Scaling PHP Applications

By Steve Corona

Scaling PHP

Steve Corona

This book is for sale at http://leanpub.com/scalingphp

This version was published on 2013-02-18

Tweet This Book!

Please help Steve Corona by spreading the word about this book on Twitter!

The suggested hashtag for this book is #scalingphp.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#scalingphp

Contents

Preface **1**

How this book came to be 1

How this book is designed 1

Who should read this book? 2

What you need for this book 2

Getting Started **3**

What's wrong with LAMP? 3

The Scalable Stack 3

DNS **7**

DNS Load Distribution 8

DNS Resolution 10

Load Balancer **14**

HAProxy 14

Choosing the best hardware 20

Automatic failover with `keepalived` 21

Tuning linux for a network heavy load 23

Issues at scale 24

App Server **26**

Choosing the right version of PHP 26

Nginx and PHP-FPM 28

Choosing the best hardware for your application server 34

Using a PHP Opcode cache 34

Tuning Linux for PHP 37

Scaling session handling 40

Database Server 45

Getting NoSQL Performance out of MySQL . . . 45
Dealing with libmysql's inability to set timeouts . . . 48
Tuning your MySQL Configuration . . . 53
Tuning Linux for an intensive database . . . 64
Load balancing MySQL Slaves . . . 69
Accepting the Master as a SPOF . . . 74
Understanding issues with NUMA . . . 77
Choosing the best hardware . . . 80
Online Schema Changes . . . 83
Further Topics . . . 85

Cache Server 86

Choosing between Redis, Memcached, and APC . . . 86
Redis Commands . . . 89
The importance of Atomic Operations . . . 93
Performance Limitations of Redis . . . 94
Installing Redis from Dotdeb . . . 94
Installing the `phpredis` C extension . . . 95
Tuning Redis for Performance . . . 96
Scaling Redis to Multiple Servers . . . 99
"The Dogpile" . . . 101
Russian Doll Caching . . . 104
Redis Bitmaps . . . 108
Redis Notification Feeds . . . 110

Worker Server 112

Why use Queues? . . . 112
Getting started with Resque and php-resque . . . 114
Writing a Resque Job . . . 118
Alternatives to Resque . . . 122
Things you should put in a queue . . . 124

Code **125**

Scaling your code 125

Capistrano for code deployment 129

Live Debugging PHP with strace 132

Parting Advice **134**

Sending Feedback and Testimonials 134

In too deep? Need help ASAP? 134

About the Author 135

Sponsors **136**

Preface

How this book came to be

In 2009, I met Noah Everett, founder and CEO of Twitpic through a job posting he advertised on Twitter. Noah is a smart, self-taught programmer, quite literally thrown into scaling when Twitpic blew up. Subsequently, my job interview was less of an interview and more me introducing Noah to memcache and how it could be setup within the next hour—up till that point the site crashed regularly. An hour of free consulting for a life changing job opportunity? Sounds good to me.

When I started at Twitpic, the infrastructure was a mix of different servers and FreeBSD versions, backed up by a steaming pile of PHP-inlined HTML. Queries were naive and uncached. Remember… Twitpic was built in a weekend—it was a side project for Noah to share pictures with a couple of his friends.

In the beginning, Noah and I had to schedule our lives around our laptops. Going to the grocery store? Bring the laptop. Headed out on a date? Don't forget the laptop. Twitpic was like a jealous girlfriend, only in this case, one waiting to crash at a moments notice. I never knew when I'd have to pop into a Starbucks to restart Apache.

Mostly out of necessity, and because no one enjoys cuddling with their laptop at night, we learned. We scaled. We improved our infrastructure. By no means is our setup perfect, but we've learned an incredible amount, and after three years I think we've built a pretty solid infrastructure. What happens when servers go down in the middle of the night today? Nothing. I sleep peacefully through the night, because we've built a horizontal, replicated system that is designed for failure.

How this book is designed

This book, although PHP centric, can be applied to any language. Most of the tricks, techniques, and design patterns will apply cleanly to Ruby, Python, Java or really anything besides Visual Basic (ok, maybe even VB too).

This book reads like a cookbook, or maybe even a choose your own adventure story—you don't need to read it in order, or cover to cover. Jump around to different chapters or sections, digging into the ones that pique your interest the most.

I've laid out the chapters so they move from the outside of the stack, level-by-level, subsequently getting deeper until we hit the code.

Who should read this book?

This book is designed for startups, entrepreneurs and smart people that love to hustle and build things the right way.

