CS 188

Scalable Internet Services

Andrew Mutz
October 4, 2015

Today's Agenda

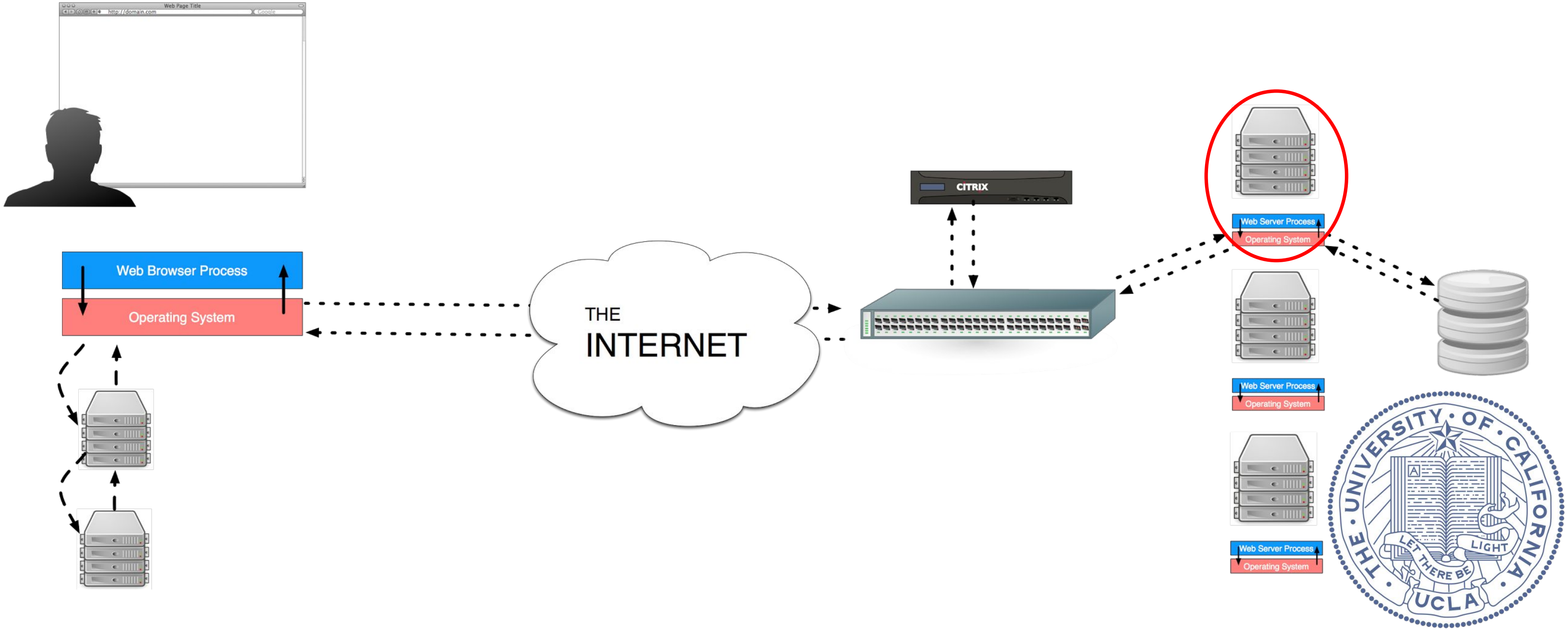
- If you don't have a group yet, you need to find one ASAP.
  - Email me if you need help finding a group.

- I will give a number of PTEs to grad students. If you are a grad student, please email me again (so I know you are attending lecture), and I will randomly assign PTEs to grad students before next lecture

Today's Agenda

- Motivation
- HTTP Servers
- Application Servers
- For next time...

Motivation



Web Page Title
http://domain.com
Google
Web Browser Process
Operating System
THE
INTERNET
CITRIX
Web Server Process
Operating System
Web Server Process
Operating System
Web Server Process
Operating System
THE UNIVERSITY OF CALIFORNIA
LET THERE BE LIGHT
UCLA

Motivation

After today, you should understand some of the tradeoffs below

Scalable Internet Services Templates

Single Instance Templates

Both the app server, and database are located on a single EC2 instance.

- **NGINX + Passenger** (Recommended for regular testing):
  NGINX handles requests to port 80 and passes connections to instances of the app through Passenger. Multiple concurrent connections are supported.
  https://scalableinternetservices.s3.amazonaws.com/SinglePassenger.json
- **Puma:**
  Puma allows both thread-based and process-based concurrency.
  https://scalableinternetservices.s3.amazonaws.com/SinglePuma.json
- **WEBrick** (Use only for slow-performance testing):
  WEBrick handles requests to port 80 directly, permitting only a single connection at a time.
  https://scalableinternetservices.s3.amazonaws.com/SingleWEBrick.json

Specify Parameters

Specify values or use the default values for the parameters that are associated with your AWS CloudFormation template.

Parameters

AppInstanceType	m3.medium	The AppServer instance type.
Branch	master	The git branch to deploy.
ProcessParallelism	2	The number of worker processes.
RubyVM	MRI	The number of threads within each worker processes.
TeamName	demo	Your team name.
ThreadParallelism	4	The number of threads within each worker processes.

Motivation

We’ve seen the HTTP protocol.

The world is full of browsers, apps & other clients that expect to be able to

- Open a TCP socket
- Send over a request (verb & resource)
- Have the request processed
- Receive data in a response
- Reuse the socket for multiple requests

The software systems that do this are generally divided into two parts

- HTTP Servers
- Application Servers

Motivation

Why not just have a single process that handles all this?

Why do we need two separate notions of an HTTP server and an App server?

Motivation

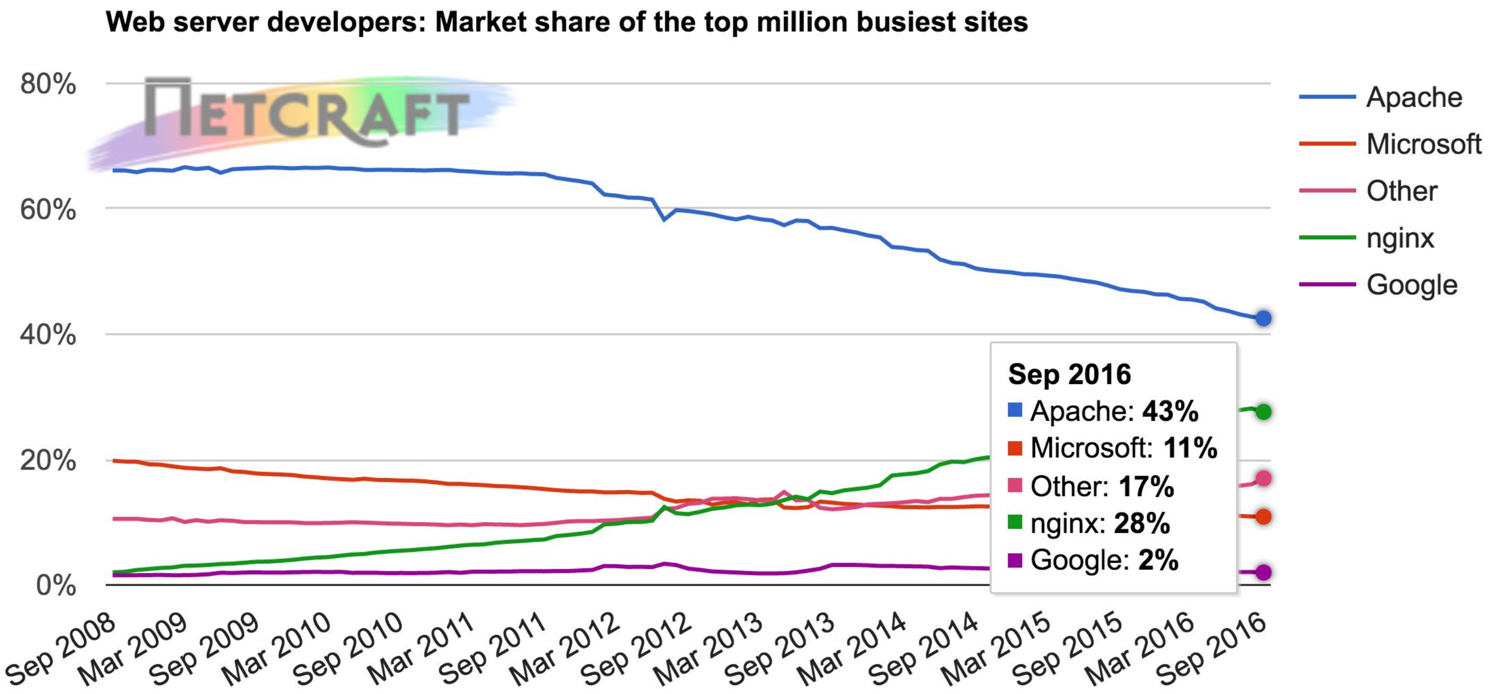
Why not just have a single process that handles all this?

