RC25090 (W1012-093) December 21, 2010
Computer Science

IBM Research Report

Regulating Concurrency vs. Regulating Rate, for CPU Overload Protection of Web Servers

Wolfgang Segmuller, Michael Spreitzer,
Malgorzata Steinder, Asser N. Tantawi
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Regulating Concurrency vs. Regulating Rate, For CPU Overload Protection of Web Servers

Wolfgang Segmuller, Michael Spreitzer, Malgorzata Steinder, and Asser N. Tantawi
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598
werewolf, mspreitz, steinder, tantawi@us.ibm.com

ABSTRACT

We compare two techniques of preventing server overload by controlling dispatch of mostly short transactional requests at the edge of a system of servers: (1) a mechanism that imposes a limit on the number of active requests and (2) a mechanism that imposes a limit on the rate at which requests are dispatched. While either technique will work well for a workload with low variation, the story gets more complicated if the workload has extreme variation in service time. We suppose a mechanism/policy separation, in which the limits imposed by a dispatch mechanism are set periodically based on policy considerations and current traffic characteristics. Since the dispatch mechanisms follow simple parameterized rules, they can not make ideal decisions for each individual request even with optimal parameter setting. We study the sensitivity of these mechanisms to the variation among requests handled by each mechanism in steady load scenarios where limits are constant or change occasionally as a result of policy-driven automation. The evaluation criterion is the quality of the regulation of the CPU utilization. We find that when the limits are manually set to fixed values neither dispatch mechanism is clearly better than the other. However, when matching automation of the limits is added then the rate-based dispatch mechanism produces more accurate results. On the other hand, the occupancy-based dispatch mechanism produces results that underload less often.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

Keywords

performance management, concurrency control, rate control, CPU overload protection, service differentiation

1. INTRODUCTION

A large body of work has been done in the area of server overload protection for web service platforms. This work gains new significance with the development of cloud computing platforms [1, 2] which host a large number of applications on shared resources and oftentimes charge users based on the resource utilization by their applications. These environments require mechanisms to accurately control resource utilization by an application or a customer.

It is adequate to consider systems with the following simplified structure. Clients issue transactional requests, which are routed to a gateway. The gateway performs various management functions, one of which is our concern here: a mechanism to judiciously dispatch the requests to the server (there is just one). The server handles a request, then returns a reply to the dispatcher, which returns the reply to the relevant client. The management goal is to achieve a given target level of CPU utilization on the server — or, when the load collectively offered by the clients is too low for that, to come as close as possible. We suppose that in the course of serving a request a server alternates between doing two things: (1) local CPU work, and (2) making a synchronous request on (i.e., waiting for the reply from) some other service of larger capacity. In particular, the response time from that other service does not strongly depend on the management of the first server (over the range of what actually happens on that first server).

This is a simplification of more realistic systems, in which requests may be rejected for the sake of overload protection and there are: multiple server processes on multiple server machines, multiple tiers, multiple gateways, multiple service classes, multiple local computational resources, and multiple interacting applications spread among the servers. The simplified system suffices to explore the topic of this paper, which is a core issue that appears in more complicated real systems. The management of those more complicated systems involves more issues, but includes the issue explored here.

We compare two techniques for preventing server overload by controlling dispatch of requests from the gateway to the server. The first mechanism imposes a limit on the number of requests that can be in process at the server concurrently. The second mechanism imposes a limit on the rate at which requests are dispatched.

Real workloads have some variability in what it takes to serve their requests. This can include variation, from request to request, in (a) the amount of local CPU work done and/or (b) the amount of time spent waiting on replies from other services. If the level of variability is modest then either of the techniques we study would work well. However, some workloads with extreme variability are managed very poorly by either or both of these techniques. In this paper we study how the two techniques react to some very problematic workloads.

We suppose a separation between a mechanism and a policy. Both studied dispatch mechanisms are components of a feedback control system that periodically adjusts dispatcher settings based on policy considerations and current traffic characteristics. In each period, the controller measures averages of request throughput and performance from one or more previous periods, and adjusts the dispatcher settings for the next period. The control logic differs based on the type of mechanism used.

We study the concurrency-based and rate-based mechanisms using a steady state load, managed in either of two modes: (1) with a constant limit setting and (2) when the setting occasionally changes as a result of feedback-driven automation. In the constant setting mode the accuracy of a dispatch mechanism in maintaining the desired server utilization is affected primarily by variations between requests; periodic feedback control introduces a sensitivity to changes in monitored statistics between cycles.

One obvious evaluation criterion is the ability of the system to avoid exceeding a given utilization level. Server overload leads to service time degradation, but in most cases some overload can be tolerated. Another, possibly more important, citerion is the ability of the system to avoid unecessary queuing, which leads to server underload. Underload resulting from unecessary queuing has the effect of reducing the server capacity for serving requests, which can be worse than slow serving in a commercial context. On the other hand, in the context of managing a more complicated system with competing classes of traffic, excessive utilization by one class can lead to the additional problem of reduced utilization by another (possibly even more valuable) class.