You should know PHP, your current stack and your way around Linux. My readers are smart and intelligent people that don't need their hands held. I explain complicated topics and provide thorough examples, but this book is NOT simply regurgitated documentation—it includes real world scenarios and use-cases.

What you need for this book

At a minimum, you need a PHP application that you want to learn how to scale. And you'll probably see the most benefit if you have a Linux server that you can deploy to. Don't have one? I really like Rackspace Cloud[1] or Amazon EC2[2] for testing because you can setup and tear down servers quickly, and launch multi-server test setups cheaply.

[1]http://www.rackspace.com/cloud/public/servers/
[2]http://aws.amazon.com/ec2/

Getting Started

What’s wrong with LAMP?

LAMP (Linux, Apache, MySQL, PHP) is the most popular web development stack in the world. It’s robust, reliable and everyone knows how to use it. So, what’s wrong with LAMP? Nothing. You can go really far on a single server with the default configurations. But what happens when you start to really push the envelope? When you have so much traffic or load that your server is running at full capacity?

You’ll notice tearing at the seams, and in a pretty consistent fashion too. MySQL is always the first to go—I/O bound most of the time. Next up, Apache. Loading the entire PHP interpreter for each HTTP request isn’t cheap, and Apache’s memory footprint will prove it. If you haven’t crashed yet, Linux itself will start to give up on you—all of those sane defaults that ship with your distribution just aren’t designed for scale.

What can we possibly do to improve on this tried-and-true model? Well, the easiest thing is to get better hardware (scale vertically) and split the components up (scale horizontally). This will get you a little further, but there is a better way. Scale intelligently. Optimize, swapping pieces of your stack for better software, customize your defaults and build a stack that’s reliable and fault tolerant. You want to spend your time building an amazing product, not babysitting servers.

The Scalable Stack

After lots of trial and error, I’ve found what I think is a generic, scalable stack. Let’s call it **LHNMPRR**… nothing is going to be as catchy as LAMP!

Linux

We still have Old Reliable, but we’re going to tune the hell out of it. This book assumes the latest version of Ubuntu Server 12.04, but most recent distributions of Linux should work equally as well. In some places you may need to substitute `apt-get` with your own package manager, but the kernel tweaks and overall concepts should apply cleanly to RHEL, Debian, and CentOS. I’ll include kernel and software versions where applicable to help avoid any confusion.

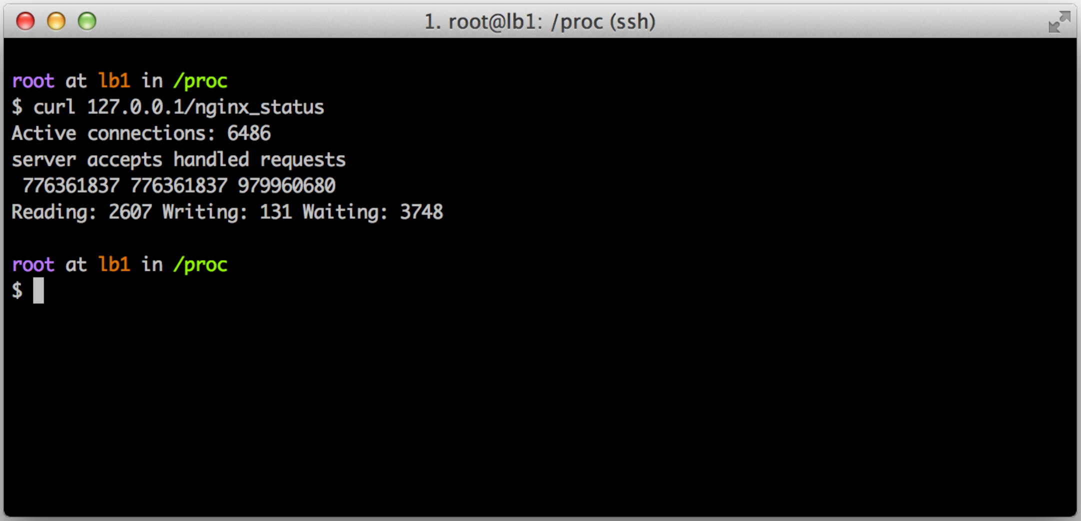
HAProxy

We’ve dumped Apache and split its job up. HAProxy acts as our load balancer—it’s a great piece of software. Many people use nginx as a load balancer, but I’ve found that HAProxy is a better choice for the job. Reasons why will be discussed in-depth in Chapter 3.

nginx

Nginx is an incredible piece of software[3]. It brings the heat without any bloat and does webserving extremely well. In addition to having an incredibly low memory and CPU footprint, it is extremely reliable and can take an ample amount of abuse without complaining.

