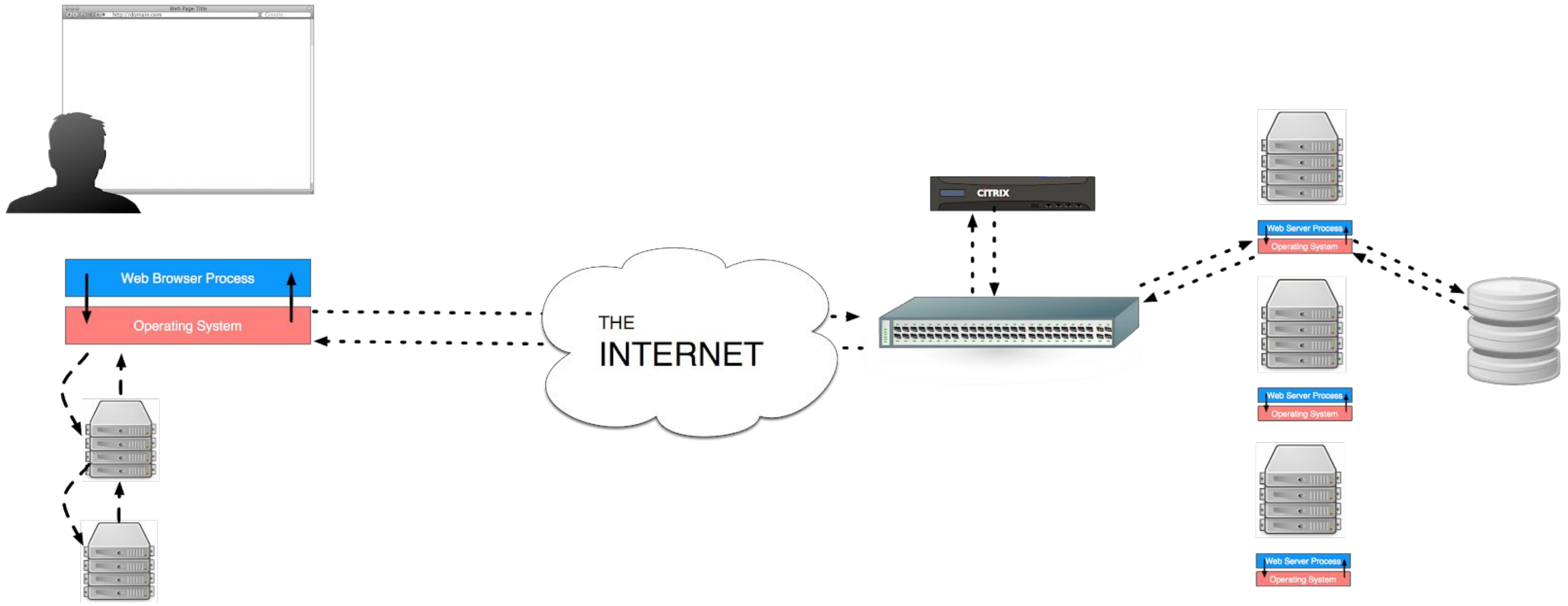
CS188
Scalable Internet Services

John Rothfels, 11/3/20

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

Motivation

Let’s say we are considering a significant change to our web application.

We think it may improve scalability, but our intuition might be wrong.

⁉️ **How should we go about testing this?**

Motivation

💡 Let’s deploy the system, send actual requests to it, and watch how it responds!

Motivation

~~💡~~ **Let's deploy the system, send actual requests to it, and watch how it responds!**

…we can test against a development server (localhost) first. 😎

Motivation

⁉️ **What should we observe while testing?**

Motivation

⁉️ **What should we observe while testing?**

- Response times (latency)
- Requests per second (RPS)
- Error rates
- Can virtual users able to complete their tasks?
- When (at what load) do problems start happening?

Motivation

⁉️ **What things should we take into account while programming load tests?**

Motivation

⁉️ **What things should we take into account while programming load tests?**

- We want a mixture of reads and writes
- Not all users have the same habits
- We want to be able to respond to application output
- Automation: should it run during CI/CD?

Motivation

⁉️ **What things should we take into account while programming load tests?**

- We want a mixture of reads and writes
- Not all users have the same habits
- We want to be able to respond to application output
- Automation: should it run during CI/CD?
- The load test runner has to be scalable too! 😅

Motivation

There are many (open source) software tools that help you create load tests.

Some have **high performance**.

Some allow **distributed deployment**.

Some have **rich feature sets**.

- HTTP/2, WebSockets, SOAP, LDAP, MySQL, GUI, ...

Some are **scriptable** (with a programming language).

apachebench

The old kid on the block. Been around since the late 90’s as part of the toolset for Apache server. Easy to use, quick and dirty. No scriptability.

```
ab -n 1000 -c 100 http://127.0.0.1:3000/app/
```

JMeter

This behemoth also comes from Apache software foundation. You use a (terrible) GUI to define your load tests. Limited scriptability.

Tsung

Old and stable. Mix of good performance with decent feature set. Limited scriptability. You define your load test in XML. 😭

```
tsung -f test.xml -k start
```

Then go to http://localhost:8091.