The study in this paper is guided by our practical experience with real-life workloads, in which we have observed that web requests can vary significantly in their service time and resource utilization. Sometimes there are no distinguishing attributes that would allow such widely varying requests to be classified into different classes.

The paper is organized as follows. Prior work is briefly reviewed in Section 2. Then, we describe the two mechanisms for flow control: concurrency-based and rate-based in Section 3. The experimental setup is provided in Section 4. The experimental results are presented in two settings. First, we present in Section 5 the experimental results without the controller, i.e. through manual setting of the controls. Then, in Section 6, we present a description of the analysis used by the controllers and the results when the controllers are engaged.

2. PRIOR WORK

The problem of controlling the use of computer and communications resources by limiting the load goes back several decades. For example, in the area of operating systems the number of concurrent active jobs is limited to a value known as the multiprogramming degree in order to avoid overloading memory and CPU resources [4]. This is an example of a concurrency control mechanism to avoid overload. Other examples are limiting the number of connections to a database system and the number of threads in web and application servers. In the area of data communication, the number of packets between a pair of sender and receiver is limited by a window size in order to control the flow and avoid overloading the links and nodes [3]. Examples of rate control mechanisms are also found in data communications systems where rate limiters in the form of leaky and/or token buckets are used to control traffic rates so as not to overload network resources. The values of the limits, whether they are concurrency levels or maximum rates, are either statically configured or dynamically adjusted according to fluctuations in traffic characteristics and resource dynamics. Typically, such systems are modeled and analyzed for the given control mechanism, e.g. [10], rather than contrasting concurrency and rate control mechanisms in an attempt to understand the nature of traffic that would make one mechanism more suitable than the other. The study in this paper is a step in that direction, focused on CPU overload in web serving systems.

A large body of work has been conducted in the related areas of overload protection, resource allocation, quality of service, service differentiation, and admission control for transactional requests to platforms providing web services. An example of rate control is the work of Kanodia and Knightly [9] where they consider the problem of achieving pre-specified response time targets to multiple streams of requests to web services. An admission control mechanism is placed in a front-end gateway to control the acceptance rate of requests. The admission control decision is made for each request and is based on statistical measurements of arrivals and services. The average and standard deviation measurements allow the controller to estimate the admission rates that satisfy the quality-of-service requirements, which are defined through percentiles of delays. The model is fairly simplistic in that it does not include a model of the workload nor resource (e.g. CPU) demand. Measurements of server resources utilization are not available to the admission controller and, hence, overload protection is achieved only indirectly by specifying smaller response time targets.

Another example of rate control is the work of Urgaonkar and Shenoy [14] where they consider the problem of CPU overload avoidance. They describe the Cataclysm system which involves a gateway function for admission control. Such a function consists of (1) guaranteeing a minimum request rate per class, (2) queueing of requests so as to achieve service differentiation by delaying lower priority requests, and (3) load balancing across servers. The first function is simply implemented using leaky buckets. The second function involves the approximate, periodical calculation of delays per class based on simple queueing models. Such calculation relies on an estimation of CPU work demand per request which is assumed to be collected offline. And, the third function is an implementation of a layer-7 load balancing.

The work of Pacifici et al. [12] describes a system employing a concurrency control mechanism. The number of concurrently executing requests for each class of service in the server complex is periodically determined using a queueing network model that is trained using observations of performance metrics during the prior period. The concurrency limits are computed in such a way to avoid CPU overloading and to provide a fair quality of service to the various traffic

classes. Concurrency limits are imposed in the gateways. Urgaonkar et al. [13] develop a queueing network model for multi-tier web systems and provide a method for accounting for concurrency limits.

A variation of the concurrency control mechanism is when requests are not homogeneous in their use of resources. For instance, consider a multitude of request types, where each type of requests is characterized by a given level of resource demand. This gives rise to a weighted concurrency control mechanism. An example of such a system is described by Elnikety et al. [5] where an admission controller and a scheduler run in a proxy in front of multi-tiered web applications. Based on online measurements of service times of different classes of requests, the load on the system is estimated and compared to a maximum load value that is obtained offline by varying the offered load until a peak performance is reached. The admission controller basically acts as a work limiter and makes its decisions at request arrival times based on the current anticipated load and the estimated load of the newly arrived request. The latter is only admitted if the resulting total system load does not violate the maximum load value.

Again, the above examples are specific in their mechanisms for controlling resource overload through limiting concurrency or rate. We are not aware of an investigation where these mechanisms are contrasted so as to identify the nature of traffic that would make one mechanism superior over the other.

3. DESCRIPTION OF CONTROL MECHANISMS

We first consider the two different mechanisms for dispatching requests from the gateway to the server. The we turn our attention to automating the setting of the dispatching parameters.