One of our busiest nginx servers at Twitpic handles well over 6,000 connections per second while using only 80MB of memory.



```
root at lb1 in /proc
$ curl 127.0.0.1/nginx_status
Active connections: 6486
server accepts handled requests
 776361837 776361837 979960680
Reading: 2607 Writing: 131 Waiting: 3748

root at lb1 in /proc
$
```

An example of a server with over 6000 active connections

PHP 5.3 / PHP-FPM

There are several ways to serve PHP applications: mod_php, cgi, lighttpd fastcgi??. None of these solutions come close to PHP-FPM, a FastCGI Process Manager written by the nginx team that's been bundled with PHP since 5.3. What makes PHP-FPM so awesome? Well, in addition to being rock-solid, it's extremely tunable, provides real-time stats and logs slow requests so you can track and analyze slow portions of your codebase.

[3]Apache is great, too. It's extremely popular and well supported, but Apache + mod_php is not the best tool for the job when you're dealing with high volume PHP apps. It's memory intensive and doesn't offer full tuning capabilities.

MySQL

Most folks coming from a LAMP stack are going to be pretty familiar with MySQL, and I will cover it pretty extensively. For instance, in Chapter 5 you'll learn how you can get NoSQL performance out of MySQL. That being said, this book is database agnostic, and most of the tips can be similarly applied to any database. There are many great databases to choose from: Postgres, MongoDB, Cassandra, and Riak to name a few. Picking the correct one for your use case is outside the scope of this book.

Redis

Redis[4] can be used as a standalone database, but its primary strength is as a datatype storage system. Think of it as memcache on steroids. You can use memcache (we still do), but Redis performance is just as good or better in most scenarios. It is persistent to disk (so you don't lose your cache in the case of a crash, which can be catastrophic if your infrastructure can't deal with 100% cache misses), and is updated extremely frequently by an opinionated, vocal and smart developer, @antirez.

Resque

Doing work in the background is one of the principal concepts of scaling. Resque[5] is one of the best worker/job platforms available. It uses Redis as a backend so it's inherently fast and scalable, includes a beautiful frontend to give you full visibility into the job queue, and is designed to fail gracefully. Later we'll talk about using PHP-Resque, along with some patches I've provided, to build elegant background workers.

What about the AWS Stack?

AWS is awesome. Twitpic is a huge AWS customer—over 1PB of our images are stored on Amazon S3 and served using CloudFront. Building that kind of infrastructure on our own would be extremely expensive and involve hiring an entire team to manage it.

That being said, if you are running servers on EC2 full-time and scaling by adding more, you're doing it wrong. It's like using the back of your iPhone to hammer a nail into the wall. It works... it will get the job done, but it's going to cost you much more than using the right tools.

Take the plunge and move to bare-metal hardware. It's cheaper, provides more consistent performance, and gives you access to a bounty of customized hardware (SSDs, RAID cards, SAS drives, 512GB of memory). We love SoftLayer, but there are plenty of reliable providers.

Note that I am not advocating against EC2, I am advocating that you use it the way that it was intended to be used: for on-demand capacity. Sudden increase in traffic? Spin up some EC2 instances

[4]http://redis.io/

[5]https://github.com/defunkt/resque

and handle the burst effortlessly. Using Amazon Virtual Private Cloud, your EC2 instances can communicate with your bare-metal servers without being exposed to the public internet.

DNS

The first layer that we are going to unravel is DNS. DNS? What!? I thought this was a book on PHP? DNS is one of those things that we don't really think about until it's too late, because when it fails, it fails in the worst ways.

Don't believe me? In 2009 Twitter's DNS was hijacked and redirected users to a hacker's website for an hour. That same year, SoftLayer's DNS system was hit with a massive DDoS attack and took down their DNS servers for more than six-hours. As a big SoftLayer customer, we dealt with this firsthand because (at the time) we also used their DNS servers.

The problem with DNS downtime is that it provides the worst user experience possible—users receive a generic error, page timeouts, and have no way to contact you. It's as if you don't exist anymore, and most users won't understand (or likely care) why.

As recently as September 2012, GoDaddy's DNS servers were attacked and became unreachable for over 24-hours. The worst part? Their site was down too, so you couldn't move your DNS servers until their website came back up (24-hours later).