Tsung

```
<?xml version="1.0"?>
<tsung loglevel="notice" version="1.0">
  <clients>
    <client host="localhost" use_controller_vm="true" maxusers="15000"/>
  </clients>
  <servers>
    <server host="www.google.com" port="80" type="tcp"/>
  </servers>
  <load>
    <arrivalphase phase="1" duration="10" unit="second">
      <users arrivalrate="1" unit="second"/>
    </arrivalphase>
  </load>
```

Tsung

```
<?xml version="1.0"?> // xml boilerplate
<tsung loglevel="notice" version="1.0">
  <clients>
    <client host="localhost" use_controller_vm="true" maxusers="15000"/>
  </clients>
  <servers>
    <server host="www.google.com" port="80" type="tcp"/>
  </servers>
  <load>
    <arrivalphase phase="1" duration="10" unit="second">
      <users arrivalrate="1" unit="second"/>
    </arrivalphase>
  </load>
```

Tsung

```
<?xml version="1.0"?>
<tsung loglevel="notice" version="1.0">
  // client-side configuration
  <clients>
    <client host="localhost" use_controller_vm="true" maxusers="15000"/>
  </clients>
  <servers>
    <server host="www.google.com" port="80" type="tcp"/>
  </servers>
  <load>
    <arrivalphase phase="1" duration="10" unit="second">
      <users arrivalrate="1" unit="second"/>
    </arrivalphase>
  </load>
```

Tsung

```
<?xml version="1.0"?>
<tsung loglevel="notice" version="1.0">
  <clients>
    <client host="localhost" use_controller_vm="true" maxusers="15000"/>
  </clients>
  // server config (where are we directing load)
  <servers>
    <server host="www.google.com" port="80" type="tcp"/>
  </servers>
  <load>
    <arrivalphase phase="1" duration="10" unit="second">
      <users arrivalrate="1" unit="second"/>
    </arrivalphase>
  </load>
```

Tsung

```
<?xml version="1.0"?>
<tsung loglevel="notice" version="1.0">
  <clients>
    <client host="localhost" use_controller_vm="true" maxusers="15000"/>
  </clients>
  <servers>
    <server host="www.google.com" port="80" type="tcp"/>
  </servers>
  // define how you want virtual users to arrive
  <load>
    <arrivalphase phase="1" duration="10" unit="second">
      <users arrivalrate="1" unit="second"/>
    </arrivalphase>
  </load>
```

Tsung

```
  // a section to set options (e.g. headers)
  <options>
    <option name="glocal_ack_timeout" value="2000"/>
    <option type="ts_http" name="user_agent">
      <user_agent probability="100">Mozilla/5.0 (Windows; U; Windows NT 5.2; fr-FR; rv:1.7.8) Gecko/20050511 Firefox/1.0.4</user_agent>
    </option>
  </options>
  <sessions>
    <session name="http-example" probability="100" type="ts_http">
      <request>
        <http url="/" version="1.1" method="GET"/>
      </request>
    </session>
  </sessions>
```

Tsung

```
  <options>
    <option name="glocal_ack_timeout" value="2000"/>
    <option type="ts_http" name="user_agent">
      <user_agent probability="100">Mozilla/5.0 (Windows; U; Windows NT 5.2; fr-FR; rv:1.7.8) Gecko/20050511
Firefox/1.0.4</user_agent>
    </option>
  </options>
  // the actual series of requests each user performs
  <sessions>
    <session name="http-example" probability="100" type="ts_http">
      <request>
        <http url="/" version="1.1" method="GET"/>
      </request>
    </session>
  </sessions>
```

Tsung

```
<load>
   <arrivalphase phase="1" duration="30" unit="second">
     <users arrivalrate="1" unit="second" />
   </arrivalphase>
   <arrivalphase phase="2" duration="30" unit="second">
     <users arrivalrate="2" unit="second" />
   </arrivalphase>
   <arrivalphase phase="3" duration="30" unit="second">
     <users arrivalrate="4" unit="second" />
   </arrivalphase>
  <arrivalphase phase="4" duration="30" unit="second">
    <users arrivalrate="8" unit="second" />
  </arrivalphase>
</load>
```

💡 Increasing the rate of requests over time lets you see more precisely when your server starts to fail!

Tsung

You can insert random realistic wait times in your simulations.

```
<!-- wait for up to 2 seconds, user is looking at posts -->
<thinktime value="2" random="true" />
```

Tsung

```
<!-- create a random number to make a unique community name -->
<setdynvars sourcetype="random_number" start="1000" end="9999999">
  <var name="random_community_name" />
</setdynvars>

<!-- post to /communities to create a community. -->
<request subst="true">
  <http
    url='/communities'
    version='1.1'
    method='POST'
    contents='community%5Bname%5D=community_name_%%_random_community_name%%&amp;commit=Create+Community'>
  </http>
</request>
```

Tsung

```
      <request subst="true">
        <dyn_variable name="created_submission_url" re="[Ll]ocation: (http://.*)\r"/>
        <dyn_variable name="created_submission_id" re="[Ll]ocation: http://.*/submissions/(.*)\r"/>
        <http
          url='/submissions'
          version='1.1'
          method='POST'

contents='submission%5Btitle%5D=link_%%_random_submission_name%%&amp;submission%5Burl%5D=http%3A%2F%2Fwww.article.com%2
F%%_random_submission_name%%&amp;submission%5Bcommunity_id%5D=%%_created_community_id%%&amp;commit=Create+Submission'>
        </http>
      </request>
```

At times you will need to read the output of one request into the next. Use dynamic variables for this.

Tsung

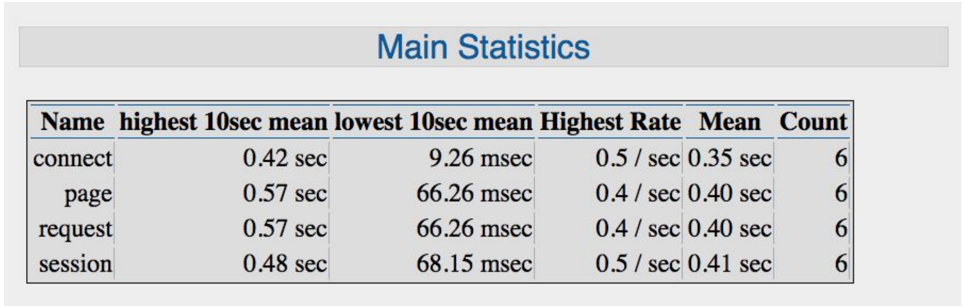
This is how you debug your dynamic variables. 😱

```
<!-- Uncomment the following to debug print your dynamic variables  -->
    <!--
    <setdynvars sourcetype="eval" code="fun( {Pid, DynVars} ) ->
      io:format([126, $p, 126, $n, 126, $n], [DynVars]),
      ok end.">
      <var name="dump" />
    </setdynvars>
    -->
```

Tsung

Tsung outputs a bunch of statistics and graphs.

