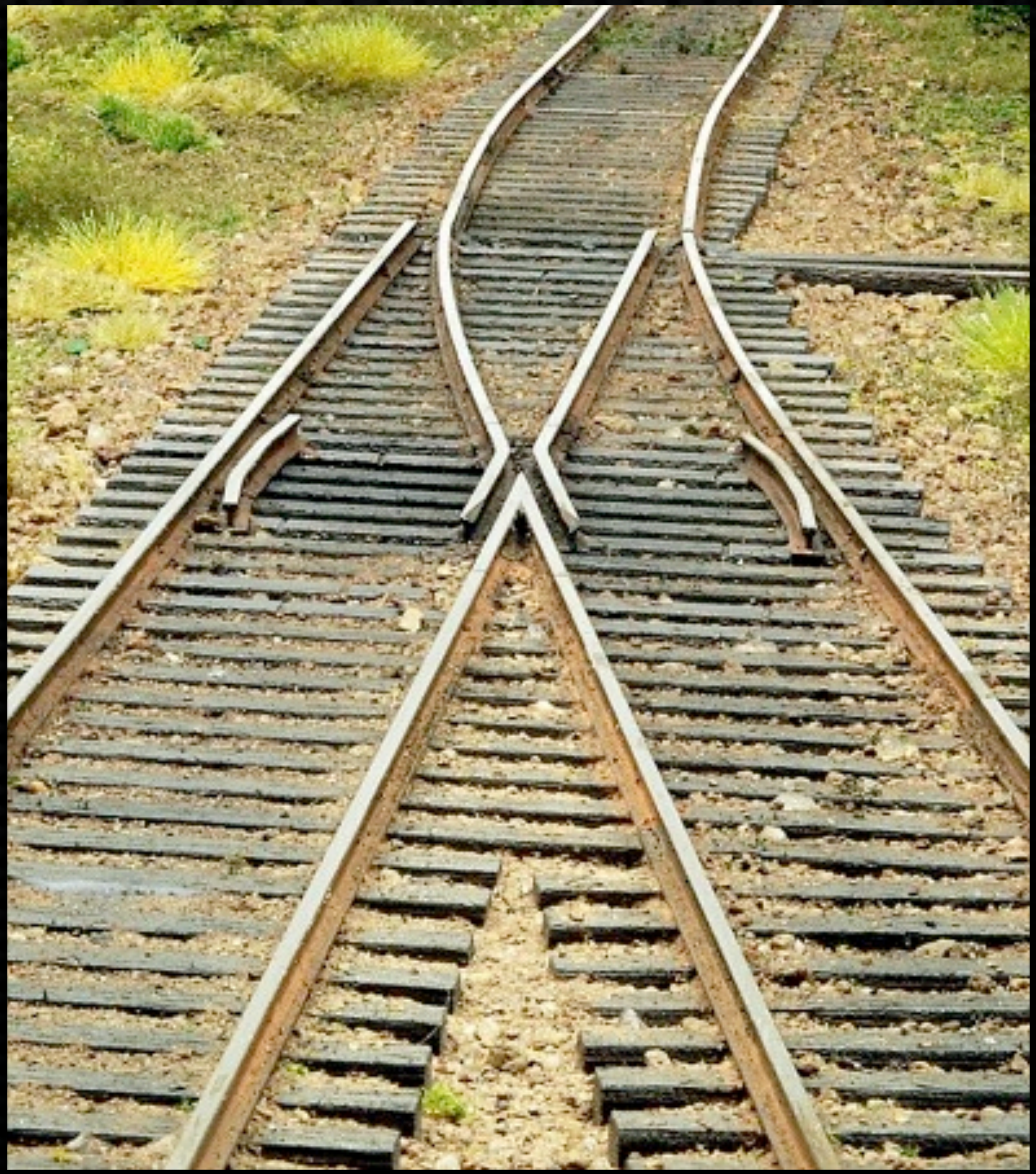


Fast Rollback

- Optimism causes outages
- Production traffic is unique
- Keep old version running
- Switch traffic to new version
- Monitor results

