Engineering Robust Server Software

Server Software

Duke
UNIVERSITY

Brian Rogers Duke ECE
Used with permission from Drew Hilton

Servers Software

- Servers accept requests from clients
  - Exchange information (take requests, give responses)
  - Generally do much of the "computing"
- We'll start with two example categories
  - Unix Daemons (sshd, httpd, …)
  - Server side code in websites (Django)
- So what is so special about server software?
  - Why is it different enough to be in the course title?

Duke UNIVERSITY

Most Code You Have Written

- Run on input, get output
  - Then done
- Error?
  - Print message and exit
- Run by you
  - Trusts user
  - On one computer…
- Deals with one input at a time
  - Serial code
  - Don't care about performance

Servers: Different

- Run "forever"
  - Implications of this?

```
while (true) {
    ......

}
```

Duke
UNIVERSITY

Run Forever

- Resource (memory, file descriptors,…) Leaks: Unacceptable
  - Restart Chrome every week b/c memory leak? Annoying
  - Restart Google every 5 minutes b/c memory leak? No way..

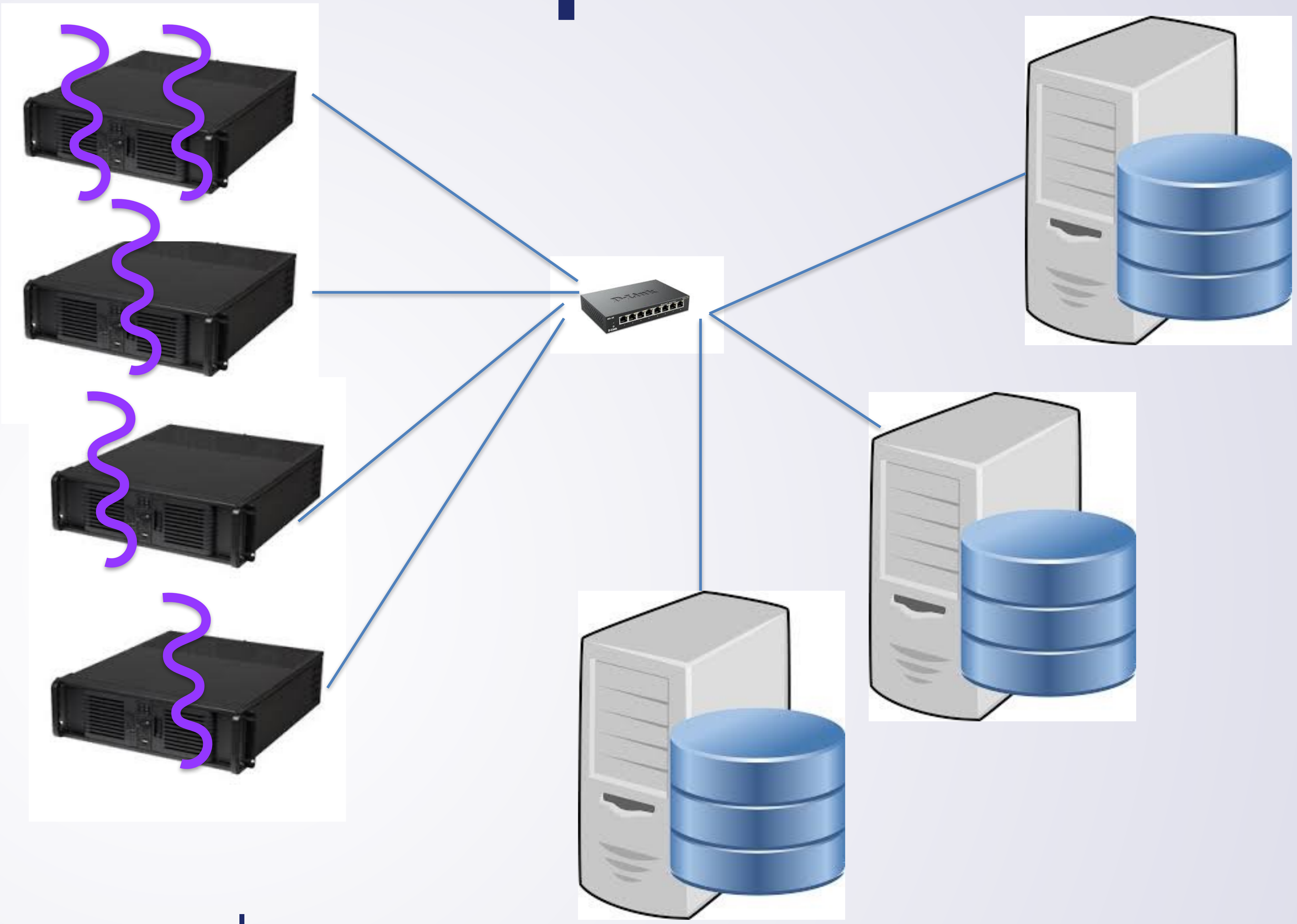
- But you all are pros at writing leak-free code by now

Duke UNIVERSITY

Run Forever

- How to handle errors?
  - abort? No way.
  - Report and keep going! Need to keep handling other requests
- Log: (Generally accepted practice in industry: **log everything**!)
  - Nobody is watching terminal.
  - Want admins to know? Need log files (/var/log/…)
- Inform user
  - Send (informative?) error response

Maybe more complex?

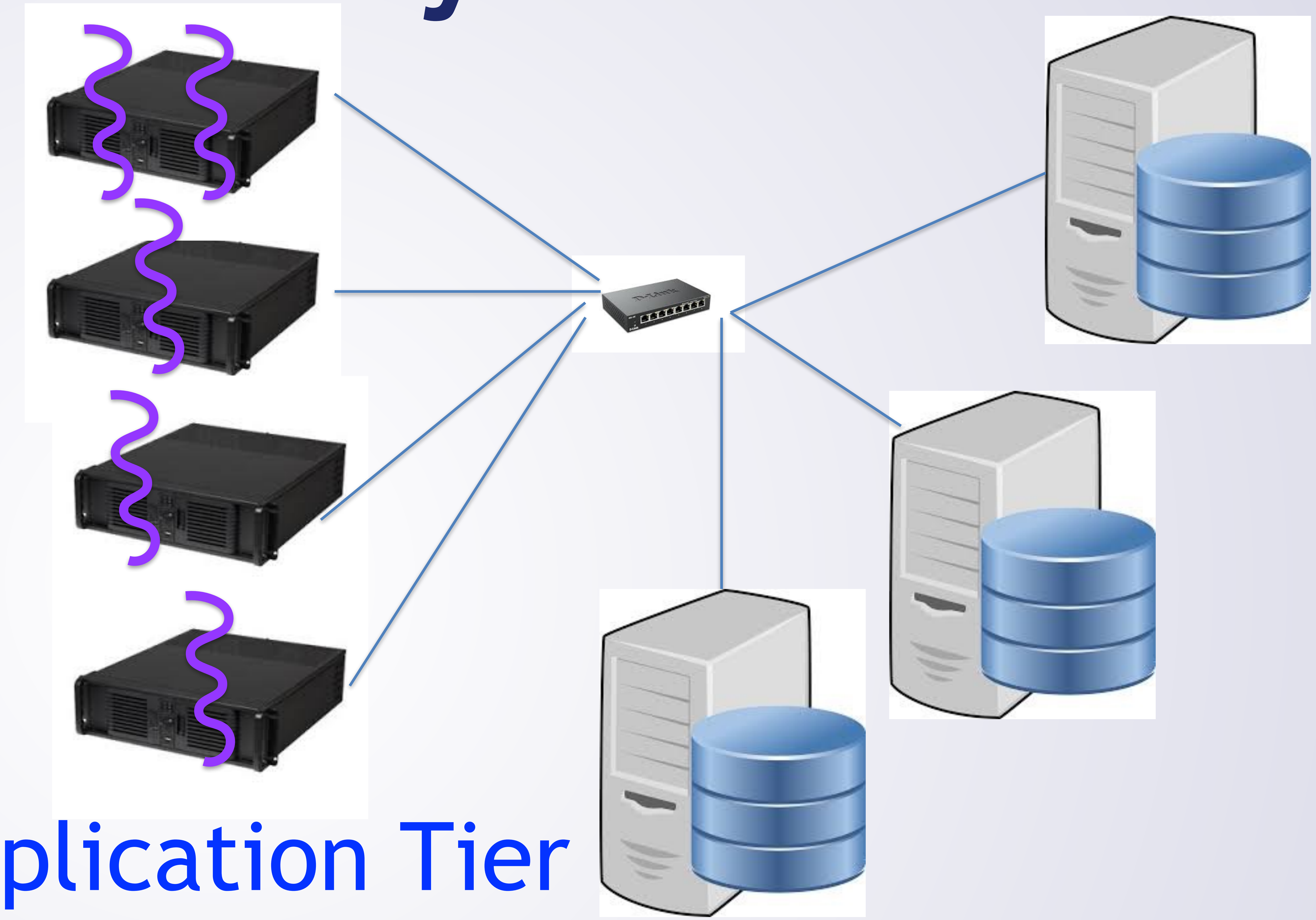


- Many server systems: more complex
  - Introduce more complexities in terms of running "forever"