- **request**: response time for each request
- **page**: response time for each set of requests (a page is a group of requests not separated by a thinktime)
- **connect**: duration of connection establishment
- **session**: duration of a user's session
- status codes
  - You want 2xx and 3xx
  - You don't want 4xx and 5xx

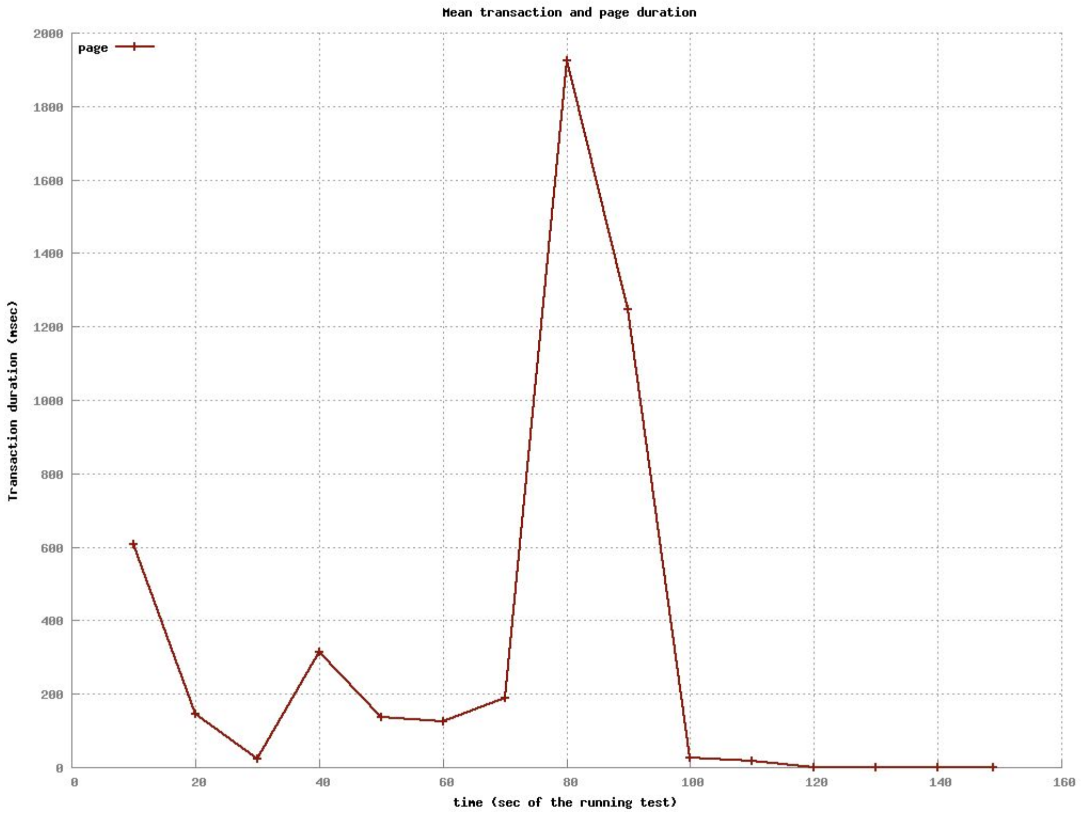


Main Statistics

Name	highest 10sec mean	lowest 10sec mean	Highest Rate	Mean	Count
connect	0.42 sec	9.26 msec	0.5 / sec	0.35 sec	6
page	0.57 sec	66.26 msec	0.4 / sec	0.40 sec	6
request	0.57 sec	66.26 msec	0.4 / sec	0.40 sec	6
session	0.48 sec	68.15 msec	0.5 / sec	0.41 sec	6

