Performance Modelling — Lecture 12: PEPA Case Study: Rap Genius on Heroku

Jane Hillston & Dimitrios Milios
School of Informatics
The University of Edinburgh
Scotland

2nd March 2017

Introduction

- As an example of a realistic case study, we consider a Platform as a Service (PaaS) system, Heroku, and study its behaviour under different policies for assigning client jobs to leased servers.
- This case study has been developed by Dimitrios Milios.

Heroku

Cloud computing describes hardware or software resources made available as a service-on-demand, typically over the Internet.

Heroku

Cloud computing describes hardware or software resources made available as a service-on-demand, typically over the Internet.

There are different service models:

Infrastructure as a Service (IaaS): computing resources, including actual hardware or virtualised computers, storage, bandwidth or other resources.

Platform as a Service (PaaS): provides a customised solution stack, including operating systems, programming languages, libraries, web servers, databases and software tools.

Software as a Service (SaaS) depends on both IaaS and PaaS, and provides access to remote software applications in a manner completely transparent to the end user.

Heroku and dynos

- Heroku is a PaaS provider offering an integrated framework enabling developers to deploy and support web-based applications. Several programming languages are supported.

Heroku and dynos

- Heroku is a PaaS provider offering an integrated framework enabling developers to deploy and support web-based applications. Several programming languages are supported.
- Clients upload the source code for their application, together with a file that describes the software dependencies.

Heroku and dynos

- Heroku is a PaaS provider offering an integrated framework enabling developers to deploy and support web-based applications. Several programming languages are supported.
- Clients upload the source code for their application, together with a file that describes the software dependencies.
- The Heroku platform then builds the application, which will be executed on one or more virtualised machines, which are known as dynos.

Web dynos and worker dynos

- According to the on-line Heroku specification documents[1], a dyno is a lightweight environment running a single command at a time.

[1] https://devcenter.heroku.com/

Web dynos and worker dynos

- According to the on-line Heroku specification documents[1], a dyno is a lightweight environment running a single command at a time.
- This functionality is implemented by an isolated virtualised server.

[1] https://devcenter.heroku.com/

Web dynos and worker dynos

- According to the on-line Heroku specification documents[1], a dyno is a lightweight environment running a single command at a time.
- This functionality is implemented by an isolated virtualised server.
- Dynos are claimed to provide a secure and performance-consistent environment to run an application.

[1] https://devcenter.heroku.com/

Web dynos and worker dynos

- According to the on-line Heroku specification documents[1], a dyno is a lightweight environment running a single command at a time.
- This functionality is implemented by an isolated virtualised server.
- Dynos are claimed to provide a secure and performance-consistent environment to run an application.
- There are two kinds of dynos available:
  - web dynos which respond to HTTP requests, and
  - worker dynos which execute background jobs.

[1] https://devcenter.heroku.com/

Concurrency and scalability

- Commands are non-interruptable; thus concurrency is achieved by employing more than one dyno.

Concurrency and scalability

- Commands are non-interruptable; thus concurrency is achieved by employing more than one dyno.
- Increasing the number of web dynos will increase the concurrency of HTTP requests, while more worker dynos provide more capacity for processes running in the background.

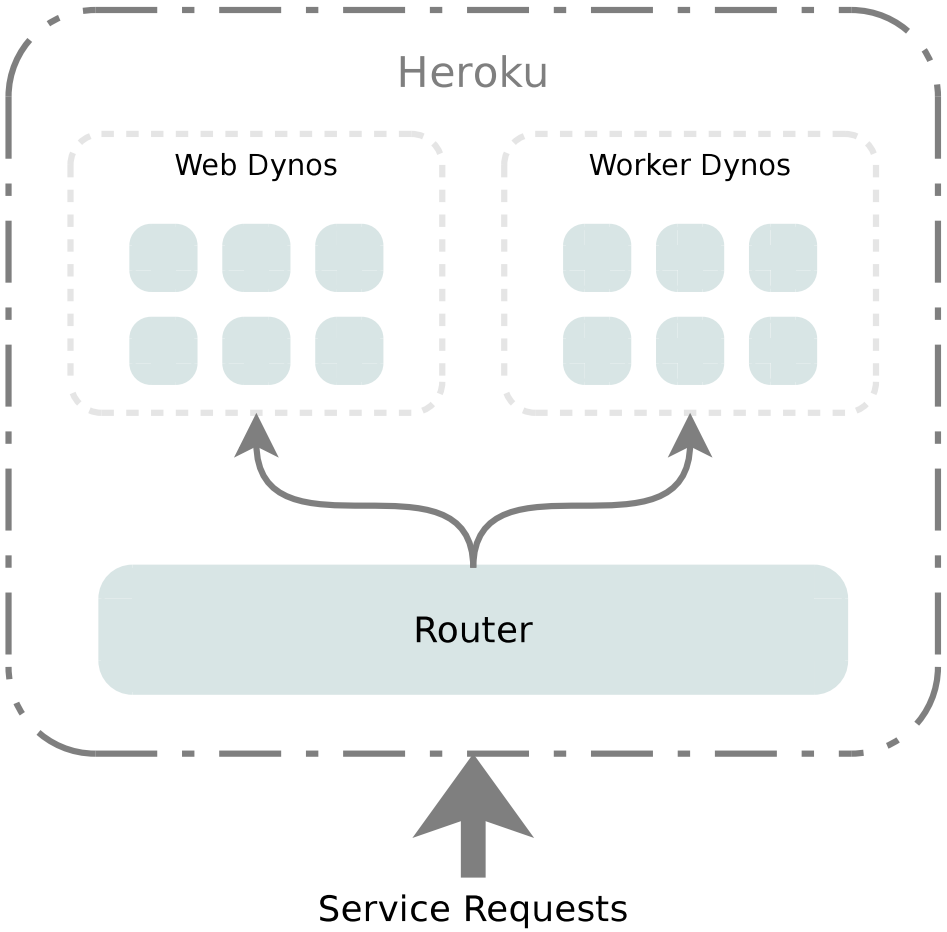
Concurrency and scalability

- Commands are non-interruptable; thus concurrency is achieved by employing more than one dyno.
- Increasing the number of web dynos will increase the concurrency of HTTP requests, while more worker dynos provide more capacity for processes running in the background.
- Therefore, all the client has to do is to upload the source code of the application and scale it to a number of dynos.

Concurrency and scalability

- Commands are non-interruptable; thus concurrency is achieved by employing more than one dyno.
- Increasing the number of web dynos will increase the concurrency of HTTP requests, while more worker dynos provide more capacity for processes running in the background.
- Therefore, all the client has to do is to upload the source code of the application and scale it to a number of dynos.
- When a service request appears, Heroku is responsible for assigning it to one of the dynos leased by the client, following a routing policy.

