12 PEPA Case Study: Rap Genius on Heroku

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.

12.1 Heroku

Cloud computing is a term that describes the access to distributed hardware or software resources that are available as a service on demand, typically over the Internet. There are different service models but one of the most popular is PaaS, which provides clients with a customised solution stack including operating systems, programming languages, libraries, web servers, databases and software tools.

Heroku is a PaaS provider which offers 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*.

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.

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. The idea is that once a service request appears, Heroku will be responsible for assigning that request to one of the dynos that have been leased by the client, by following a routing policy as outlined in Figure 31.

The routing policy is the key component that we shall investigate in this case study. These 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.

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

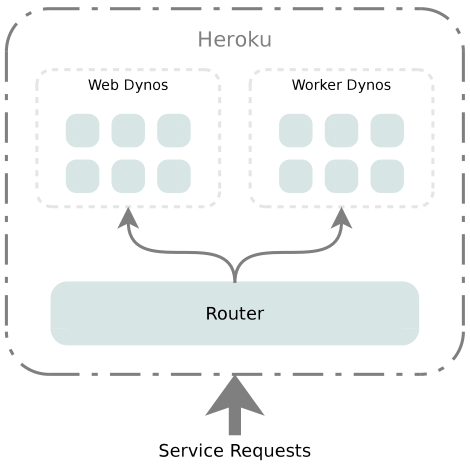


Figure 30: The basic structure of Heroku

12.1.1 Detecting and investigating performance problems

Each client must determine how many dynos should be leased. Of course, this depends on the workload. Naturally, the heavier the workload is, the more dynos will be needed. In the ideal case, every service should be tailed 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. 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.

Genius[2] (formerly Rap Genius[3]) is a website that aims to provide a critical and artistic insight into the lyrics of rap songs. The website users have access to content via HTTP requests, and they are able to add annotations to content. Rap Genius makes this service available via Heroku.

In the beginning of 2013, Rap Genius reported unusually long average response times, despite the large number of dynos leased by the website[4]. 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. 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*[5].

Here, our objective is not to assess the quality of service provided by Heroku, or recreate

[2]`http://genius.com/`
[3]`http://rapgenius.com/`
[4]`http://rapgenius.com/James-somers-herokus-ugly-secret-lyrics`
[5]`http://www.wired.com/2013/03/hieroku`

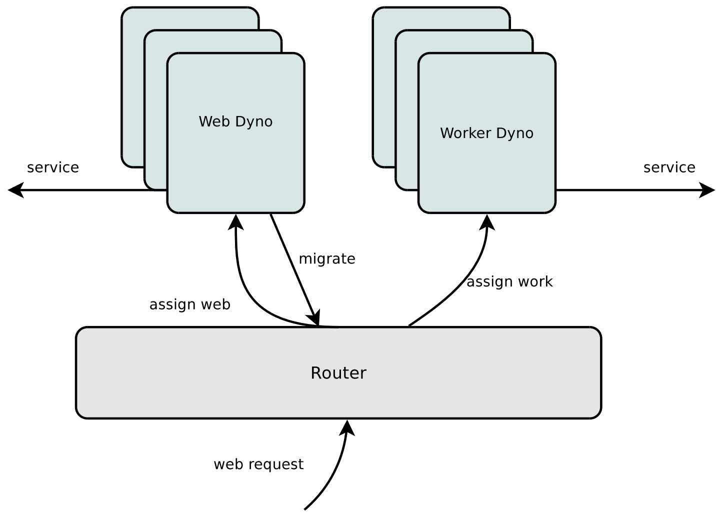


Figure 31: The Heroku configuration considered

the situation experienced by Rap Genius. Instead, we demonstrate how modelling with Markov chains, via a high-level modelling language, can capture the effect of different routing policies.

12.2 Modelling Heroku Routing with PEPA

Before modelling the routing policies, we describe the basic components and the interactions between them in an abstract way. As illustrated in Figure 32, the model we consider involves two classes of dynos, web and worker, and a router component.

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. This 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 of requests that are migrated is small; more specifically, we consider a migration probability equal to 1/9.

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. Table 6 summarises the rates of the events considered. The request arrival rate $r_{request}$ will control the assumed workload in the system. It is actually 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.