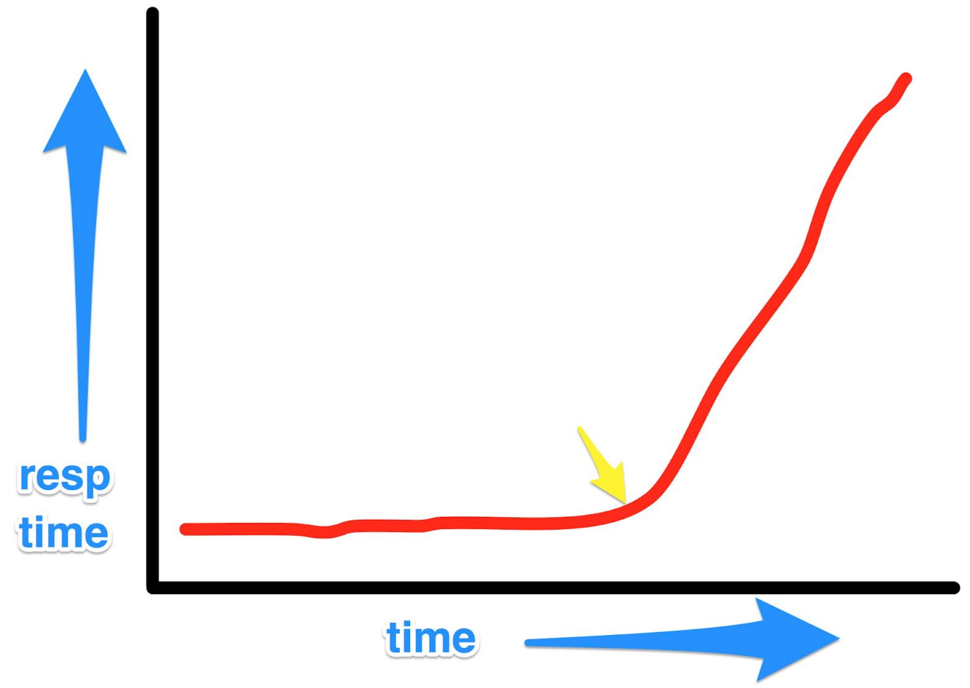
Tsung

Response time



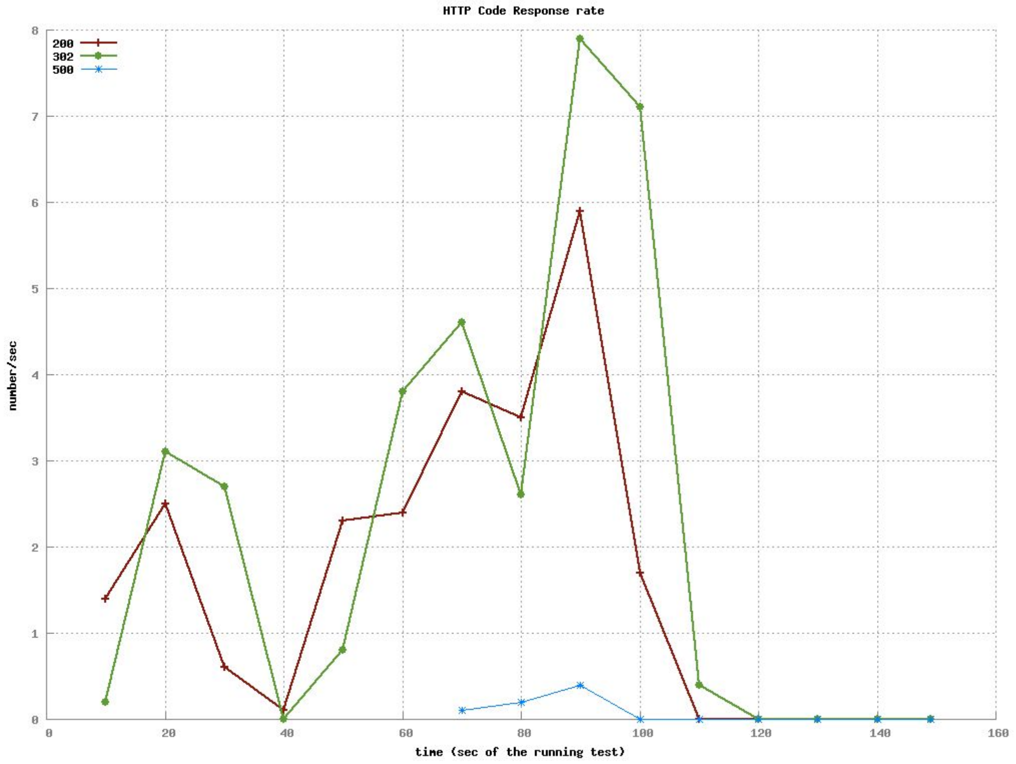
Tsung

The yellow arrow indicates where the system starts to fail.



