Recognizing

ZOMBIES, Black H  les, and Tribbl  s

before they eat you

(or Techniques to Avoid Cascading Failure)

Avery Regier
@averyregier avery.regier@gmail.com

Acknowledgements

Luther Tegtmeier

Jason Lackore

Mahesh Gaya

Brian Danenhauer

Ryan Pritchard

Abhinav Sharma

Manish Patil

Ravi Yerramsetti

Joel Rindfleisch

Brian Ashmore

Scott Fegenbush

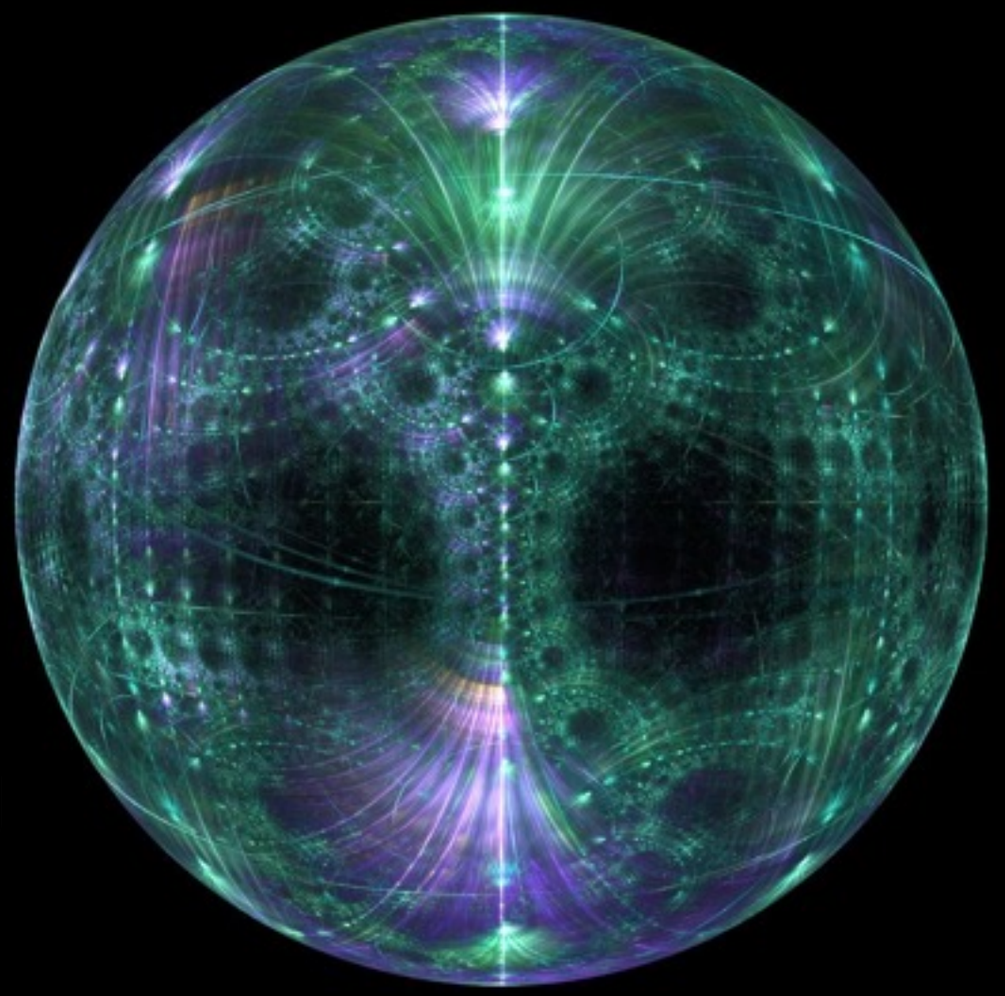
Ryan Bergman

Jason Bogh

Josh Carson

Foretellings…

- Motivations
- Observability
- Recognizing and Killing ZOMBIES
- Recognizing and Avoiding Black H les
- Handling Tribbl s



Motivations

- Large Monolithic API
  - with ~50 dependencies
  - does many mostly independent things
  - often a victim
- Failure Themes
  - Resource Exhaustion
    - Long Running Work
    - Getting Slammed
  - Cascading Failure
  - There are warning signs - use them!

Observability

- For your code
  -> Automated resiliency

- For you (centralized logging)
  -> Dashboards & Alerting

Foundations: Work Tracking

- A good work tracker will:
  - Not block your real work
  - Order by oldest first
    - Be cheap to delete (no searching for the item)
    - Clean up after itself
    - Such a collection does not exist in the JDK
- API Design: Take a ticket, return the ticket, and stream tickets by age.

Watch this space

https://github.com/JohnDeere

```
public class Outstanding<T> implements Iterable<T> {
    public abstract class Ticket implements AutoCloseable {
        ...
    }
    ...

public class WorkTracker implements Filter {
    private Outstanding<Work> outstanding = new Outstanding<>();

    @Override
    public void doFilter(ServletRequest req, ServletResponse res,
                         FilterChain chain)
            throws IOException, ServletException
    {
        try (Outstanding<T>.Ticket ignored =
                     outstanding.create(mapWorkIdentity(req))) {
            chain.doFilter(req, res);
        }
    }
}
```