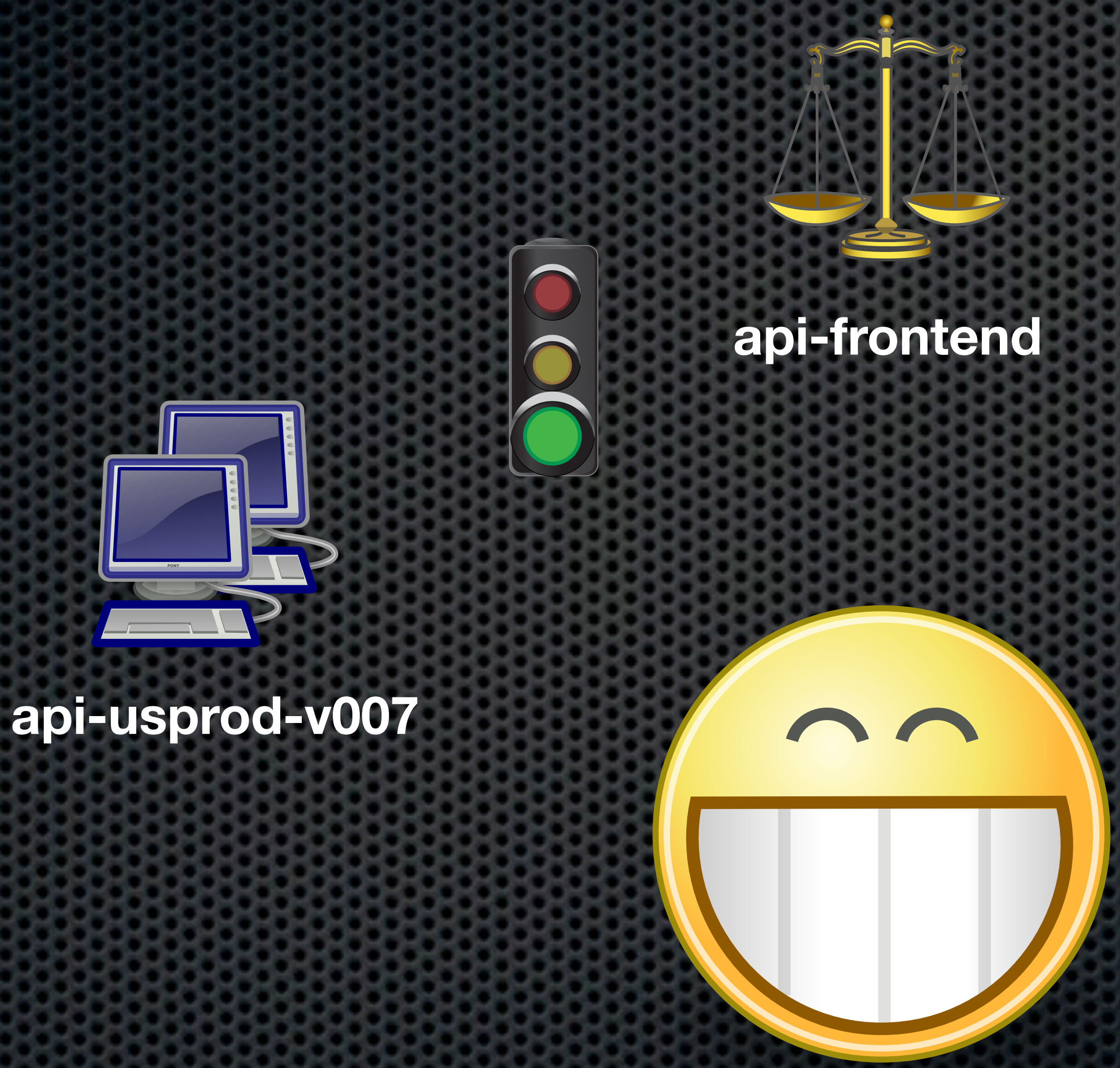
Fast Rollback

- Optimism causes outages
- Production traffic is unique
- Keep old version running
- Switch traffic to new version
- Monitor results
- Revert traffic quickly