Duke UNIVERSITY

Three Tier System

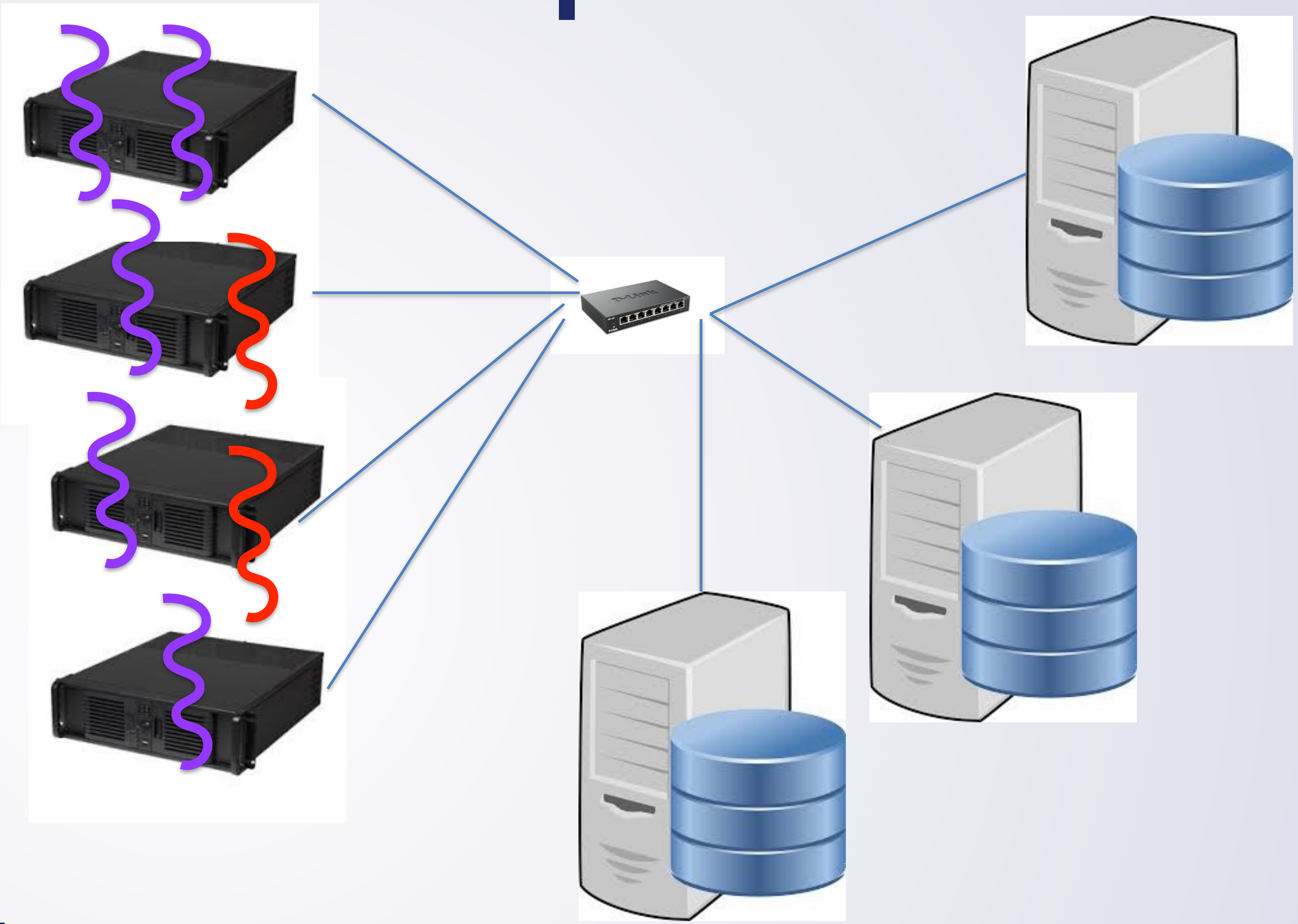
1. Presentation Tier



2. Application Tier (Business Logic)

3. Storage Tier

Maybe more complex?



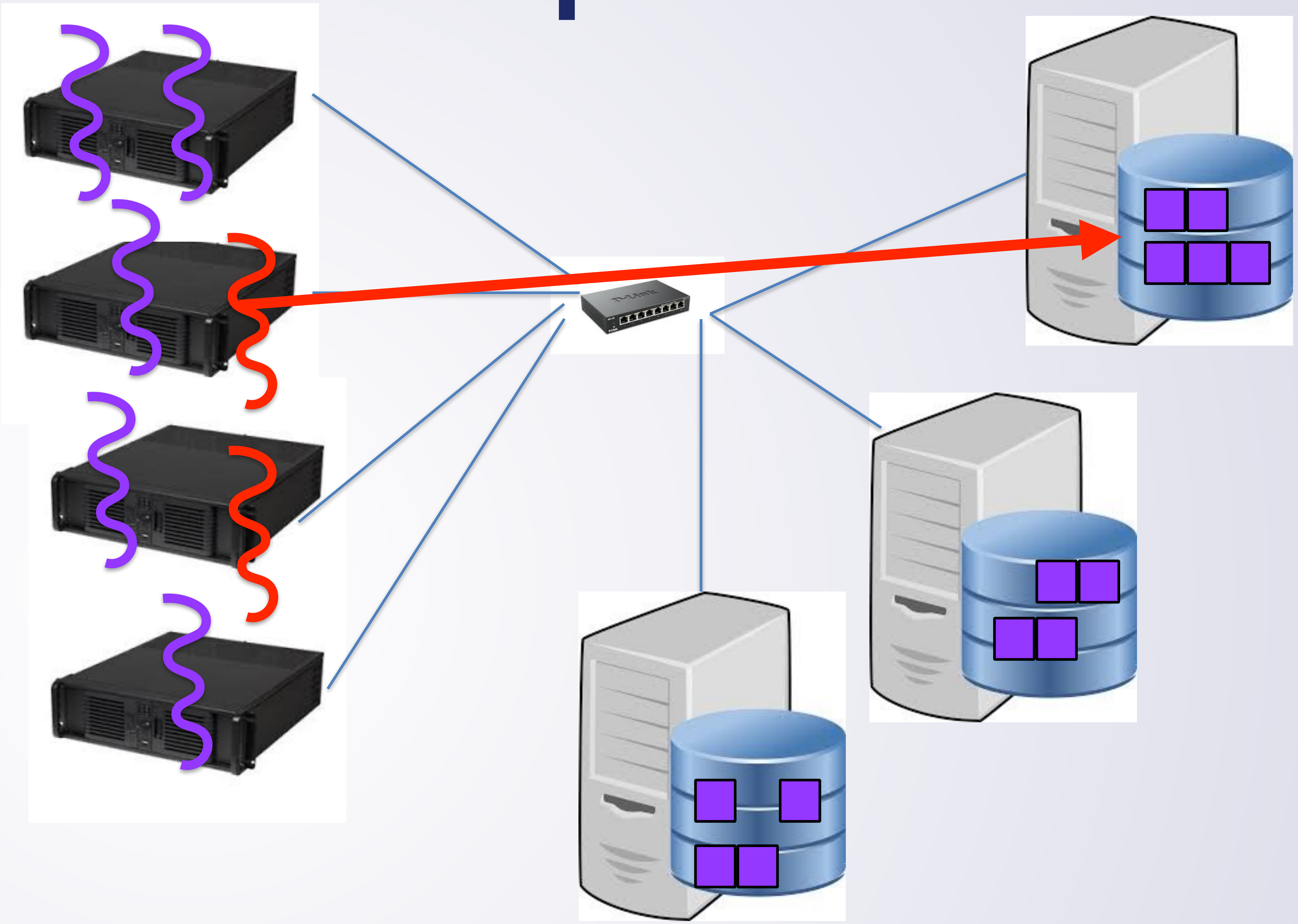
- Maybe we want to upgrade v1.0 to v2.0
  - Now have v1.0 and v2.0 running at same time: difficulties?

What if we just shut everything down?

Hypothetical picture of what would happen if Google or Facebook were down for 1 minute

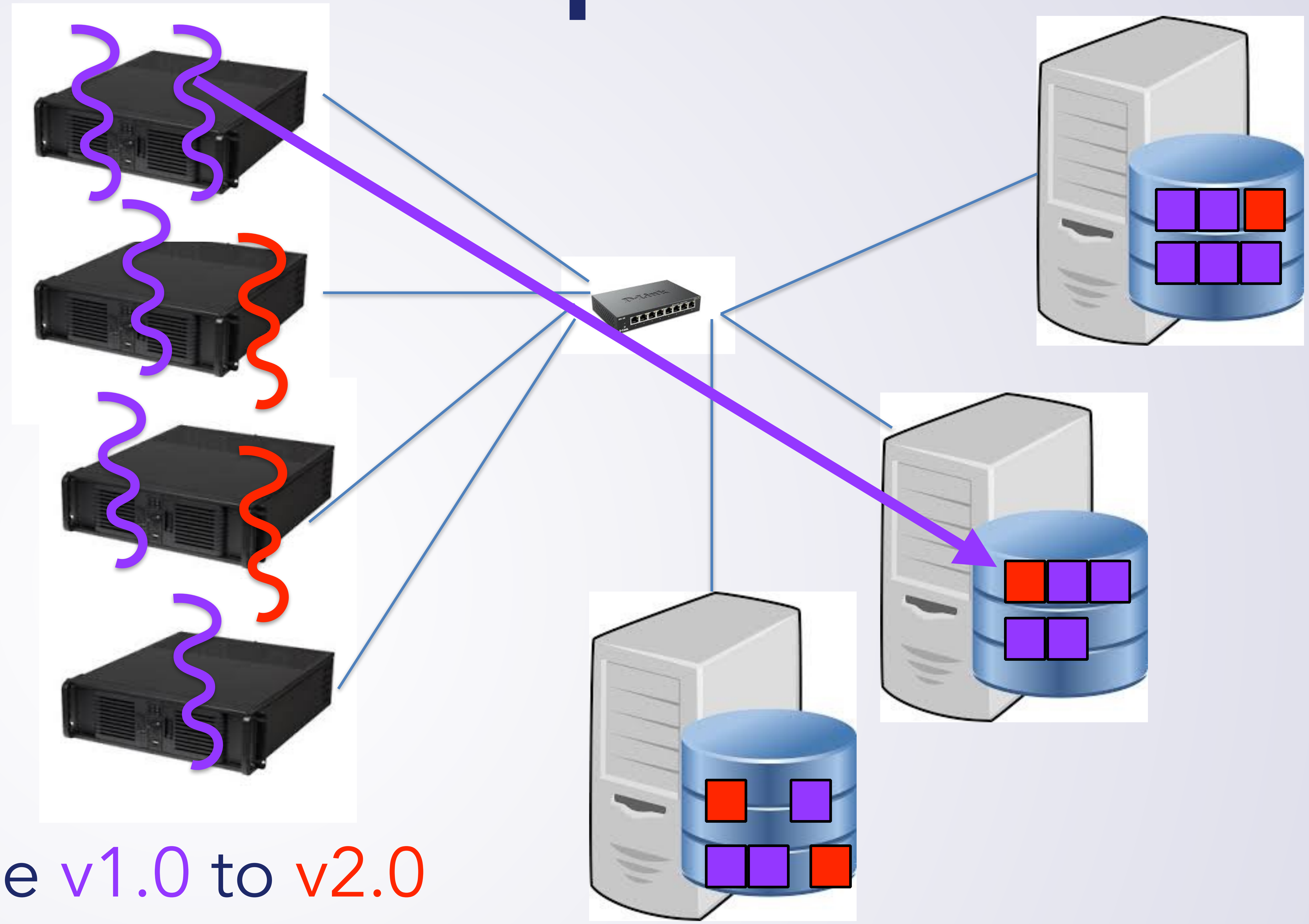
- Couldn't we just shut the whole thing down, and upgrade?

Maybe more complex?



- Maybe we want to upgrade v1.0 to v2.0
  - Version 1.0 data: accessed by v2.0 software..

Maybe more complex?



- Maybe we want to upgrade v1.0 to v2.0
  - Version 2.0 data, accessed by version 1.0 software…
  - Why is this a bigger problem?