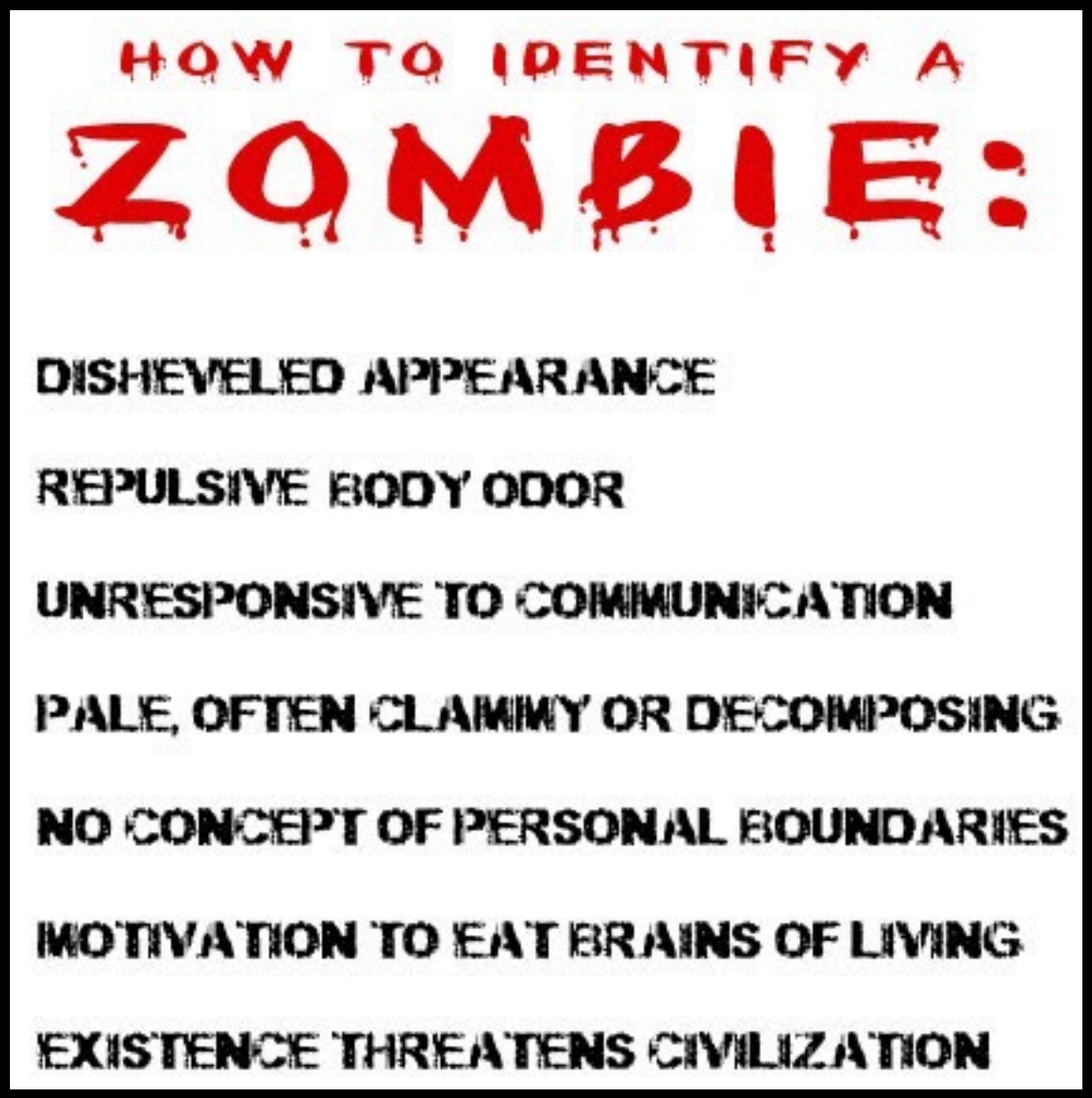
```java
private Work mapWorkIdentity(ServletRequest request) {
    Work bean = new Work();
    bean.setRemoteAddress(request.getRemoteAddr());
    if(bean instanceof HttpServletRequest) {
        HttpServletRequest httpRequest = (HttpServletRequest) request;
        bean.setPath(httpRequest.getMethod()+" "+httpRequest.getRequestURI());
        bean.setRemoteUser(httpRequest.getRemoteUser());
        bean.setSessionId(
                    Optional.ofNullable(httpRequest.getSession(false))
                            .map(HttpSession::getId)
                            .orElse(null));
    }
    return bean;
}
```

```java
public abstract class Work {
    private long start = System.currentTimeMillis();
    private Thread thread = Thread.currentThread();
    private String requestId = UUID.randomUUID().toString();

    public long getElapsedMillis() {
        return start - System.currentTimeMillis();
    }

    public void setMDC() {
        MDC.put("request_id", requestId);
    }

    …
}
```

ZOMBIES

ZOMBIES are undead (long running, but abandoned requests) that eat brains (system memory) and cause havoc (crash the system) long after its progenitor (user who conjured it) thought it dead (gave up) and probably made more (tried again), often resulting in a **ZOMBIE HORDE**.

Declaring a ZOMBIE



- Web Server timeout settings (ours are 5 minutes)
- Whitelist the good stuff you know just takes a while (like downloading a large file).
- Perfection is not required. Goals:
  - Get it before it causes instability
    - (It can take more than half an hour to eat all the memory in a large JVM using I/O)
  - Get it before a human would
- Keep it visible: Log long running work using a background thread and your work tracker. (Every 30 seconds works fine.)

“In those moments where you're not quite sure if the undead are really dead, dead, don't get all stingy with your bullets.”