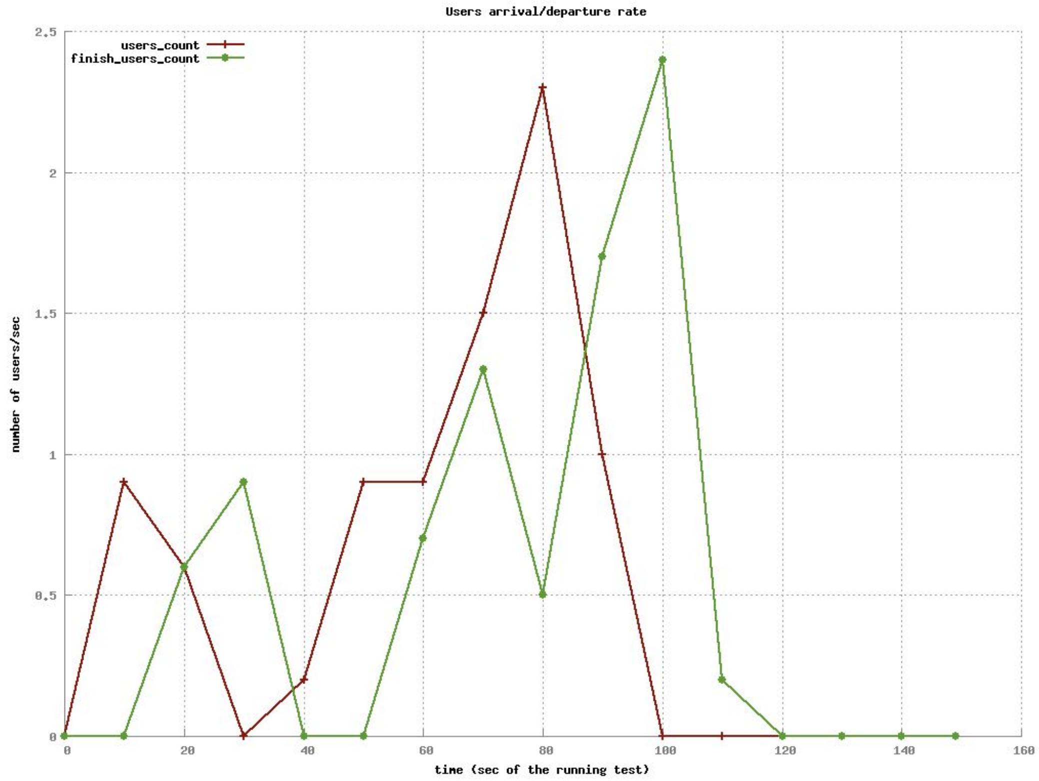
Tsung

Error rates



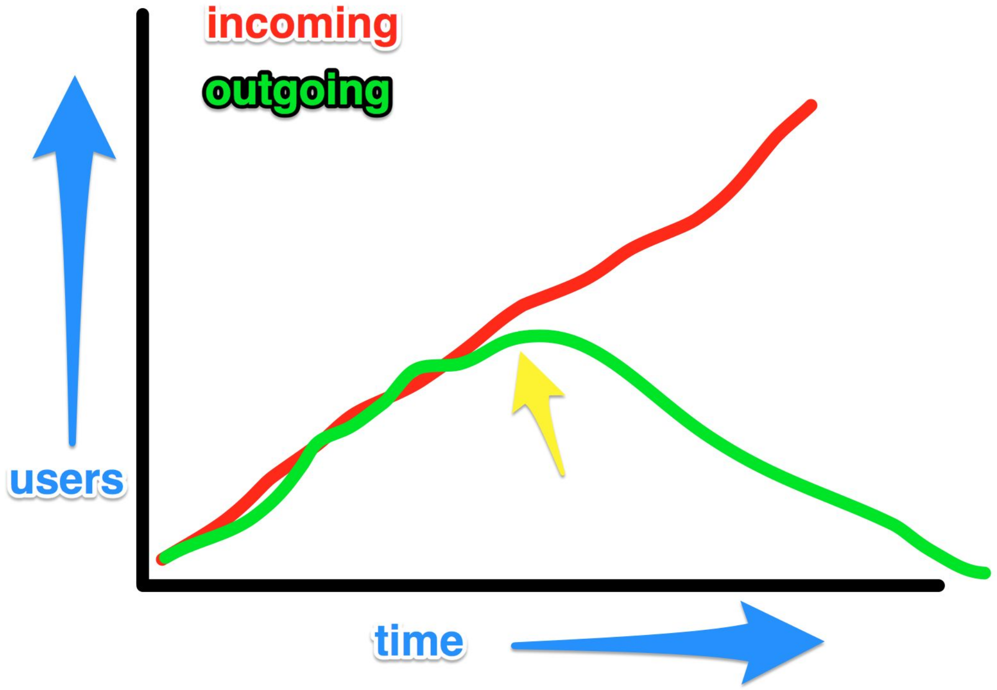
Tsung

Arrival/departure rate



Tsung

The yellow arrow indicates where the system starts to fail.



k6

Newer, more actively developed alternative to Tsung. Implemented in Go, performant, and scriptable with JavaScript.

You write your test cases in pure JavaScript. You have the entire programming language at your disposal. 😍

k6

Newer, more actively developed alternative to Tsung. Implemented in Go, performant, and scriptable with JavaScript.

You write your test cases in pure JavaScript. You have the entire programming language at your disposal. 😍

⚠️ JavaScript is a bridge to the underlying Go process; you're *not* running JavaScript with node. You can't npm install, import node modules, etc.

k6

K6 is a significantly easier alternative to load testing w/ Tsung.

One missing feature is graphs, which can help you identify at what load your server starts to fail. But you can work around this by stitching data together yourself.

For this class, **you should try using both k6 and Tsung**.

Hey

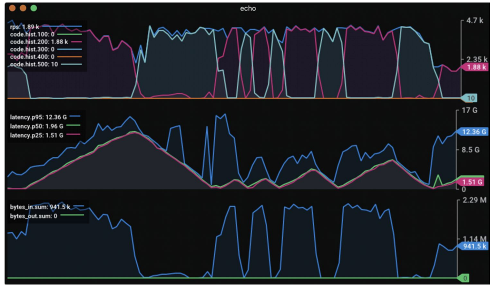
Like `apachebench`, but modern and with better performance.

```
hey -c 10 -z 10s 'http://127.0.0.1:3000/app/index'
```

Vegeta

Like `apachebench`, but with nifty features.

```
echo 'GET http://localhost:3000/app/index' | \
    vegeta attack -rate 50 -duration 10m | vegeta encode | \
    jaggr @count=rps \
          hist\[100,200,300,400,500\]:code \
          p25,p50,p95:latency \
          sum:bytes_in \
          sum:bytes_out | \
    jplot rps+code.hist.100+code.hist.200+code.hist.300+code.hist.400+code.hist.500 \
          latency.p95+latency.p50+latency.p25 \
          bytes_in.sum+bytes_out.sum
```