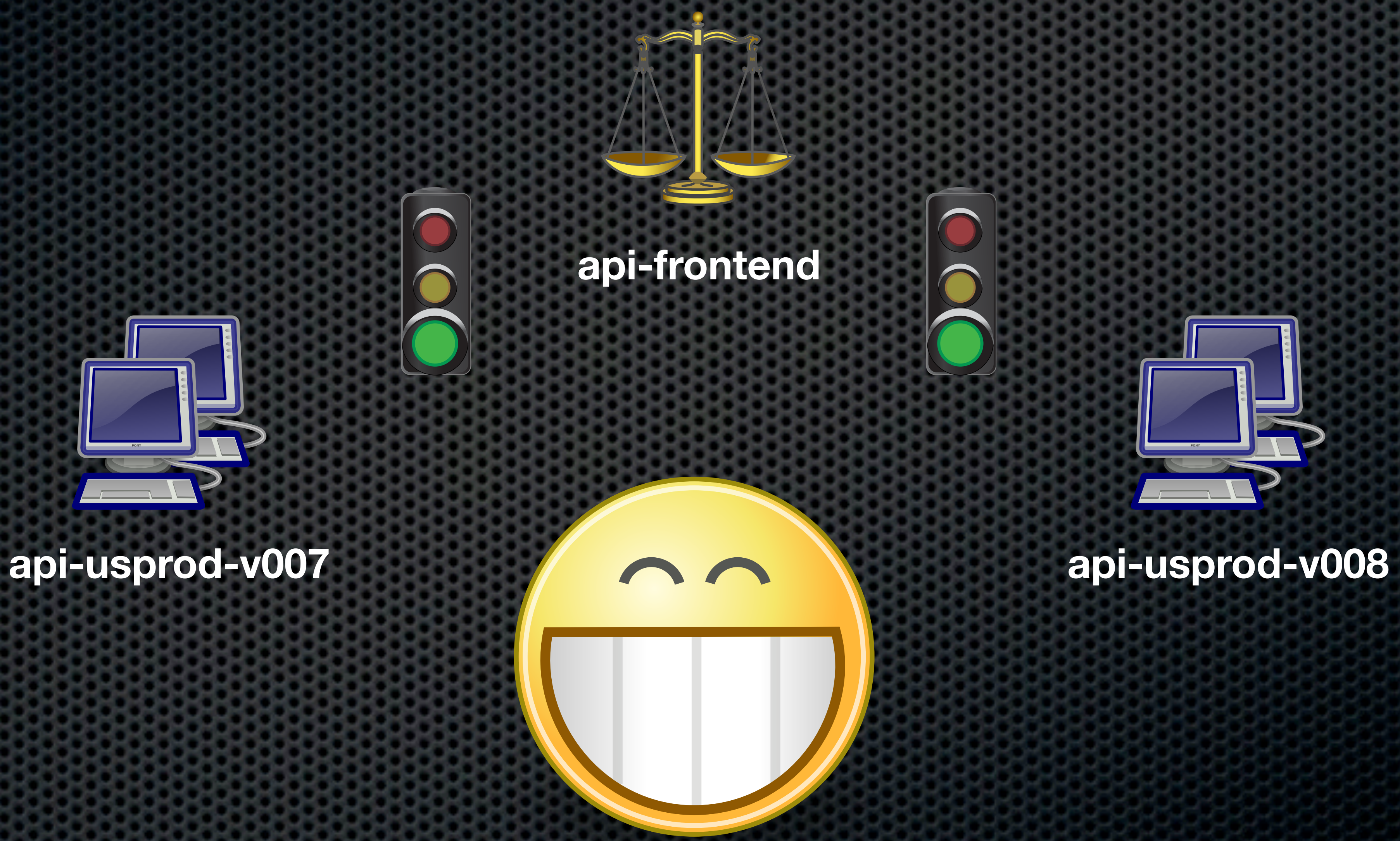


Fast Rollback

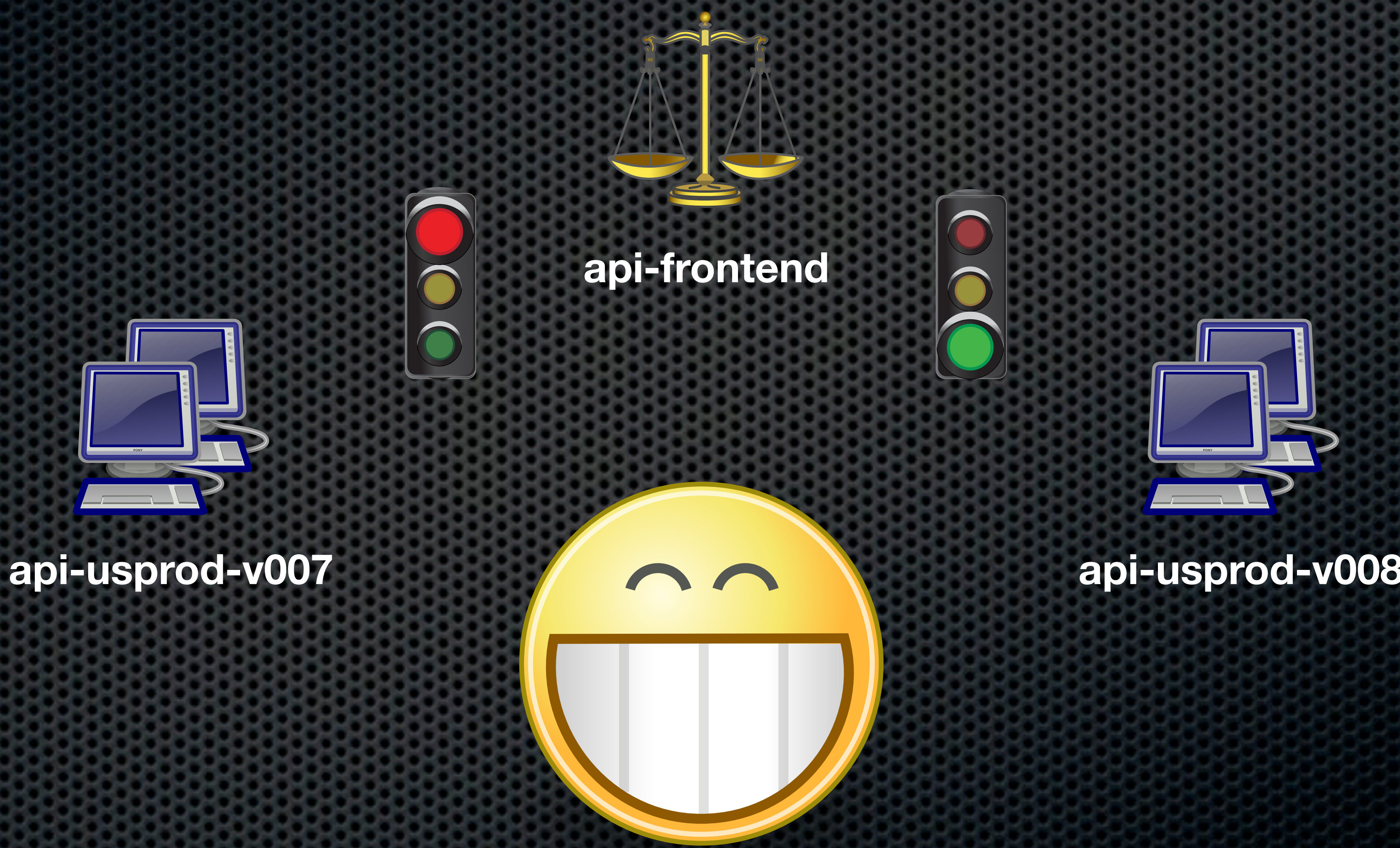