3.1 Gateway Mechanisms

3.1.1 Concurrency Control Gateway

In concurrency control, the dispatcher is given a limit on the number of requests that may be concurrently active in the server. We suppose that each request's response flows back through the dispatcher, or the dispatcher is otherwise informed of the completion of each request. The dispatcher can thus keep track of the active number of its requests. The dispatcher has a FIFO queue of requests. Whenever a request arrives from outside, it is first put into that queue. Whenever the queue is non-empty and the number of active requests is below the given limit, another request is dispatched to the server system.

3.1.2 Rate Control Gateway

In rate control, the dispatcher also has a FIFO queue; the dispatch criterion is a token bucket. The parameters are a rate (tokens per time) and a size (number of tokens). The bucket is continuously filled with tokens at the given rate, but can not hold more than its size (excess spills over the lip of the bucket). A dispatch happens whenever the queue is non-empty and the bucket holds at least 1.0 tokens. Note that this mechanism pays no attention to responses.

3.2 Automatic Control Mechanisms

In automatic mode there is a policy agent, or controller, that monitors certain aspects of the running system and periodically changes the parameters of the dispatcher. A policy agent's job can be broken down into the following parts. One is to create a model of the offered load. Another is to develop a performance model that relates dispatch parameter values with server utilization, for the modeled offered load. Finally, the performance model is solved to find the dispatch parameter values that correspond to the desired level of utilization. When the offered load is insufficient, the problem is to find dispatch parameter values that would be appropriate if the offered load were to suddenly increase.

3.2.1 Automatic Concurrency Control Mechanism

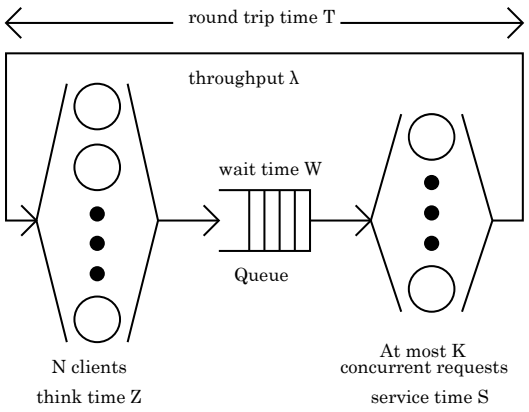


Figure 1: Model outline

Figure 1 outlines the queuing system used in both (a) modeling the offered load and (b) performance modeling. There is some number N of clients, each of which alternately makes a synchronous invocation and sleeps with a mean sleep time of Z. An invocation is queued and dispatched, by one of the mechanisms discussed earlier; the mean wait time is W, and the mean service time is S. The total round trip time T averages $Z + W + S$, and the N clients doing this collectively average a throughput of λ, which Little's Result tells us equals $N/(Z + W + S)$ in long term averages. The policy agent gets the statistics, which give sample averages for W, S, and λ; the load modeling problem is to invent N and Z that give a reasonable model of the offered load. This starts by using a case analysis, on the average utilization of the allocated concurrency in the preceding control cycle, to set a provisional estimate. If the average concurrency was less than 40% of the concurrency limit, then the provisional estimate is $N = \lceil 2(W + S)\lambda \rceil$ with Z to fit Little's Result; if greater than 95%, then $N = \lceil (W + S)\lambda \rceil$ and Z is fit to Little's Result; otherwise the provisional estimate is made by picking the N and Z that are within reasonable bounds and close to the intersection of Little's Result and the Machine Repairmen model [6] (which gives a set of recursive equations that relate those quantities — with one degree of freedom). The series of provisional Z estimates is subjected to exponential smoothing and then re-fitting to observation[1]

[1]setting N to the nearest integer related by Little's Result

to give the series of estimates.

One part of the performance model is simple: the utilization ρ of the outer server is proportional to the request throughput, and the constant of proportionality is what we call the *utilization factor* and denote by the variable α.

$$\rho = \lambda\alpha$$

The utilization factor is a characeristic of the traffic, which we presume may drift over time. The value of α is estimated on-line based on observations of ρ and λ. In reality the problem is more complex due to the presence of multiple classes of traffic [11]. The α estimates are smoothed over time to cope with various difficulties of doing this estimation. Thus, the α estimates do not depend on the latest interval's statistics as strongly as do the other quantities used by the performance agent.