Why do we need two separate notions of an HTTP server and an App server?

The general answer is the two have separate concerns and separate design goals.

- HTTP Server:
  - High performance HTTP implementation
  - Stable, secure, relatively static
  - Highly configurable and language/framework agnostic
  - Concurrency concerns dealt with here (mostly)
- App Server:
  - Specific language, frequently lower-performance
  - Contains business logic and is very dynamic
  - More concerned with optimizing human resources
    - Commonly a large MVC architecture

HTTP Servers



HTTP Servers

HTTP Server's responsibilities:

- Parse HTTP requests and and craft HTTP responses very fast
- Dispatch to the appropriate handler and return response
- Be stable and secure
- Provide clean abstraction for backing applications

Many possible ways to architect an HTTP server:

- Single Threaded
- Process per request
- Thread per request
- Process/thread worker pool
- Event-driven

HTTP Servers - Single Threaded

Single threaded approach:

- Bind() to port 80 and listen()
- Loop forever and...
  - Accept() a socket connection
  - While we can still read from it
    - Read a request
    - Process that request
    - Write response
  - Close connection

If another request comes in before we get back around to accept() another, what happens?

HTTP Servers - Single Threaded

Problem!

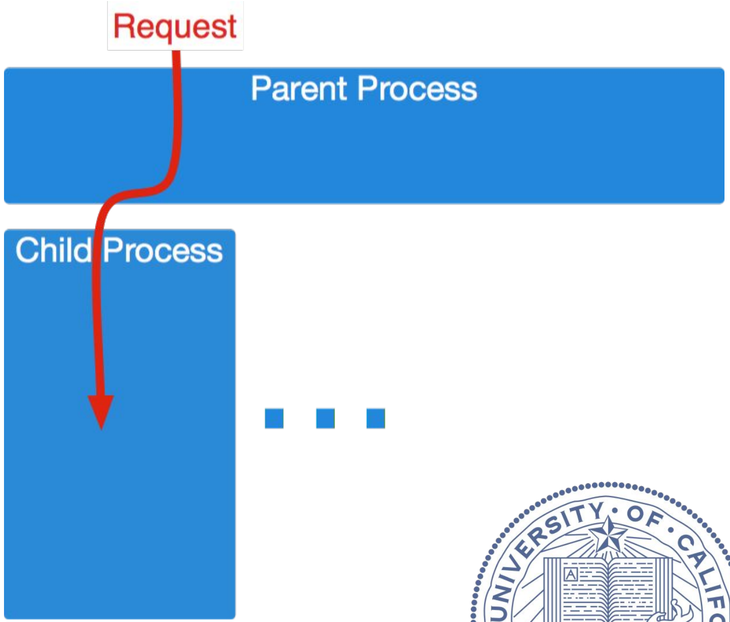
- If we don't quickly get back to accepting more connections, clients end up waiting or worse
- We are building web applications, not web sites:
  - These requests are usually much more than simply serving a file from disk
  - It is common to have a web request doing a significant amount of computation and business logic
  - It is common to have a web request talk to multiple external services: databases, caching stores, SOA services
  - These requests can be anything: lightweight or heavyweight, IO intensive or CPU intensive

We can solve these problems if the thread of control that processes the request is separate from that listening and accepting new connections.

HTTP Servers - Process Per Request

Why not handle each requests as a subprocess?

- Bind() to port 80 and listen()
- Loop forever and...
    - Accept() a socket connection
    - if fork() == 0
        - While we can still read from it
            - Read a request
            - Process that request
            - Write response
        - Close connection, exit



Strengths? Weaknesses?

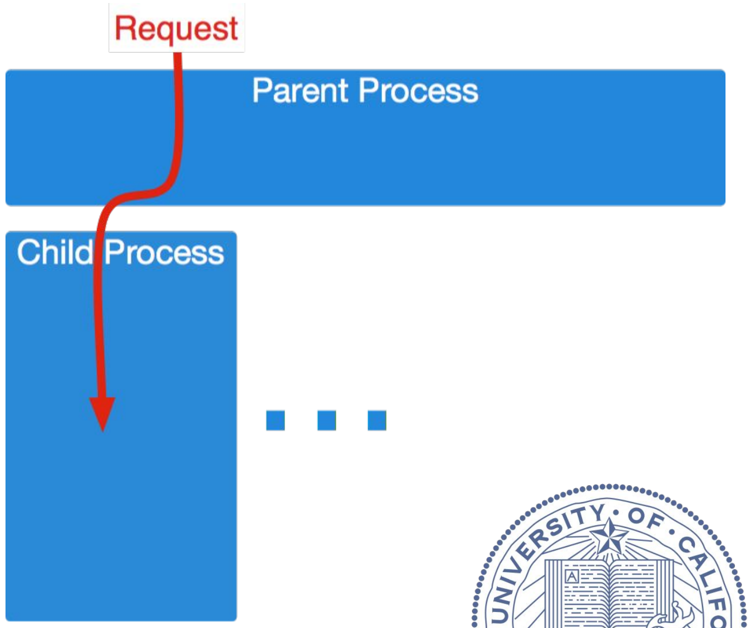
HTTP Servers - Process Per Request

Strengths:

- Simple
- Great isolation between requests
- No problems with multiple threads

Weaknesses:

- Does each request duplicate process memory?
- What happens when load keeps rising?
- Is it efficient to be firing up a process on each request?
    - Each of these does setup work

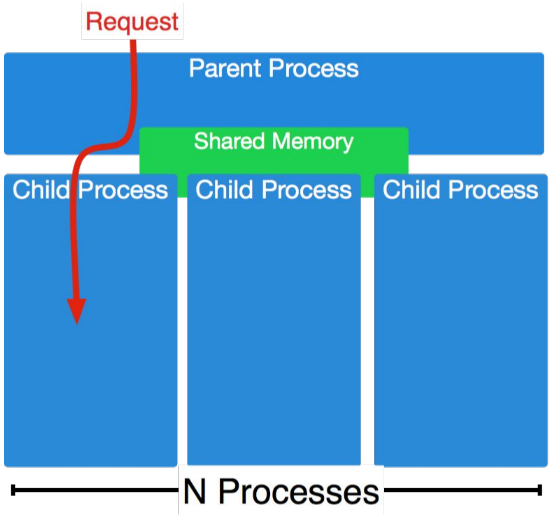


HTTP Servers - Process Pool

Instead of spawning a new process each time we get a request, we can create a pool of N processes at the beginning and dole out requests to them.

The children are responsible for accepting incoming connections, and use shared memory to coordinate.

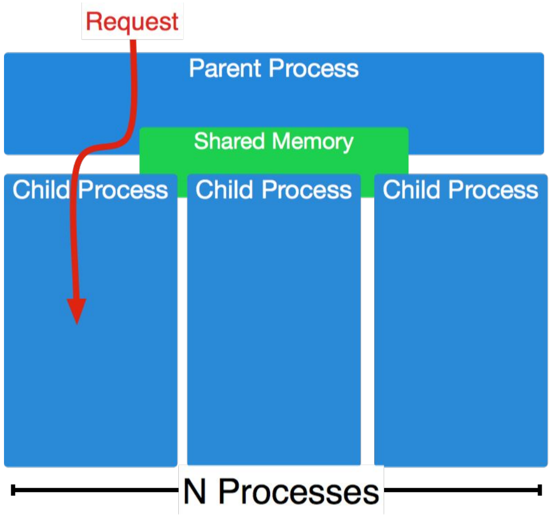
The parent process watches the level of busy-ness of the children and adjusts the number of children as needed.



HTTP Servers - Process Pool

Strengths:

- Great isolation between requests. Children die after M requests to avoid memory leakage.
- Process startup/setup costs are avoided
- More predictable behavior under high load.
- Still no problems with multiple threads



Weaknesses:

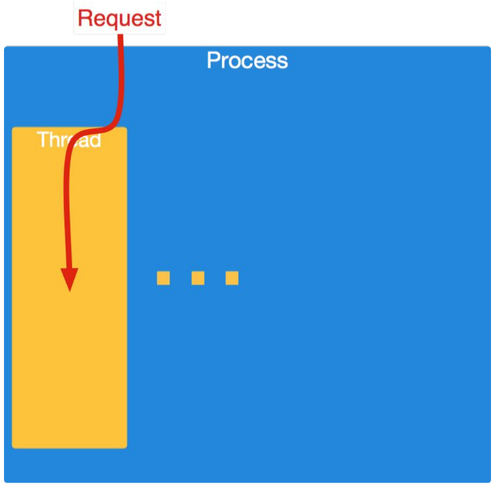
- System more complex than before
- Many processes can mean a lot of memory consumption

This basic structure is Apache 2.x MPM “Prefork”.

HTTP Servers - Thread per request

Why use multiple processes at all? Why not just have a single process, and each time we get a new connection we spawn another thread?