The basic structure of Heroku



We will investigate the routing policy performance.

Heroku routing policies

Two routing policies have historically been used by Heroku:

- Random Routing — a new request is directed to a randomly-selected web dyno. The premise of random routing is that the load is balanced across the dynos in the long term.
- Smart Routing — the availability of each dyno is tracked and the load is directed accordingly, thus minimising the number of idle dynos.

Heroku routing policies

Two routing policies have historically been used by Heroku:

- Random Routing — a new request is directed to a randomly-selected web dyno. The premise of random routing is that the load is balanced across the dynos in the long term.
- Smart Routing — the availability of each dyno is tracked and the load is directed accordingly, thus minimising the number of idle dynos.

Although explicit information on the implementation of these policies is not available, it is straightforward to model the desired behaviour for each policy at a high-level.

Determining the number of dynos needed

- Each client must determine how many dynos should be leased.

Determining the number of dynos needed

- Each client must determine how many dynos should be leased.
- This depends on the workload: the heavier the workload is, the more dynos will be needed.

Determining the number of dynos needed

- Each client must determine how many dynos should be leased.
- This depends on the workload: the heavier the workload is, the more dynos will be needed.
- In the ideal case, every service should be tailored to the needs of the corresponding client.

Determining the number of dynos needed

- Each client must determine how many dynos should be leased.
- This depends on the workload: the heavier the workload is, the more dynos will be needed.
- In the ideal case, every service should be tailored to the needs of the corresponding client.
- Typically, clients may have a rough idea of the expected workload. However, they may find it difficult to accurately estimate the number of the machines needed.

Applying performance modelling techniques

- Performance modelling is a natural way to produce such estimates in a rigorous manner.

Applying performance modelling techniques

- Performance modelling is a natural way to produce such estimates in a rigorous manner.
- Despite the fact that modelling relies on rather strong assumptions, if done appropriately it can provide us with useful insight into the behaviour of a system.

Applying performance modelling techniques

- Performance modelling is a natural way to produce such estimates in a rigorous manner.
- Despite the fact that modelling relies on rather strong assumptions, if done appropriately it can provide us with useful insight into the behaviour of a system.
- Even just having some expectations about the system can help the client to detect when something has gone wrong.

Rap Genius/Genius

- Genius[2] (formerly known as Rap Genius[3]) is a website that aims to provide a critical and artistic insight into the lyrics of rap songs.

[2] http://genius.com/
[3] http://rapgenius.com/

Rap Genius/Genius

- Genius[2] (formerly known as Rap Genius[3]) is a website that aims to provide a critical and artistic insight into the lyrics of rap songs.
- The cultural contribution of Genius is remarkable, however, in terms of the current poetically sterile course, we shall focus on some technical aspects only.

[2] http://genius.com/
[3] http://rapgenius.com/

Rap Genius/Genius

- Genius[2] (formerly known as Rap Genius[3]) is a website that aims to provide a critical and artistic insight into the lyrics of rap songs.
- The cultural contribution of Genius is remarkable, however, in terms of the current poetically sterile course, we shall focus on some technical aspects only.
- The website users have access to content via HTTP requests, and they are able to add annotations to content.

[2] http://genius.com/
[3] http://rapgenius.com/

Rap Genius/Genius

- Genius[2] (formerly known as Rap Genius[3]) is a website that aims to provide a critical and artistic insight into the lyrics of rap songs.
- The cultural contribution of Genius is remarkable, however, in terms of the current poetically sterile course, we shall focus on some technical aspects only.
- The website users have access to content via HTTP requests, and they are able to add annotations to content.
- Genius makes this service available via Heroku.

[2] http://genius.com/
[3] http://rapgenius.com/

The problem experienced

- In the beginning of 2013, Rap Genius reported unusually long average response times, despite the large number of dynos leased by the website.

The problem experienced

- In the beginning of 2013, Rap Genius reported unusually long average response times, despite the large number of dynos leased by the website.
- The average response time reported by the Heroku platform was as low as 40 ms, while the response time experienced by the users had been 6330 ms.

The problem experienced

- In the beginning of 2013, Rap Genius reported unusually long average response times, despite the large number of dynos leased by the website.
- The average response time reported by the Heroku platform was as low as 40 ms, while the response time experienced by the users had been 6330 ms.
- This difference was attributed to requests waiting in the local queues at the dynos.

The problem experienced

- In the beginning of 2013, Rap Genius reported unusually long average response times, despite the large number of dynos leased by the website.
- The average response time reported by the Heroku platform was as low as 40 ms, while the response time experienced by the users had been 6330 ms.
- This difference was attributed to requests waiting in the local queues at the dynos.
- Therefore, given that the actual service had not been any slower than usual, this suggested that the system had simply been overloaded.

Diagnosing the problem

- Nevertheless, according to Rap Genius, there had not been any significant change in the workload, which had been as high as 9000 requests per minute.

[4] http://www.wired.com/2013/03/hieroku

Diagnosing the problem

- Nevertheless, according to Rap Genius, there had not been any significant change in the workload, which had been as high as 9000 requests per minute.
- Eventually, this considerable increase in the response time was blamed on the fact that the Heroku routing policy has been changed from smart to random[4].