Duke UNIVERSITY

v1.0 can handle v2.0 data

- Easy: v1.0 and v2.0 have same data layout/constraints
- Only **add** fields and/or **tighten** constraints
  - v1.0 has (name, grade) and v2.0 has (name, grade, bday)
  - v1.0 requires x >= 0 and v2.0 writes data with x > 0
- v2.0 must be written to handle v1.0 data
  - e.g. missing bday
  - x = 0
  - This is ok: we know these requirements when we write v2.0

Duke UNIVERSITY

What if v1.0 Cannot Handle v2.0 Data?

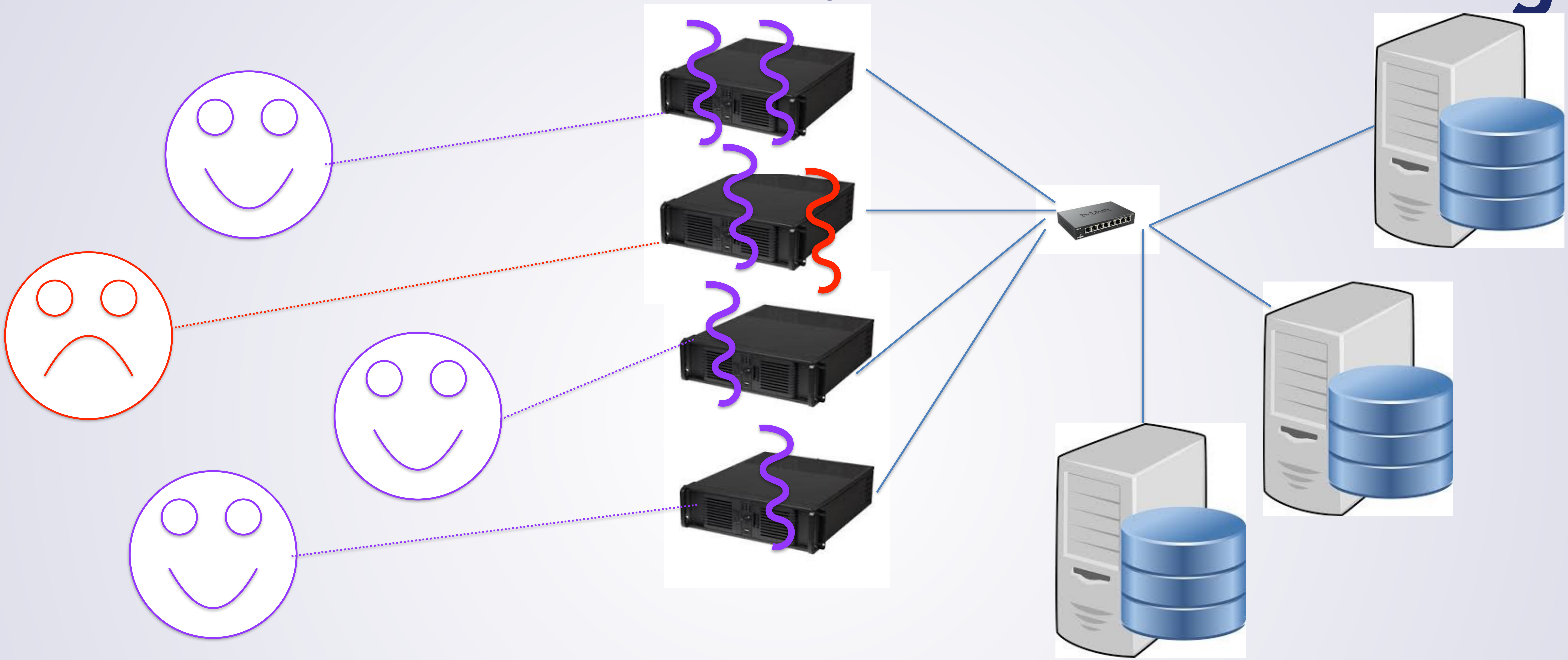
- Suppose we make some change that v1.0 cannot handle
  - v1.0 expects a field to be an int, but v2.0 writes arbitrary strings
    - (relaxes constraints)
  - v2.0 removes/renames fields [hint: try not to do this!]
- Solution: make v1.9
  - Writes v1.0 compatible data
  - Can read/handle v2.0 data
  - Spin up v1.9, until all v1.0s replaced
  - Then spin up v2.0 to replace v1.9

Migrating Data?

- Migrating Data is tricky
  - E.g., change storage tier itself itself?
- Reading:
  - How to Survive a Ground-up Rewrite Without Losing Your Sanity by Dan Milstein

Duke UNIVERSITY

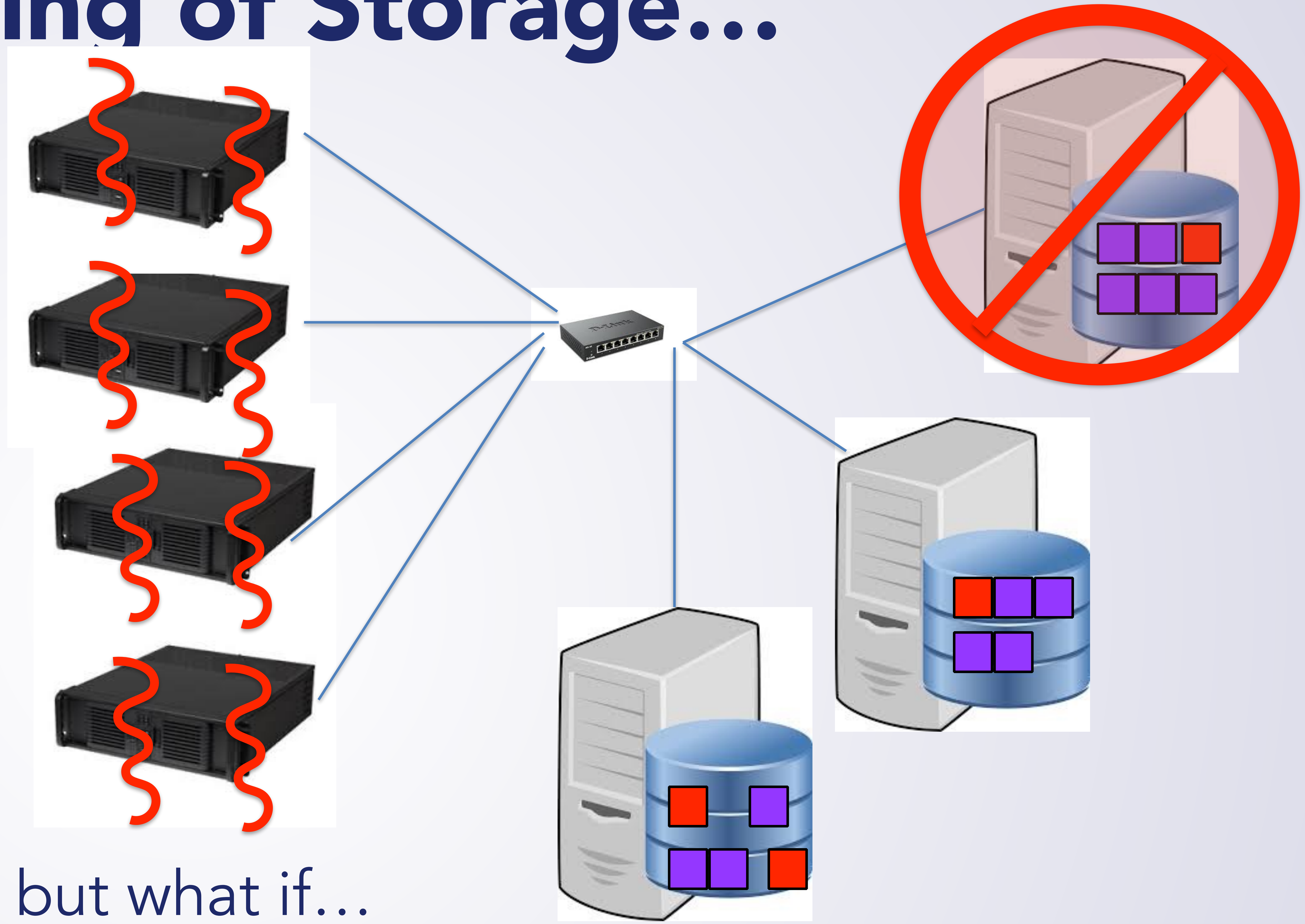
Another Reason for Slow Rollout: Testing



- Suppose v2.0 has some bug we didn't catch in testing

Duke UNIVERSITY

Speaking of Storage…



- Our code is running happily, but what if…
  - A storage server fails? Temporarily or Permanently
  - This is what we will cover later in High Availability & Disaster Recovery

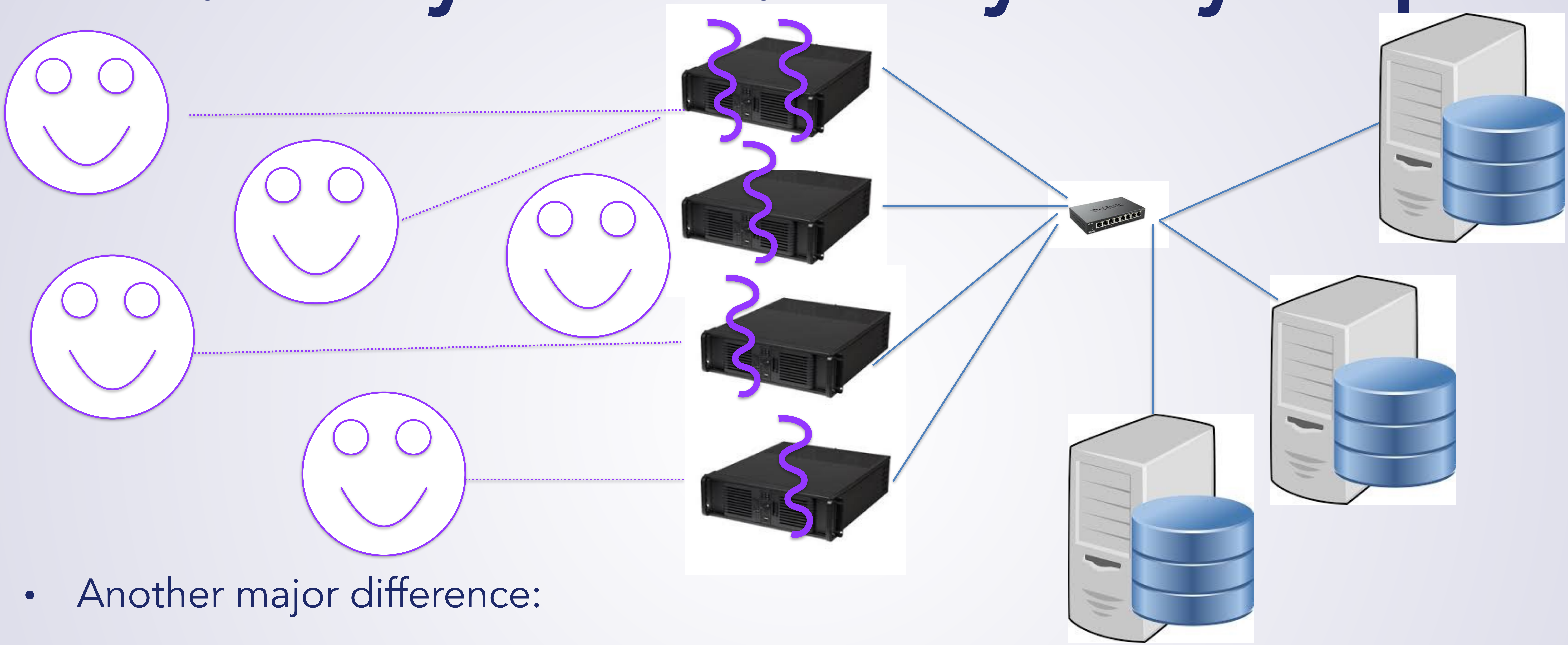
Duke
UNIVERSITY

Another Major Issue: Configuration!