Load testing comparison

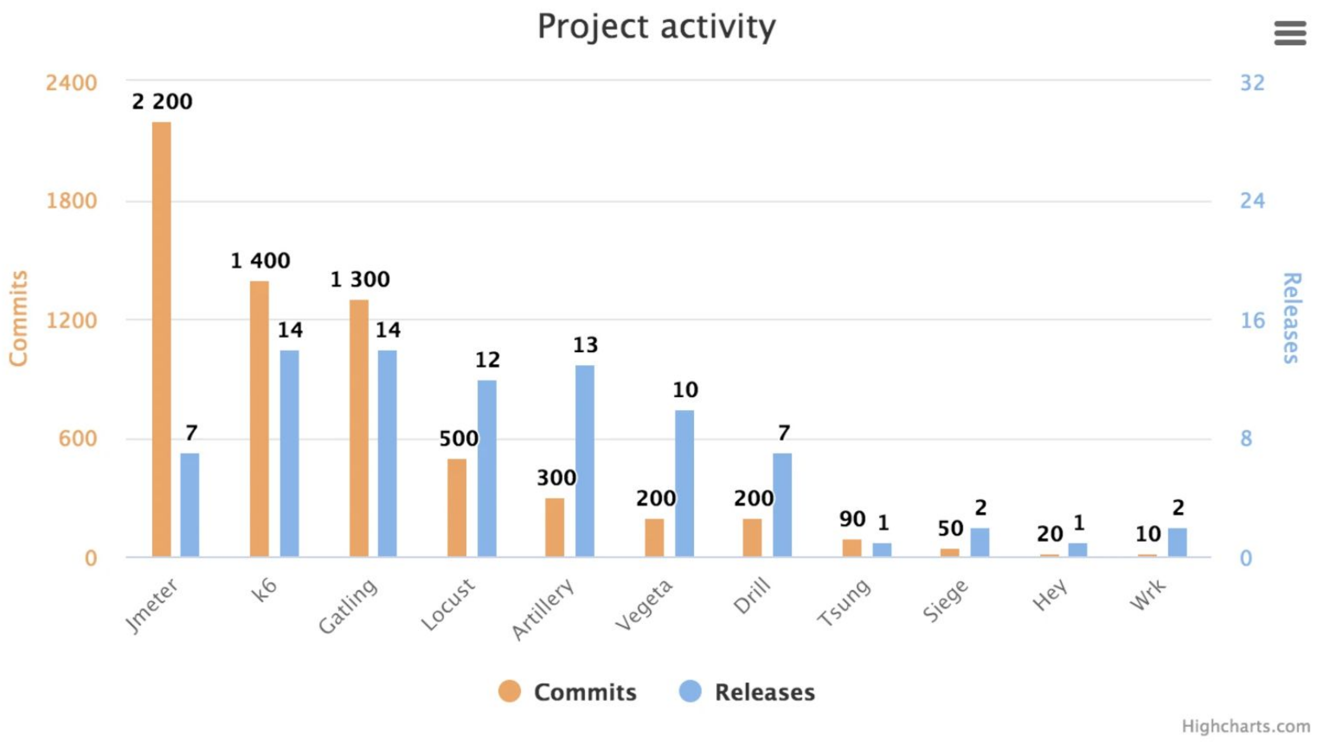
⁉️ **How do you compare the different load testing tools, in terms of their performance?**

Load testing comparison

⁉️ **How do you compare the different load testing tools, in terms of their performance?**

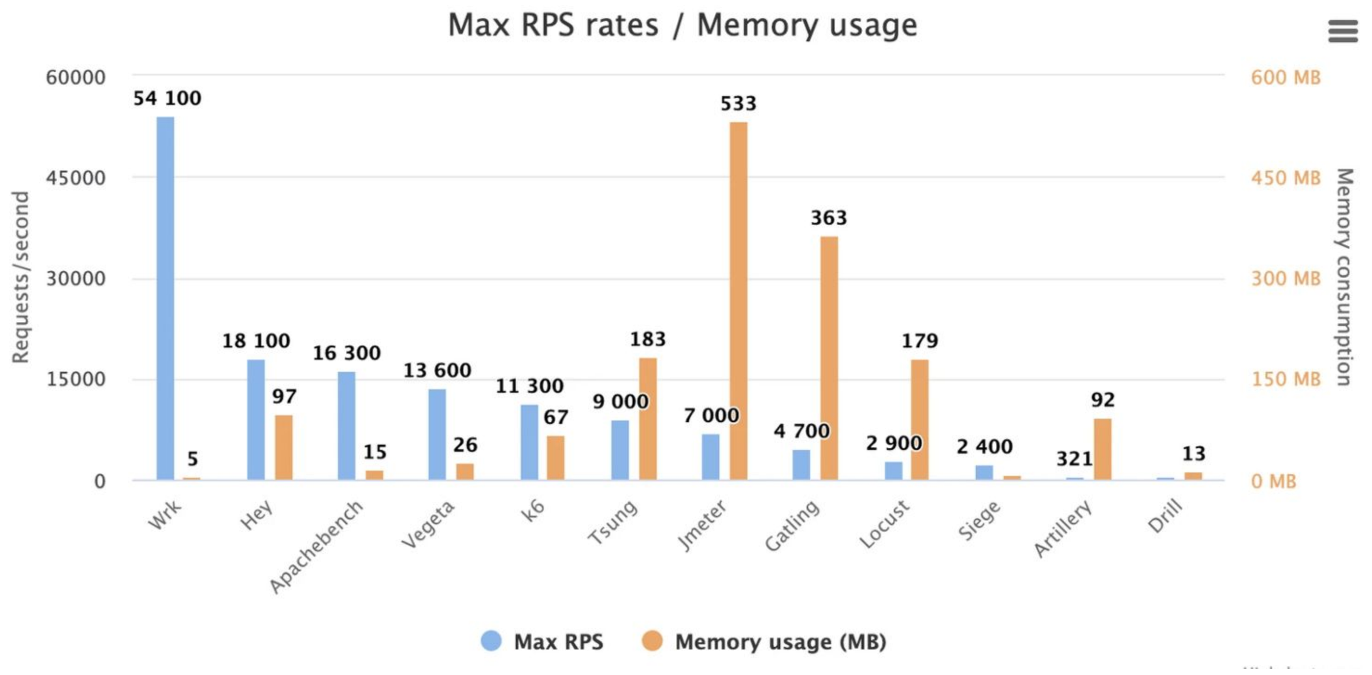
- Load test them!
- Don’t test them against your code (it’s slow), test against a third party thing that’s known to be performance (e.g. nginx hello world)
- Don’t forget to take into account network delays and network bandwidth!
- … and RAM, CPU, file descriptors, …