Too many companies are using their domain registrar or hosting provider's DNS configuration—that is WRONG! Want to know how many of the top 1000 sites use GoDaddy's DNS? None of them.

So, should I run my own DNS server?

It's certainly an option, but I don't recommend it. Hosting DNS "the right way" involves having many geographically dispersed servers and an Anycast network. DDoS attacks on DNS are extremely easy to launch and if you half-ass it, your DNS servers will be the Achilles' heel to your infrastructure. You could have the best NoSQL database in the world but it won't matter if people can't resolve your domain.

Almost all of the largest websites use an external DNS provider. That speaks volumes as far as I'm concerned—the easiest way to learn how to do something is to imitate those that are successful.

- Reddit: Akamai[6]
- Twitter: Dynect[7]
- Amazon: UltraDNS[8] and Dynect
- LinkedIn: UltraDNS

You should plan to pay for a well known, robust DNS SaaS. At Twitpic, we use Dynect Managed DNS[9]. It has paid for itself—we've had no downtime related to DNS outages since switching. Make

[6]http://www.akamai.com/html/solutions/enhanced_dns.html
[7]http://dyn.com
[8]http://ultradns.com
[9]http://dyn.com/dns/

sure you choose a DNS provider that has a presence close to your target audience, too, especially if you have a large international userbase.

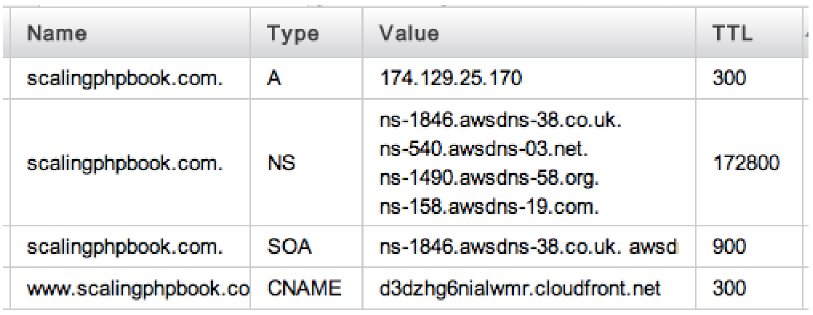
Here's what you should look for in a DNS provider:

- Geographically dispersed servers
- Anycast Network
- Large IP Space (to handle network failures)

What is Anycast?

Anycast is a networking technique that allows multiple routers to advertise the same IP prefix—it routes your clients to the "closest" and "best" servers for their location. Think of it as load balancing at the network level.

In addition to the providers listed above, Amazon Route53[10] is a popular option that is incredibly cheap, offers a 100% SLA, and has a very user-friendly web interface. Since it's pay-as-you-go, you can easily get started and adjust for exact usage needs as you grow.



Name	Type	Value	TTL
scalingphpbook.com.	A	174.129.25.170	300
scalingphpbook.com.	NS	ns-1846.awsdns-38.co.uk. ns-540.awsdns-03.net. ns-1490.awsdns-58.org. ns-158.awsdns-19.com.	172800
scalingphpbook.com.	SOA	ns-1846.awsdns-38.co.uk. awsd	900
www.scalingphpbook.co	CNAME	d3dzhg6nialwmr.cloudfront.net	300

DNS Load Distribution

Besides scaling DNS, we can use DNS to help us scale, too. The HAProxy load balancer is integral to our scalable stack—but what happens when it loses network connection, becomes overloaded, or just plain crashes? Well it becomes a single point of failure, which equals downtime—it's a question of when, not if it will happen.

[10]http://aws.amazon.com/route53/

Traditional DNS load balancing involves creating multiple A records for a single host and passing all of them back to the client, letting the client decide which IP address to use. It looks something like this.