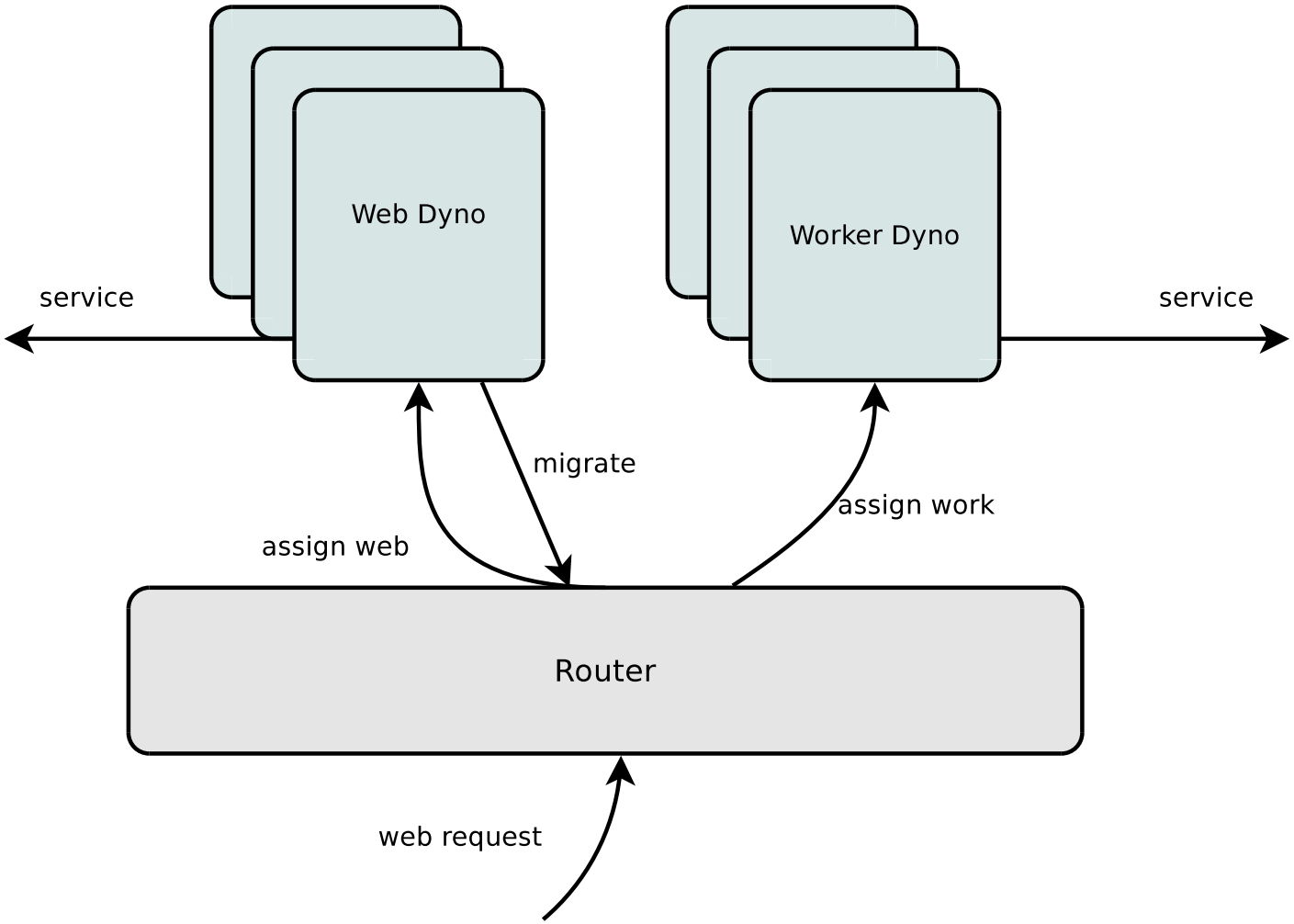
[4] http://www.wired.com/2013/03/hieroku

Diagnosing the problem

- Nevertheless, according to Rap Genius, there had not been any significant change in the workload, which had been as high as 9000 requests per minute.
- Eventually, this considerable increase in the response time was blamed on the fact that the Heroku routing policy has been changed from smart to random[4].
- Here, our objective is not to assess the quality of service provided by Heroku, or recreate the situation experienced by Rap Genius. Instead, we demonstrate how modelling with PEPA can capture the effect of different routing policies.

[4] http://www.wired.com/2013/03/hieroku

The Heroku configuration considered



Modelling assumptions

- We assume the web requests arrive at the router in a Poisson stream.

Modelling assumptions

- We assume the web requests arrive at the router in a Poisson stream.
- The router component is responsible for forwarding each request to a web dyno.

Modelling assumptions

- We assume the web requests arrive at the router in a Poisson stream.
- The router component is responsible for forwarding each request to a web dyno.
- When a web dyno receives a request, there are two possibilities:
  - it can either service the request directly or
  - create a new request to be serviced by a worker dyno.

Modelling assumptions

- We assume the web requests arrive at the router in a Poisson stream.
- The router component is responsible for forwarding each request to a web dyno.
- When a web dyno receives a request, there are two possibilities:
  - it can either service the request directly or
  - create a new request to be serviced by a worker dyno.
- In the latter case, the current job will migrate from a web dyno to a worker dyno, and the router is responsible for redirecting the request accordingly.

Modelling assumptions and parameters

- Migration from a web dyno to a worker dyno is the only way a worker dyno may be accessed, as the users are assumed to produce HTTP requests only.
- The generation of a worker request captures the possibility that a job may require some background computation.
- It is assumed that the fraction requests that are migrated is small; more specifically, we consider a migration probability equal to 1/9.

Activity types

- We can identify some activity types representing behaviour in our model, regardless of the routing policy.
- These activities are *request*, *assign*, *web*, *migrate*, *worker* and *response* and each will be associated with an exponentially distributed duration.
- The request arrival rate $r_{request}$ will control the assumed workload in the system.
  - This is the variable we are going to experiment with, so it will take values within a range from 40 to 150 sec^{-1}, which corresponds to 9000 requests per minute.

Rate values

Variable Name	Value (sec^{-1})
$r_{request}$	[40, 60, 150]
r_{web}	8
$r_{migrate}$	1
r_{worker}	4
$r_{response}$	20
r_{assign}	500

Service rates and dynos

- While both types of dynos are identical with respect to their computational capabilities, the worker dynos deal with more demanding tasks, which is reflected in a lower service rate.
- Therefore the average web service time is $1/r_{web} = 0.125$ sec, while for the worker dyno services we have an average time of $1/r_{worker} = 0.25$ sec.
- The response part represents the time needed by a dyno to transmit the results to the user. It is considered to be identical in both cases, as it only depends on the network.
- Moreover, response takes place at a considerably higher rate than the actual service, so it has rate $r_{response} = 20$.

Migration and migration probability

- It is assumed that there is a race condition between migration and web service.
- Thus, the rate of migration will control the migration probability.
- By considering $r_{migrate} = 1$ and given that we have $r_{web} = 8$, we impose a migration probability equal to $1/9$.

Assignment of jobs to dynos

- Finally, it is assumed that assignment happens almost instantaneously, since it depends only on the resources allocated to the routing component.
- It is fair to expect that any decision will take place very quickly based on the current state of the system.
- This is reflected by the high rate $r_{assign} = 500$, or 2 milliseconds average duration.

PEPA model

- In the following, we present two PEPA models that implement the two routing policies.
- We assume that each dyno has its own queue, thus we are interested in observing how the local dyno queues are affected by each policy.
- In the PEPA models we have components for the web dynos, the worker dynos, and the system router which keeps queues for requests coming from the web and migration requests from the web dynos for the worker dynos.

Random Routing Policy

- A dyno can be idle, occupied or with one or more requests in its local queue.
- According to the random routing policy, the router randomly assigns jobs to dynos, regardless of their state.

PEPA components

- Web dynos are represented by components $WebDyno_i$, where the subscript i denotes the number of requests in the local dyno queue.

PEPA components

- Web dynos are represented by components $WebDyno_i$, where the subscript i denotes the number of requests in the local dyno queue.
- For $WebDyno_i$, three activities are possible; *service* represents the main web service part, whose completion proceeds to the response stage, carried out by $WebDyno_{ia}$. Since a response cannot be interrupted, no new job can be assigned or enqueued at this point.