— *Zombieland*

Killing a (Java) ZOMBIE

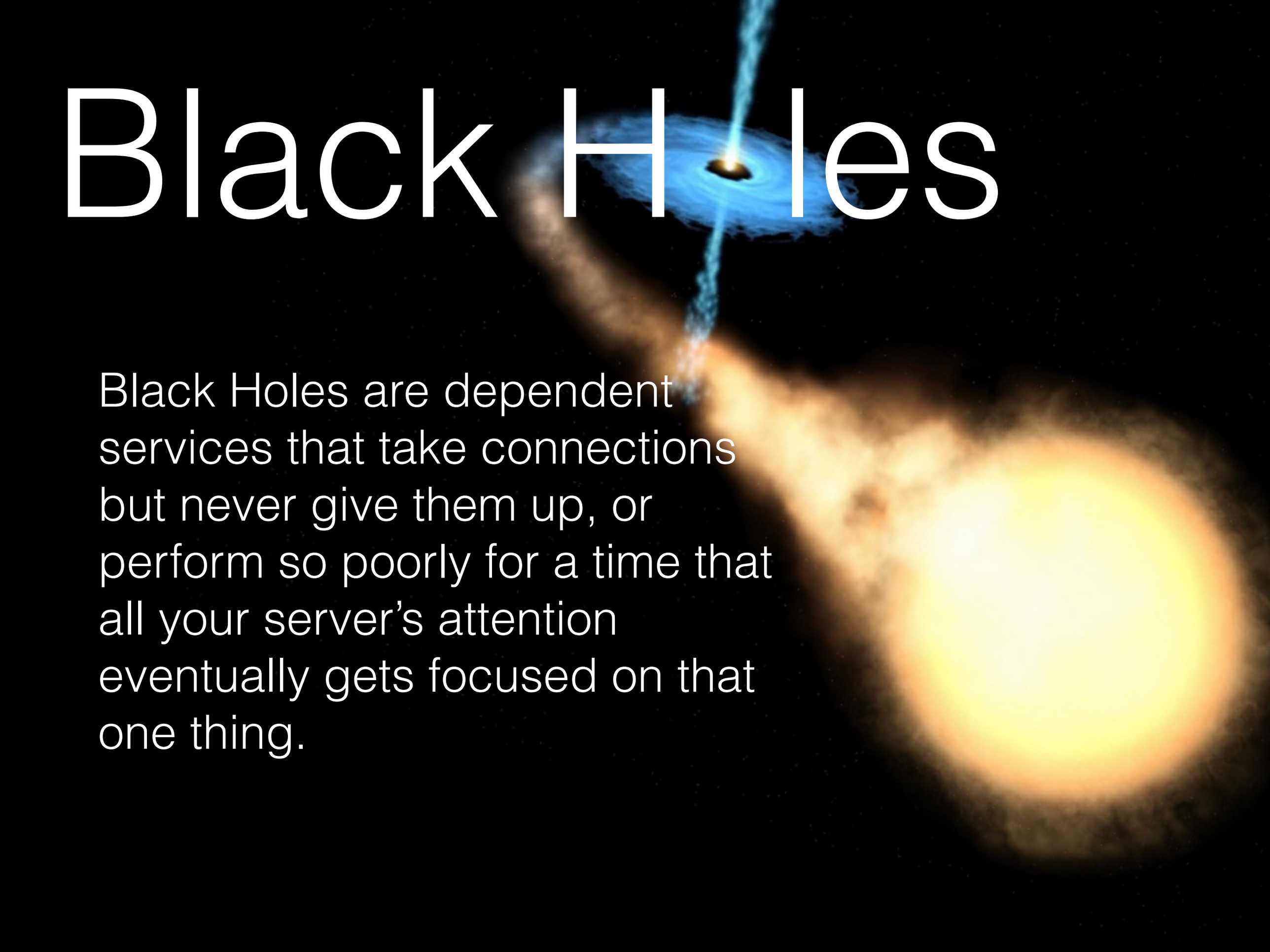
- 3 types:
  - Waiting on stuck I/O - using interrupt() works fine.
  - Making lots of calls to backends - use a generic proxy that checks against the clock before making the call and throws an error.
  - Long running single backend call - prevention is key! For this we need to discover…

```java
public class ZombieDetector {
    …
    private Outstanding.Threaded<? extends Work> outstanding;

    public ZombieDetector(Outstanding.Threaded<? extends Work> outstanding) {
        this.outstanding = outstanding;
    }

    private void doWork() {
           outstanding.stream()
                      .filter(this::isLoggable)
                      .peek(this::logIt)
                      .filter(this::isZombie)
                      .forEach(this::killZombie);
    }

    private void killZombie(Work work) { … }

    private void logIt(Work work) {
        logger.info("ZOMBIE", work.getMetadata());
    }

    private boolean isLoggable(Work work) {
        return work.getElapsedMillis() > 30 * 1000;
    }

    public boolean isZombie(Work work) {
        return work.getElapsedMillis() > 5 * 60 * 1000;
    }

     public void killRunaway() {
         if (outstanding.current().map(this::isZombie).isPresent()) {
             throw new ShootBrainz.Error();
         }
     }
}
```

Killing a (Java) ZOMBIE

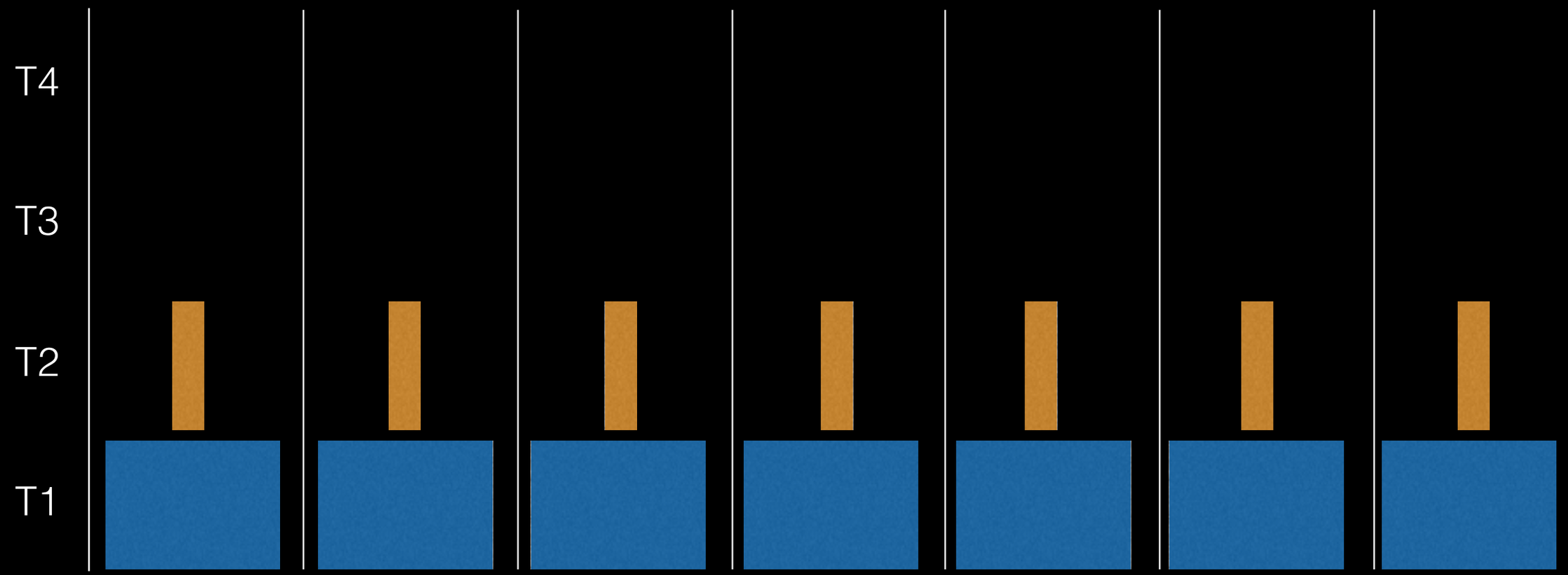
- 3 types:
  - Waiting on stuck I/O - using interrupt() works fine.
  - Making lots of calls to backends - use a generic proxy that checks against the clock before making the call and throws an error.
  - Long running single backend call - prevention is key! For this we need to discover…