The rest of the performance model is the Machine Repairmen model. Given N, Z, S, and K, it predicts values for λ and W. The policy agent solves for K incrementally [8] by considering successively larger values for K until it finds the largest that keeps $\lambda\alpha$ within the desired utilization limit. Note that this iteration is naturally limited to $K \leq N$ — there is no gain in allowing more concurrency than there are clients. If there is still power to spare at $K = N$ then additional increments in K are made, making the conservative assumption that increasing K by 1 will increase λ by $1/S$. To gain a little bit finer control, the considered values of K are not just whole numbers but rather multiples of 1/8. For non-integer K greater than 1, the Machine Repairmen results from the nearest two integer K are linearly interpolated; for $0 < K < 1$ the Machine Repairmen model is tweaked by using a single repairman, with S/K in place of S

3.2.2 Automatic Rate Control Mechanism

The load modeling used by the policy agent for a rate controlling dispatcher can also be explained by reference to Figure 1. It starts with Little's Result, $N = (Z+W+S)\lambda$. Because the control variable is a throughput limit rather than a concurrency limit, the Machine Repairmen model does not apply. Little's Result is only one equation, not enough to fix two unknowns with a single observation of $\langle W, S, \lambda\rangle$. Instead, two successive observations are used, $\langle W_1, S_1, \lambda_1\rangle$ from the previous cycle and $\langle W_2, S_2, \lambda_2\rangle$ from the latest cycle. We start with a provisional estimate of N_2, done in one of two ways depending on the relative amount of change in throughput ($|\lambda_2 - \lambda_1|/\lambda_2$). If that difference is relatively small then the provisional N_2 is $[(Z_1 + W_2 + S_2)\lambda_2]$, otherwise the following series of steps is done. First, solve for the Z that, together with some N, would fit both observations; this is

$$Z = \frac{(W_1 + S_1)\lambda_1 - (W_2 + S_2)\lambda_2}{\lambda_2 - \lambda_1}$$

Take that Z or zero, whichever is higher, and subject to exponential smoothing. Using the smoothed Z, set the provisional N_2 to $[(Z+W_2+S_2)\lambda_2]$. Finally, regardless of which case applied, N_2 is the maximum of 1 and the provisional N_2. Lastly,

$$Z_2 = \max(0, N_2/\lambda_2 - W_2 - S_2)$$

The performance modeling proceeds as follows. As in the concurrency control case,

$$\rho = \lambda\alpha$$

The remainder of the performance model predicts λ given the control setting — which is a limit on λ. Although the current traffic characteristics impose another limit, $N/(Z + S)$, this could change during the upcoming control cycle. So the model is simply that the achieved throughput will be the control setting. The policy agent solves for the limit λ by considering $\lambda = K\delta$ for a small δ and successive whole numbers K, looking for the largest that respects the desired utilization limit (this looks unnecessarily complex in this simple setting, but makes more sense in the real world of multiple service classes, multiple applications, multiple servers, multiple dispatchers).

4. EXPERIMENTAL SETUP

Both dispatch mechanisms perform well when workload is characterized by relatively small variance in service time and compute work done, such as that observed in exponential distribution. However, in practice, some workloads exhibit a lot higher variability: we have dealt with real life scenarios where workload parameters of a small fraction of all requests were orders of magnitude higher than the average. While it is reasonable to design a dispatch mechanism for the most common workloads, it is also important to understand how the design copes with workloads whose characteristics are uncommon and extreme.

We will compare the two dispatch mechanisms, in each of two scenarios, which represent extreme conditions modelled after those we have observed in real-life deployments. In each scenario the request population is a mixture of two kinds: 99% of the requests are "normal", 1% are "long" (i.e., have an exceptionally long service time). Both kinds of requests have exponentially distributed service times; the mean service time of the long requests is about two orders of magnitude larger than the mean of the normal requests. In the constant-ratio scenario, the amount of computational work needed to serve a request is proportional to its service time; in the constant-work scenario, the mean computational work needed to service a request is the same for both normal and long requests.

Notice that the chosen scenarios favor one or the other mechanism. Concurrency control is robust against variations in requests that maintain a constant ratio of service time to compute work done. Rate control is robust against variations in requests that change the service time but have no effect on compute work done. Our purpose is to quantify the performance of each mechanism in extreme scenarios that favor and disfavor its design.

The overall request rates in the experiments are relatively slow, making detailed instrumentation easy to do relatively cheaply.

to Z and the latest observation, then setting Z to the non-negative value most closely related by Little's Result to N and the observation

4.1 The Constant-Work Scenario

In this scenario we use a simple micro-benchmark that works as follows. A single server handles all (outer) requests. This server handles such a request by executing a loop that does some arithmetic; the number of iterations is exponentially distributed with a certain mean. Every so many iterations, there is a synchronous[2] request to an inner server. In particular, there are 3 inner requests generated by an outer request that does the mean number of iterations. Each inner server's service logic is simply an invocation of Java's `Thread.sleep()` operation, which is parameterized by the amount of time to pause the thread. For the normal outer requests, that inner sleep is 65 milliseconds in manual trials and automatic mode trials with about 1 long request per control cycle, 110 milliseconds in automatic mode trials with about 100 longs/cycle; for long outer requests, that inner sleep is 19.866 seconds. The inner requests are load balanced across a distinct batch of servers, keeping each one's CPU utilization low[3].