PEPA components

- Web dynos are represented by components $WebDyno_i$, where the subscript i denotes the number of requests in the local dyno queue.
- For $WebDyno_i$, three activities are possible; *service* represents the main web service part, whose completion proceeds to the response stage, carried out by $WebDyno_{ia}$. Since a response cannot be interrupted, no new job can be assigned or enqueued at this point.
- The *migrate* activity generates a migration request and decreases the queue length at this web dyno.

PEPA components

- Web dynos are represented by components $WebDyno_i$, where the subscript i denotes the number of requests in the local dyno queue.
- For $WebDyno_i$, three activities are possible; *service* represents the main web service part, whose completion proceeds to the response stage, carried out by $WebDyno_{ia}$. Since a response cannot be interrupted, no new job can be assigned or enqueued at this point.
- The *migrate* activity generates a migration request and decreases the queue length at this web dyno.
- Finally, the $assign_{web}$ activity adds a request to the queue from the client.

The Random *WebDyno* component in PEPA

$$
\begin{aligned}
WebDyno &\stackrel{def}{=} (assign_{web}, \top).WebDyno_0 \\
WebDyno_i &\stackrel{def}{=} (service, r_{web}).WebDyno_{ia} \\
&+ (migrate, r_{migrate}).WebDyno_{i-1} \\
&+ (assign_{web}, \top).WebDyno_{i+1} \\
WebDyno_{ia} &\stackrel{def}{=} (response, r_{response}).WebDyno_{i-1}
\end{aligned}
$$

The Random *WebDyno* component in PEPA

$$
\begin{aligned}
WebDyno &\stackrel{def}{=} (assign_{web}, \top).WebDyno_0 \\
WebDyno_i &\stackrel{def}{=} (service, r_{web}).WebDyno_{ia} \\
&\quad + (migrate, r_{migrate}).WebDyno_{i-1} \\
&\quad + (assign_{web}, \top).WebDyno_{i+1} \\
WebDyno_{ia} &\stackrel{def}{=} (response, r_{response}).WebDyno_{i-1}
\end{aligned}
$$

The Random *WebDyno* component in PEPA

$$
\begin{aligned}
WebDyno &\stackrel{def}{=} (assign_{web}, \top).WebDyno_0 \\
WebDyno_i &\stackrel{def}{=} (service, r_{web}).WebDyno_{ia} \\
&+ (migrate, r_{migrate}).WebDyno_{i-1} \\
&+ (assign_{web}, \top).WebDyno_{i+1} \\
WebDyno_{ia} &\stackrel{def}{=} (response, r_{response}).WebDyno_{i-1}
\end{aligned}
$$

The Random *WebDyno* component in PEPA

$$
\begin{aligned}
WebDyno &\stackrel{def}{=} (assign_{web}, \top).WebDyno_0 \\
WebDyno_i &\stackrel{def}{=} (service, r_{web}).WebDyno_{ia} \\
&+ (migrate, r_{migrate}).WebDyno_{i-1} \\
&+ (assign_{web}, \top).WebDyno_{i+1} \\
WebDyno_{ia} &\stackrel{def}{=} (response, r_{response}).WebDyno_{i-1}
\end{aligned}
$$

The Random *WebDyno* component in PEPA

$$
\begin{aligned}
WebDyno &\stackrel{def}{=} (assign_{web}, \top).WebDyno_0 \\
WebDyno_i &\stackrel{def}{=} (service, r_{web}).WebDyno_{ia} \\
&\quad + (migrate, r_{migrate}).WebDyno_{i-1} \\
&\quad + (assign_{web}, \top).WebDyno_{i+1} \\
WebDyno_{ia} &\stackrel{def}{=} (response, r_{response}).WebDyno_{i-1}
\end{aligned}
$$

The Random *WebDyno* component in PEPA

$$
\begin{aligned}
WebDyno &\stackrel{def}{=} (assign_{web}, \top).WebDyno_0 \\
WebDyno_i &\stackrel{def}{=} (service, r_{web}).WebDyno_{ia} \\
&+ (migrate, r_{migrate}).WebDyno_{i-1} \\
&+ (assign_{web}, \top).WebDyno_{i+1} \\
WebDyno_{ia} &\stackrel{def}{=} (response, r_{response}).WebDyno_{i-1}
\end{aligned}
$$

Idle and busy states

$WebDyno_i$ and $WebDyno_{ia}$ represent the two stages of a web service. In both cases, the web dyno is considered to be occupied. The idle state is denoted by $WebDyno$.

The Random *WorkerDyno* component in PEPA

The worker dynos have a similar but simpler structure, as in this case there is no job migration option.

$$
\begin{aligned}
WorkerDyno &\stackrel{def}{=} (assign_{worker}, \top).WorkerDyno_0 \\
WorkerDyno_i &\stackrel{def}{=} (service, r_{worker}).WorkerDyno_{ia} \\
&\quad + (assign_{worker}, \top).WorkerDyno_{i+1} \\
WorkerDyno_{ia} &\stackrel{def}{=} (response, r_{response}).WorkerDyno_{i-1}
\end{aligned}
$$

The Random *WorkerDyno* component in PEPA

The worker dynos have a similar but simpler structure, as in this case there is no job migration option.

$$
\begin{aligned}
WorkerDyno &\stackrel{def}{=} (assign_{worker}, \top).WorkerDyno_0 \\
WorkerDyno_i &\stackrel{def}{=} (service, r_{worker}).WorkerDyno_{ia} \\
&\quad + (assign_{worker}, \top).WorkerDyno_{i+1} \\
WorkerDyno_{ia} &\stackrel{def}{=} (response, r_{response}).WorkerDyno_{i-1}
\end{aligned}
$$

The Random *WorkerDyno* component in PEPA

The worker dynos have a similar but simpler structure, as in this case there is no job migration option.

$$
\begin{aligned}
WorkerDyno &\stackrel{def}{=} (assign_{worker}, \top).WorkerDyno_0 \\
WorkerDyno_i &\stackrel{def}{=} (service, r_{worker}).WorkerDyno_{ia} \\
&\quad + (assign_{worker}, \top).WorkerDyno_{i+1} \\
WorkerDyno_{ia} &\stackrel{def}{=} (response, r_{response}).WorkerDyno_{i-1}
\end{aligned}
$$

The Random *WorkerDyno* component in PEPA

The worker dynos have a similar but simpler structure, as in this case there is no job migration option.

$$
\begin{aligned}
WorkerDyno &\stackrel{def}{=} (assign_{worker}, \top).WorkerDyno_0 \\
WorkerDyno_i &\stackrel{def}{=} (service, r_{worker}).WorkerDyno_{ia} \\
&\quad + (assign_{worker}, \top).WorkerDyno_{i+1} \\
WorkerDyno_{ia} &\stackrel{def}{=} (response, r_{response}).WorkerDyno_{i-1}
\end{aligned}
$$

Routing

- The routing component is characterised by a set of states that denote the number of requests in the router queue.
- In any state, the router can accept a web request or a migration request, and add it to the router queue.
- If one or more jobs are in the queue, the router will attempt to direct them to any of the web or worker dynos, depending on the type of the request.
- It is convenient to model the router as two queues, one for each type of dyno.