Black Holes

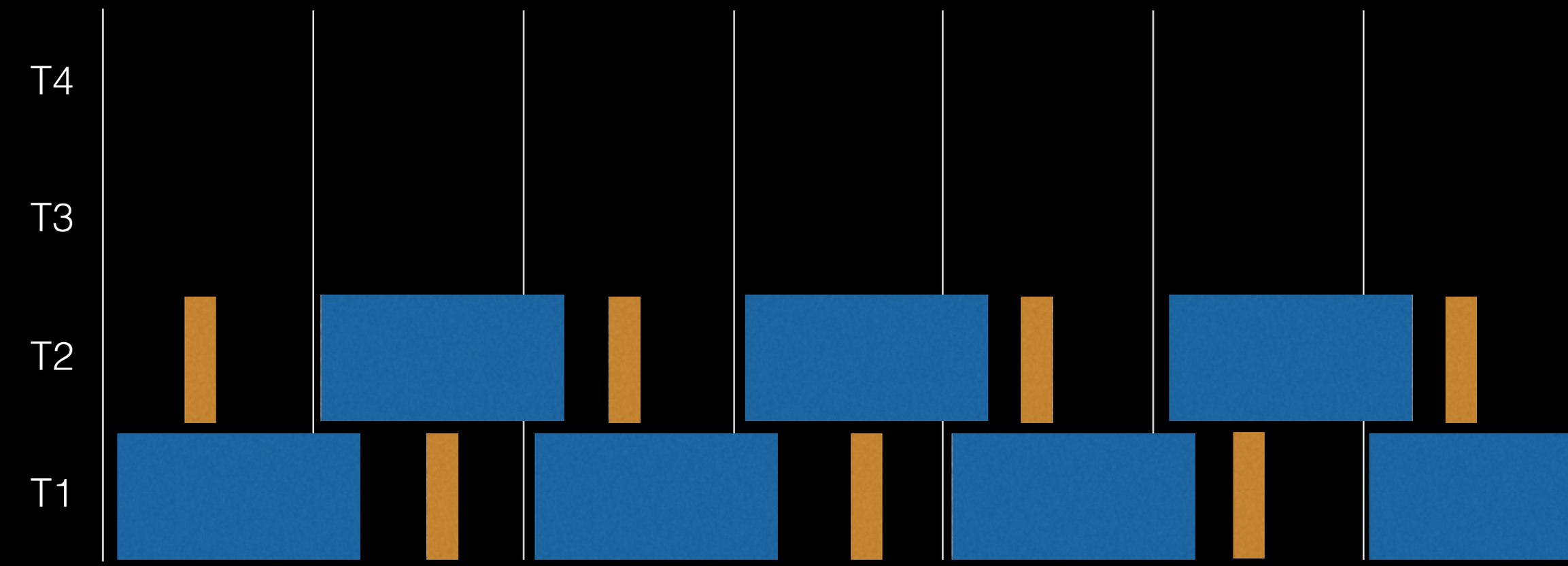


Black Holes are dependent services that take connections but never give them up, or perform so poorly for a time that all your server's attention eventually gets focused on that one thing.