- Code you have written:
  - Minimal, if any configuration. Likely read at startup
- Servers:
  - Much more configuration: see /etc/ssh/sshd_config, /etc/apache2/*, etc..
  - Re-read/change while running?
- Warning: changing config as dangerous as changing code!
  - Reading 2:
    - Google Compute Engine Incident #16007

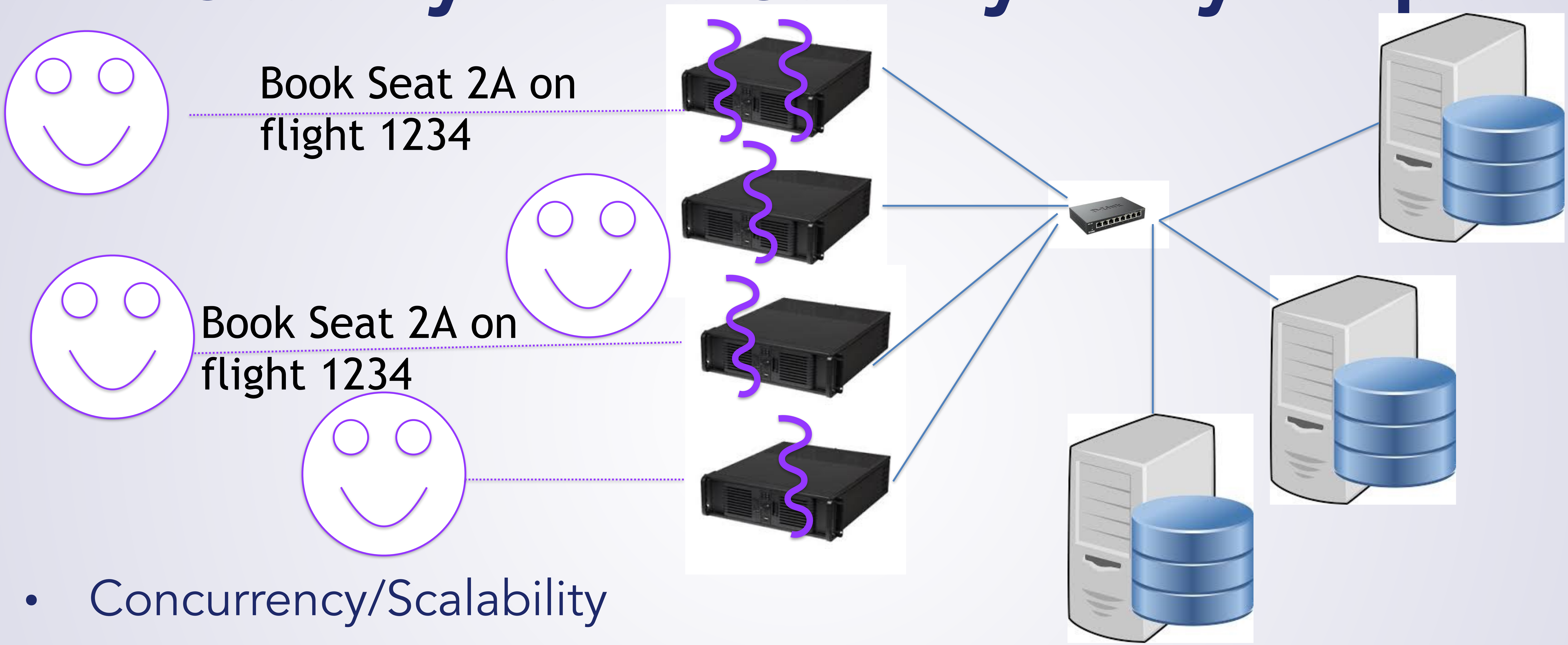
Duke UNIVERSITY

Used By You vs Used By Many People



- Another major difference:
  - Things you have written: used by you
  - Server Software: used by (many?) other people…
  - **Complexities**?

Used By You vs Used By Many People



- Concurrency/Scalability
  - Many things going on at once in system
  - Need to handle many requests efficiently

Duke UNIVERSITY

Performance: I feel the need for speed

- Performance: Users care about **speed**
  - Want system to be fast!
- From system perspective:
  - Many users
  - Want to be fast for all of them at once…
- Performance comes in two metrics:
  - Latency: time to complete one request
  - Throughput: requests/second
- Not the same, but they do interact…
- Let us look at non-software example…

Latency vs Throughput

- Here is a "road".
  - 1 lane
  - 70 mph
  - 700 miles long

Duke
UNIVERSITY

Latency vs Throughput

- Latency: 700 miles @ 70 mph= 10 hours to travel

Latency vs Throughput

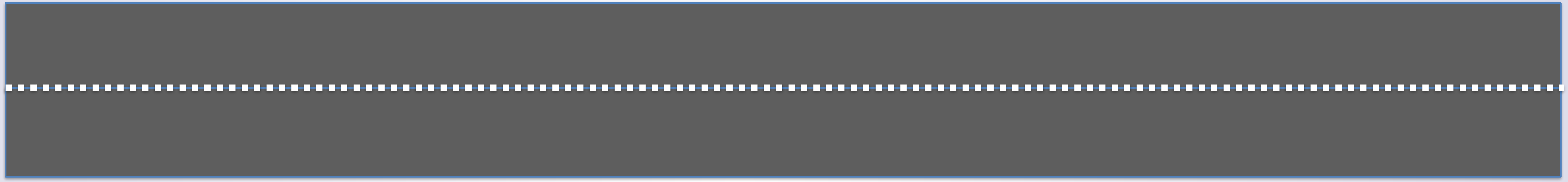
- Latency: 700 miles @ 70 mph= 10 hours to travel
- Throughput: 1 car/ 10 hours = 0.000028 cars/second ?

Latency vs Throughput

- Latency: 700 miles @ 70 mph= 10 hours to travel
- ~~Throughput: 1 car/ 10 hours = 0.000028~~ cars/second ?
- Throughput, for example: 0.3 cars / second
  - Pipeline of cars on the road at one time

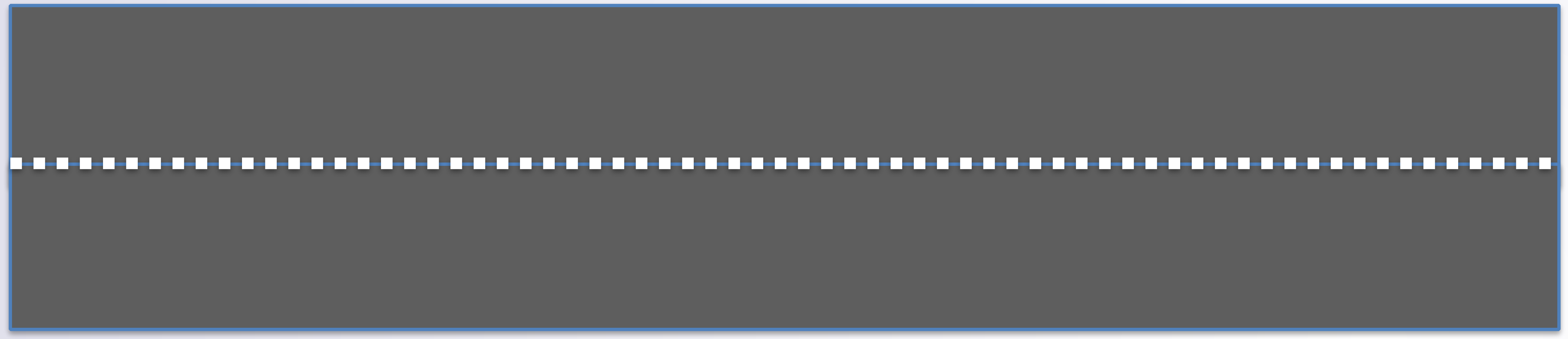
Duke
UNIVERSITY

Latency vs Throughput



- Different things: can affect one without changing other
  - Another lane? Throughput **improves**, latency **unchanged**

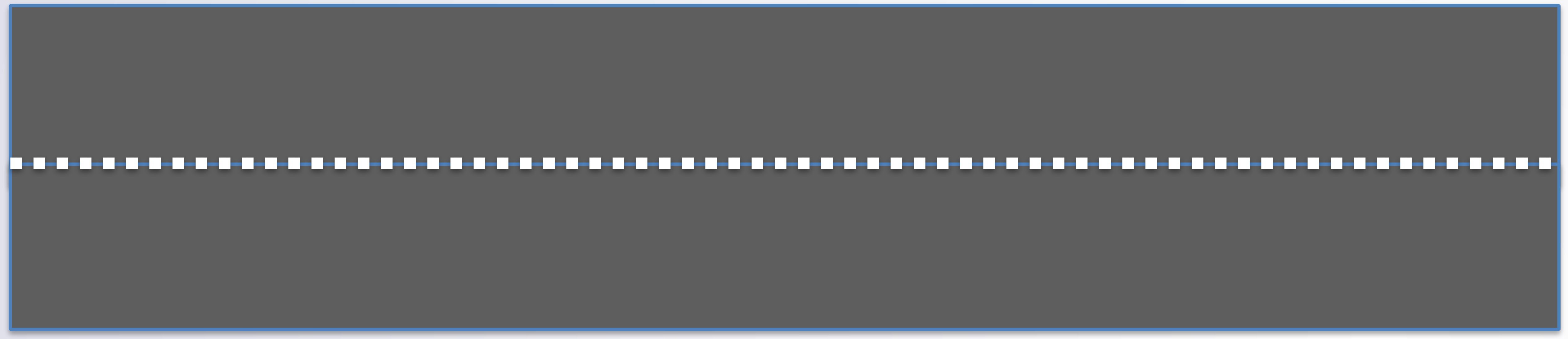
Latency vs Throughput



- Different things: can affect one without changing other
  - Another lane? Throughput **improves**, latency **unchanged**
  - Shorter road? Throughput **unchanged**, latency **improves**