The Random *WebRouter* component

$$
\begin{aligned}
WebRouter_0 &\stackrel{def}{=} (request, r_{request}).WebRouter_1 \\
WebRouter_i &\stackrel{def}{=} (request, r_{request}).WebRouter_{i+1} \\
&\quad + (assign_{web}, r_{assign}).WebRouter_{i-1} \\
WebRouter_n &\stackrel{def}{=} (request, r_{request}).WebRouter_n \\
&\quad + (assign_{web}, r_{assign}).WebRouter_{n-1}
\end{aligned}
$$

where n denotes the maximum size for the corresponding queue.

The Random *WebRouter* component

$$
\begin{aligned}
WebRouter_0 &\stackrel{def}{=} (request, r_{request}).WebRouter_1 \\
WebRouter_i &\stackrel{def}{=} (request, r_{request}).WebRouter_{i+1} \\
&\quad + (assign_{web}, r_{assign}).WebRouter_{i-1} \\
WebRouter_n &\stackrel{def}{=} (request, r_{request}).WebRouter_n \\
&\quad + (assign_{web}, r_{assign}).WebRouter_{n-1}
\end{aligned}
$$

where n denotes the maximum size for the corresponding queue.

The Random *WebRouter* component

$$
\begin{aligned}
WebRouter_0 &\stackrel{def}{=} (request, r_{request}).WebRouter_1 \\
WebRouter_i &\stackrel{def}{=} (request, r_{request}).WebRouter_{i+1} \\
&\quad + (assign_{web}, r_{assign}).WebRouter_{i-1} \\
WebRouter_n &\stackrel{def}{=} (request, r_{request}).WebRouter_n \\
&\quad + (assign_{web}, r_{assign}).WebRouter_{n-1}
\end{aligned}
$$

where n denotes the maximum size for the corresponding queue.

The Random *WebRouter* component

$$
\begin{aligned}
WebRouter_0 &\stackrel{def}{=} (request, r_{request}).WebRouter_1 \\
WebRouter_i &\stackrel{def}{=} (request, r_{request}).WebRouter_{i+1} \\
&\quad + (assign_{web}, r_{assign}).WebRouter_{i-1} \\
WebRouter_n &\stackrel{def}{=} (request, r_{request}).WebRouter_n \\
&\quad + (assign_{web}, r_{assign}).WebRouter_{n-1}
\end{aligned}
$$

where n denotes the maximum size for the corresponding queue.

Overflow behaviour

If the maximum size is reached, it is assumed that any new requests will be discarded until the queue is not full.

The Random *WorkerRouter* component

$$
\begin{aligned}
WorkerRouter_0 &\stackrel{def}{=} (migrate, \top).WorkerRouter_1 \\
WorkerRouter_i &\stackrel{def}{=} (migrate, \top).WorkerRouter_{i+1} \\
&\quad + (assign_{worker}, r_{assign}).WorkerRouter_{i-1} \\
WorkerRouter_n &\stackrel{def}{=} (migrate, \top).WorkerRouter_n \\
&\quad + (assign_{worker}, r_{assign}).WorkerRouter_{n-1}
\end{aligned}
$$

The Random *WorkerRouter* component

$$
\begin{aligned}
WorkerRouter_0 &\stackrel{def}{=} (migrate, \top).WorkerRouter_1 \\
WorkerRouter_i &\stackrel{def}{=} (migrate, \top).WorkerRouter_{i+1} \\
&\quad + (assign_{worker}, r_{assign}).WorkerRouter_{i-1} \\
WorkerRouter_n &\stackrel{def}{=} (migrate, \top).WorkerRouter_n \\
&\quad + (assign_{worker}, r_{assign}).WorkerRouter_{n-1}
\end{aligned}
$$

The Random *WorkerRouter* component

$$
\begin{aligned}
WorkerRouter_0 &\stackrel{def}{=} (migrate, \top).WorkerRouter_1 \\
WorkerRouter_i &\stackrel{def}{=} (migrate, \top).WorkerRouter_{i+1} \\
&\quad + (assign_{worker}, r_{assign}).WorkerRouter_{i-1} \\
WorkerRouter_n &\stackrel{def}{=} (migrate, \top).WorkerRouter_n \\
&\quad + (assign_{worker}, r_{assign}).WorkerRouter_{n-1}
\end{aligned}
$$

Smart routing policy

- In the smart routing case, the dyno components are similar to those in the random routing case, as both web and worker dynos have the same states and the same rates.
- The only difference is that we now have two distinct action types for assigning a job to a dyno.
- We want to capture the fact that a job may be either assigned to an idle dyno, or enqueued to an occupied dyno.

The Smart *WebDyno* component in PEPA

For the web dynos, only a *WebDyno* component will now be able to perform an $assign_{web}$ activity, as it denotes that the dyno is idle.

$$
\begin{aligned}
WebDyno &\stackrel{def}{=} (assign_{web}, \top).WebDyno_0 \\
WebDyno_i &\stackrel{def}{=} (service, r_{web}).WebDyno_{ia} \\
&+ (migrate, r_{migrate}).WebDyno_{i-1} \\
&+ (enqueue_{web}, \top).WebDyno_{i+1}
\end{aligned}
$$

$WebDyno_{ia}$ remains unchanged.

The Smart *WorkerDyno* component in PEPA

Similarly, an $assign_{worker}$ activity can only be performed by *WorkerDyno*, while for the $WorkerDyno_i$ component we have only *service* and $enqueue_{worker}$

$$
\begin{aligned}
WorkerDyno &\stackrel{def}{=} (assign_{worker}, \top).WorkerDyno_0 \\
WorkerDyno_i &\stackrel{def}{=} (service, r_{worker}).WorkerDyno_{ia} \\
&\quad + (enqueue_{worker}, \top).WorkerDyno_{i+1}
\end{aligned}
$$

$WorkerDyno_{ia}$ remains unchanged.