Fast Rollback



Fast Rollback

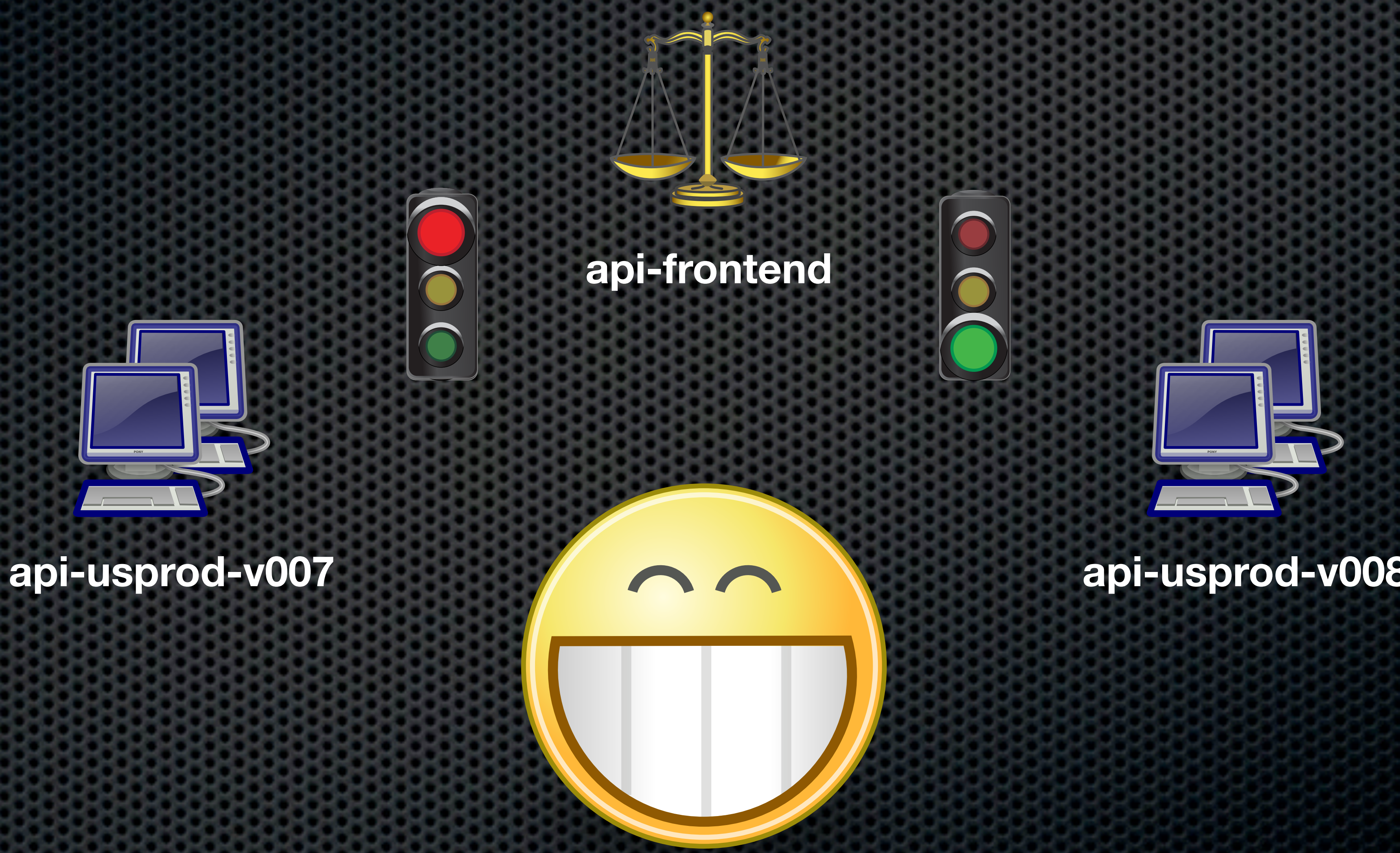


Fast Rollback



Fast Rollback