Load testing comparison



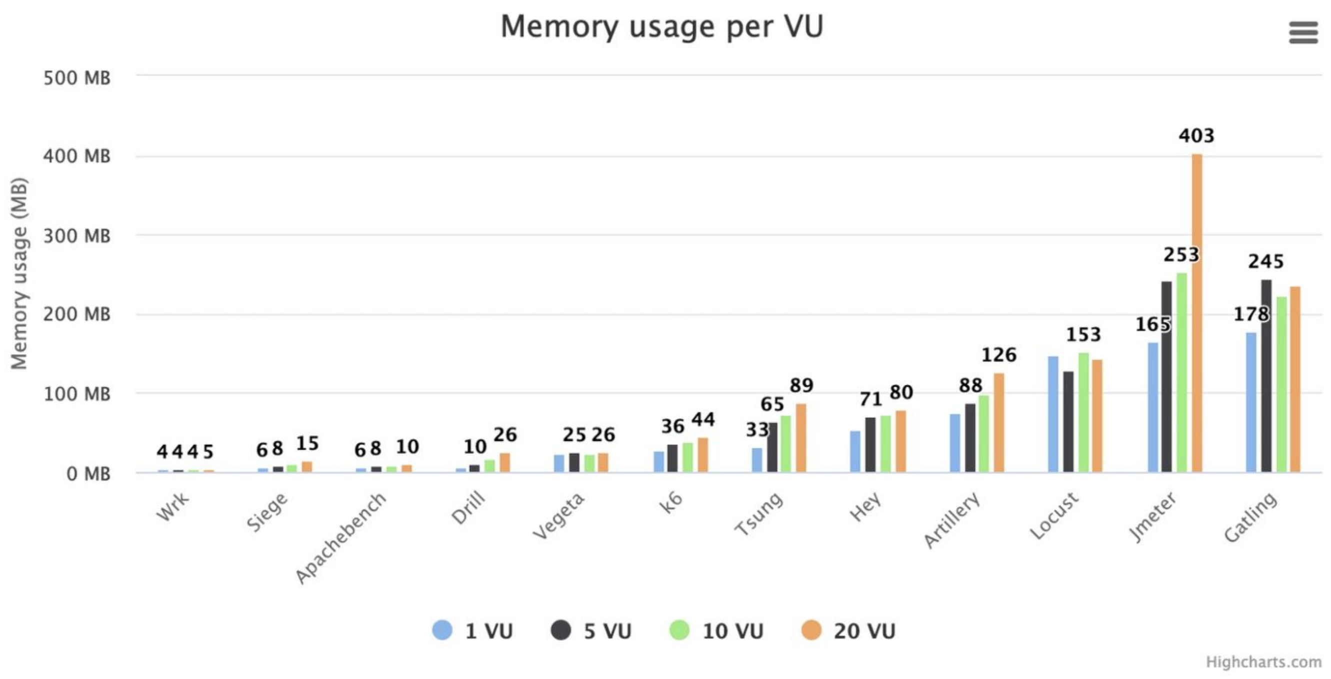
source

Load testing comparison



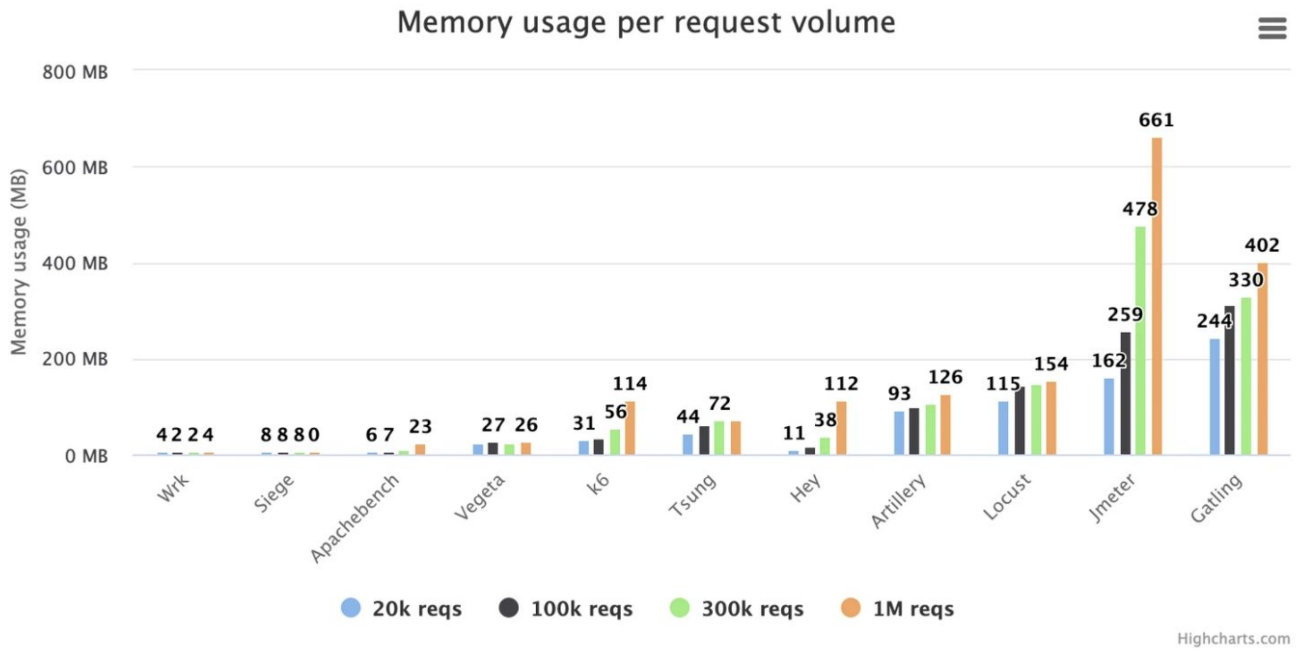
source

Load testing comparison