Representing smart routing in PEPA

- Smart routing directs a request to an available dyno.
  - If more than one dyno is available, then the router will randomly select a dyno.
  - If there are no dynos available, the request will be randomly enqueued to any dyno.
- The routing algorithm has a deterministic step: the dyno availability check. Our model probabilistically favours assigning jobs to free dynos over placing them in queues.
- The idea is that the router will delay directing a request until a dyno is available.
- If too many requests arrive, then the router will decrease its queue length by directing the requests to random dynos.

The Smart *WebRouter* component

We assume that the *WebRouter* component has a maximum queue length of n. Then for any queue length $i < n$, the requests are assigned to web dynos that can perform an $assign_{web}$ activity; i.e. the dyno is currently idle.

$$\begin{aligned} WebRouter_0 &\stackrel{def}{=} (request, r_{request}).WebRouter_1 \\ WebRouter_i &\stackrel{def}{=} (request, r_{request}).WebRouter_{i+1} \\ &+ (assign_{web}, r_{assign}).WebRouter_{i-1} \end{aligned}$$

The Smart *WebRouter* component

If the queue length reaches its maximum size n, this means that no dyno has been available for a long time.

It is then acceptable to send the request to the queue of any dyno thus $WebRouter_n$ will either assign or enqueue a request.

$$\begin{aligned} WebRouter_n \stackrel{def}{=} & \ (request, r_{request}).WebRouter_n \\ + & \ (assign_{web}, r_{assign} \times 0.5).WebRouter_{n-1} \\ + & \ (enqueue_{web}, r_{assign} \times 0.5).WebRouter_{n-1} \end{aligned}$$

The Smart *WorkerRouter* component

Analogously, the migration queue on the router side will be modified as follows:

$$
\begin{aligned}
WorkerRouter_0 &\stackrel{def}{=} (migrate, \top).WorkerRouter_1 \\
WorkerRouter_i &\stackrel{def}{=} (migrate, \top).WorkerRouter_{i+1} \\
&+ (assign_{worker}, r_{assign}).WorkerRouter_{i-1} \\
WorkerRouter_n &\stackrel{def}{=} (migrate, \top).WorkerRouter_n \\
&+ (assign_{worker}, r_{assign} \times 0.5).WorkerRouter_{n-1} \\
&+ (enqueue_{worker}, r_{assign} \times 0.5).WorkerRouter_{n-1}
\end{aligned}
$$

where n denotes the maximum queue length, and $0 < i < n$.

Summary

- When the queue of the router is not full, then the router works according to its "smart" mode of operation — it directs any requests to idle dynos only.
- New requests wait in the router queue before being assigned.
- However, if the router queue reaches maximum capacity, this is an indication that the system is congested, suggesting that there are no idle dynos available.
- The router will then enter its "random" mode of operation, and will decrease its queue by randomly directing requests to any dyno; $enqueue_{web}$ and $enqueue_{worker}$ can only be performed if the corresponding router queue is full.

Evaluation of routing policies

- We experimentally evaluate how the routing policies respond to different workloads.
- We consider a system featuring eight web dynos and eight worker dynos. We have two models that implement the two routing policies; these are $Random_{8:8}$ and $Smart_{8:8}$.

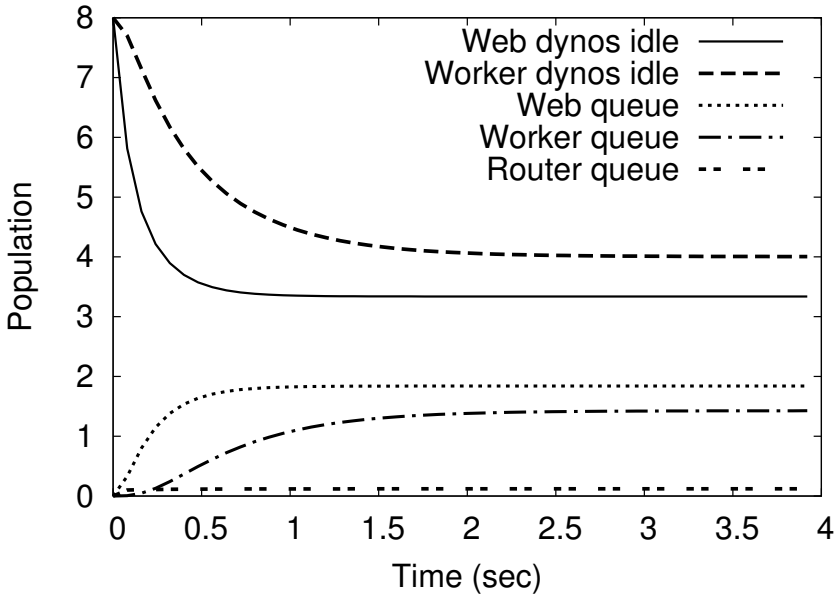
Evaluation of routing policies

- We experimentally evaluate how the routing policies respond to different workloads.
- We consider a system featuring eight web dynos and eight worker dynos. We have two models that implement the two routing policies; these are $Random_{8:8}$ and $Smart_{8:8}$.
- The models have been solved for their transient and steady-state behaviour.
  - The $Random_{8:8}$ model has $3,920,400$ states and took $10,250$ seconds for steady state solution and $39,000$ seconds for transient analysis.
  - The $Smart_{8:8}$ model has $3,849,444$ states and took $11,370$ seconds for steady state solution and $43,000$ seconds for transient solution.

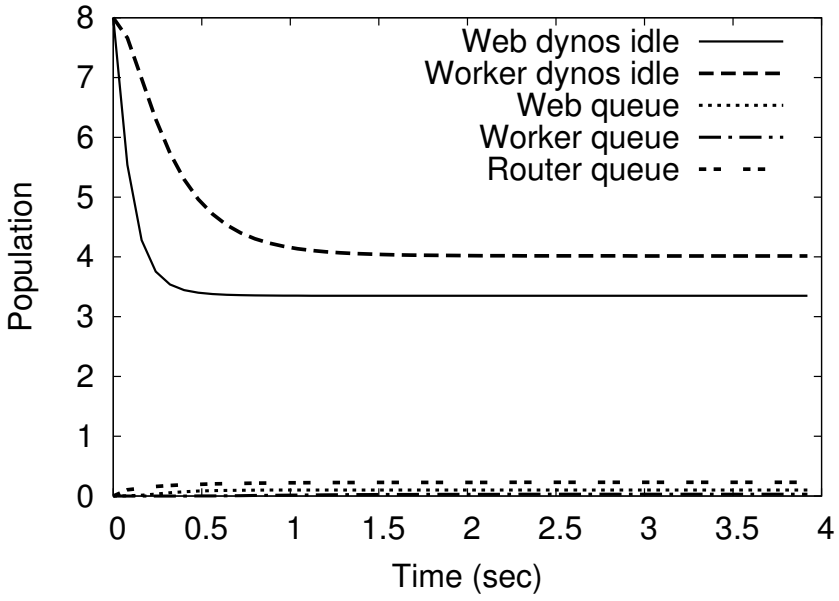
Experimentation with the workload

- We experimented with two different values for the request rate 40 and 60, in order to observe how the two routing policies respond to different workloads.
- The effects of each policy are reflected in the average dyno queue length and in the number of dynos that remain idle.