Fast Rollback

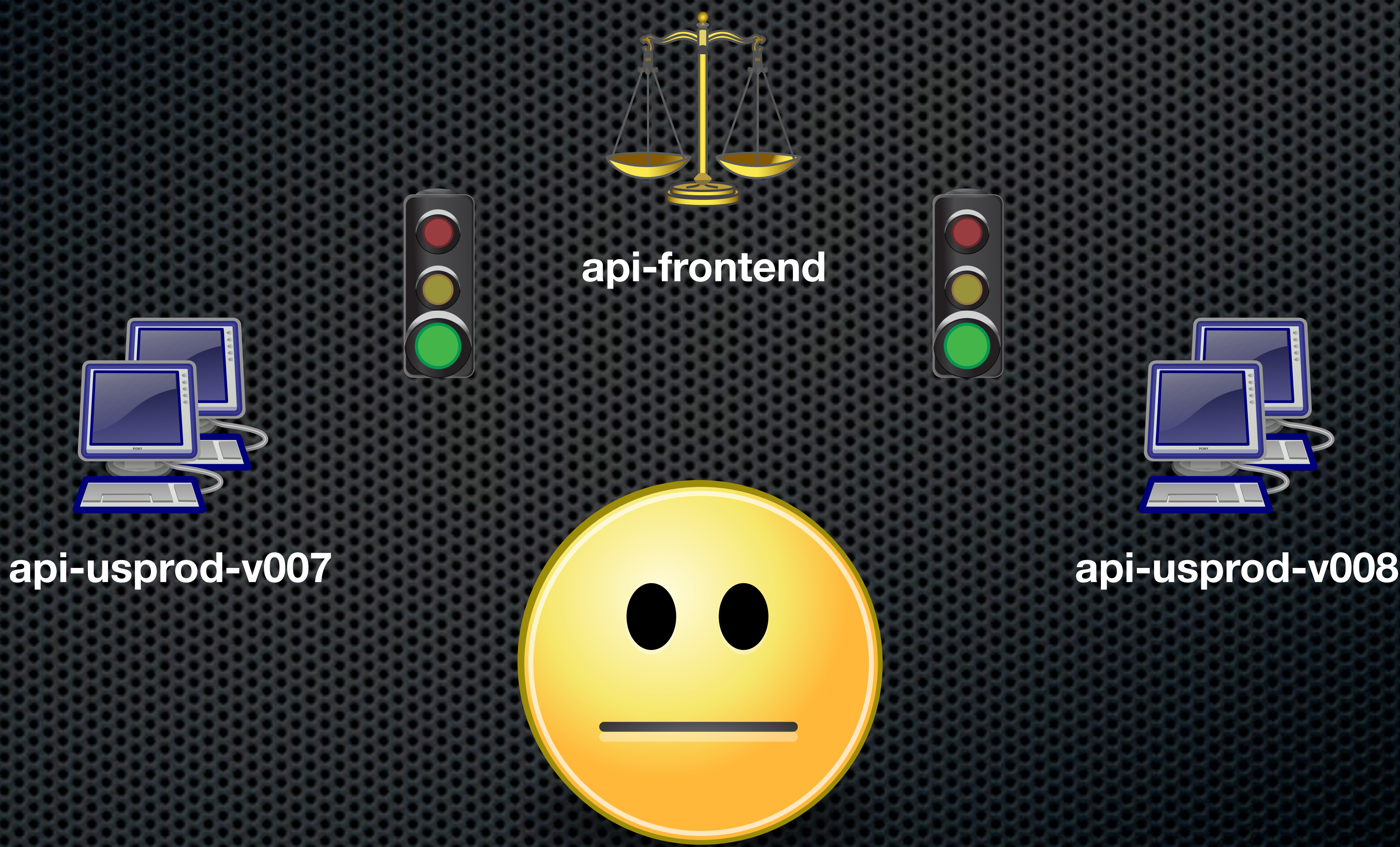


Fast Rollback

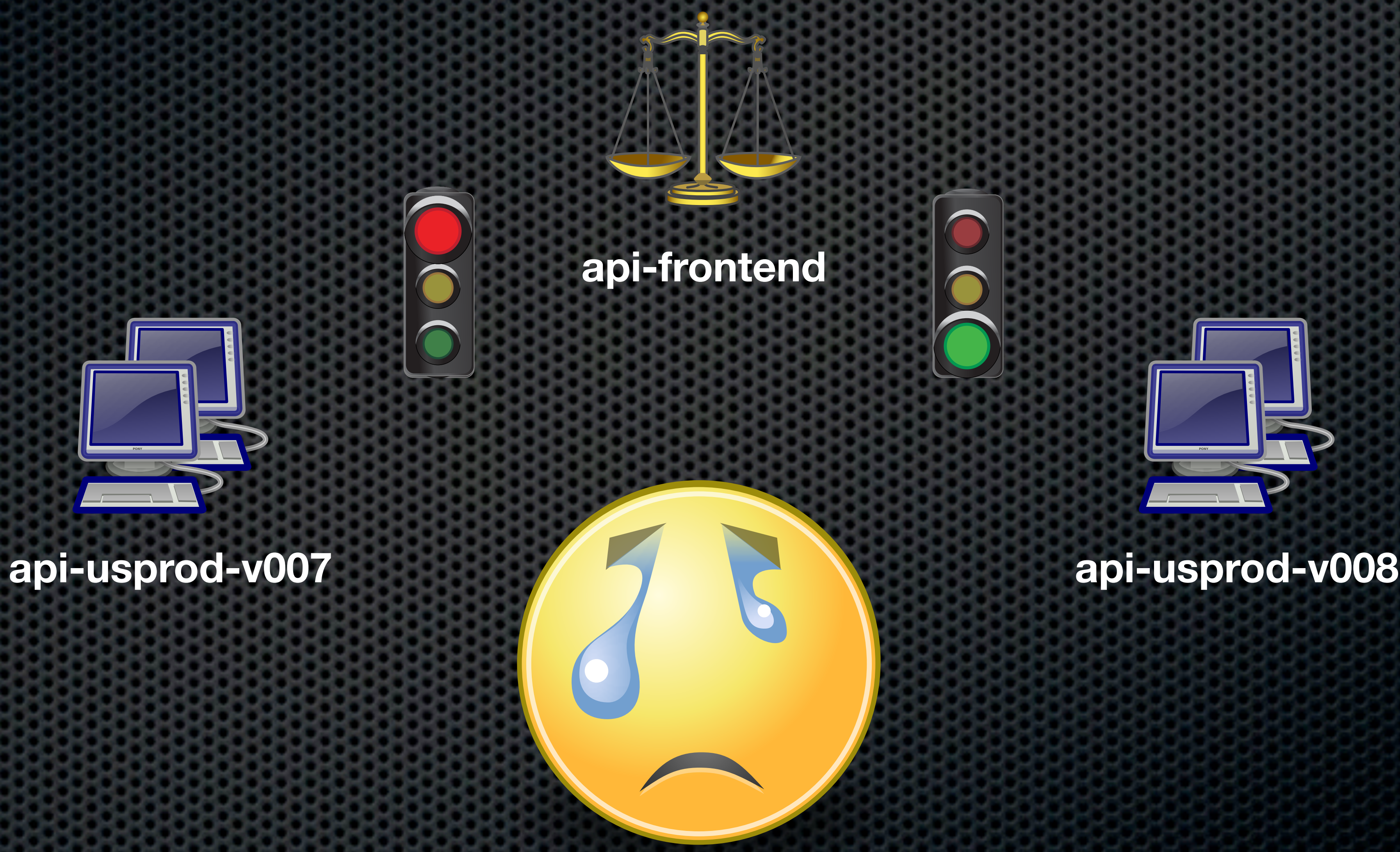