- Bind() to port 80 and listen()
- Loop forever and...
    - Accept() a socket connection
    - pthread_create a function that will...
        - While we can still read from it
            - Read a request
            - Process that request
            - Write response
        - Close connection, thread dies



Strengths? Weaknesses?

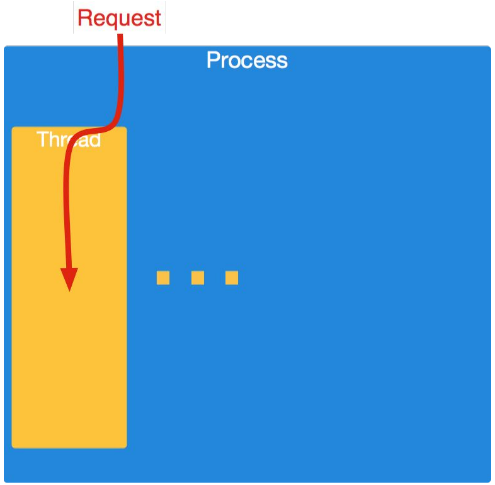
HTTP Servers - Thread per request

Strengths:

- Fairly simple
- Memory footprint is reduced versus processes

Weaknesses:

- The code handling each request must be thread safe
- Pushing thread-safety on to the application developer isn't ideal
- Setup (database connections, etc.) needs to happen each time



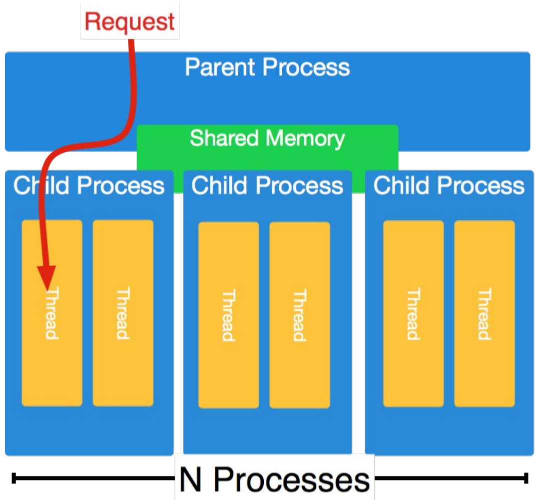
HTTP Servers - Process/Thread Pool

Can we see benefit from combining these techniques?

Master process spawns processes, each with many threads. Master maintains process pool.

Processes coordinate through shared memory to accept requests.

Fixed threads per request, scaling is done at the process level.



HTTP Servers - Process/Thread Pool

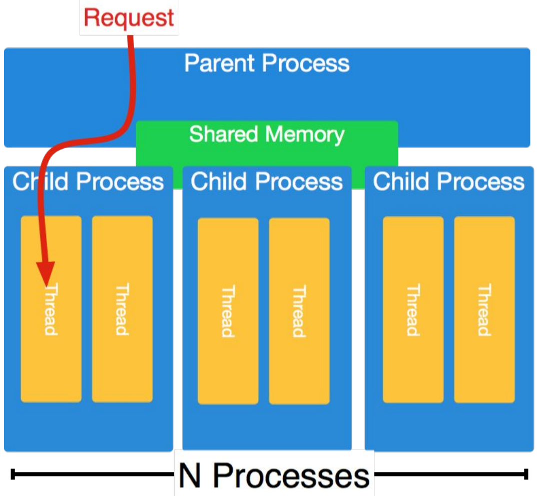
Strengths:

- Faults isolated between processes, but not threads
- Threads reduce our memory footprint and we still get a tuneable level of isolation
- Controlling the number of processes and threads allows predictable behavior under load

Weaknesses:

- Need thread-safe code
- Uses more memory than an all-thread based approach

This is Apache 2.x MPM “Worker”



HTTP Servers

Next we will discuss event-driven architectures and nginx.

But first, a thought experiment: the C10K problem.

HTTP Servers

C10K Problem, originally posed in 2001

- Given a 1ghz machine with 2gb of RAM, and a gigabit ethernet card, can we support 10,000 simultaneous connections?
    - 10,000 clients means…
    - 100Khz CPU, 200Kbytes RAM, 100Kbits/second network for each
    - Shouldn't we be able to move 4kb from disk to network once a second?

This is difficult, but it seems like it shouldn't be.

What are we spending time doing?

HTTP Servers

Lets say I've got 10K connections. Each is doing something like this:

```
Read from the network socket
Parse the request
Open the correct file on disk
Read the file into memory
Write the memory to network
```

HTTP Servers

Lets say I've got 10K connections. Each is doing something like this:

```
Read from the network socket (system call - WAIT)
Parse the request
Open the correct file on disk (system call - WAIT)
Read the file into memory (system call - WAIT)
Write the memory to network (system call - WAIT)
```

HTTP Servers

Each time I'm waiting on I/O, I'm not runnable, but I'm not cost-free.

- I need to be considered every time the scheduler does anything.
- Before I waited, my memory accesses pushed others' data out of caches

This massive concurrency slows down all processes.

HTTP Servers

Since much of these problems have their root in these blocking system calls, can we accomplish all the same tasks without blocking?

Yes, with asyncronous io:

- select(): Here is a list of file descriptors. Block until ready for IO.
- epoll_*(): Lets keep a list of FDs in kernel space. Block until ready.

HTTP Servers - Event Driven

Let’s say we have a list of sockets called `fd_list`

```
loop forever:
    select(fd_list, ...)  //block until one of this list is ready
    for each fd in fd_list
        if fd is ready for IO
            some_handler(fd)
        else do nothing.
```

- `some_handler` can include socket acceptance.
- `some_handler` absolutely can’t do blocking IO.
    - How do we handle this IO?
- What do we do if `some_handler` is doing a lot of computation?

HTTP Servers - Event Driven

These systems are called event driven systems.

- Only need a single thread (although can support more)
- Well known examples:
  - nginx
  - OpenResty
  - Tengine
  - LightTPD
  - netty (java)
  - node.js (javascript)
  - eventmachine (ruby)
  - twisted (python)

HTTP Servers - Event Driven

Strengths:

- High performance under high load
- Predictable performance under high load
- No need to be thread-proof

Weaknesses:

- Poor Isolation
  - If you include a library that does blocking syscall what happens?
- Fewer extensions, since code can't use blocking syscalls
- Very complex
  - See next slide...

HTTP Servers - Event Driven

Code is dominated by callbacks:

```
EM.run {
  page = EM::HttpRequest.new('http://google.ca/').get
  page.errback { p "Google is down! terminate?" }
  page.callback {
    a = EM::HttpRequest.new('http://google.ca/search?q=em').get
    a.callback { # callback nesting, ad infinitum }
    a.errback  { # error-handling code }
  }
}
```

This can lead to code that is confusing and hard to maintain.

HTTP Servers

To recap, there are many possible ways to architect an HTTP server:

- Single Threaded
- Process per request
  - Greatest isolation, largest memory footprint
- Thread per request
  - Smaller memory footprint, less isolation
- Process/thread worker pool
  - Tuneable compromise between processes & threads
- Event-driven
  - Great performance under high load
  - Harder to extend and reduced isolation

Application Servers

We are building web applications, so we will need complex server-side logic.

We can extend our HTTP servers to do this through modules, but there are benefits to breaking out application servers to a distinct process:

- Application logic will be dynamic, whereas HTTP is more static
- Application logic regularly uses high level (slow) languages vs. needs of high-performance
- Security concerns are easier: HTTP server can shield the app server from some things
- Startup/setup costs can be amortized if the app server is running continuously

Instead, we can have a separate Application server and forward each request to it for handling.

We will be looking primarily at Ruby application servers.

Application Servers

Our HTTP server needs to communicate each request to the App server, and the response needs to be sent back.

How is this done?

- CGI - Spawn a process, pass in HTTP headers as ENV variables
- FastCGI, SCGI - modifications to CGI to allow persistent processes.
- HTTP - Essentially a reverse-proxy configuration
    - Why does it make sense to have an HTTP server in front of a server that speaks HTTP?

Application Servers

Many of the same questions regarding concurrency haven't gone away:

- Should we handle these requests via processes? Threads? Evented?

Today we will take a quantitative look at various approaches

- We will not be looking at evented application servers
- We can build evented HTTP Application servers in Ruby (EventMachine), but not Rails.