The service rate is dependent on the type of dyno. In both cases, we assume that service is broken down in two parts: the actual service and the response. The actual service part covers the amount of work that a dyno needs to do to produce a result — the

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

Table 4: The rate values used in the examples

service time depends on the type of the job. 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 $r_{response} = 20$.

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.

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.

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, the system router which keeps queues for requests coming from the web and migration requests from the web dynos for the worker dynos.

12.2.1 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 assign jobs to dynos, regardless of their state.

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* realises 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. Given that the response rate is significantly higher than the web service rate (see Table 6), this has limited impact on the availability of the dyno. 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.

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

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

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

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. So for the web requests we have:

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

where n denotes the maximum size for the corresponding queue. If the maximum size is reached, it is assumed that any new requests will be discarded until the queue is not full. Analogously, for migration requests we have:

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

Finally, the router will be the parallel composition of the components above.

Figure 33 gives the complete model of the random routing policy. Note that the model imposes a maximum dyno queue length of 1. The main reason behind this modelling choice is to keep the component state-space at relatively low levels, in order to avoid excessive state-space explosion. As we shall see later in Section 6.3, this model is adequate to observe the qualitative difference between a random and a smart routing policy.

$$
\begin{array}{rcl}
\mathit{WebDyno} & \stackrel{\mathit{def}}{=} & (\mathit{assign}_{\mathit{web}}, \top).\mathit{WebDyno}_0 \\
\mathit{WebDyno}_0 & \stackrel{\mathit{def}}{=} & (\mathit{service}, r_{\mathit{web}}).\mathit{WebDyno}_{0a} + (\mathit{migrate}, r_{\mathit{migrate}}).\mathit{WebDyno} \\
 & + & (\mathit{assign}_{\mathit{web}}, \top).\mathit{WebDyno}_1 \\
\mathit{WebDyno}_{0a} & \stackrel{\mathit{def}}{=} & (\mathit{response}, r_{\mathit{response}}).\mathit{WebDyno} \\
\mathit{WebDyno}_1 & \stackrel{\mathit{def}}{=} & (\mathit{service}, r_{\mathit{web}}).\mathit{WebDyno}_{1a} + (\mathit{migrate}, r_{\mathit{migrate}}).\mathit{WebDyno}_0 \\
\mathit{WebDyno}_{1a} & \stackrel{\mathit{def}}{=} & (\mathit{response}, r_{\mathit{response}}).\mathit{WebDyno}_0 \\
\\
\mathit{WorkerDyno} & \stackrel{\mathit{def}}{=} & (\mathit{assign}_{\mathit{worker}}, \top).\mathit{WorkerDyno}_0 \\
\mathit{WorkerDyno}_0 & \stackrel{\mathit{def}}{=} & (\mathit{service}, r_{\mathit{worker}}).\mathit{WorkerDyno}_{0a} + (\mathit{assign}_{\mathit{worker}}, \top).\mathit{WorkerDyno}_1 \\
\mathit{WorkerDyno}_{0a} & \stackrel{\mathit{def}}{=} & (\mathit{response}, r_{\mathit{response}}).\mathit{WorkerDyno} \\
\mathit{WorkerDyno}_1 & \stackrel{\mathit{def}}{=} & (\mathit{service}, r_{\mathit{worker}}).\mathit{WorkerDyno}_{1a} \\
\mathit{WorkerDyno}_{1a} & \stackrel{\mathit{def}}{=} & (\mathit{response}, r_{\mathit{response}}).\mathit{WorkerDyno}_0 \\
\\
\mathit{WebRouter}_0 & \stackrel{\mathit{def}}{=} & (\mathit{request}, r_{\mathit{request}}).\mathit{WebRouter}_1 \\
\mathit{WebRouter}_1 & \stackrel{\mathit{def}}{=} & (\mathit{request}, r_{\mathit{request}}).\mathit{WebRouter}_2 + (\mathit{assign}_{\mathit{web}}, r_{\mathit{assign}}).\mathit{WebRouter}_0 \\
\mathit{WebRouter}_2 & \stackrel{\mathit{def}}{=} & (\mathit{request}, r_{\mathit{request}}).\mathit{WebRouter}_3 + (\mathit{assign}_{\mathit{web}}, r_{\mathit{assign}}).\mathit{WebRouter}_1 \\
\mathit{WebRouter}_3 & \stackrel{\mathit{def}}{=} & (\mathit{request}, r_{\mathit{request}}).\mathit{WebRouter}_3 + (\mathit{assign}_{\mathit{web}}, r_{\mathit{assign}}).\mathit{WebRouter}_2 \\
\\
\mathit{WorkerRouter}_0 & \stackrel{\mathit{def}}{=} & (\mathit{migrate}, \top).\mathit{WorkerRouter}_1 \\
\mathit{WorkerRouter}_1 & \stackrel{\mathit{def}}{=} & (\mathit{migrate}, \top).\mathit{WorkerRouter}_2 + (\mathit{assign}_{\mathit{worker}}, r_{\mathit{assign}}).\mathit{WorkerRouter}_0 \\
\mathit{WorkerRouter}_2 & \stackrel{\mathit{def}}{=} & (\mathit{migrate}, \top).\mathit{WorkerRouter}_3 + (\mathit{assign}_{\mathit{worker}}, r_{\mathit{assign}}).\mathit{WorkerRouter}_1 \\
\mathit{WorkerRouter}_3 & \stackrel{\mathit{def}}{=} & (\mathit{migrate}, \top).\mathit{WorkerRouter}_3 + (\mathit{assign}_{\mathit{worker}}, r_{\mathit{assign}}).\mathit{WorkerRouter}_2 \\
\\
\mathit{Random}_{N:M} & \stackrel{\mathit{def}}{=} & \mathit{WebDyno}[N] \parallel \mathit{WorkerDyno}[M] \underset{\mathcal{L}_{\mathit{random}}}{\bowtie} (\mathit{WebRouter}_0 \parallel \mathit{WorkerRouter}_0) \\
\\
\text{where } \mathcal{L}_{\mathit{random}} & = & \{\, \mathit{assign}_{\mathit{web}}, \mathit{assign}_{\mathit{worker}}, \mathit{migrate} \,\}
\end{array}
$$