Fast Rollback

Fast Rollback

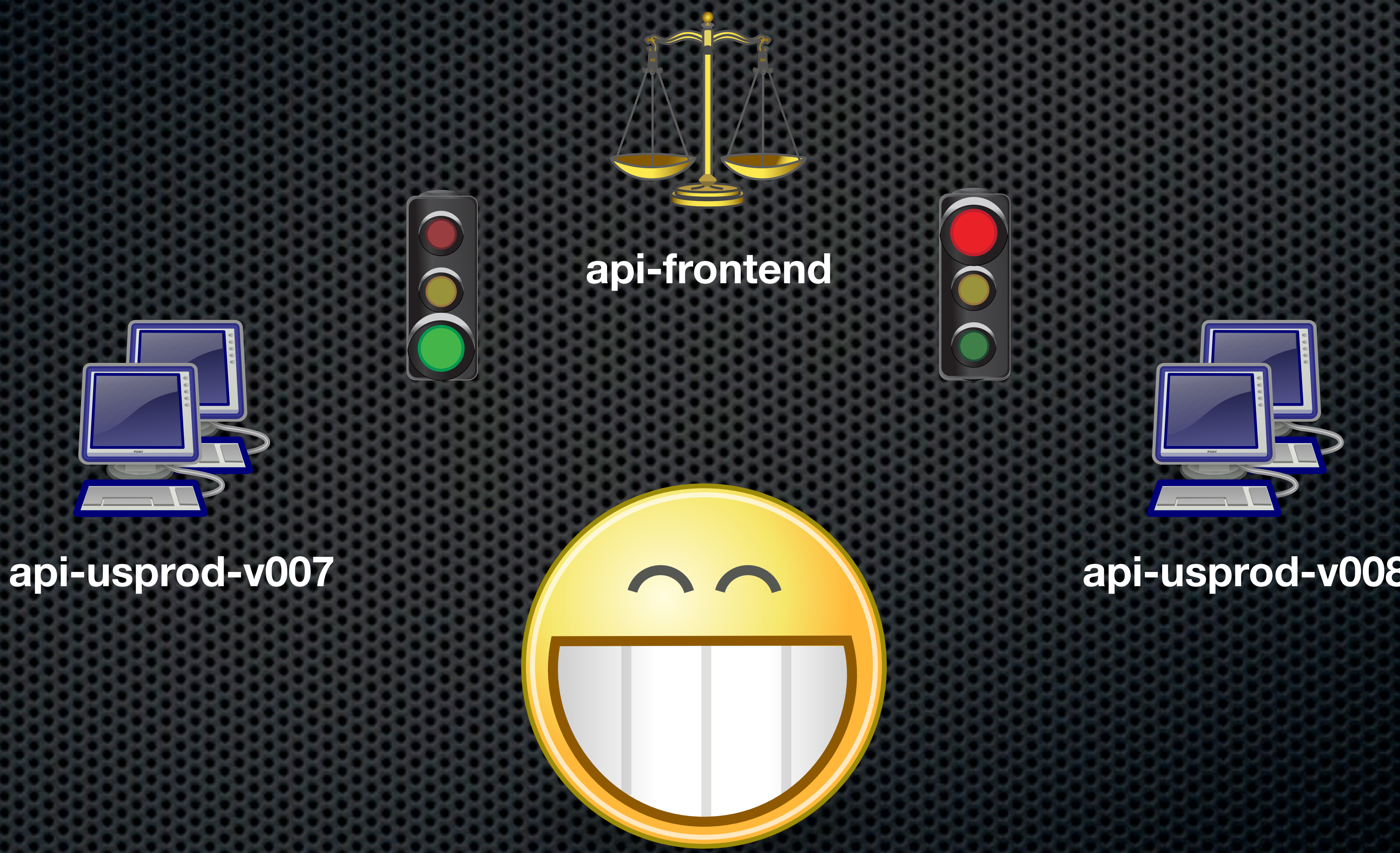
Fast Rollback



Fast Rollback



Fast Rollback



Fast Rollback

Acknowledgements

Platform Engineering

- Sudhir Tonse
- Pradeep Kamath

Engineering Tools

- Joe Sondow

Streaming Server

- Ranjit Mavinkurve

Questions?

Sid Anand

@r39132