But before we look at that, let's examine our testing setup

The Demo App

Demo App

Submissions

Title	Url	Community	
Reprehenderit facilis qui illo minima tempora modi ab.	http://rosenbaumryan.biz/irving_padberg	Consequatur velit voluptas perferendis maiores eos.	20 comments
Ex in rerum et voluptas quos dolores.	http://kris.info/kieran.hintz	Consequatur velit voluptas perferendis maiores eos.	20 comments
Vel tempore neque deleniti fugiat.	http://robel.biz/kyle_hahn	Consequatur velit voluptas perferendis maiores eos.	20 comments

A link sharing website

- Multiple communities
- Each community has submitted links
- Each submitted link has a tree of comments

The Demo App

We will simulate users arriving at the website and doing the following:

1. Going to the homepage
2. Waiting for up to 2 seconds
3. Requesting a form to create a new community
4. Waiting for up to 2 seconds
5. Submitting the new community
6. Requesting a form to create a new link submission
7. Waiting for up to 2 seconds
8. Submitting the new link
9. Waiting for up to 2 seconds
10. Delete the link
11. Waiting for up to 2 seconds
12. Delete the community

The Demo App

When we test we will have six phases, of 60 seconds each:

1. Phase 1: every second 1 new user arrives
2. Phase 2: every second 1.5 new users arrives
3. Phase 2: every second 2 new users arrives
4. Phase 2: every second 2.5 new users arrives
5. Phase 2: every second 3 new users arrives
6. Phase 2: every second 3.5 new users arrives

Remember these are users, not requests. There will be many requests per user, and users stay for 5-10s.

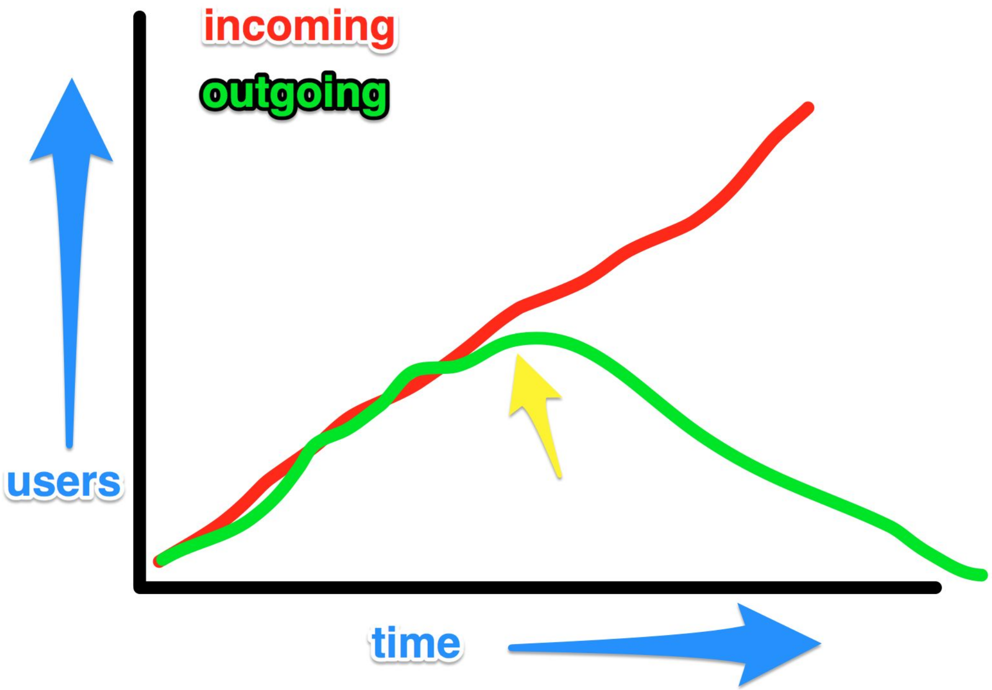
The Demo App

- All tests are conducted on M3-Medium instance.
  - (1 vCPU, 3.75 GB Memory)
- All tests use the single Puma CF template, unless otherwise specified.
  - https://scalableinternetservices.s3.amazonaws.com/SinglePuma.json
- All tests use the `database_optimizations` branch of the demo app:
  - https://github.com/scalableinternetservices/demo/tree/database_optimizations

The Demo App

How to read these graphs: arrival/departure rate

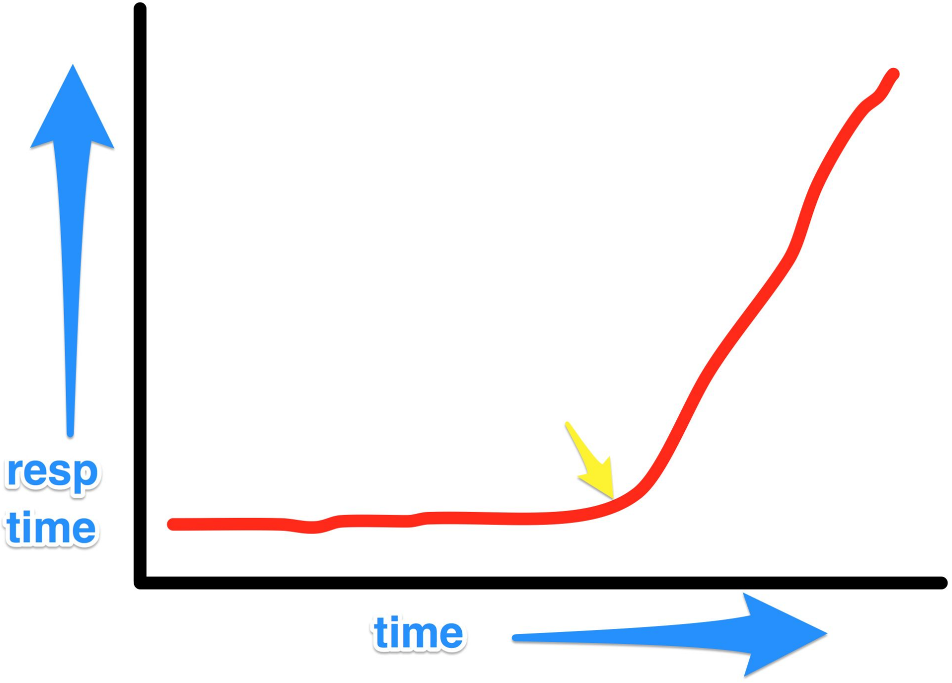
Yellow arrow indicates where system starts to fail