Figure 32: PEPA model for random Heroku routing with N web dynos and M worker dynos.

12.2.2 Smart Routing Policy

In the smart routing case the dyno components are not substantially different from those in the random routing case, as both web and worker dynos are characterised by 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. 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. For a $WebDyno_i$ component, which denotes an occupied web dyno with i requests in its local queue, jobs can only be enqueued as follows:

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

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

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

The choice between assignment or placement in the local queue is not under the dyno's control; it is responsibility of the routing policy.

The $WebDyno_{ia}$ and $WorkerDyno_{ia}$ remain unchanged.

The smart routing policy consists of directing a request to a dyno that is available. 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 involves a deterministic step, which is the dyno availability check. Such a deterministic behaviour cannot be directly modelled in PEPA. What we can do instead is to probabilistically favour assigning jobs to free dynos rather than placing them in queues. The idea is that the router will delay directing a request until a dyno is available. This delay should not be infinite however; if too many requests arrive, then the router will decrease its queue length by directing the requests to random dynos.

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.

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

If the queue length reaches its maximum size n, that probably 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.

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

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

$$\begin{aligned}
\mathit{WorkerRouter}_0 &\stackrel{\mathit{def}}{=} (\mathit{migrate}, \top).\mathit{WorkerRouter}_1\\
\mathit{WorkerRouter}_i &\stackrel{\mathit{def}}{=} (\mathit{migrate}, \top).\mathit{WorkerRouter}_{i+1}\\
&\quad + (\mathit{assign}_{\mathit{worker}}, r_{\mathit{assign}}).\mathit{WorkerRouter}_{i-1}\\
\mathit{WorkerRouter}_n &\stackrel{\mathit{def}}{=} (\mathit{migrate}, \top).\mathit{WorkerRouter}_n\\
&\quad + (\mathit{assign}_{\mathit{worker}}, r_{\mathit{assign}} \times 0.5).\mathit{WorkerRouter}_{n-1}\\
&\quad + (\mathit{enqueue}_{\mathit{worker}}, r_{\mathit{assign}} \times 0.5).\mathit{WorkerRouter}_{n-1}
\end{aligned}$$

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

To summarise, 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. Any new requests will have to 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; $\mathit{enqueue}_{\mathit{web}}$ and $\mathit{enqueue}_{\mathit{worker}}$ can only be performed if the corresponding router queue is full. The complete model for the smart routing policy is shown in Figure 34.