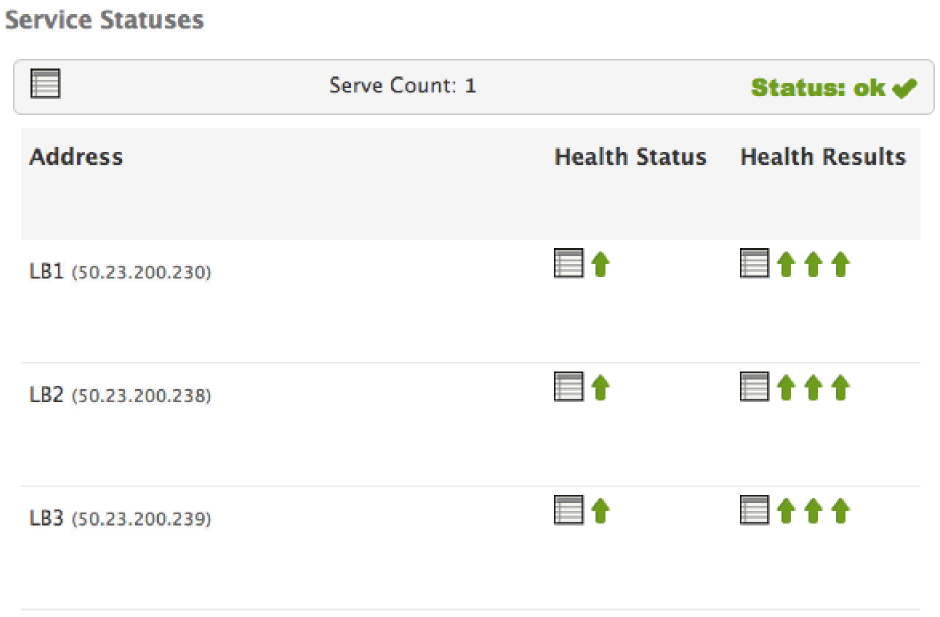
```
> dig A example.com
; <<>> DiG 9.7.3-P3 <<>> A example.com

;; QUESTION SECTION:
;example.com.                        IN        A

;; ANSWER SECTION:
example.com.                287        IN        A        208.0.113.36
example.com.                287        IN        A        208.0.113.34
example.com.                287        IN        A        208.0.113.38
example.com.                287        IN        A        208.0.113.37
example.com.                287        IN        A        208.0.113.35
```

There are a few drawbacks to this, however. First of all, less-intelligent DNS clients will always use the first IP address presented, no matter what. Some DNS providers (Dynect and Route53, for example) overcome this by using a round-robin approach whereby they change the order of the IPs returned everytime the record is requested, helping to distribute the load in a more linear fashion.

Another drawback is that round-robin won't prevent against failure, it simply mitigates it. If one of your servers crashes, the unresponsive server's IP is still in the DNS response. There are two solutions that work together to solve this.

1. Use a smart DNS provider that can perform health checks. Dynect offers this feature and it's possible to implement it yourself on Route53 using their API. If a load balancer stops responding or becomes unhealthy, it gets removed from the IP pool (and readded once it becomes healthy again).
2. Even if you remove a server's IP from the DNS pool, users that have the bad IP cached will still experience downtime until the record's TTL expires. Anywhere from 60-300s is a recommended TTL value, which is acceptable, but nowhere near ideal. We'll talk about how servers can "steal" IPs from unhealthy peers using `keepalived` in Chapter 4.

Dynect has a very intuitive interface for load balancing and performing health checks on your hosts:



Service Statuses

Serve Count: 1 Status: ok ✓

Address	Health Status	Health Results
LB1 (50.23.200.230)	▤ ↑	▤ ↑ ↑ ↑
LB2 (50.23.200.238)	▤ ↑	▤ ↑ ↑ ↑
LB3 (50.23.200.239)	▤ ↑	▤ ↑ ↑ ↑

DNS Resolution

The last, and very often overlooked, part of scaling DNS is internal domain resolution. An example of this is when a PHP application calls an external API and has to resolve the domain of the API host. Let's say the application is using the Twitter API—every time you post something to Twitter, PHP has to look up and determine the IP of `api.twitter.com`.

PHP accomplishes this by using the libc system call `gethostbyname()`, which uses nameservers set in `/etc/resolv.conf` to look up the IP address. Usually this is going to be set to a public DNS resolver (like `8.8.8.8`) or a local resolver hosted by your datacenter.

So, what's wrong with this setup? Well, two things:

1. It's another server that you don't control. What happens when it's down? Slow? They block you? The lowest timeout allowed in `/etc/resolv.conf` is one second (and the default is FIVE!), which is too slow for a high-volume website and can cause domino-effect failure at scale.
2. Most Linux distributions don't provide a DNS cache by default and that adds extra network latency to every single DNS lookup that your application has to make.