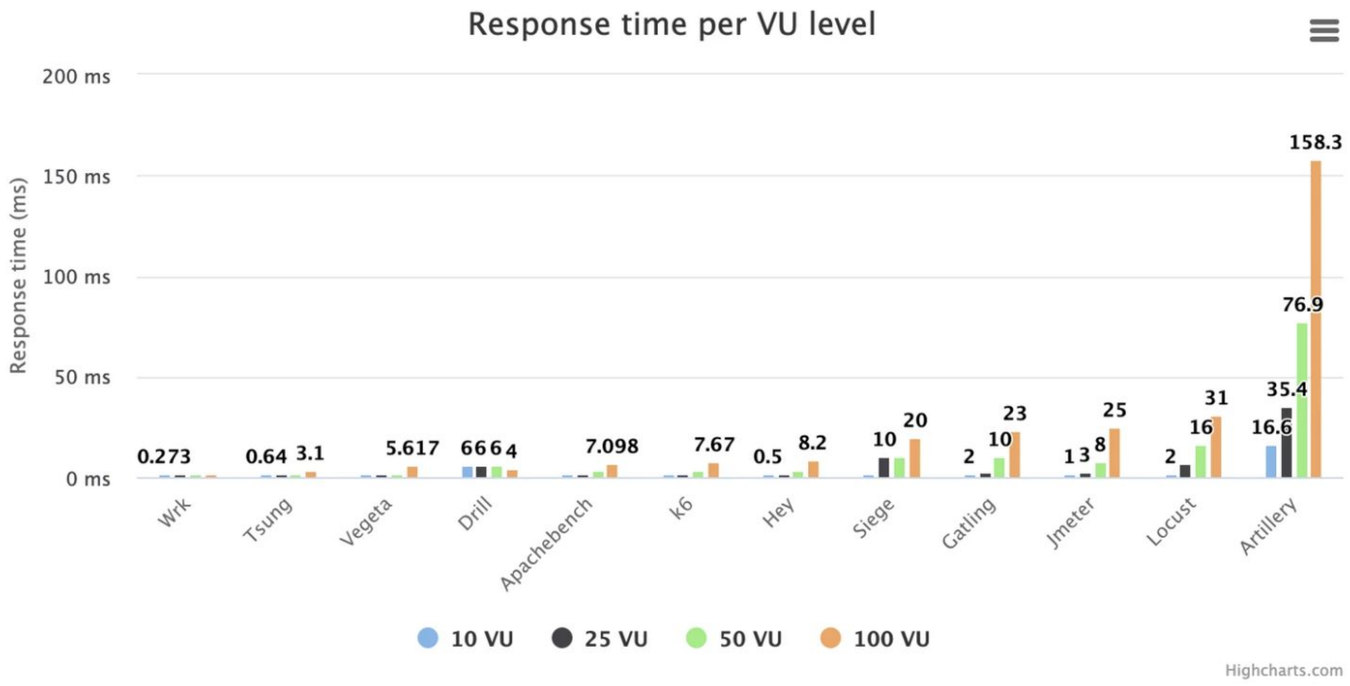
source

Load testing comparison



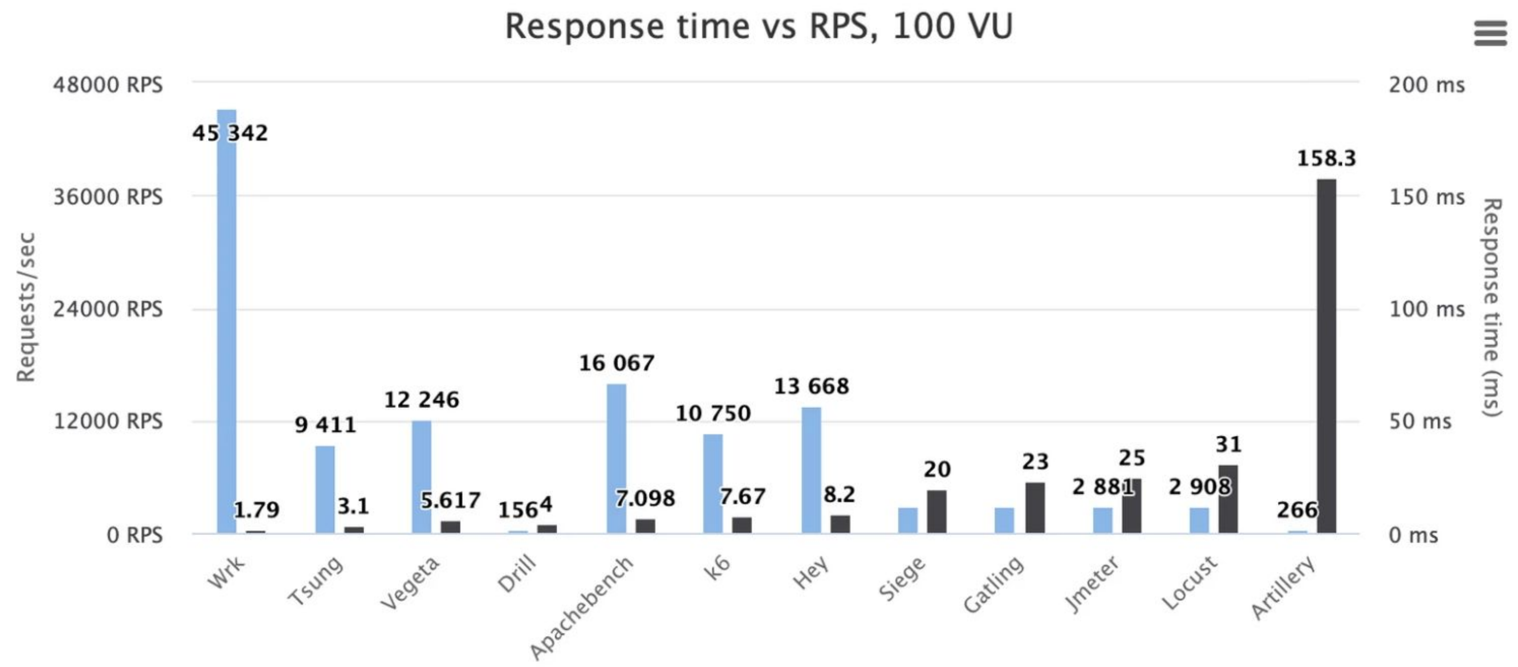
source

Load testing comparison



source

Load testing comparison



source