12.3 Evaluation of Routing Policies

In this section, we present some experimental results in order to compare the two routing policies, based on numerical analysis of the Markov process underlying the PEPA models. These models would also be amenable to discrete event simulation (to be discussed in subsequent lectures) and in that case the queue sizes could be increased.

12.3.1 Experimentation with the Workload

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

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. Figure 35 outlines the transient behaviour for request arrival rate equal to 40 sec^{-1}, or 2400 requests per minute. The data plotted depict how the average population of idle dynos and average local queue lengths change during the first four seconds of the system being online. We can see that after these four seconds, the system appears to be in steady state. Figure 35 (a) depicts results for the random routing policy, while the Figure 35 (b) depicts smart routing.

$$
\begin{aligned}
\mathit{WebDyno} &\stackrel{\text{def}}{=} (\mathit{assign}_{\mathit{web}}, \top).\mathit{WebDyno}_0 \\
\mathit{WebDyno}_0 &\stackrel{\text{def}}{=} (\mathit{service}, r_{\mathit{web}}).\mathit{WebDyno}_{0a} + (\mathit{migrate}, r_{\mathit{migrate}}).\mathit{WebDyno} \\
&\;+\; (\mathit{enqueue}_{\mathit{web}}, \top).\mathit{WebDyno}_1 \\
\mathit{WebDyno}_{0a} &\stackrel{\text{def}}{=} (\mathit{response}, r_{\mathit{response}}).\mathit{WebDyno} \\
\mathit{WebDyno}_1 &\stackrel{\text{def}}{=} (\mathit{service}, r_{\mathit{web}}).\mathit{WebDyno}_{1a} + (\mathit{migrate}, r_{\mathit{migrate}}).\mathit{WebDyno}_0 \\
\mathit{WebDyno}_{1a} &\stackrel{\text{def}}{=} (\mathit{response}, r_{\mathit{response}}).\mathit{WebDyno}_0
\end{aligned}
$$

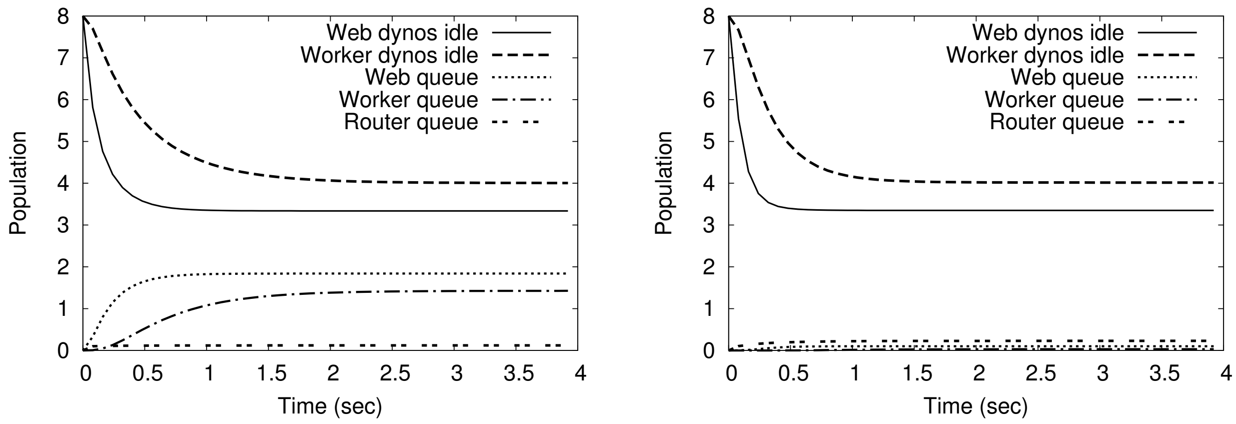
$$
\begin{aligned}
\mathit{WorkerDyno} &\stackrel{\text{def}}{=} (\mathit{assign}_{\mathit{worker}}, \top).\mathit{WorkerDyno}_0 \\
\mathit{WorkerDyno}_0 &\stackrel{\text{def}}{=} (\mathit{service}, r_{\mathit{worker}}).\mathit{WorkerDyno}_{0a} + (\mathit{enqueue}_{\mathit{worker}}, \top).\mathit{WorkerDyno}_1 \\
\mathit{WorkerDyno}_{0a} &\stackrel{\text{def}}{=} (\mathit{response}, r_{\mathit{response}}).\mathit{WorkerDyno} \\
\mathit{WorkerDyno}_1 &\stackrel{\text{def}}{=} (\mathit{service}, r_{\mathit{worker}}).\mathit{WorkerDyno}_{1a} \\
\mathit{WorkerDyno}_{1a} &\stackrel{\text{def}}{=} (\mathit{response}, r_{\mathit{response}}).\mathit{WorkerDyno}_0
\end{aligned}
$$