Duke UNIVERSITY

Latency vs Throughput



- Different things: can affect one without changing other
  - Another lane? Throughput **improves**, latency **unchanged**
  - Shorter road? Throughput **unchanged**, latency **improves**
  - Cars drive faster? **Both improve (*)**
    - (*) Except that you need more space for safety…

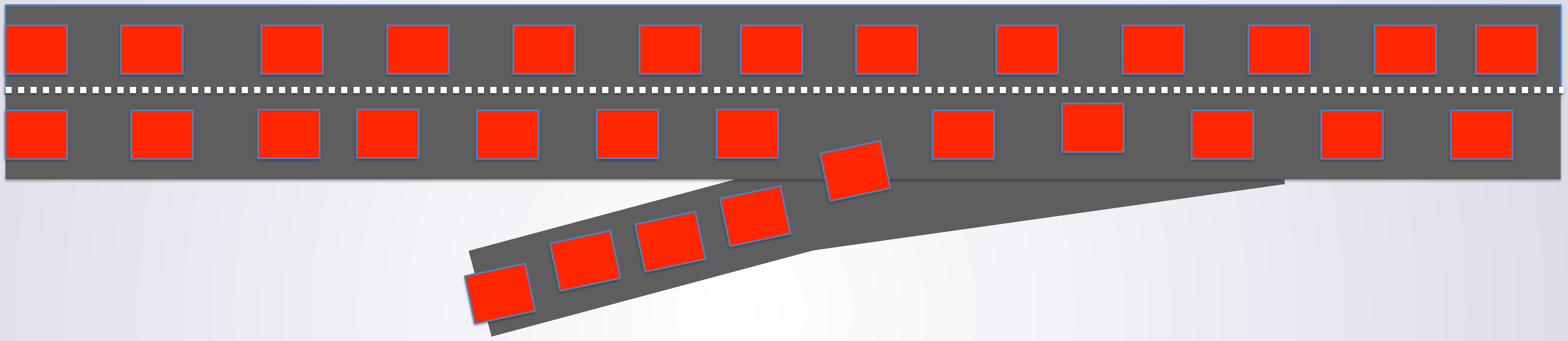
Duke UNIVERSITY

So Which Do We Care About?

- What matters? Latency or throughput?
  - From a **user's** perspective: **latency**
- From a **system** perspective, **both** matter
  - Need high throughput to get low latency for many users
  - Latency goes up with resource contention and queueing delays
- Back to our road example…

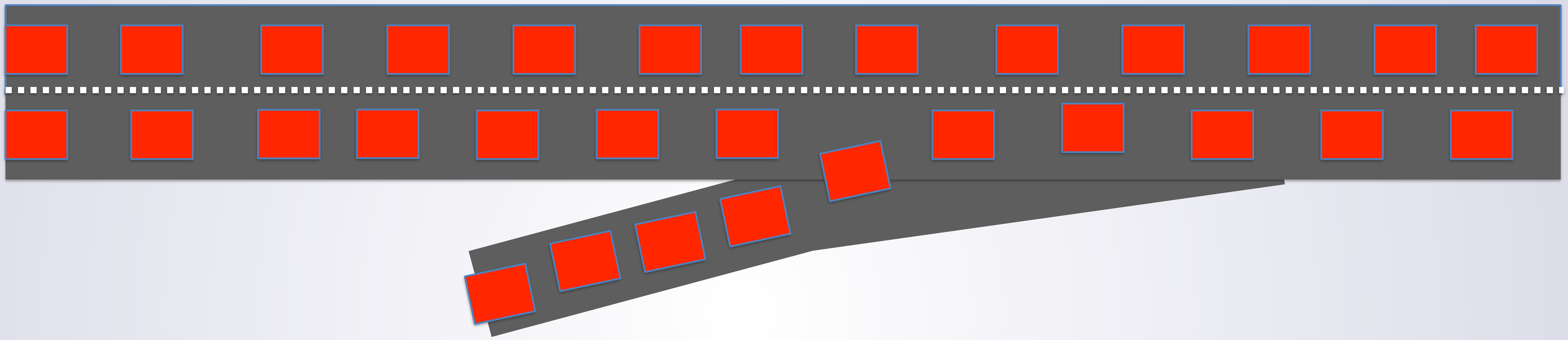
Duke
UNIVERSITY

Latency vs Throughput



- Heavy traffic, more cars merging in.. What happens?

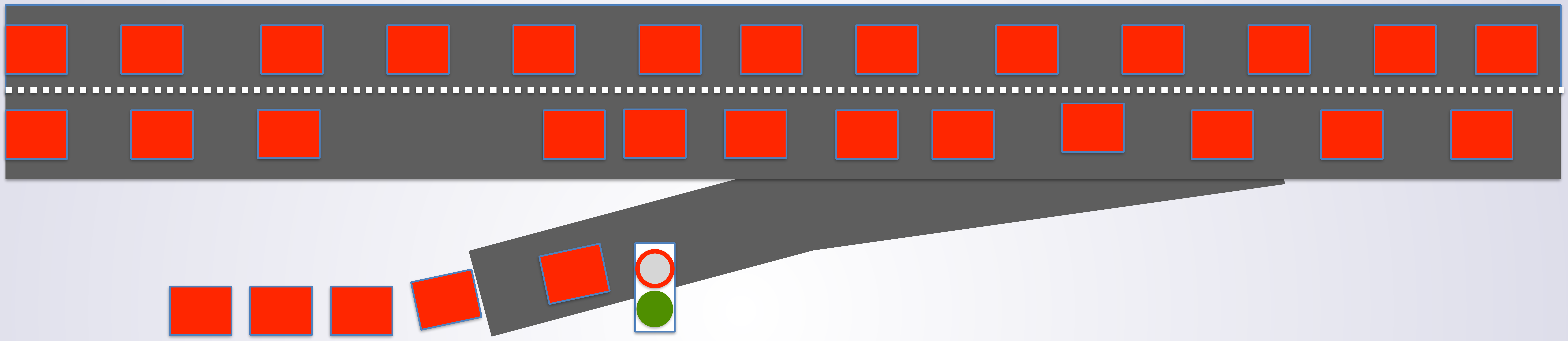
Latency vs Throughput



- Heavy traffic, more cars merging in.. What happens?
  - Latency goes up
    - Cars **slow down** due to resource (road space) contention

Duke UNIVERSITY

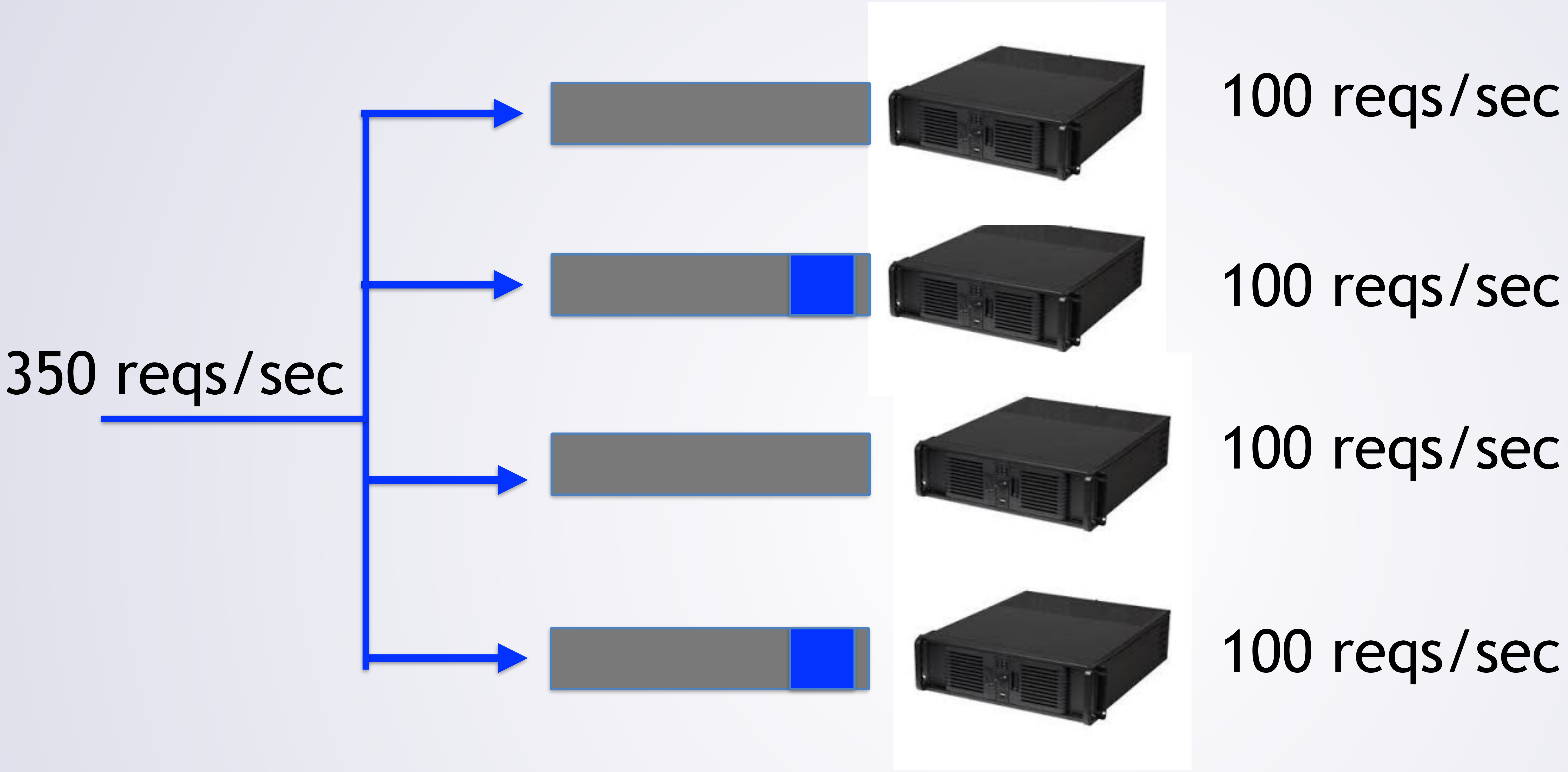
Latency vs Throughput



- Alternative: merge traffic lights
  - Traffic queues up (at on ramp)
  - Reduce resource contention (keep speeds higher)
  - Ideally: maintain speed, extra latency comes in queue