The outer requests are generated by a fixed number of clients; that number is 100. Each client runs a simple infinite loop; the body of the loop issues an outer request, reads the reply once it's available, and then does a stochastic pause. The length of the pause is 745 ms plus a random number of ms that has an exponential distribution and a mean of 800. In this way we ensure there is ample offered load to keep the outer server busy without having to worry about queue overflows.

4.2 The Constant-Ratio Scenario

This scenario uses a similar micro-benchmark. The total number of iterations in an outer request is exponentially distributed, but the two different kinds of requests have different means. The mean number of iterations for long requests is about 90 times the mean number of requests for the normal requests in most of the trials, 100 times larger in the low-target trials. Independently for each outer request a nominal number ν of inner requests is chosen uniformly at random from the interval $[0.6, 3.4]$. The "get interval" for the request is set to the quotient of the mean number of iterations (for the kind of request at hand) divided by ν. This is the number of iterations between synchronous calls to the inner server. From this, and knowledge of the phase at which the inner calls are made, is calculated the number n of inner requests that would be done if the outer request did the mean number of iterations for its kind (normal or long). For this outer request the length of the sleep done while serving an inner request is $[333/n]$ ms for normal requests, $[33400/n]$ ms for long requests, in most of the trials; $[430/n]$ and $[43000/n]$, respectively, in the low-target trials.

The outer requests are generated in the a similar fashion as in the constant-work scenario: there are simulated clients repeatedly issuing synchronous outer requests; in this scenario the number of simulated clients is small and they do not pause between invocations.

In both the constant-work and constant-ratio scenarios, in the manual mode trials and the automatic control trials with about one long request per control cycle, the outer requests are served on a machine with a single processor so that we can have a significant CPU utilization with a very low concurrency (which highlights the effects of the mixed nature of the workload). The server machines have two processors, but one was disabled. In the automatic control trials with about 100 long requests per control cycle, both processors were enabled.

[2]That is, the outer loop does not continue until the reply to this inner request is received

[3]This is required for good behavior of `Thread.sleep()`.

4.3 Experiments outline

We perform two kinds of experiments. First, we evaluate both dispatch mechanisms with both workloads with a static configuration of the control setting. The control settings are obtained manually by applying the modeling logic described in Section 3 to long-term workload averages, which are measured in an initial profiling stage. They are selected so as to obtain the average CPU utilization of 66%.

Then, we evaluate the dispatch mechanisms with automated control. We study two different lengths of control cycles.

All experiments run for several hours. During this time, we collect various aggregate statistics such as: request arrivals & dispatches & completions, average occupancy (number of outer requests active), average inner and outer service timse, average CPU utilization on each inner and each outer server. These statistics are collected every 15 seconds. In the evaluation we show both the 15-second raw measurements and their 25-minute averages. From each experiment we discard an initial warm up phase and only analyze data collected while the system is in steady state.

5. RESULTS IN MANUAL CONTROL

In this section we consider the following question. Supposing the parameter(s) of the dispatch mechanism are held fixed for a long time, how well can that mechanism cope with the variation in the requests?

5.1 Constant-Ratio workload

We applied concurrency control dispatch mechanism to the constant-ration workload. The stati concurrency limit was 2. In the experiment, the average throughput was about 1.62 requests/sec, which means the expected number of long requests per statistics period (15 sec) was about 1/4. Figures 2 and 3 show time series of the outer server's CPU utilization obtained in the experiment averaged over 15-second intervals and over 25-minute intervals, and a histogram of 15-second CPU utilization averages, respectively. These results show that with concurrency control mechanism, the CPU utilization is fairly constant, indicating that this mechanism is indeed robust in constant-ration scenario despite its high variance of service time. Figures 4 and 5 show the time series of the outer service times and a histogram of the 15-second service time averages, respectively. They show quite a lot of variation in service time: the distribution has quite a long tail, because sometimes the outer server is entirely occupied by long requests — whose completions are thus averaged with few or no normal requests.

With rate-control mechanism we set the rate limit to 1.4667 req/sec. The average throughput was about 1.5 req/sec,

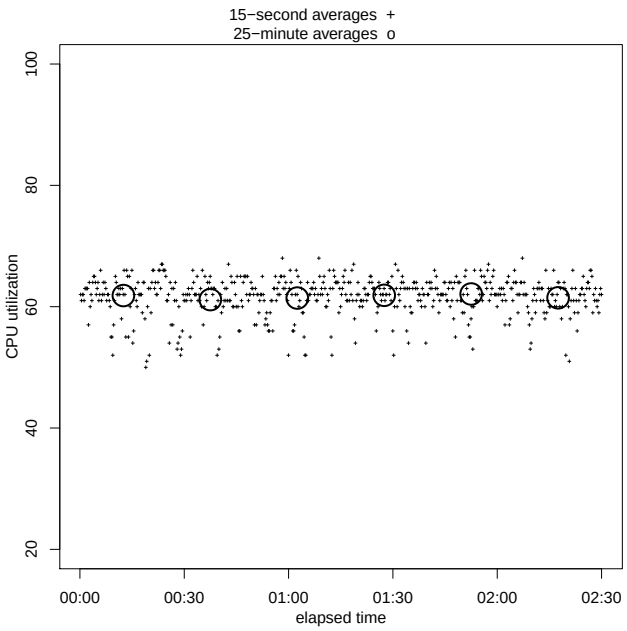


Figure 2: CPU Utilization in Constant-ratio, Concurrency Control, Manual Mode scenario