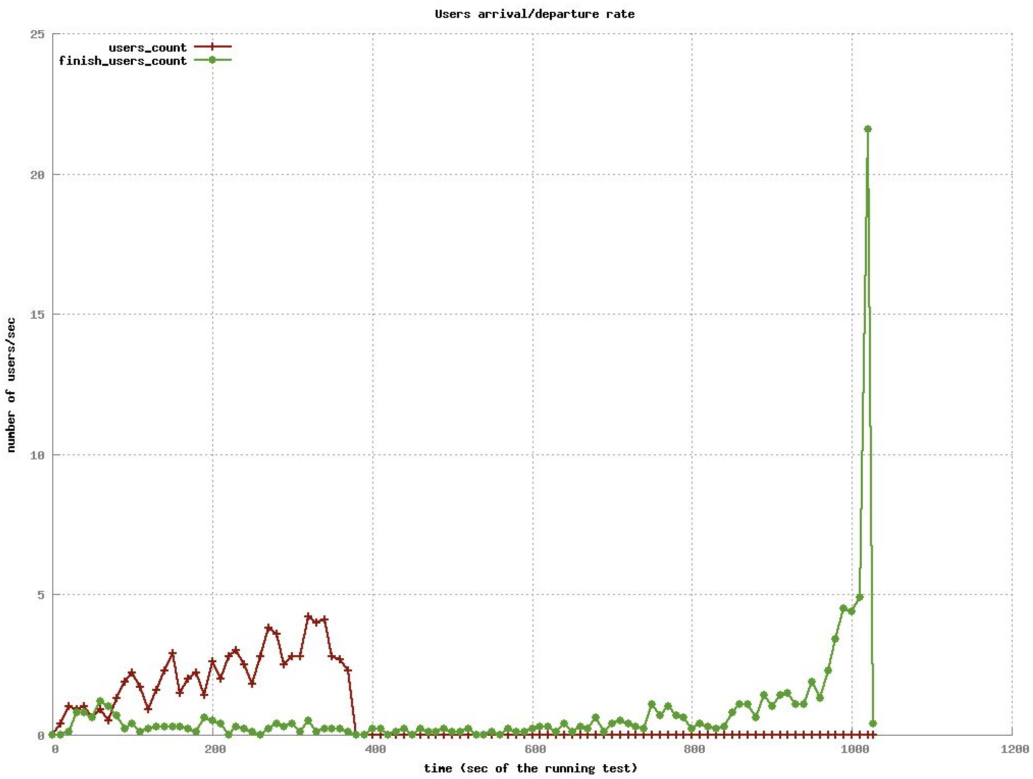
The Demo App

How to read these graphs: response time

Yellow arrow indicates where system starts to fail

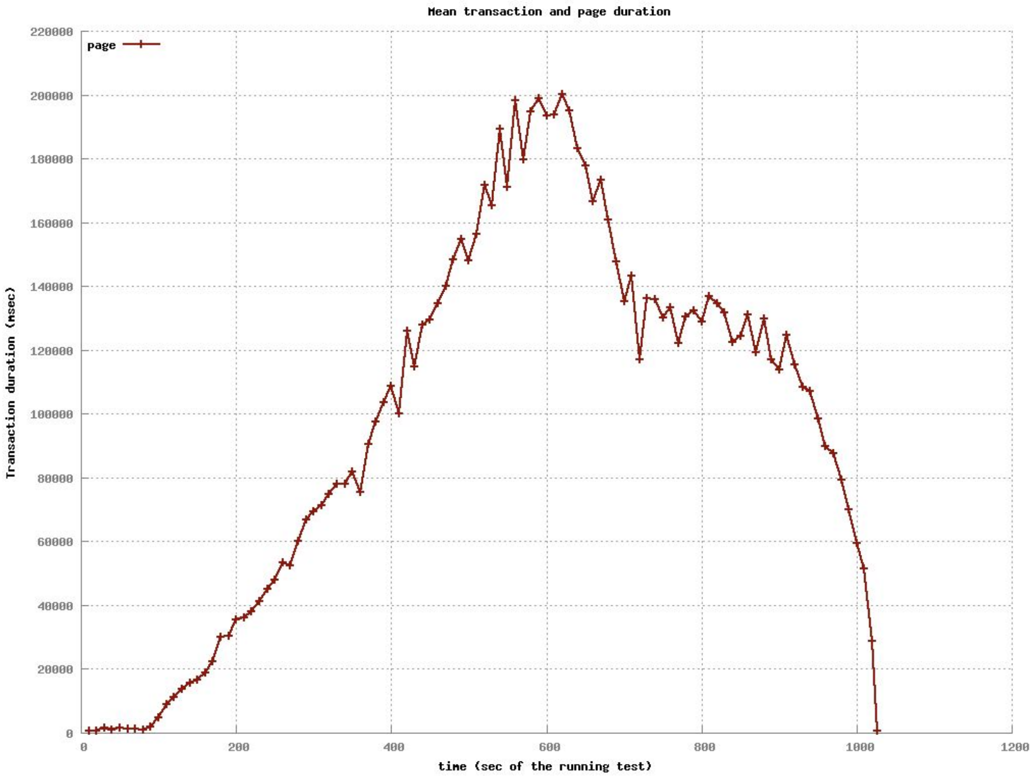


The Demo App - 1/1



Lets see what this looks like for a single-threaded, single-process model.

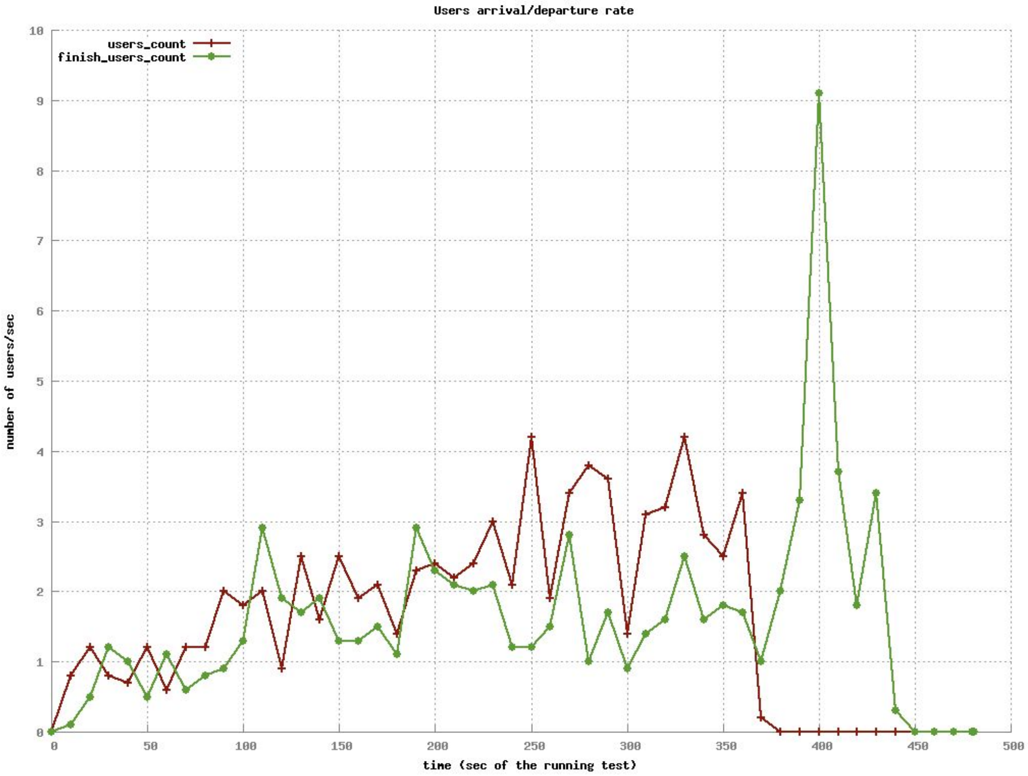
The Demo App - 1/1



Past 1.5 users per second, things fall apart.

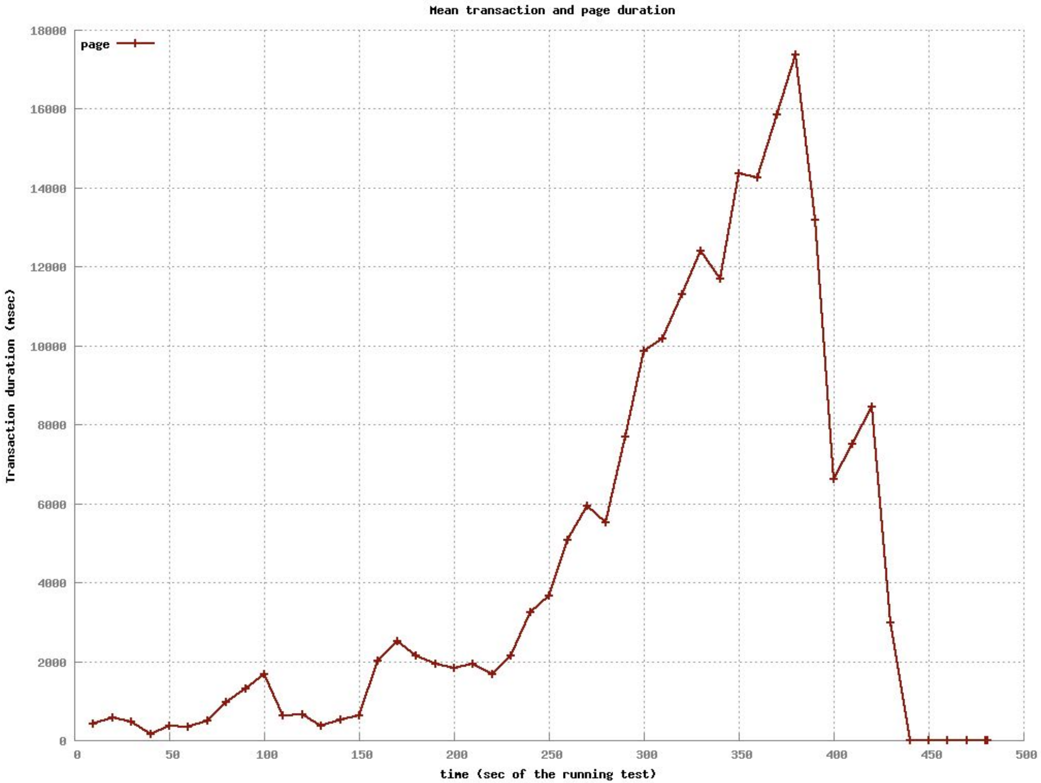
What do you think the effect of increasing to 4 processes will be?

The Demo App - 4/1



With 4 worker processes, we are able to keep up much longer.

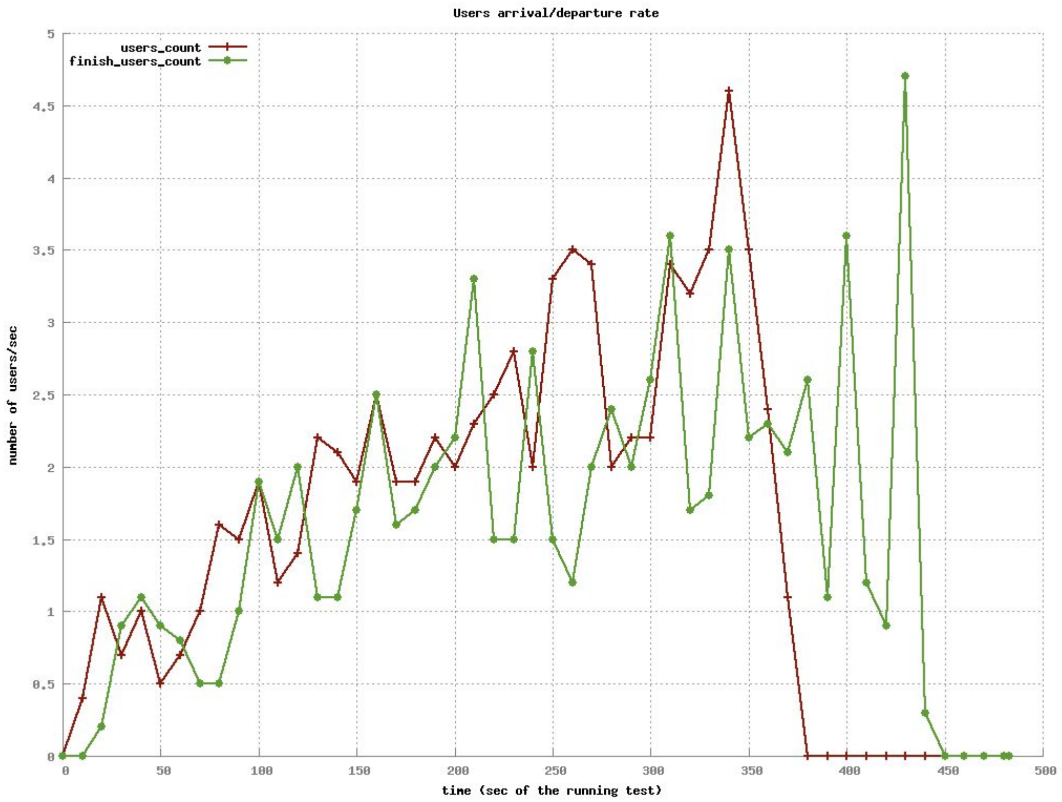
The Demo App - 4/1



Up until 2 users per second, response times are *okay*.