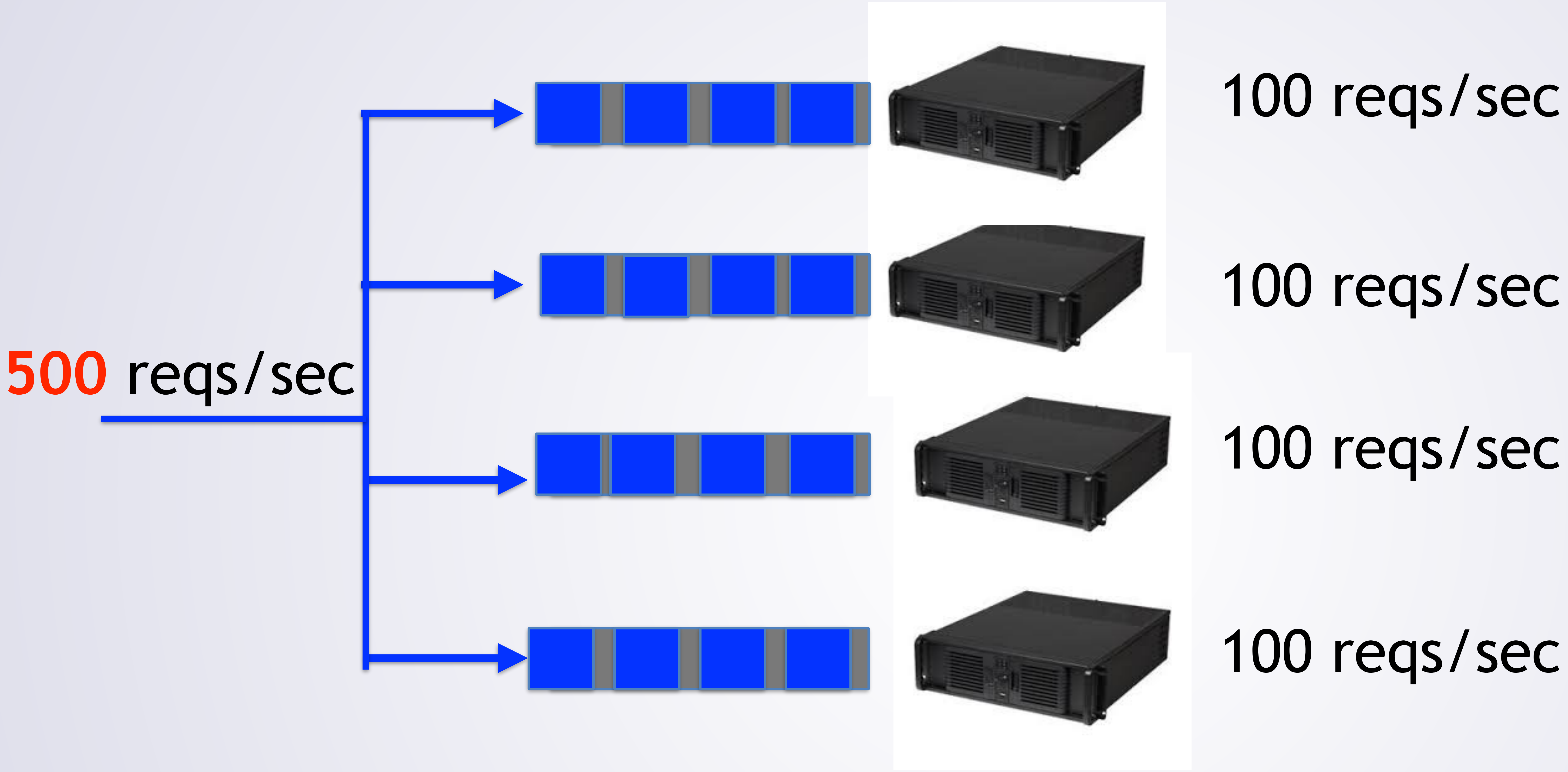
Duke UNIVERSITY

Latency vs Throughput



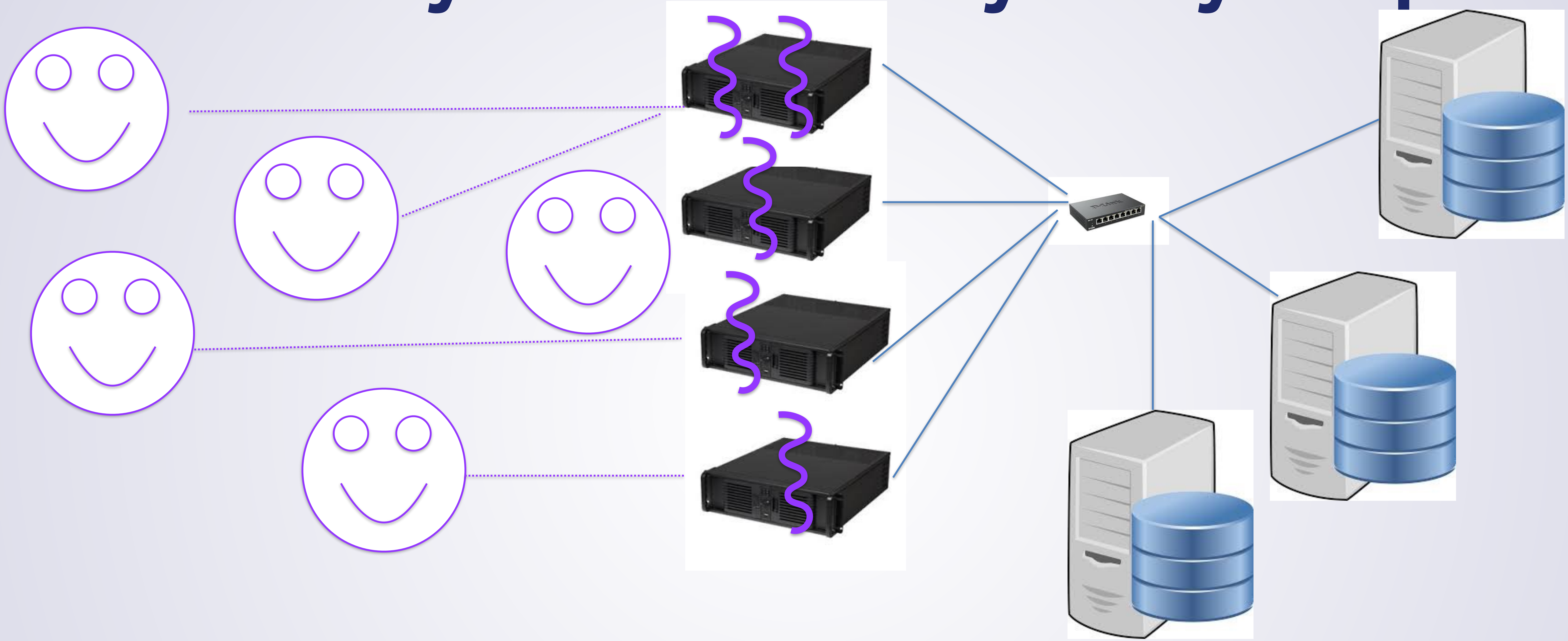
- Adding more systems won't help latency (probably)
  - May experience resource contention (cache, locks, etc…)

Latency vs Throughput



- System is oversubscribed: **queuing delays** add to latency
    - Adding more throughput would reduce latency!

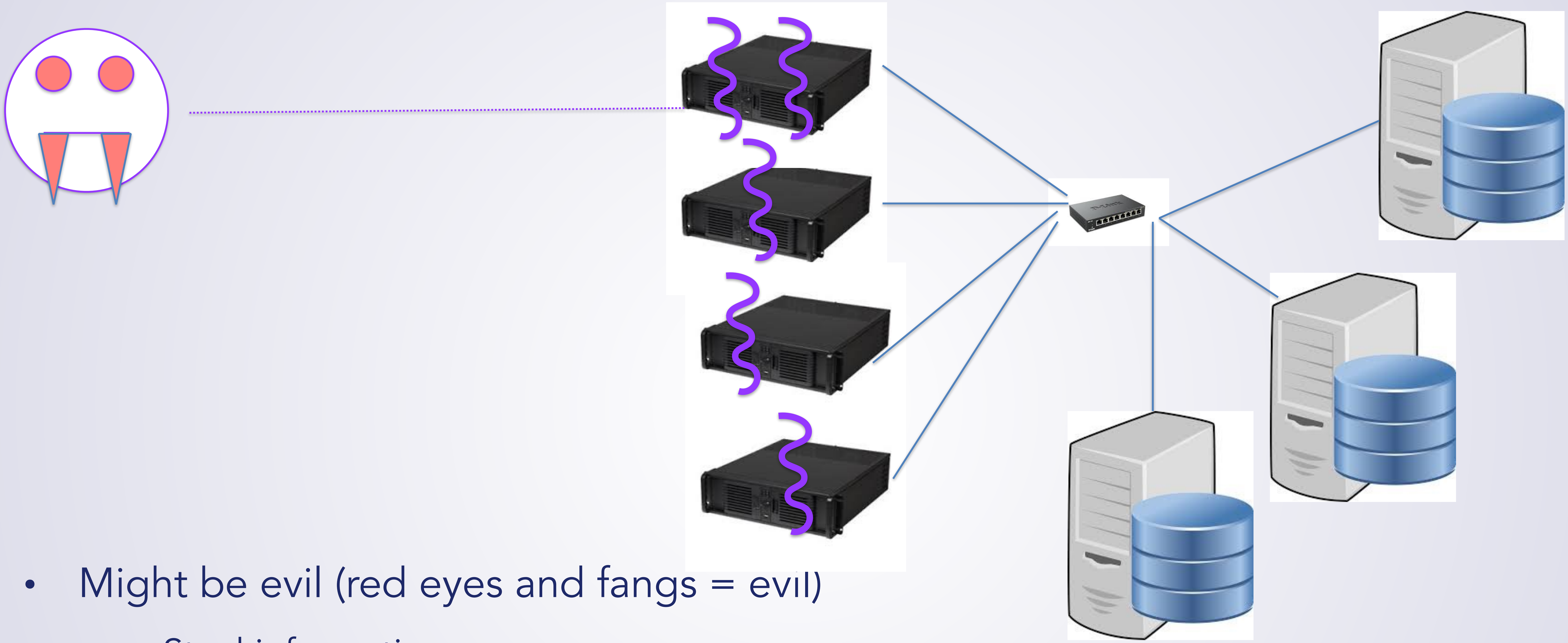
Used By You vs Used By Many People



- Another complexity: **trust**
  - Are all those users out there good?