$$
\begin{aligned}
\mathit{WebRouter}_0 &\stackrel{\text{def}}{=} (\mathit{request}, r_{\mathit{request}}).\mathit{WebRouter}_1 \\
\mathit{WebRouter}_1 &\stackrel{\text{def}}{=} (\mathit{request}, r_{\mathit{request}}).\mathit{WebRouter}_2 + (\mathit{assign}_{\mathit{web}}, r_{\mathit{assign}}).\mathit{WebRouter}_0 \\
\mathit{WebRouter}_2 &\stackrel{\text{def}}{=} (\mathit{request}, r_{\mathit{request}}).\mathit{WebRouter}_3 + (\mathit{assign}_{\mathit{web}}, r_{\mathit{assign}}).\mathit{WebRouter}_1 \\
\mathit{WebRouter}_3 &\stackrel{\text{def}}{=} (\mathit{request}, r_{\mathit{request}}).\mathit{WebRouter}_3 \\
&\;+\; (\mathit{assign}_{\mathit{web}}, r_{\mathit{assign}} \times 0.5).\mathit{WebRouter}_2 \\
&\;+\; (\mathit{enqueue}_{\mathit{web}}, r_{\mathit{assign}} \times 0.5).\mathit{WebRouter}_2
\end{aligned}
$$

$$
\begin{aligned}
\mathit{WorkerRouter}_0 &\stackrel{\text{def}}{=} (\mathit{migrate}, \top).\mathit{WorkerRouter}_1 \\
\mathit{WorkerRouter}_1 &\stackrel{\text{def}}{=} (\mathit{migrate}, \top).\mathit{WorkerRouter}_2 + (\mathit{assign}_{\mathit{worker}}, r_{\mathit{assign}}).\mathit{WorkerRouter}_0 \\
\mathit{WorkerRouter}_2 &\stackrel{\text{def}}{=} (\mathit{migrate}, \top).\mathit{WorkerRouter}_3 + (\mathit{assign}_{\mathit{worker}}, r_{\mathit{assign}}).\mathit{WorkerRouter}_1 \\
\mathit{WorkerRouter}_3 &\stackrel{\text{def}}{=} (\mathit{migrate}, \top).\mathit{WorkerRouter}_3 \\
&\;+\; (\mathit{assign}_{\mathit{worker}}, r_{\mathit{assign}} \times 0.5).\mathit{WorkerRouter}_2 \\
&\;+\; (\mathit{enqueue}_{\mathit{worker}}, r_{\mathit{assign}} \times 0.5).\mathit{WorkerRouter}_2
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Smart}_{N:M} &\stackrel{\text{def}}{=} \mathit{WebDyno}[N] \parallel \mathit{WorkerDyno}[M] \underset{\mathcal{L}_{\mathit{smart}}}{\bowtie} (\mathit{WebRouter}_0 \parallel \mathit{WorkerRouter}_0) \\
\text{where } \mathcal{L}_{\mathit{smart}} &= \{\, \mathit{assign}_{\mathit{web}}, \mathit{enqueue}_{\mathit{web}}, \mathit{assign}_{\mathit{worker}}, \mathit{enqueue}_{\mathit{worker}}, \mathit{migrate} \,\}
\end{aligned}
$$