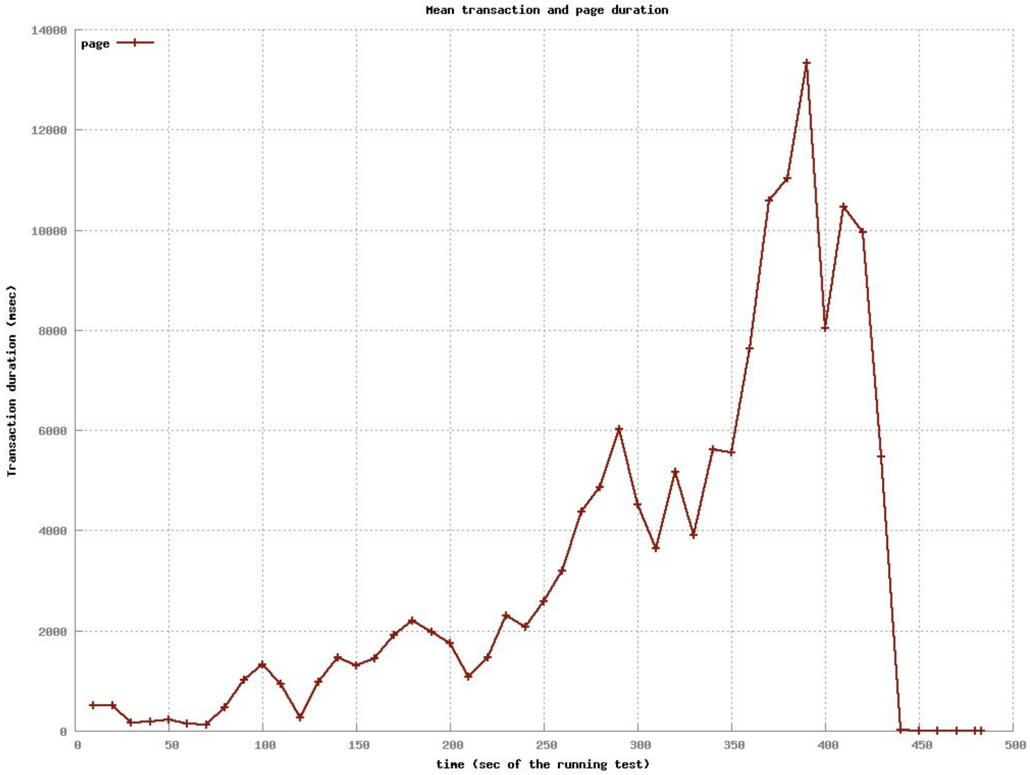
Things fall apart past then.

The Demo App - 16/1



With 16 worker processes, are an improvement over 4, but not a huge improvement.

The Demo App - 16/1



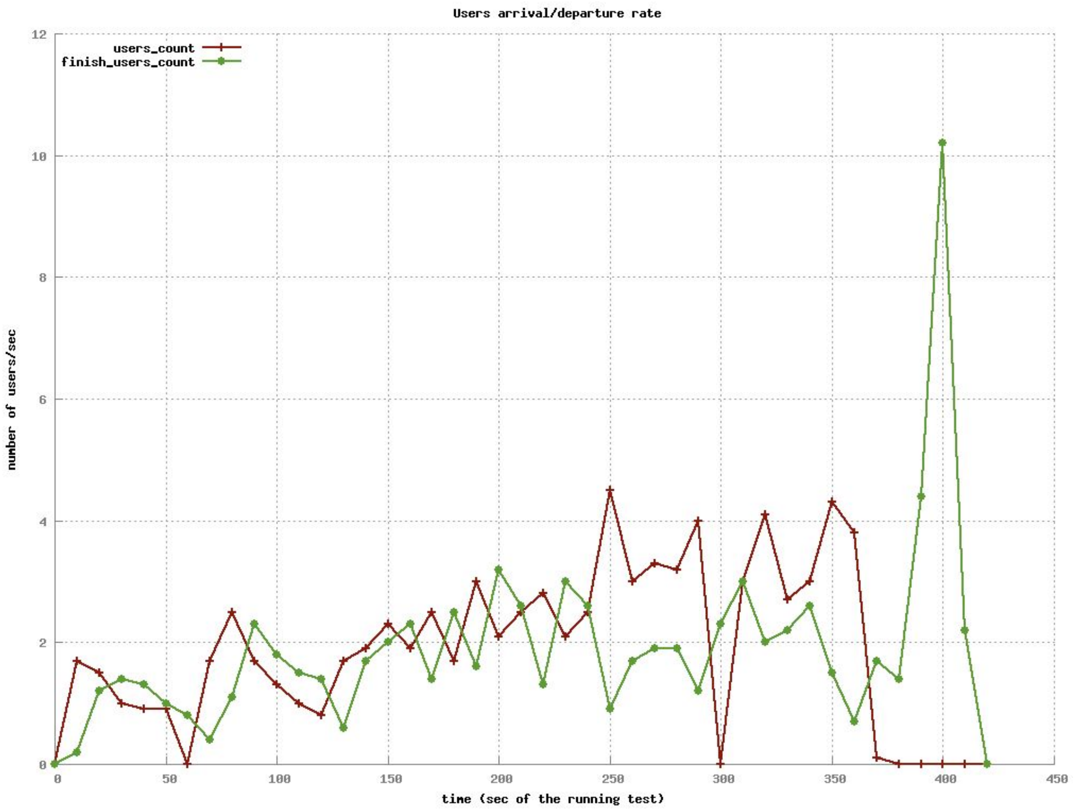
With 16 worker processes, are an improvement over 4, but not a huge improvement.

The Demo App - Threads

Lets try this with threads instead of processes.

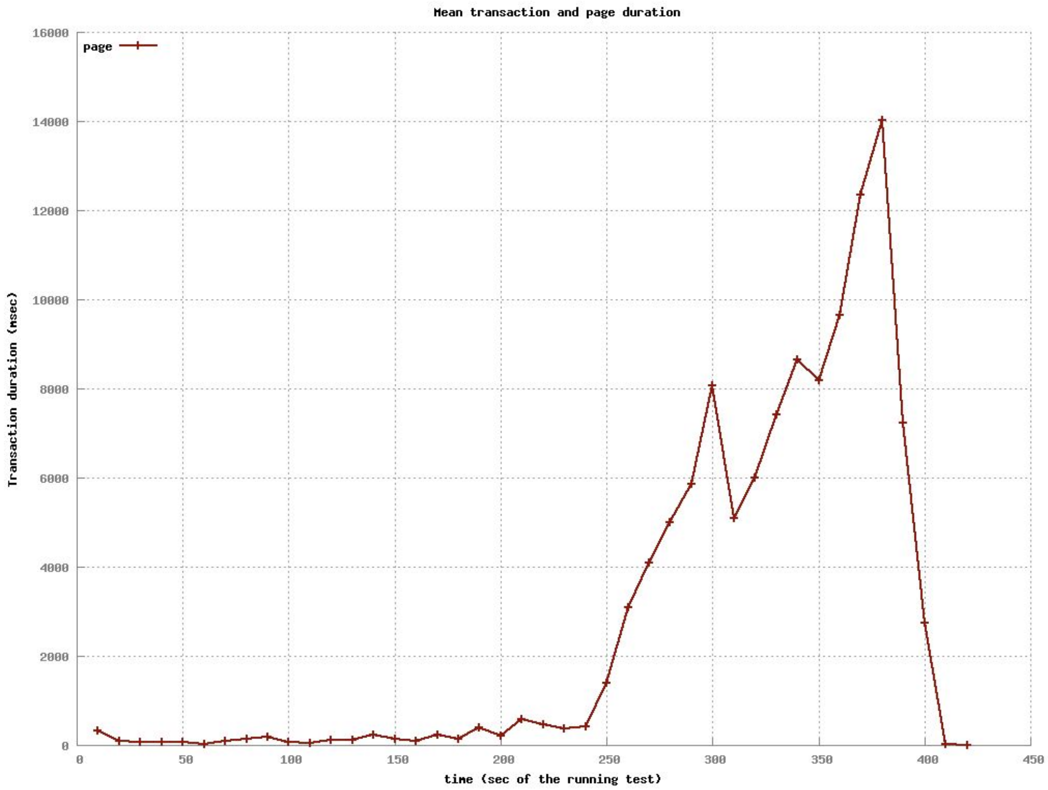
What do you think we will see?