Duke UNIVERSITY

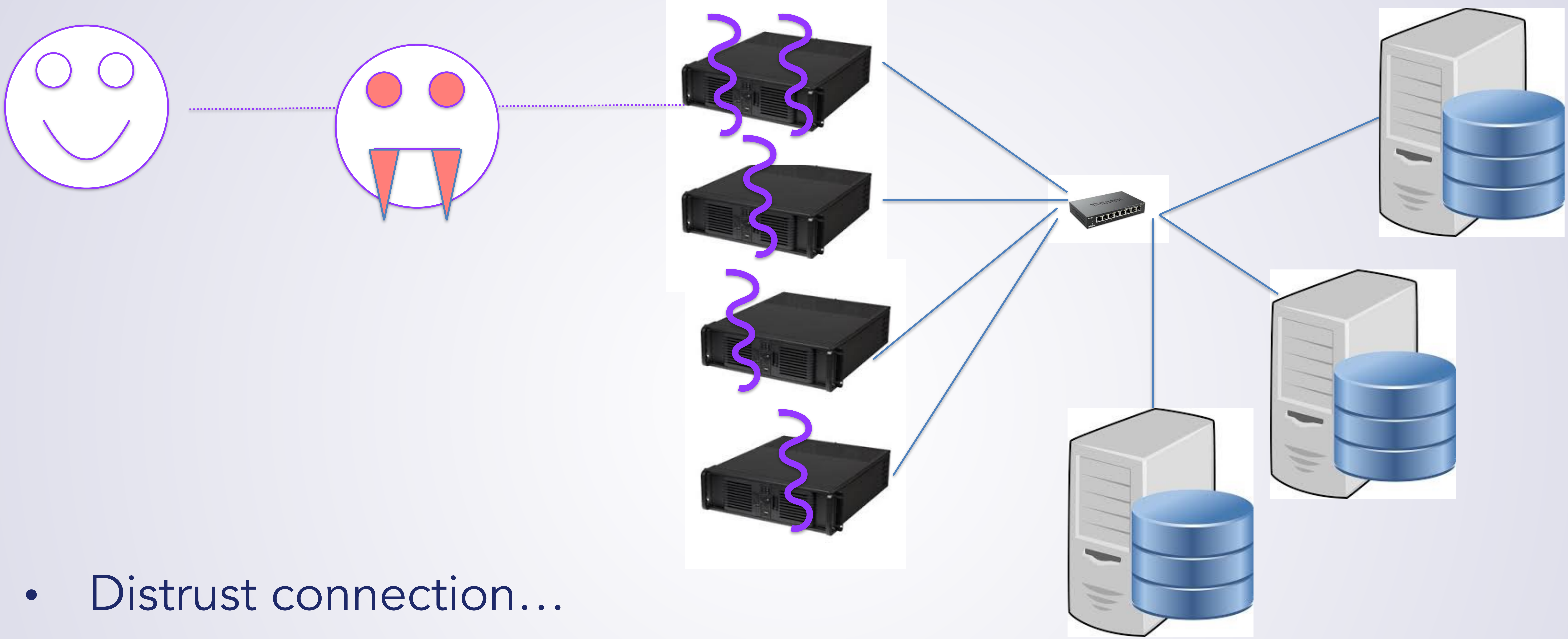
Trust?



- Might be evil (red eyes and fangs = evil)
  - Steal information
  - Modify information
  - Use server for nefarious purposes (spam,…)

Duke UNIVERSITY

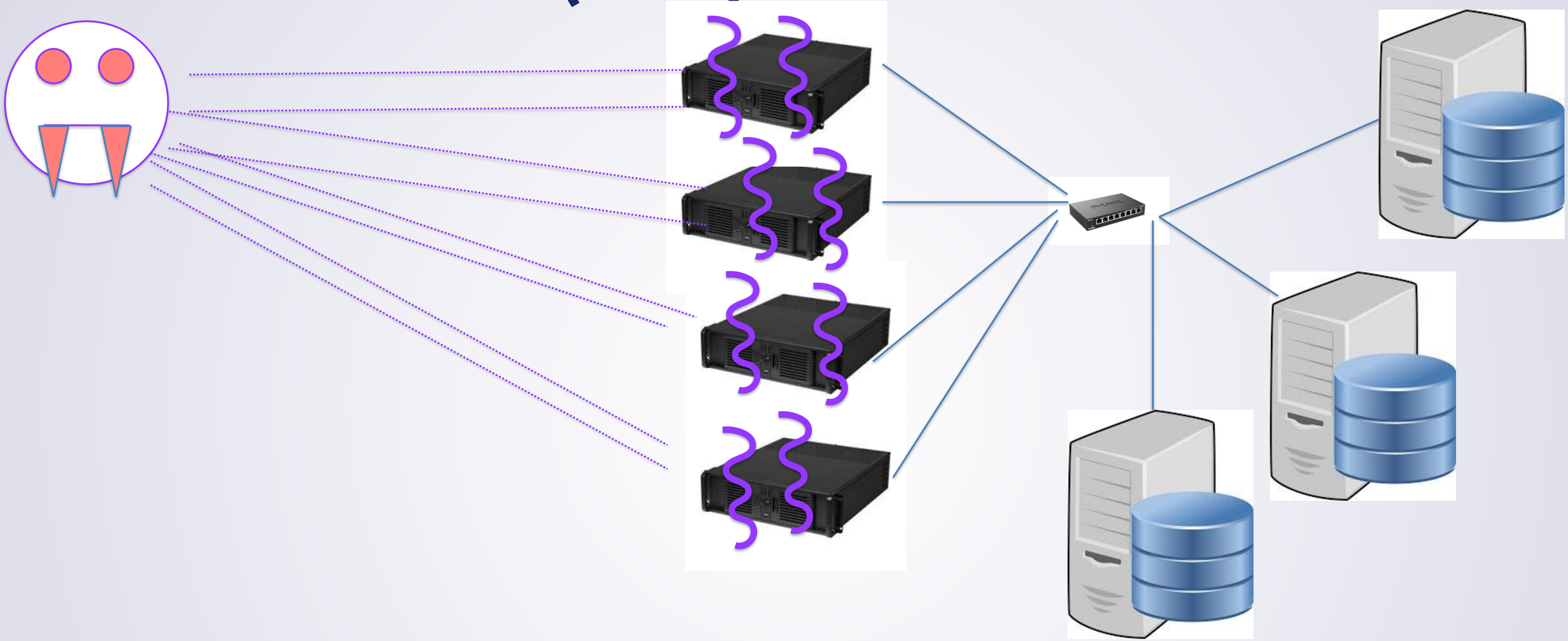
Trust?



- Distrust connection…
  - Adversary might eavesdrop (passively gather information)
  - Or tamper with connection (actively change what is sent)

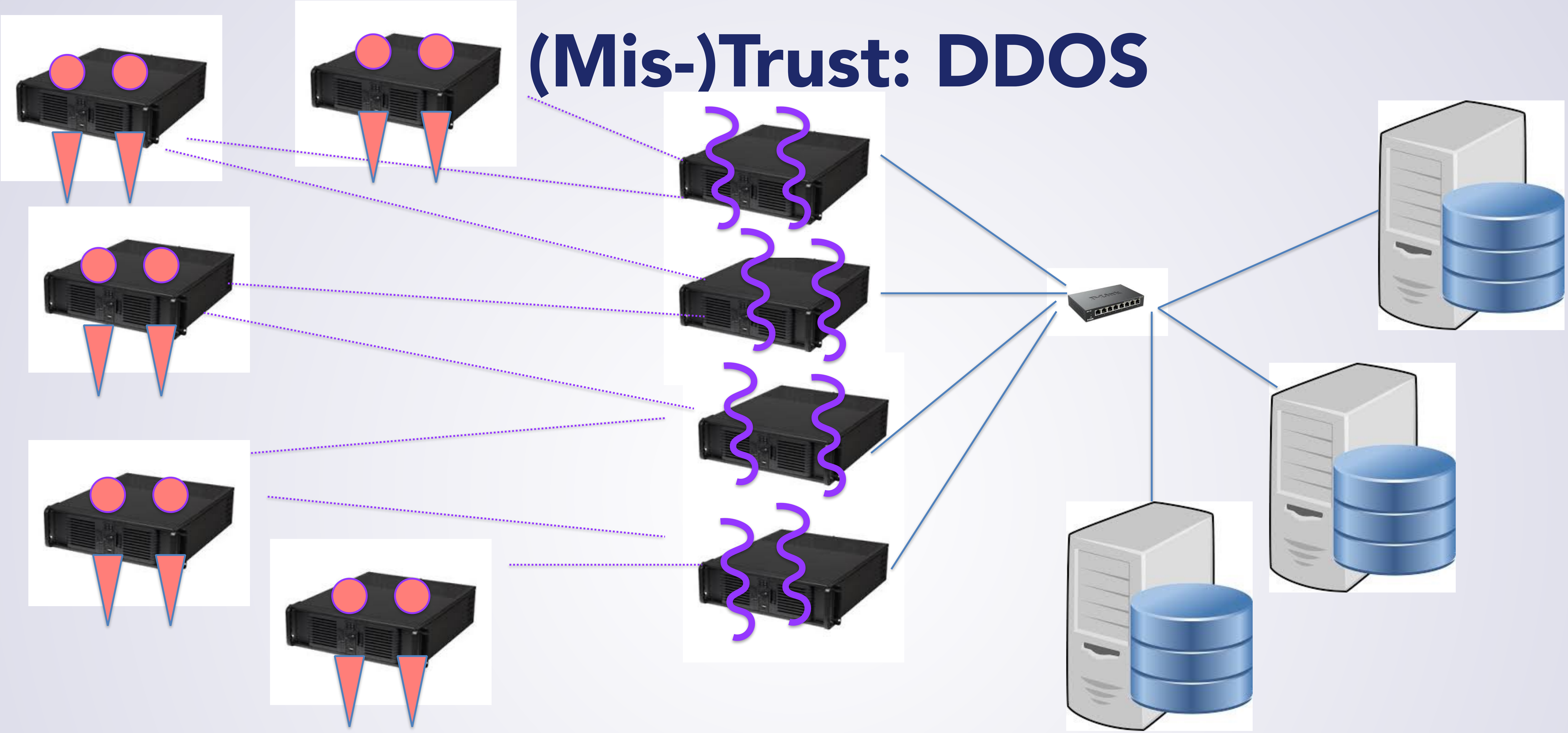
Duke UNIVERSITY

(Mis-)Trust: DOS



- Malicious user may also attempt to deny service
  - DOS = Denial of Service

(Mis-)Trust: DDOS



- Malicious user may also attempt to deny service
  - DOS = Denial of Service
  - DDOS = Distributed Denial of Service