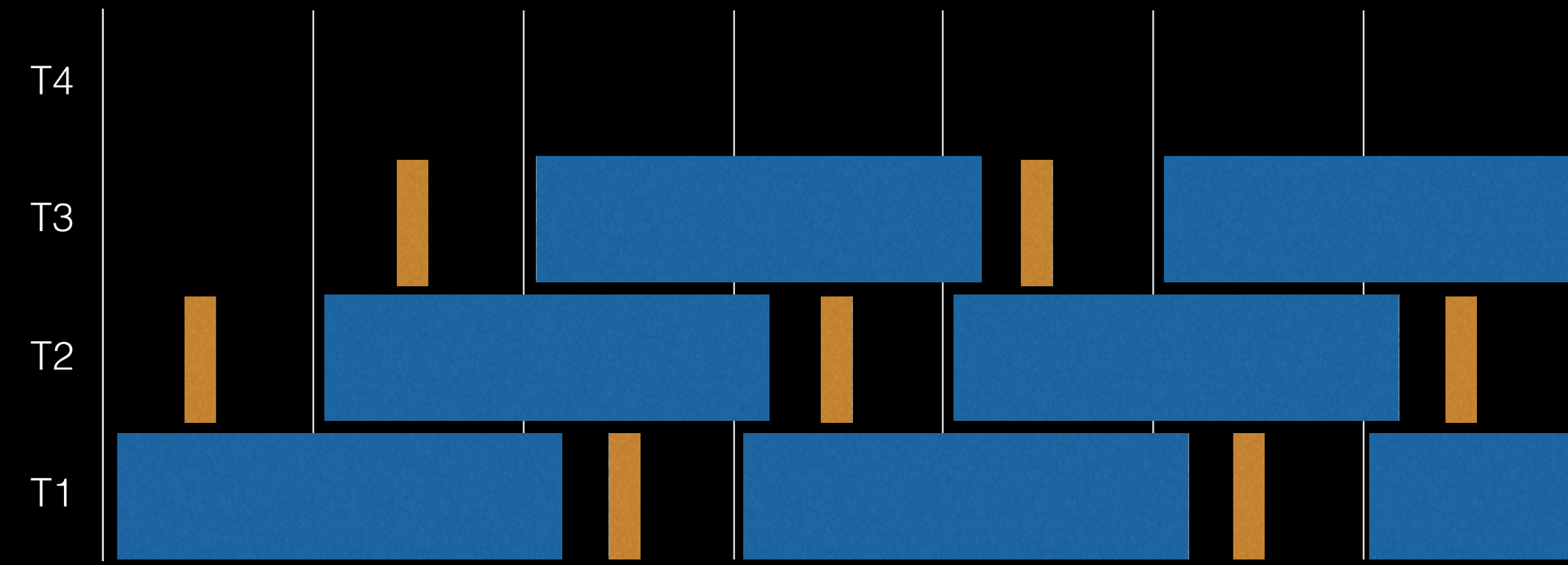
Performance Based Resource Exhaustion



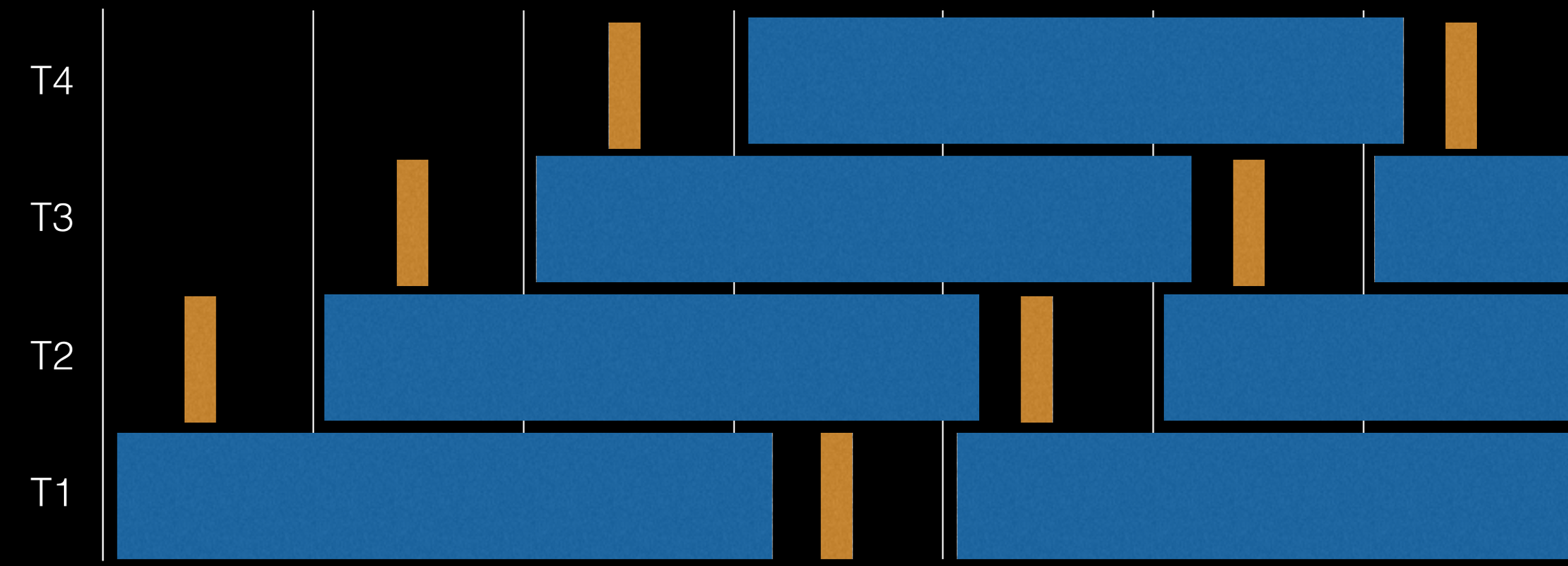
Performance Based Resource Exhaustion



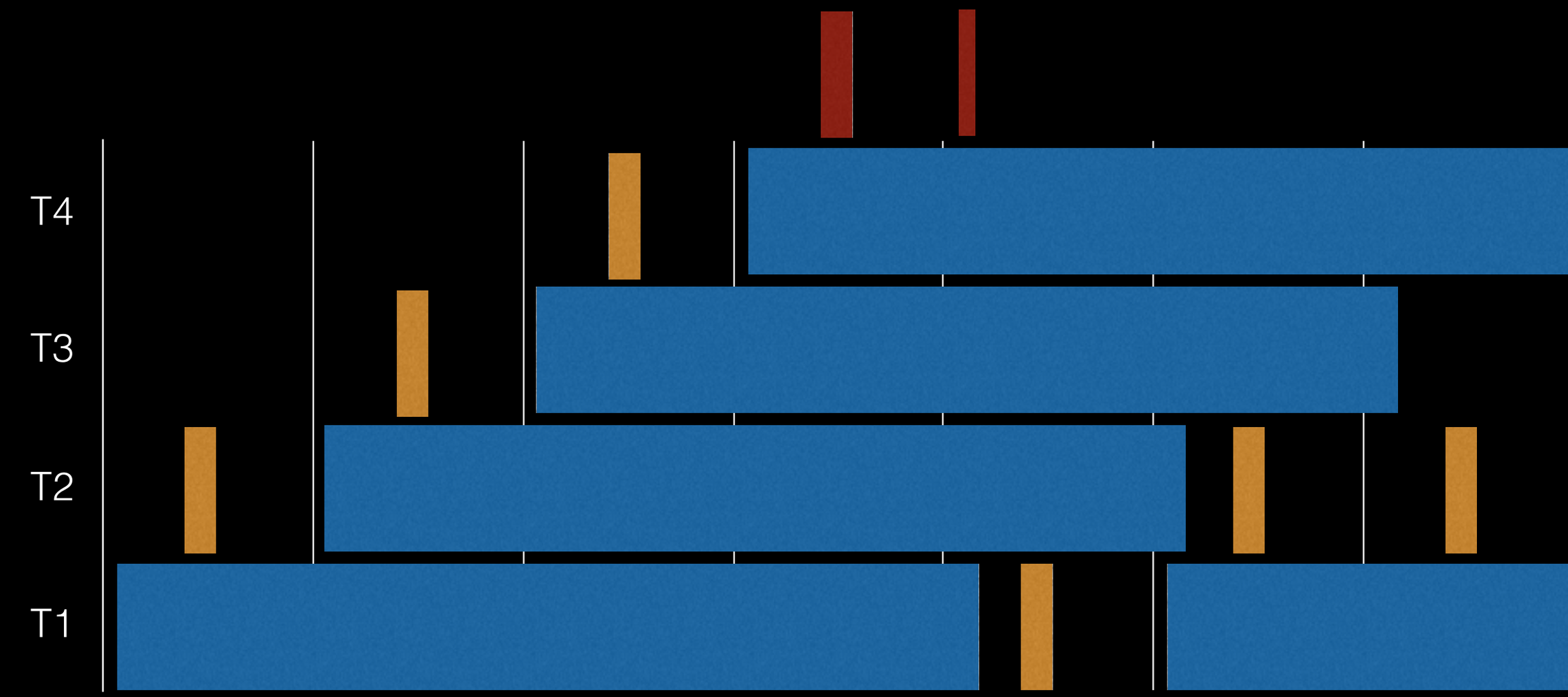
Performance Based Resource Exhaustion



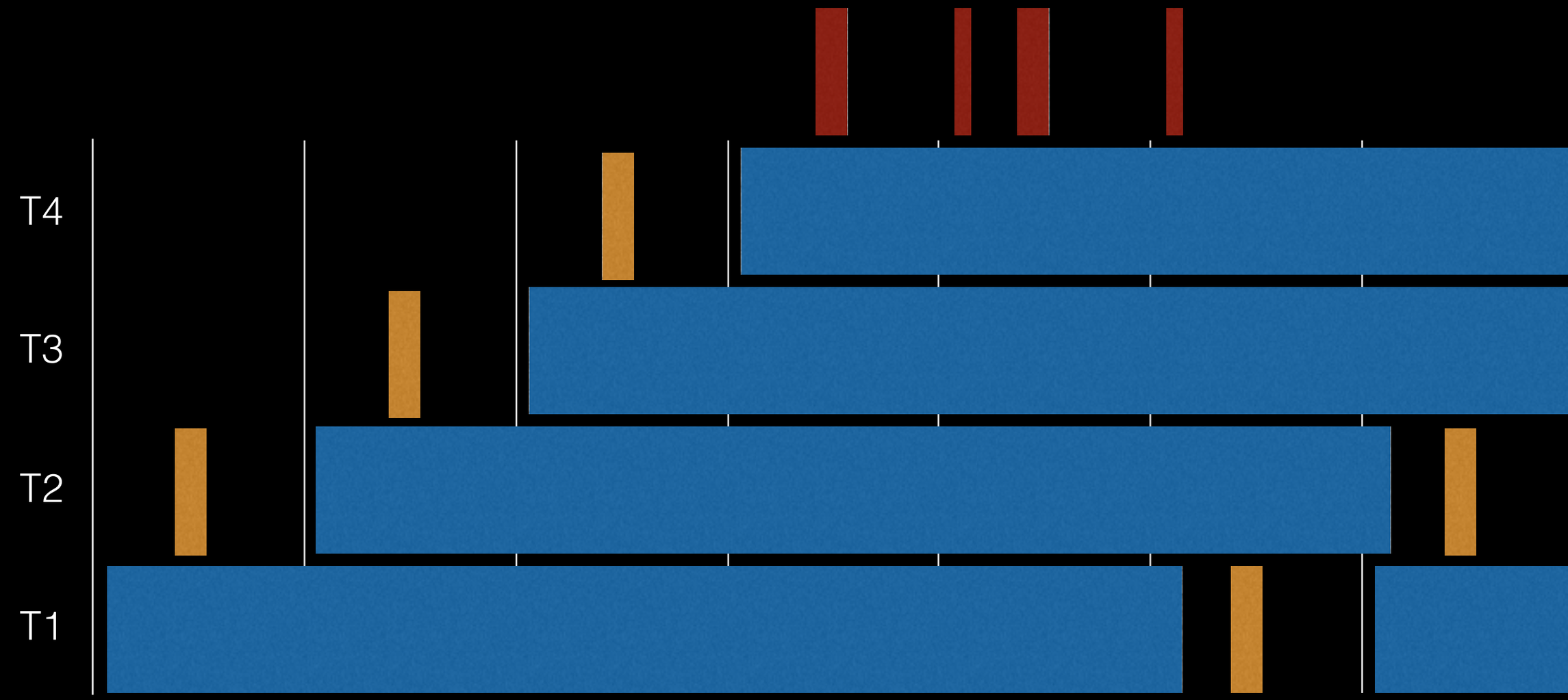
Performance Based Resource Exhaustion



Performance Based Resource Exhaustion



Performance Based Resource Exhaustion



The Circuit Breaker Analogy

- Detects an overload, and ‘opens’ to protect appliances, prevent fires, etc.

- The terminology sometimes seems backwards:

  - Closed = Letting the current pass through

  - Open = Current cannot pass

- Can be ‘closed’ (typically manually) when the event is over.

  Q: What’s the main problem with circuit breakers?

Goals

- No human intervention necessary.
  - Dynamically understands what normal is.
  - Detects when things aren't normal.
  - Prevents sending more requests into a non-normal situation.
  - Detects when things are normal again and heals the system automatically.
- Protecting the service over protecting the traffic.
  - Allow some requests to enter the black hole to detect the situation.
- Human intervention is possible:
  - Lock closed: I know its OK, let the work flow (but keep learning)
  - Lock open: Prevent all usage because a task is going on.
    - Close: Return to normal and continue looking for non-normal.
    - Open: Prevent most usage but start testing for normal.
- Doesn't limit scalability or impact users in normal situations.

"Logic clearly dictates that the needs of the many outweigh the needs of the few."

–Spock

"Or the one."