$Random_{8:8}$ and $Smart_{8:8}$ results for $r_{request} = 40$



(a) Random routing



(b) Smart routing

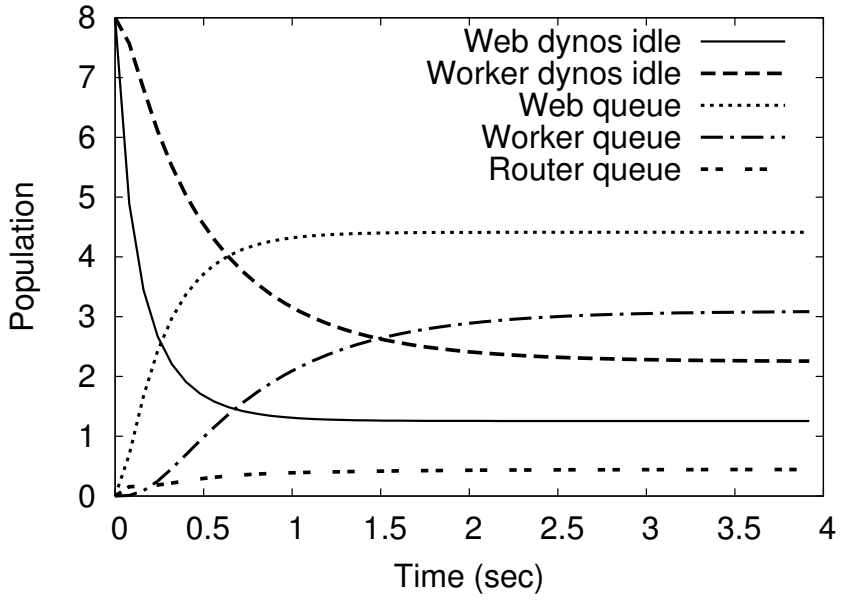
Comments on the results

- The results show that part of the system is underused for both smart and random routing, as there are a significant number of idle dynos in both cases.
- However, the average dyno queue lengths are noticeably higher for random routing which means that some requests might be waiting in the queue while there are dynos available.
- That is not the case for smart routing however, where the dyno queues are almost empty.

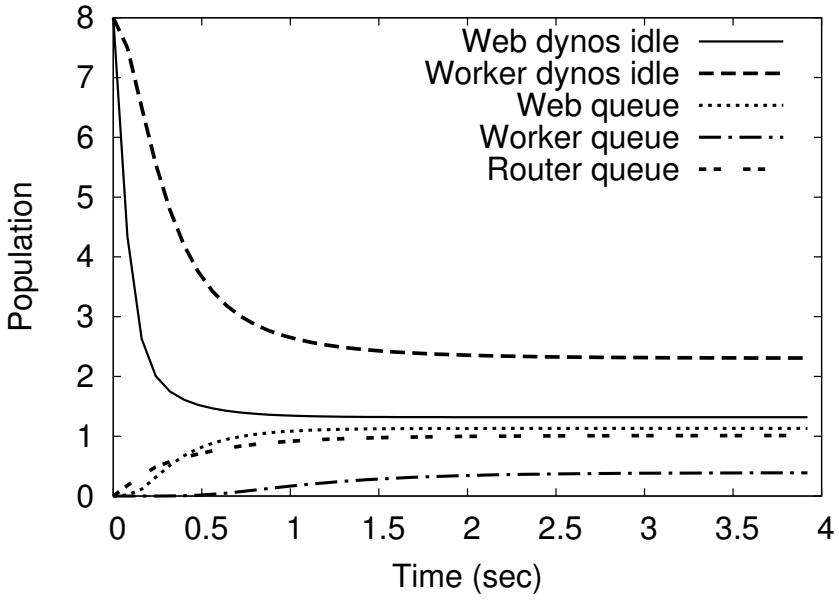
Increasing the request rate

- We investigated how the routing policies are affected by a higher workload, by increasing the request arrival rate to 60 per second, or 3600 requests per minute.

$Random_{8:8}$ and $Smart_{8:8}$ results for $r_{request} = 60$



(a) Random routing



(b) Smart routing

Comments on the results

- Here, the system usage is similar for both random and smart routing.
- For the smart system, the dyno queues have significantly shorter length when compared to the random routing policy, implying that the requests wait less time for service.

Comments on the results

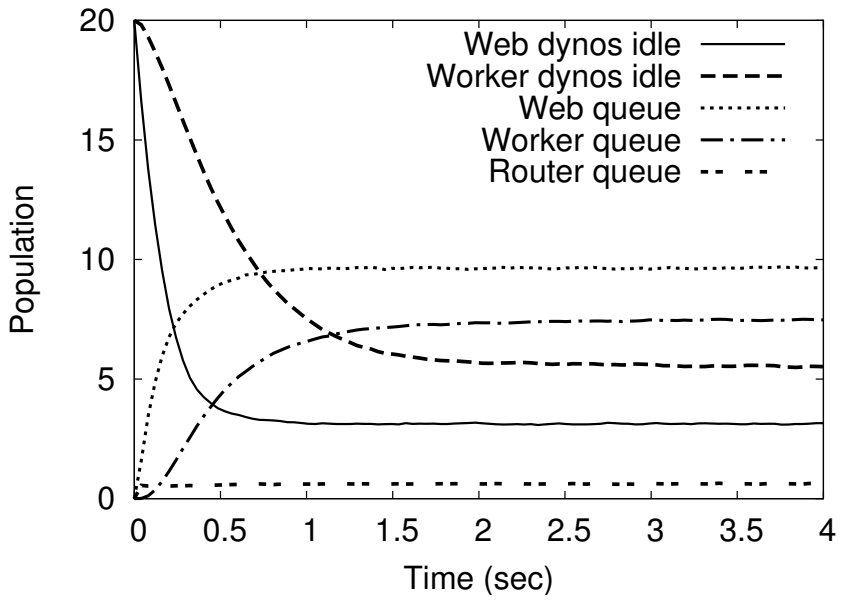
- Here, the system usage is similar for both random and smart routing.
- For the smart system, the dyno queues have significantly shorter length when compared to the random routing policy, implying that the requests wait less time for service.

- To summarise, the smart routing policy results in better utilisation of the system resources compared to random routing, judging by the number of requests that remain in the queues at the dyno level.
- Smart routing results in a significantly shorter average queue length, regardless of the workload.