Figure 33: PEPA model for smart Heroku routing with N web dynos and M worker dynos

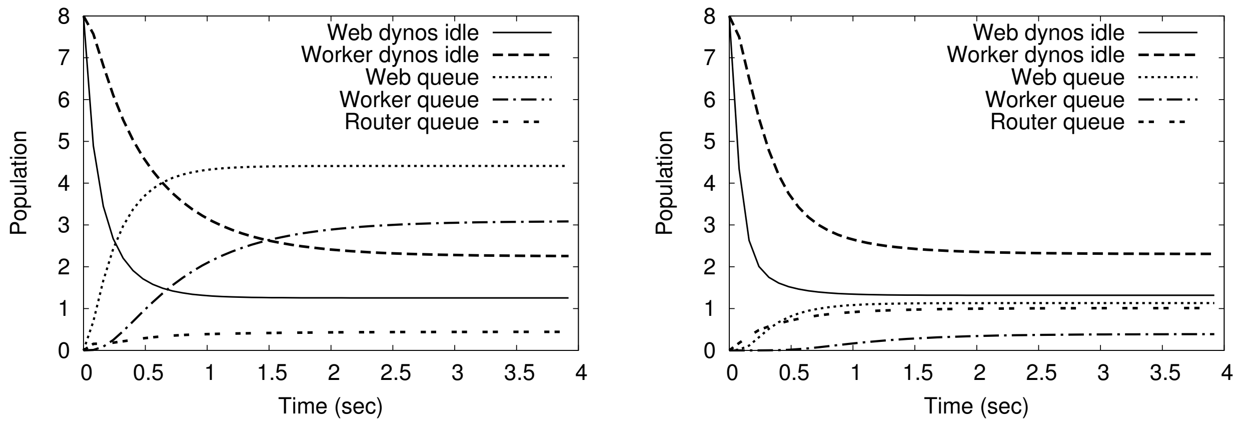


(a) Random routing (b) Smart routing

Figure 34: $Random_{8:8}$ and $Smart_{8:8}$ results for $r_{request} = 40$

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. This 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. In other words, the smart routing fully exploits the capacity of the Heroku configuration, in contrast with the random routing policy.

Figure 36 gives the results of the experiment in which we investigated how the routing policies are affected by a higher workload, by increasing the request arrival rate to 60 sec^{-1}, or 3600 requests per minute. 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 until they are serviced.



(a) Random routing (b) Smart routing

Figure 35: $Random_{8:8}$ and $Smart_{8:8}$ results for $r_{request} = 60$

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.

12.3.2 Experimentation with the System Size

The medium-sized system that we have examined in the previous section has shown that there is a significant difference in terms of performance between the two routing policies considered. Our objective now is to investigate how many dynos are required to service 9000 requests per minute, translated into a request arrival rate of 150 sec^{-1} which is the reported workload for Rap Genius. In this experiment, we consider a fixed arrival rate equal to 150, while we experiment with the size of the system, in order to determine how many dynos have to be leased, to minimise both the number of idle dynos and the queue length at the dynos.

Figure 37 outlines the transient behaviour for 20 web dynos and 20 worker dynos. Each sub-figure describes how the average population of idle dynos and the average queue lengths at the dyno level change through time. More specifically, in Figure 37 (b) we see that we have only a small number of idle dynos, while the number of jobs queued at the dyno-level remains small. Therefore, the system of this size has been found to be adequate to service 9000 requests per minute using the smart routing policy. According to Figure 37 (a) however, the queue lengths are considerably larger for the random policy.