Duke UNIVERSITY

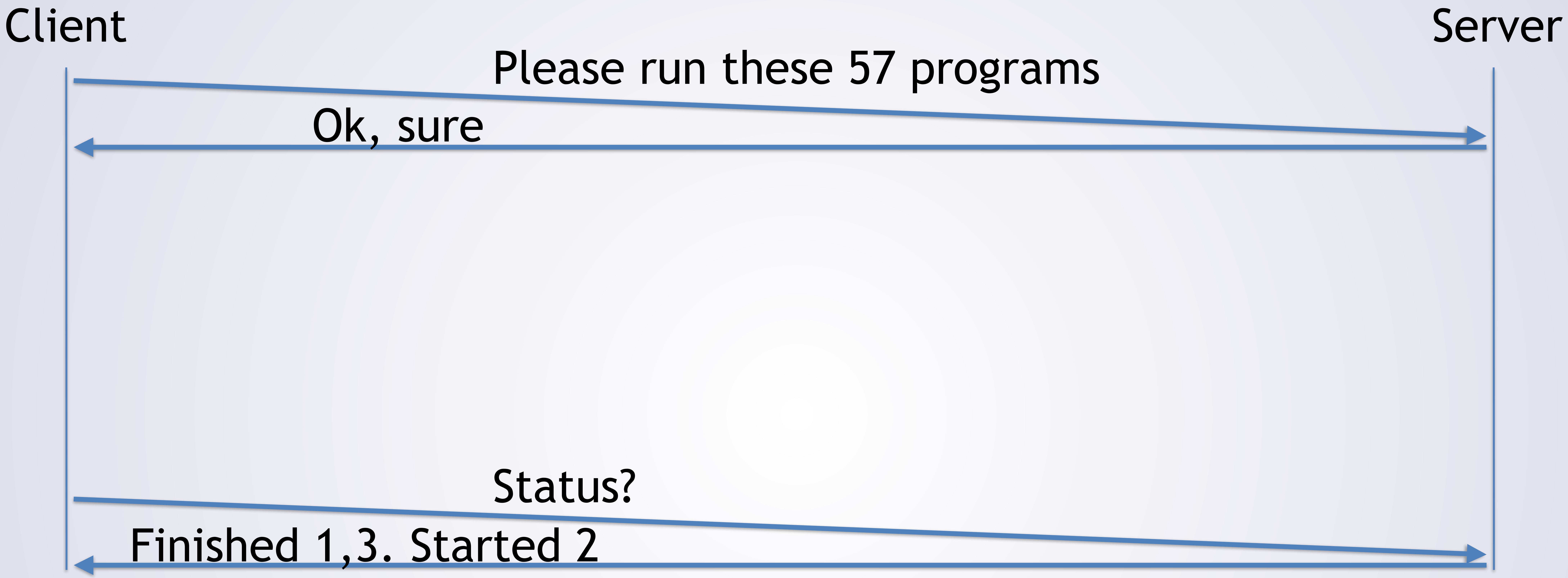
What Does The Server Look Like?

- Now, we've seen a bunch of differences in constraints/requirements
- But what does the server itself look like?
  - …it depends…

Always the answer in CE

Duke
UNIVERSITY

Batch Servers

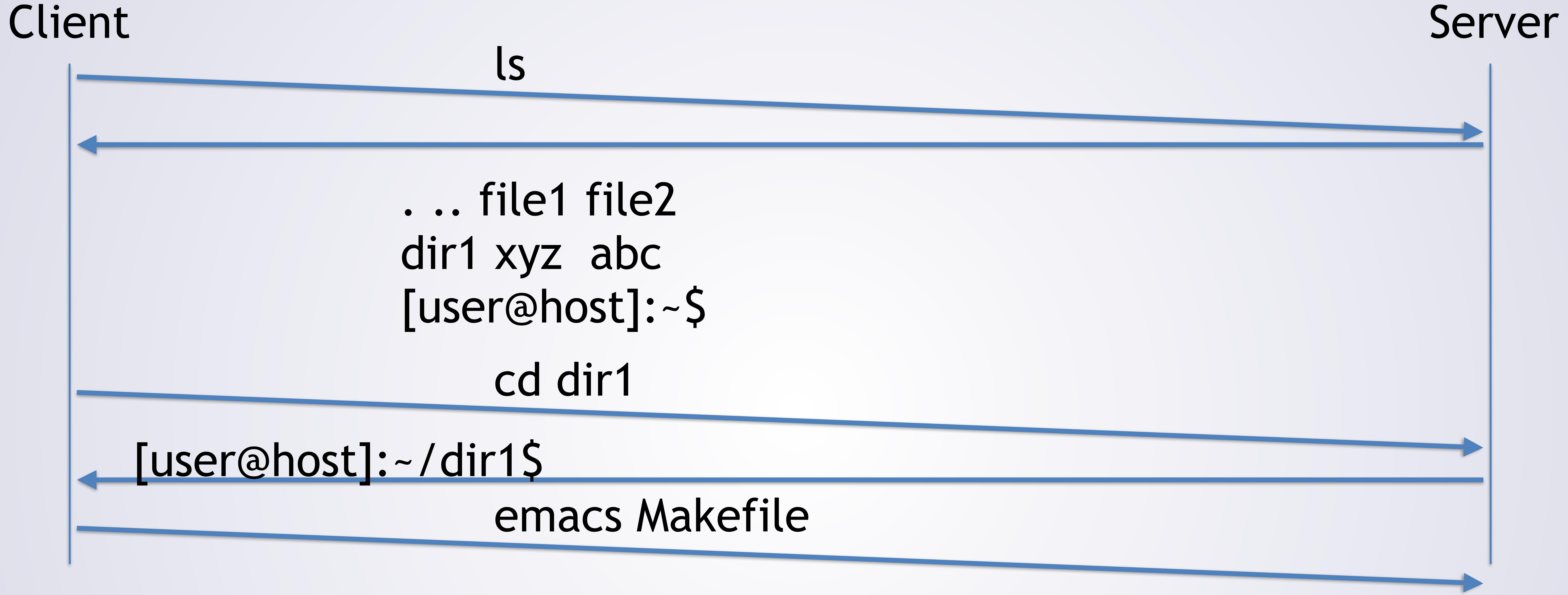


- Submit jobs (possibly in bulk)
- Server will do them later (when it can)

Batch Servers

- Examples:
  - Sun Grid Engine, Condor, Platform LSF
- Mostly queue requests
  - Possibly with priorities
- Most concerned with throughput
  - Overhead latency << job latency
- Running code for user?
  - Generally more trust than most systems

Interactive Servers



- (Many ?) requests, sent/handled frequently

Interactive Servers

- Examples:
  - sshd
  - Game servers (Fortnite, WoW, etc.)
- Latency is critical
- Web-servers similar,
  - Just flurry of requests, then close connection

Database Servers / DBMS

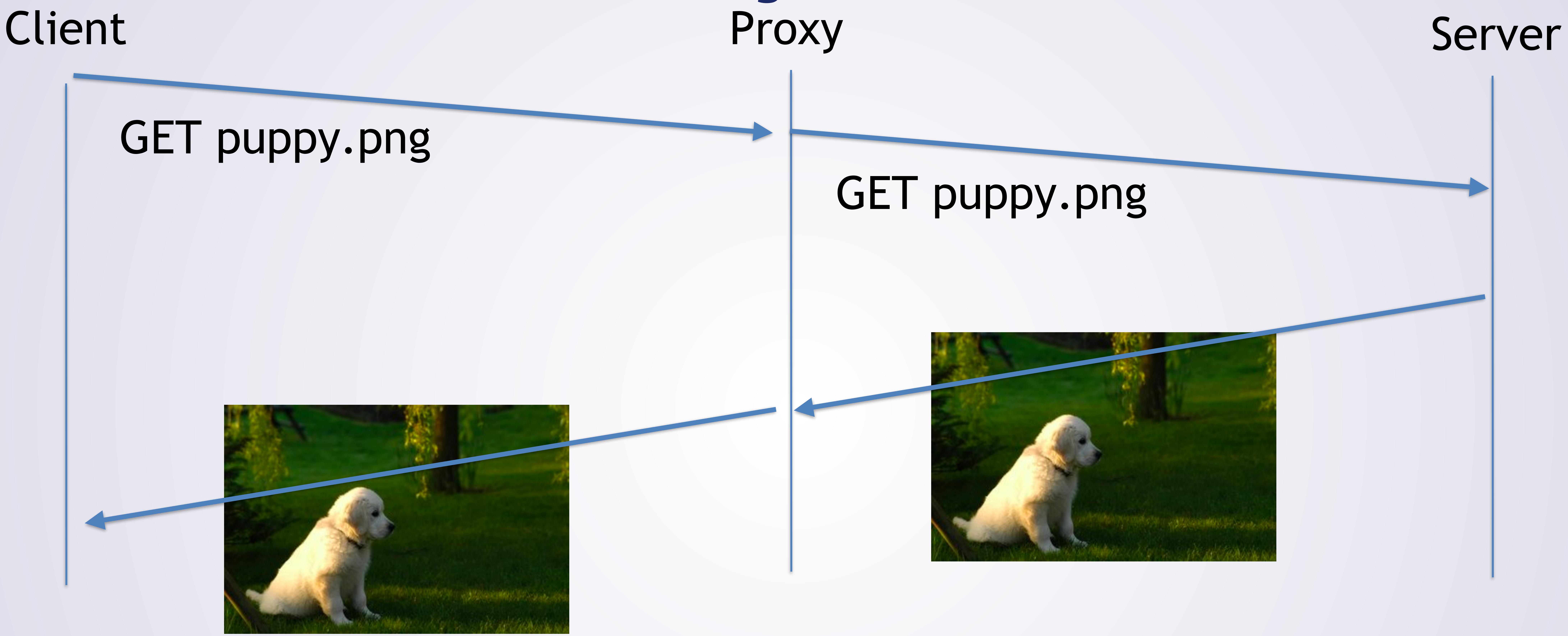
- Process queries from clients
- Often must efficiently process many tuples to satisfy query
  - High tuple throughput -> low response latency
- Often have special IO needs, require much RAM
- Quite a complex beast (topic of advanced database classes)
- Examples: Postgres, MySQL, Oracle,….

File Servers

- Put filesystem on remote server
- Why?
  - Use same files on many systems
  - E.g., login to any lab computer, have same home directory
- Compute requirements << IO requirements
  - IO slower than compute anyways
- Examples: NFS, AFS,…

Duke
UNIVERSITY

Proxy Servers



- Pass requests to "actual" server

...but really...all the same

```
while (true) {
    req = accept_incoming_request();
    resp = process_request(req);
    send_response(req, resp);
}
```

Note: really need some parallelism

- Pretty much all of these have a unix daemon that
  - Accepts requests
  - Processes them
  - Sends responses

Duke UNIVERSITY

Coming soon: Unix Daemons

```
while (true) {
    req = accept_incoming_request();
    resp = process_request(req);
    send_response(req, resp);
}
```

- Soon: all the details of how to make this work
  - You'll write a particular type of Daemon
- Will utilize 650 knowledge: concurrency + socket programming

Duke UNIVERSITY

Coming soon: Unix Daemons

```
while (true) {
    req = accept_incoming_request();
    resp = process_request(req);
    send_response(req, resp);
}
```

- Server side web development
  - How to process the request
  - Web-servers (Apache,...) have ways to "hook up" to code to generate content

Duke
UNIVERSITY

Coming Up…

- Web Protocols & Technologies

- Protocol/API/Server Concepts
  - Asynchronous requests
  - At least or at most once
  - Idempotent Operations