The Demo App - 1/4



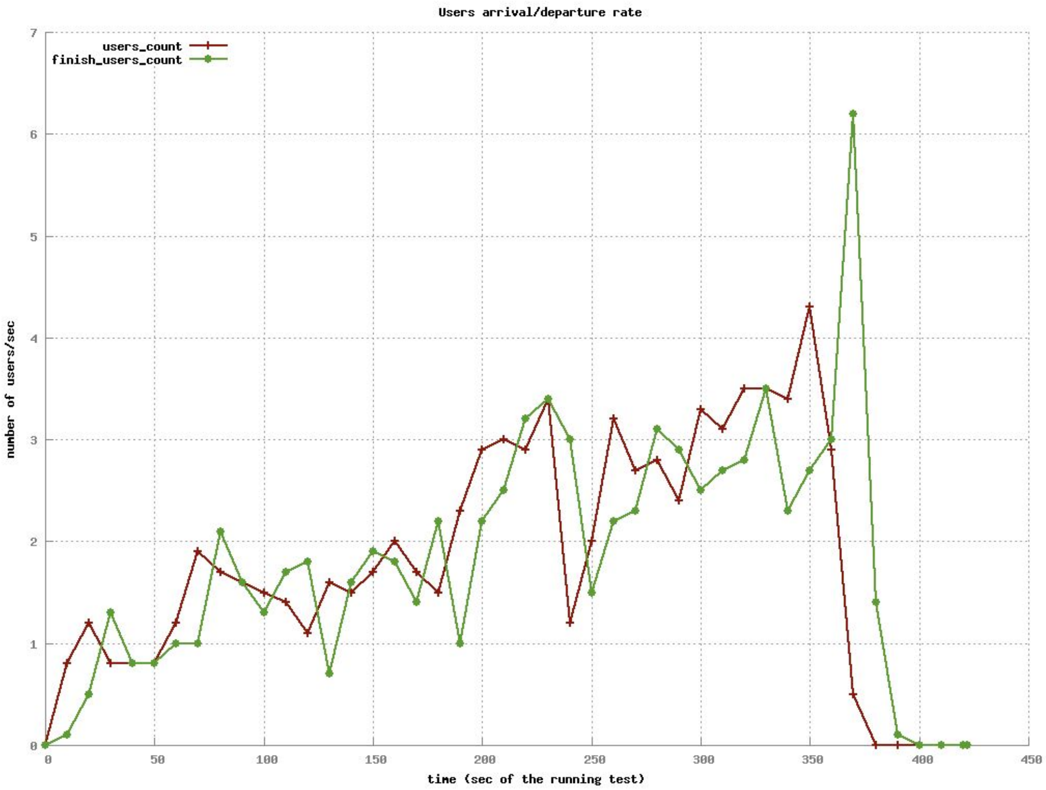
4 threads deliver much better performance than 4 processes

The Demo App - 1/4



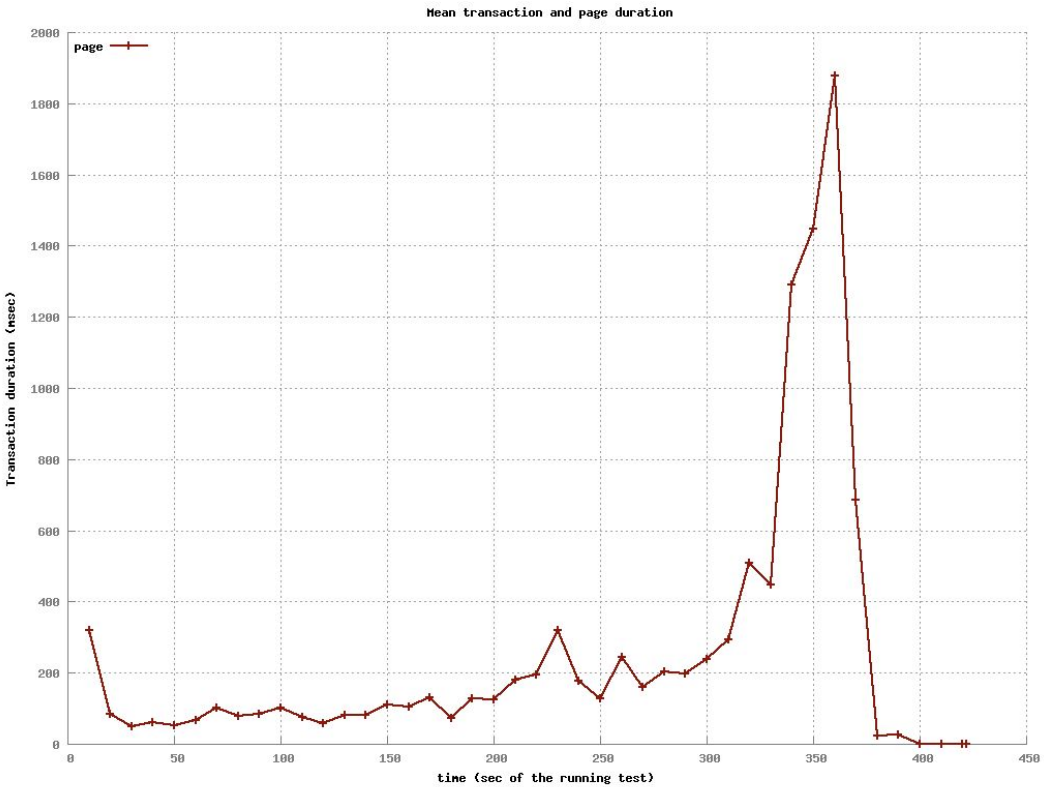
4 threads deliver much better performance than 4 processes

The Demo App - 1/32



32 threads deliver much better performance

The Demo App - 1/32



32 threads deliver much better performance

Application Servers

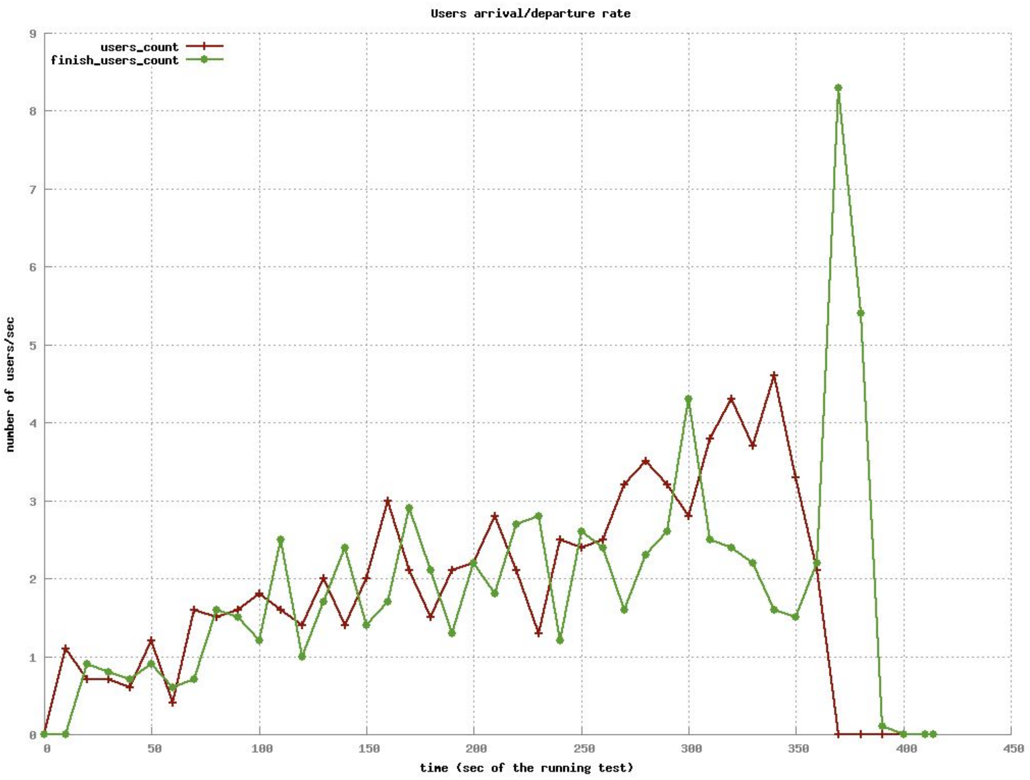
If we are using the standard C Ruby interpreter (Matz's Ruby Interpreter), then we have a Global Interpreter Lock to deal with.