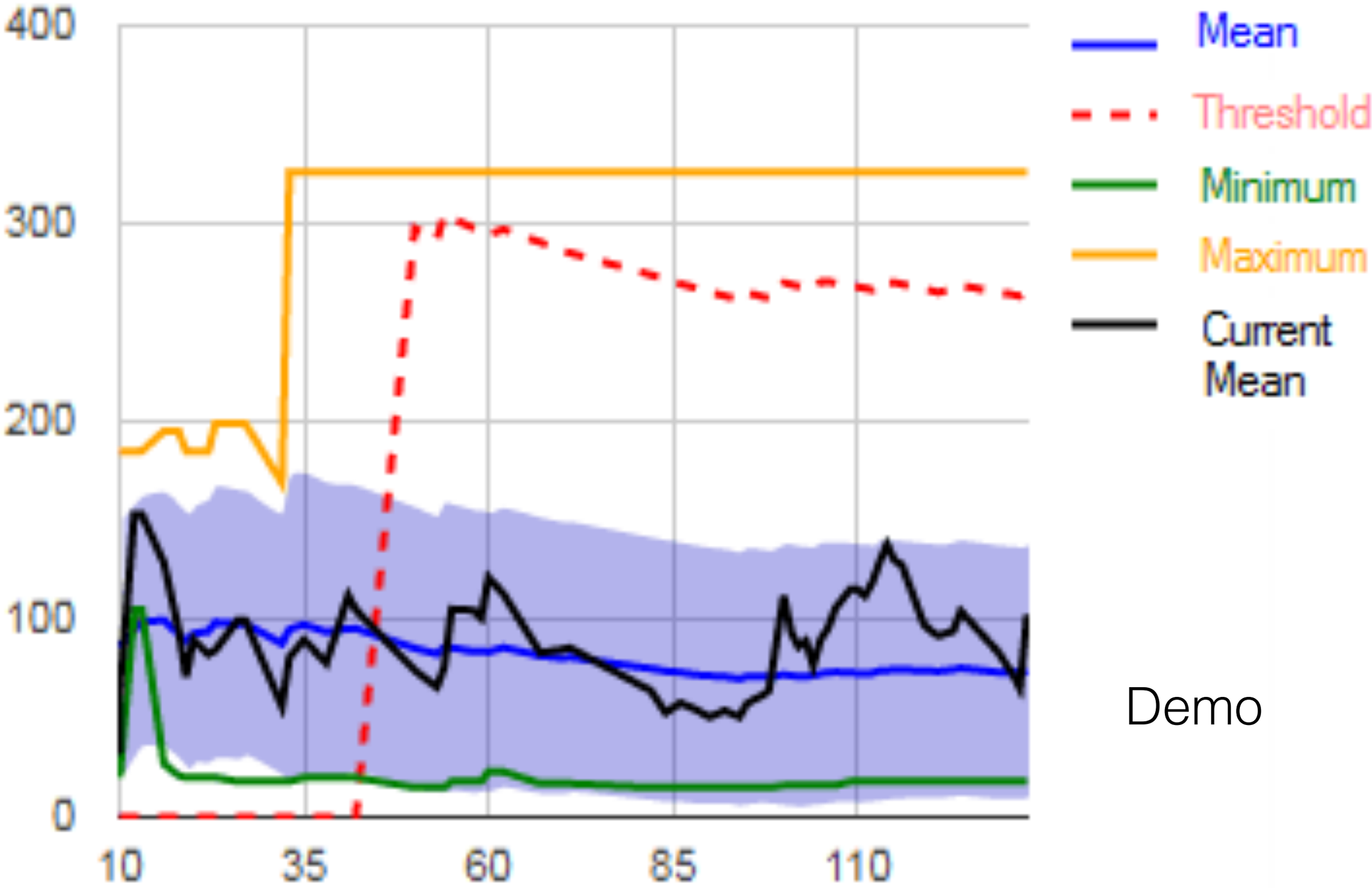
–Kirk

The Plan

- Understanding Normal
- It's Messy
- Intervention
- Demonstration

Circuit Breaker Basics

- Elapsed Time of successful hits
- Find standard deviation
- Mean + 5 x standard deviation ≈ 99.5%. This is your threshold.
- Use your work tracker!
- If the average of in-progress is above the threshold, kill the circuit.


400
300
200
100
0
10
35
60
85
110
Mean
Threshold
Minimum
Maximum
Current Mean

Demo

It gets messy
(use heuristics)

- Poison messages that aren't representative of the situation, OR rare, long running legitimate traffic:
  - Don't consider the oldest thread when calculating your average for threshold detection.
- Low usage services
  - Wait for a certain number of hits before you trust your stats.
  - Let the Zombie detector deal with it.
  - Require a minimum # of outstanding threads before you check against threshold.
- Some calls are naturally slower, like lists versus single items. (Two humped camel distribution problem)
  - Divide elapsed time by the weight of the call before using it for threshold calculation or detection.
  - Typical weight =
    # of items requested *
    # of attributes requested
- What about exceptions?
  - Ignore them. Make sure those aren't used in establishing a threshold.
- Flood of calls
  - For that we need to prevent…

Tribbles

Tribbles are similar requests that you normally invite, but they come too many, too fast for your service to handle as they take your attention and eat up all your resources.

“Too much of anything, Lieutenant, even love, isn't necessarily a good thing.”

*–**Kirk** to Uhura, on the love of a tribble*

How do you get  s?

- Friendly, unintentional Denial of Service
  - Innocent Parallelism
    - A service returns a long list, then the client asks for more details on each item in parallel.
    - The client wants to load a lot of data and get it to their user faster, so…
  - Overwhelming Retries
    - I'm handling my Exceptions, right?
- Tribble Glasses (Looks like a Tribble to me!)
  - Failure to Scale
    - Slow Performance (Looks like a Black Hole)
  - Cost Allocation Failure
    - No incentive created for clients to limit service usage
  - Architectural Misuse
    - Client wants immediate updates, or to copy all information, but all you provide is a list API they poll.

How do  hurt you?

- Take up all your database connections or threads.
- When ‘riding the limit’, deadlocks often result.
- Overwhelm downstream dependencies.
- Queue up, leading to client timeouts (which you fulfill anyways), retries, and being further overwhelmed.

“And as captain, I want two things done.
First, find Cyrano Jones.
And second ... close that door.”

–***Kirk***, *after an avalanche of tribbles falls on him*

Tribble Detection

- Use your work tracker!
- Enforce an overall limit
  - ~90% of resources in incoming request count
- Detect similarities in incoming traffic
  - User
  - Client Software
  - API