The solution is to run a DNS cache daemon like `nscd`, `dnsmasq`, or `bind`. I won't cover BIND because it's overkill to run it simply as a cache, but I will talk about `nscd` and `dnsmasq`, which work in slightly different ways.

nscd

`nscd` (nameserver cache daemon) is the simplest solution to setting up your own internal DNS cache. Whether or not it's installed by default is dependent on your Linux distro (it's not on Ubuntu or Debian). It's easy to install and needs zero-configuration. The main difference between `nscd` and `dnsmasq` is that `nscd` runs locally on each system while `dnsmasq` is exposed as a network service and can provide a shared DNS cache for multiple servers.

```
> apt-get install nscd
> service nscd start
```

Pros

- Extremely easy to setup
- Zero configuration

Cons

- Runs locally only, so each server needs to have it's own install

dnsmasq

`dnsmasq` is a lightweight DNS cache server that provides nearly the same feature-set as `nscd`, but as a network service.

You setup `dnsmasq` on a server, let's call it `198.51.100.10`, and set `198.51.100.10` as the nameserver for all of your other servers. `dnsmasq` will still go out onto the internet to look up DNS queries for the first time, but it will cache the result in memory for subsequent requests, speeding up DNS resolution and allowing you to gracefully deal with failure.

Additionally, you can use `dnsmasq` as a lightweight internal DNS server with the `addn-hosts` configuration option, allowing you to use local hostnames without having to hardcode IP addresses in your code (i.e, `$memcache->connect('cache01.example')` instead of `$memcache->connect('198.51.100.15')`).

Assuming a network setup based on the table below, here is how we'd setup `dnsmasq` and point our servers to it:

Hosts	IP
dnsmasq server	192.51.100.10
server01.example	192.51.100.16
server02.example	192.51.100.17

On your dnsmasq server:

```
> apt-get install dnsmasq
> vi /etc/dnsmasq.conf

cache-size=1000
listen-address=198.51.100.10
local-ttl=60
no-dhcp-interface=eth0
no-dhcp-interface=eth1
addn-hosts=/etc/dnsmasq.hosts

> vi /etc/dnsmasq.hosts

198.51.100.16 server01.example
198.51.100.17 server02.example

> service dnsmasq restart
```

`local-ttl` sets the time-to-live for any hosts you define in /etc/hosts or /etc/dnsmasq.hosts

`cache-size` defines the size of the DNS cache.

`no-dhcp-interface` disables all services provided by dnsmasq except for dns. Without this, dnsmasq will provide dhcp and tftp as well, which you do not want in most scenarios.

On EC2, after restarting `dnsmasq` you may need to add the following line to `/etc/hosts`:

```
127.0.0.1 ip-10-x-x-x
```

And then on your other hosts:

```
> vi /etc/resolv.conf

nameserver 198.51.100.10
options rotate timeout:1
```

The `rotate` option tells linux to rotate through the nameservers instead of always using the first one. This is useful if you have more than one nameserver and want to distribute the load.

The `timeout:1` option tells linux that it should try the next nameserver if it takes longer than 1-second to respond. You can set it to any integer between 1 and 30. The default is 5-seconds and it's capped at 30-seconds. Unfortunately, the minimum value is 1-second, it would be beneficial to set the timeout in milliseconds.

When do I need this?

There is virtually no negative impact to implementing a caching nameserver early; however, it does add another service to monitor—it's not a completely free optimization. Essentially, if any of your pages require a DNS resolution, you should consider implementing a DNS cache early.

Want to know how many uncached DNS requests your server is currently sending? With `tcpdump` you can see all of the DNS requests going out over the network.

```
> apt-get install tcpdump
> tcpdump -nnp dst port 53

09:52 IP 198.51.100.16 > 198.51.100.10.53: A? graph.facebook.com.
09:52 IP 198.51.100.16 > 198.51.100.10.53: AAAA? graph.facebook.com.
09:52 IP 198.51.100.16 > 198.51.100.10.53: A? api.twitter.com.
09:52 IP 198.51.100.16 > 198.51.100.10.53: AAAA? api.twitter.com.
```

The `-nn` option ensures tcpdump itself does not resolve IP Addresses or protocols to names.

The `-p` option disables promiscuous mode, to minimize adverse impact to running services.

The `dst port 53` only shows DNS requests sent (for brevity), to see how long the application may have blocked waiting for a response (i.e. to see the request along with the response), exclude the 'dst' portion.

The above example shows how a single HTTP GET could cause four DNS requests to be sent to 3rd party APIs that you may be using. Using a caching DNS resolver, such as `dnsmasq` or `nscd`, helps reduce the blocking period and possible cascading failures.