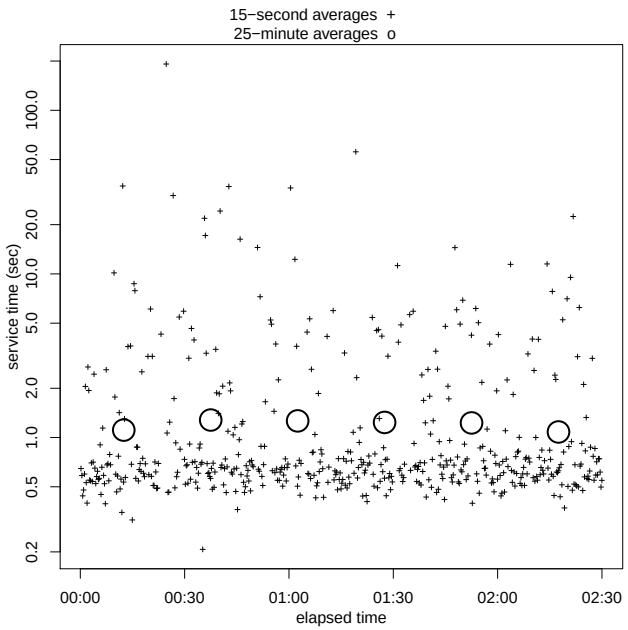


Figure 4: Average Service Time in Constant-ratio, Concurrency Control, Manual Mode scenario

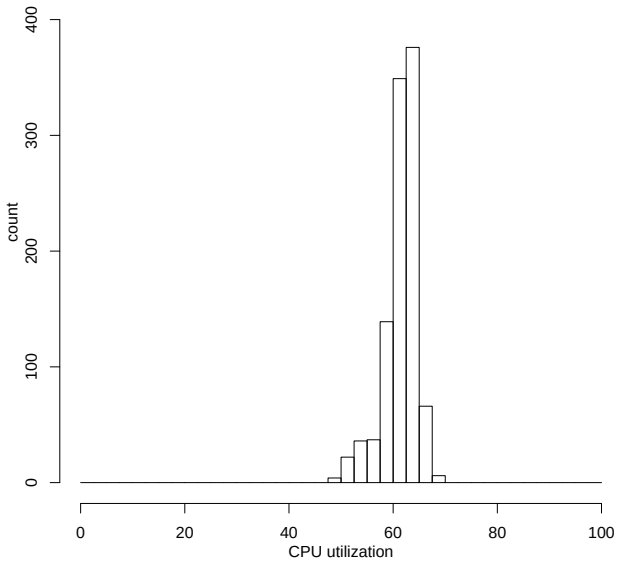


Figure 3: Histogram of 15-second CPU Utilization in Constant-ratio, Concurrency Control, Manual Mode scenario

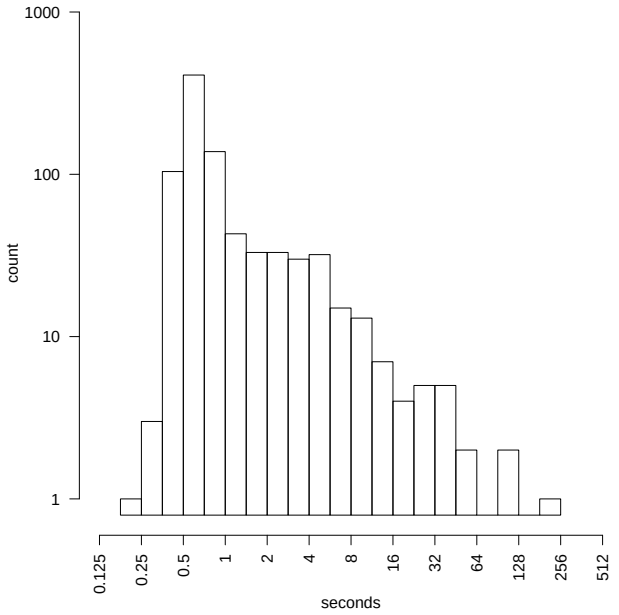


Figure 5: Histogram of 15-second Average Service Time in Constant-ratio, Concurrency Control, Manual Mode scenario