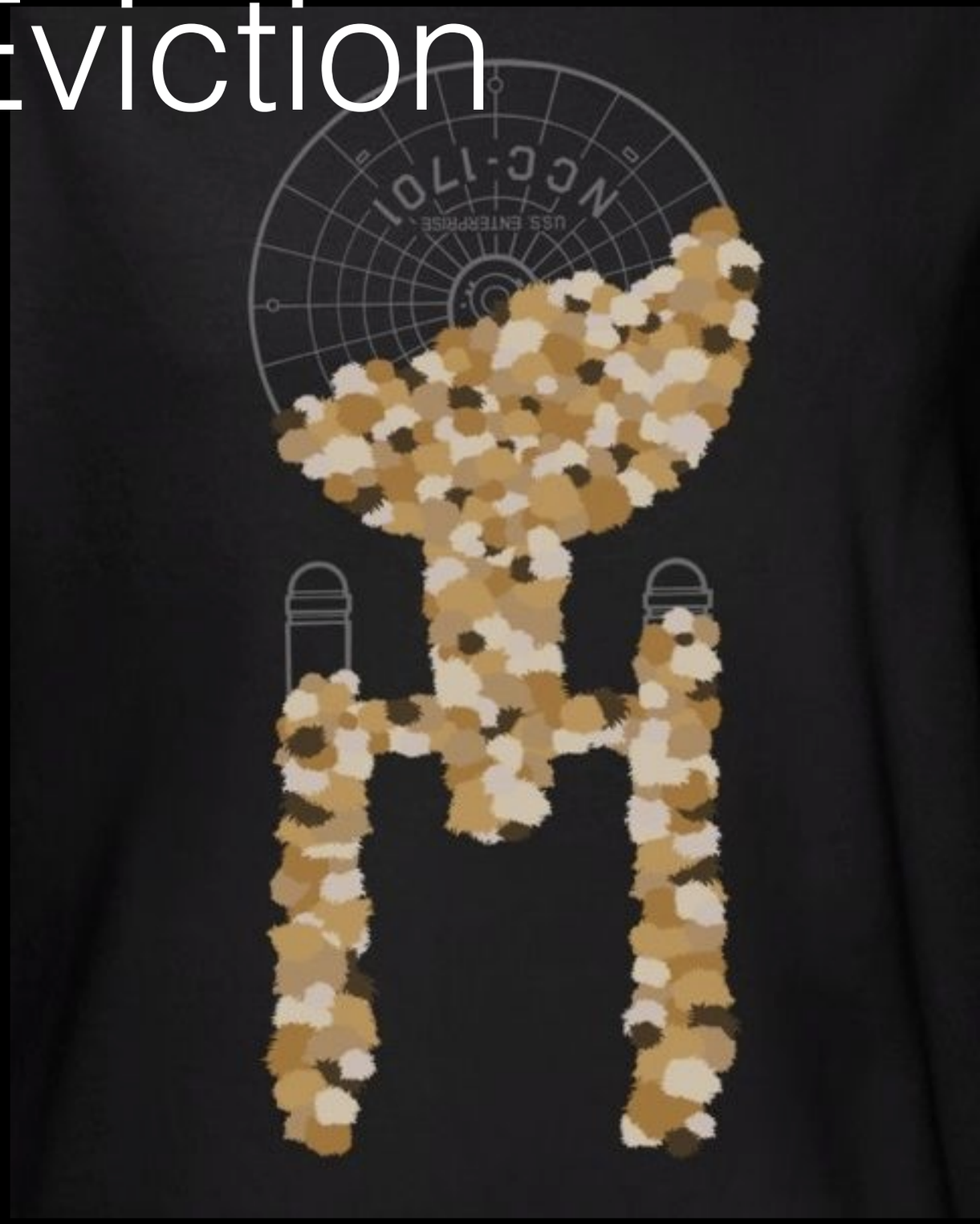
- Be careful not to use up resources before you make a decision
  - Use header information
  - Use in-memory cached data
  - Avoid database access, unless feeding the cache
- Log your metadata
- Alert yourself

“I want these things off my ship! I don't care if it takes every man we've got—I want them off the ship!”

–***Kirk**, determined to rid the ship of the Tribbles after discovering them in his food*

Tribble Eviction

- Allow spikes but give headroom for normal traffic
    - Know the percentage of your resources you use normally.
    - Evict similar hits when total of those hits exceed
      Allowed = 100% - Normal usage% - Headroom%
- HTTP Status Code 429 - Too Many Requests
- Be Friendly: Retry-After possibilities:
    - You have statistics on similar requests in your work tracker.
    - Give a time in the future as long as the longest ongoing request time.



“I gave them to the Klingons, sir.”

“The Klingons?”

“Aye, sir. Before they went into warp I transported the whole kit and kaboodle into their engine room, where they'll be no tribble at all.”

– ***Kirk*** *and* ***Scotty****, discussing what happened to all the tribbles that were aboard the Enterprise*

Things we didn't talk about…

Hystrix
Coordinating nodes
Thread.stop()
Logging metadata
WebSockets
Exceptions
Downstream failure
Thread pools
Dashboards
Human intervention
Alerting

Things I'd like to hear about…

Better math
Different solutions
Your experiences
Other stacks?
Alternative analogies
Horror stories
Whatever is on your mind

@averyregier
avery.regier@gmail.com
https://github.com/johndeere

What about Hystrix?

The precise way that the circuit opening and closing occurs is as follows:

1. Assuming the volume across a circuit meets a certain threshold (HystrixCommandProperties.circuitBreakerRequestVolumeThreshold())...

2. And assuming that the error percentage exceeds the threshold error percentage (HystrixCommandProperties.circuitBreakerErrorThresholdPercentage())...

3. Then the circuit-breaker transitions from CLOSED to OPEN. While it is open, it short-circuits all requests made against that circuit-breaker.

4. After some amount of time (HystrixCommandProperties.circuitBreakerSleepWindowInMilliseconds()), the next single request is let through (this is the HALF-OPEN state). If the request fails, the circuit-breaker returns to the OPEN state for the duration of the sleep window. If the request succeeds, the circuit-breaker transitions to CLOSED and the logic in 1. takes over again.

https://github.com/Netflix/Hystrix/wiki/How-it-Works#flow7

What about Hystrix?

- It checks errors.
  - In a black hole situation there won't be an error.
  - How about letting the underlying service determine failure?
- It "solves" the Black Hole problem by:
  - Using timeouts:
    - Timeouts interrupt legitimate long running work.
    - Tend to become excessively long to avoid affecting legitimate cases.
  - limiting the amount of work that can be done at once.
    - Pool limits restrict scalability in normal situations.
    - Pool limits require a quick runtime configuration option.
- Its a different, valid approach, but doesn't meet our stated goals.

- If a client library is misconfigured, the health of a thread pool will quickly demonstrate this (via increased errors, latency, timeouts, rejections, etc.) and you can handle it (typically in real-time via dynamic properties) without affecting application functionality.

…

In short, the isolation provided by thread pools allows for the always-changing and dynamic combination of client libraries and subsystem performance characteristics to be handled gracefully without causing outages.

https://github.com/Netflix/Hystrix/wiki/How-it-Works#benefits-of-thread-pools