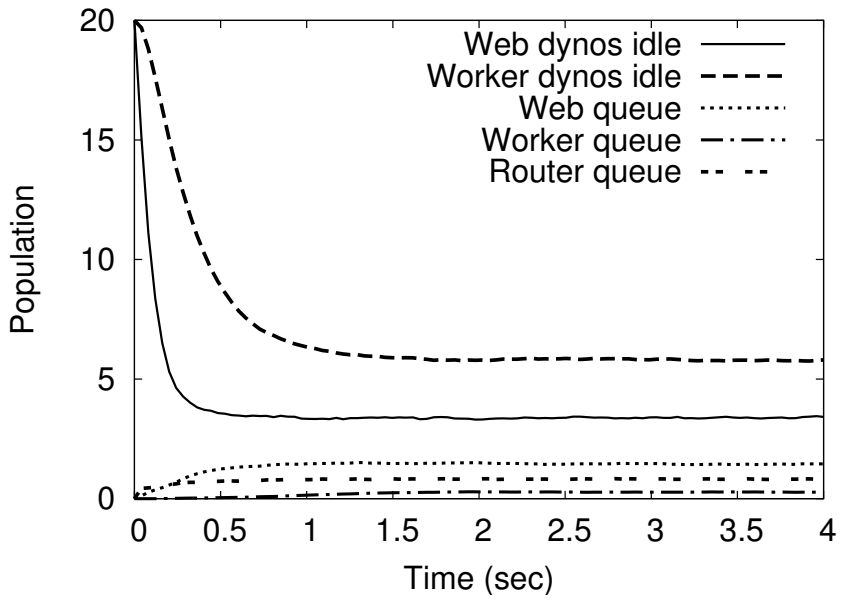
Experimentation with the system size

- The previous results show that in a medium-sized system there is a significant difference in terms of performance between the two routing policies considered.
- We now investigate how many dynos are required to service 9000 requests per minute, translated into a request arrival rate of 150 per second which is the reported workload for Rap Genius.
- In this experiment, we consider a fixed arrival rate equal to 150, and experiment with the size of the system to determine how many dynos have to be leased, to minimise both the number of idle dynos and the queue length at the dynos.

$Random_{20:20}$ and $Smart_{20:20}$ results for $r_{request} = 150$



(a) Random routing



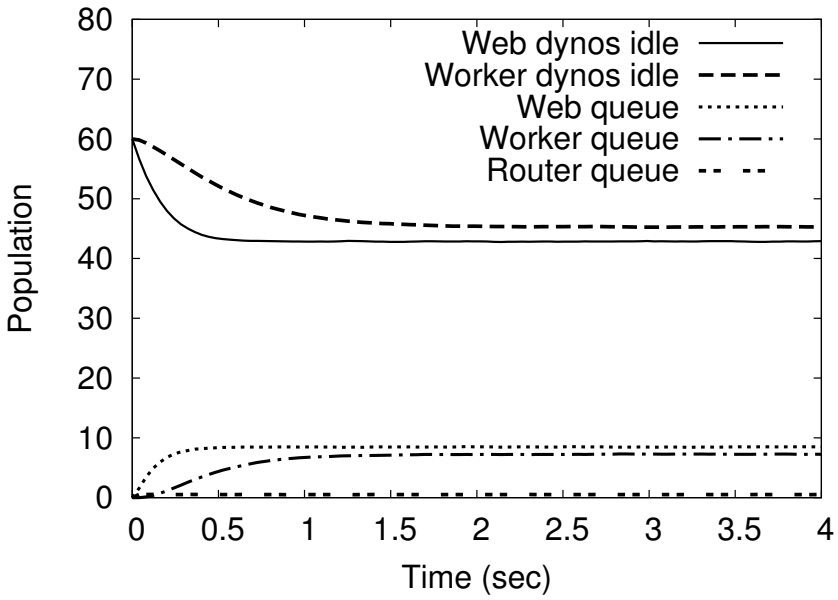
(b) Smart routing

In (b) the number of idle dynos is small, while the number of jobs queued at the dyno-level is acceptably small. However, in (a) the queue lengths are considerably larger.

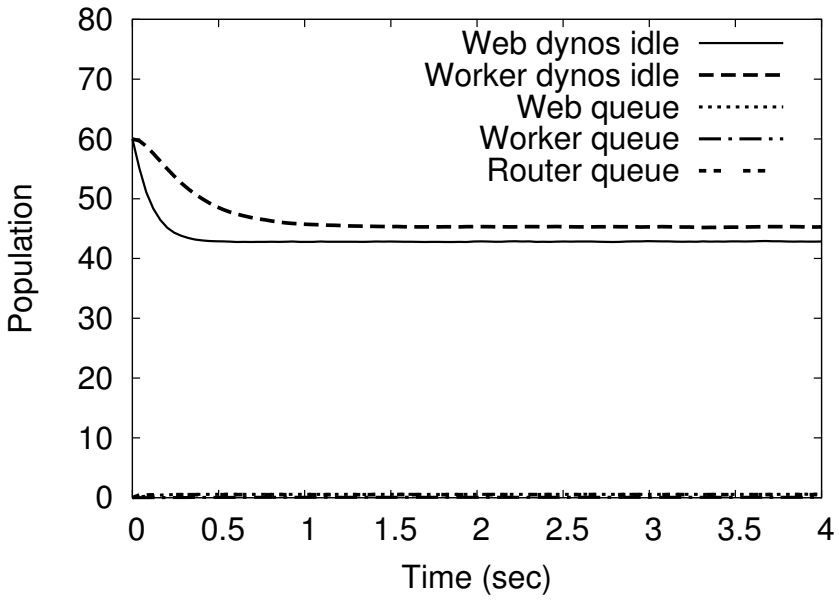
More experimentation with the system size

- We also considered a system with 60 web dynos and 60 worker dynos.
- For the random routing policy we have a relatively small but non-zero number of requests in the dyno queues. It appears that a random routing policy has a negative impact on the request waiting time, regardless of the size of the system.
- For the smart routing policy there almost no requests waiting. But note that, in both cases, a large part of the system remains idle, meaning that the use of 60 dynos of each kind is a waste of resources considering the given workload.

$Random_{60:60}$ and $Smart_{60:60}$ results for $r_{request} = 150$



(a) Random routing



(b) Smart routing

Summary

- The example used has been motivated by a particular incident involving the Rap Genius website, where a change in the routing policy has been reported to negatively affect the quality of service experienced by clients.
- Our model does not aspire to be an accurate representation of Rap Genius/Heroku. Nevertheless it provides a realistic representation of a system of that scale.
- Our experimentation shows that a smart routing policy results in a significantly smaller number of requests waiting to be serviced, compared to a random policy.