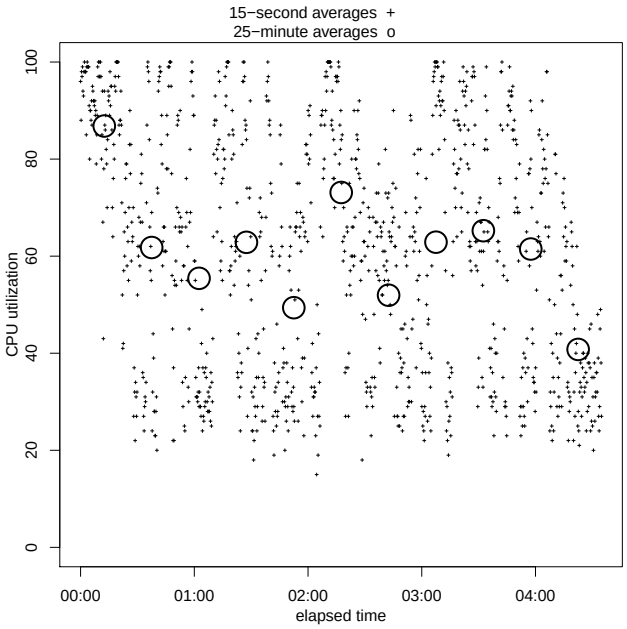


Figure 6: CPU Utilization in Constant-ratio, Rate Control, Manual Mode scenario

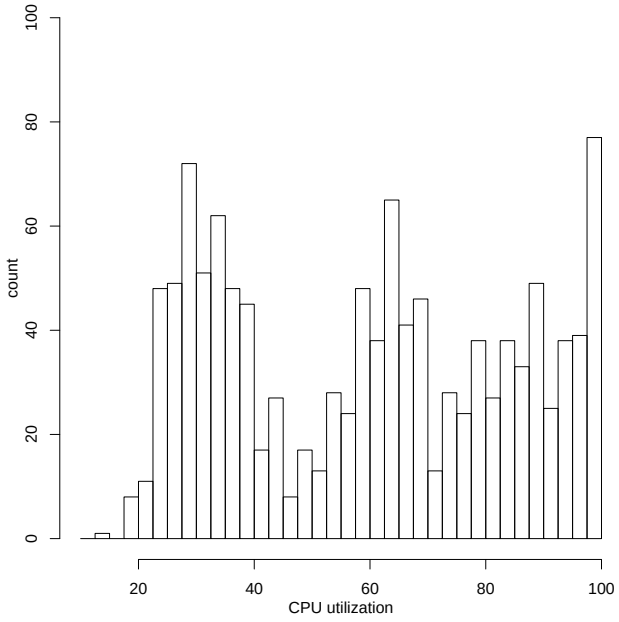


Figure 7: Histogram of 15-second CPU Utilization in Constant-ratio, Rate Mode, Manual Control scenario

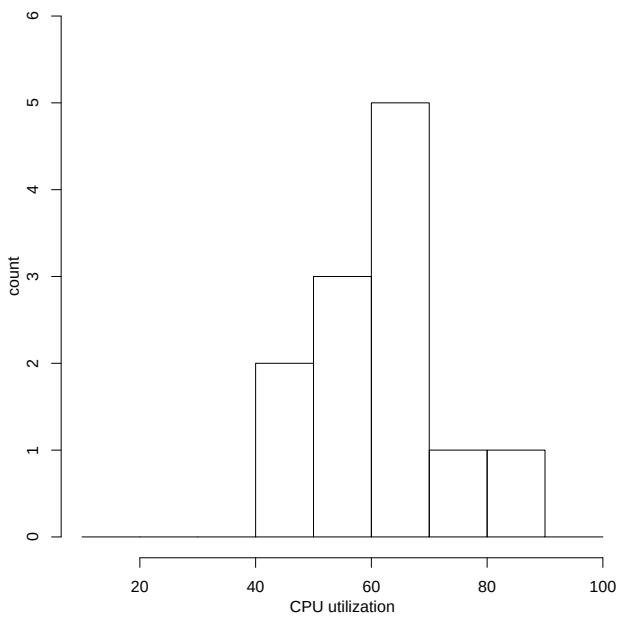


Figure 8: Histogram of 25-minute CPU Utilization in Constant-ratio, Rate Mode, Manual Control scenario

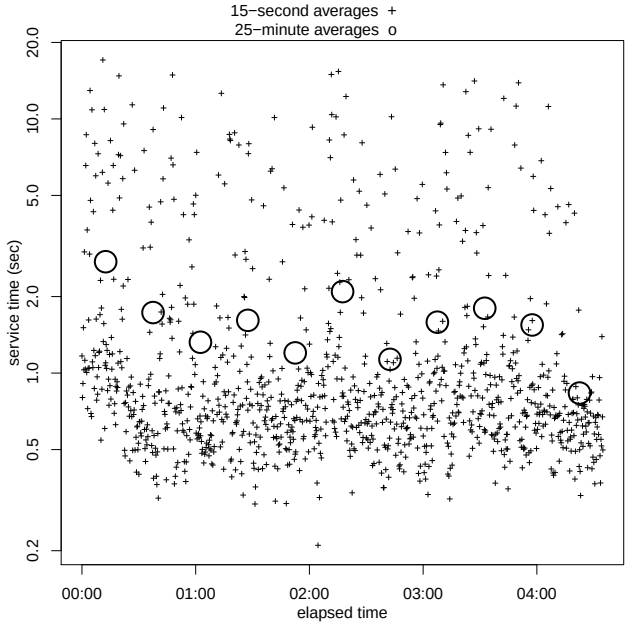


Figure 9: Average Service Time in Constant-ratio, Rate Control, Manual Mode scenario