- Only one thread of control can be executing in a given Ruby process at a given time
- JRuby has no GIL

The previous slides have all been using JRuby, to avoid the GIL.

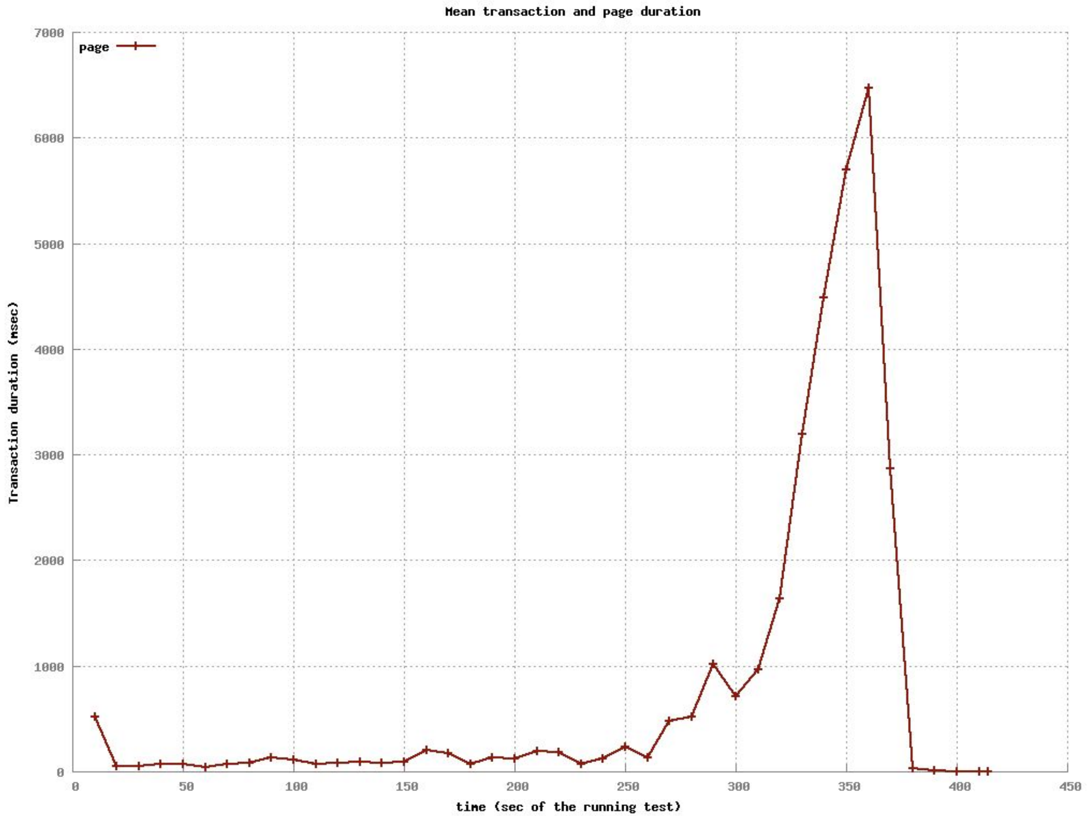
What do you expect will happen if we use MRI with threads?

The Demo App - MRI 1/64



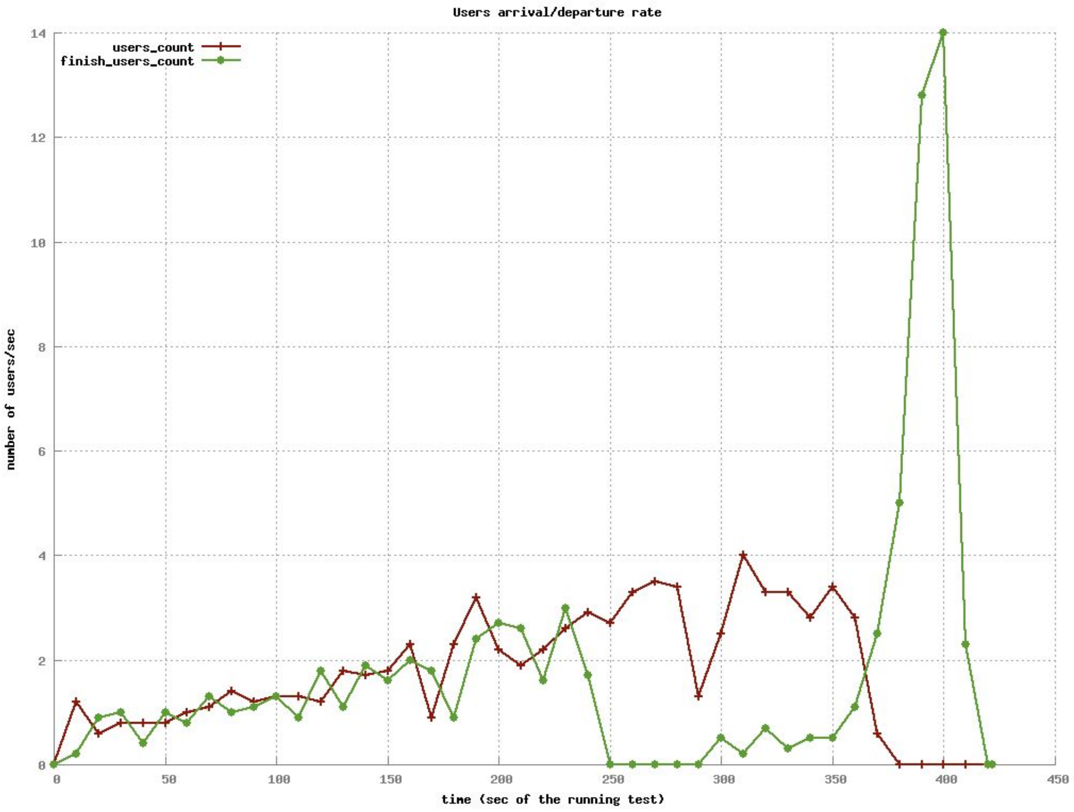
Performs surprisingly well!

The Demo App - 1/64 MRI



Why doesn’t the GIL get in the way?

The Demo App - 1/64 MRI - mysql



It turns out the mysql2 gem is designed with the GIL in mind.

Application Servers

In this class you will have some options to choose from when it comes to application servers:

- Webrick
- Puma
  - JRuby
  - MRI
- Phusion Passenger
- Advanced students can install others
  - Unicorn, Thin, Mongrel (?)

Application Servers

Phusion Passenger

- A passenger module is added to Apache or nginx
- The code running inside the HTTP server knows what it is load balancing and actively controls the size of the pool.
- Two advantages:
    - Simple mechanism to increase/decrease the pool
    - Processes can be forked after ruby/rails is loaded.
        - Why is this good?

Application Servers

Puma

- Originally designed for Rubinius
    - Works well with JRuby
- If we move away from the GIL, we can avoid process-based parallelism and choose threads instead
- Common setups involve a load balancer in front of multiple Ruby processes, each with multiple threads.
    - We can tune the isolation vs. memory footprint

Application Servers

Making your code thread safe isn't always obvious.

```
class PaymentSystem
    def self.transact(amount)
        transact(amount, self.credentials)
    end
    def self.credentials
        @credentials ||= get_credentials()
    end
end
PaymentSystem.transact(100)
```

Application Servers

Making your code thread safe isn't always obvious.

```
app/models/order.rb

class Order
    belongs_to :user, klass: User
end
```

Motivation

After today, you should understand some of the tradeoffs below

Scalable Internet Services Templates

Single Instance Templates

Both the app server, and database are located on a single EC2 instance.

- **NGINX + Passenger** (Recommended for regular testing):
  NGINX handles requests to port 80 and passes connections to instances of the app through Passenger. Multiple concurrent connections are supported.
  https://scalableinternetservices.s3.amazonaws.com/SinglePassenger.json
- **Puma:**
  Puma allows both thread-based and process-based concurrency.
  https://scalableinternetservices.s3.amazonaws.com/SinglePuma.json
- **WEBrick** (Use only for slow-performance testing):
  WEBrick handles requests to port 80 directly, permitting only a single connection at a time.
  https://scalableinternetservices.s3.amazonaws.com/SingleWEBrick.json

Specify Parameters

Specify values or use the default values for the parameters that are associated with your AWS CloudFormation template.

Parameters

AppInstanceType	m3.medium	The AppServer instance type.
Branch	master	The git branch to deploy.
ProcessParallelism	2	The number of worker processes.
RubyVM	MRI	The number of threads within each worker processes.
TeamName	demo	Your team name.
ThreadParallelism	4	The number of threads within each worker processes.

For Next Time...

- **Find a group!**
- By Thursday finish up to chapter 9 in Hartl's Rails book
- Finish Ruby Code Academy