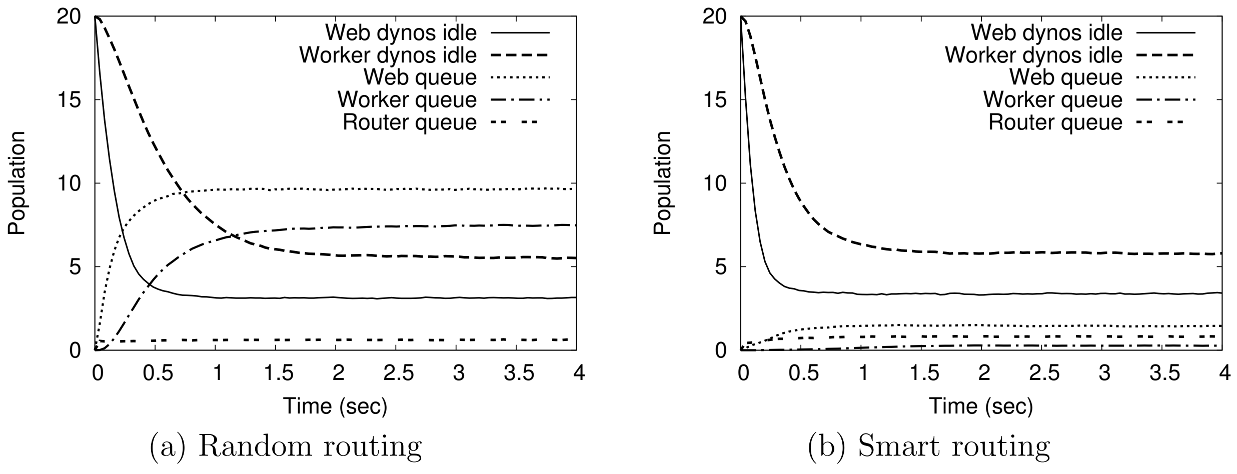


Figure 36: $Random_{20:20}$ and $Smart_{20:20}$ results for $r_{request} = 150$

We have also considered a system with 60 web dynos and 60 worker dynos, whose results are summarised in Figure 38. For the random routing policy in Figure 38 (a), 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. The picture is quite different for the smart policy in Figure 38 (b), where almost no requests are 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.

Thus, a system of 20 web and 20 worker dynos featuring a smart routing policy should be enough to service the typical workload of a website such as Rap Genius. Replacing the smart policy with a random policy will increase the number of dynos required to service the same workload at the same rate, and or will diminish the quality of service provided to the clients if the number stays the same.

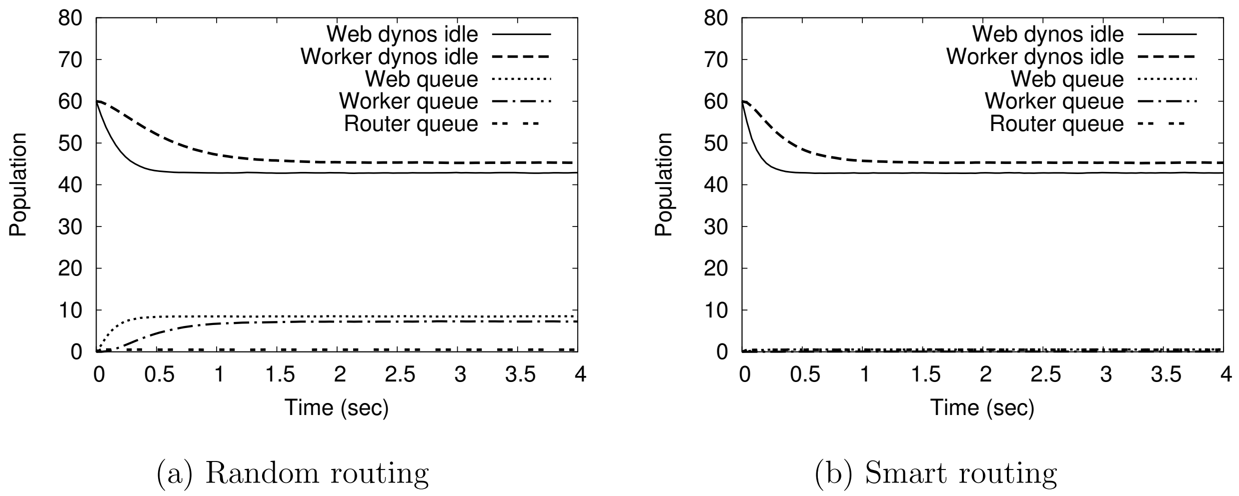


Figure 37: $Random_{60:60}$ and $Smart_{60:60}$ results for $r_{request} = 150$

12.4 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.

Jane Hillston ⟨Jane.Hillston@ed.ac.uk⟩. February 20, 2017.