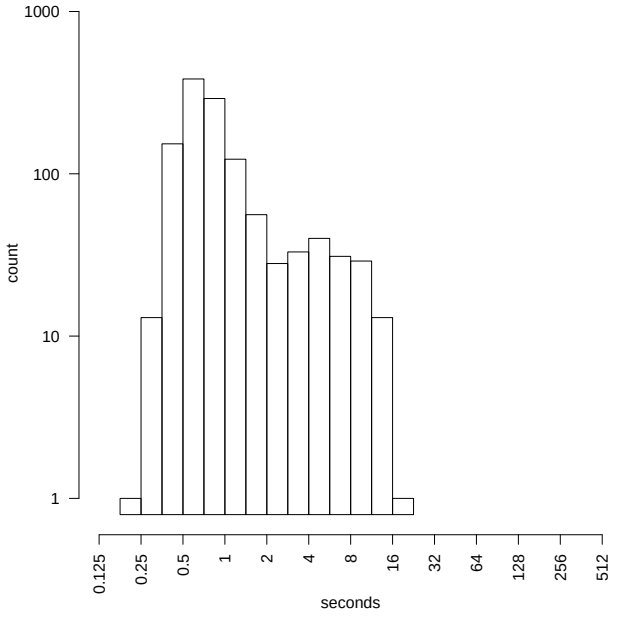


Figure 10: Histogram of 15-second Average Service Time in Constant-ratio, Rate Control, Manual Mode scenario

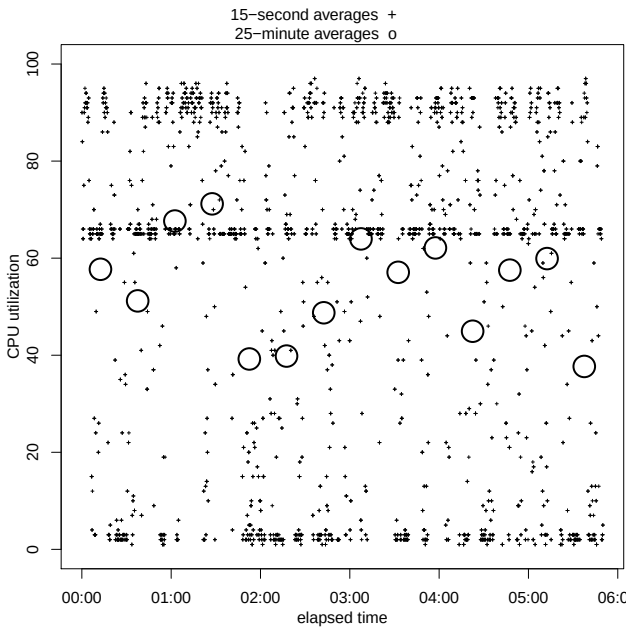


Figure 11: CPU Utilization in Constant-work, Concurrency Control, Manual Mode scenario

thus expecting about 1/4 long requests per 15-second interval. The time series of the outer server's CPU utilization presented in Figure 6 show a lot of variation. Notice that even 25-minute averages diverge from the target quite considerably, showing that the mechanism is ineffective in controlling CPU utilization by this workload. The histogram shown in Figure 7 reveals three clusters, which correspond to the number of long requests being served: low utilization cluster centered around 30% when no long requests are active, middle cluster centered around 65% when one long request is active, and very high utilization cluster reaching 100% when two or more long requests are active. Service time distribution shown in Figures 9 and 10 does not have the long tail observed with the concurrency-control mechanism (Figure 5), because long requests do not stop normal ones from entering the system and so every long service time is averaged with many normal ones.

The results of this study clearly demonstrate that for constant-ratio workload, the concurrency control mechanism performs significantly better than the rate-control mechanism when manual control is applied.

5.2 Constant-Work, Manual Control Results

5.2.1 Constant-Work, Manual Concurrency Control Results

We ran the constant-work scenario for several hours, with the concurrency limit set to 2, collecting the same kinds of statistics. The average throughput was about 1.47 requsts/sec, so the expected number of long requests per 15-second interval was about 0.22. Figure 11 shows time series of the outer server's CPU utilization, averaged over 15-second intervals and over 25-minute intervals. Figure 12

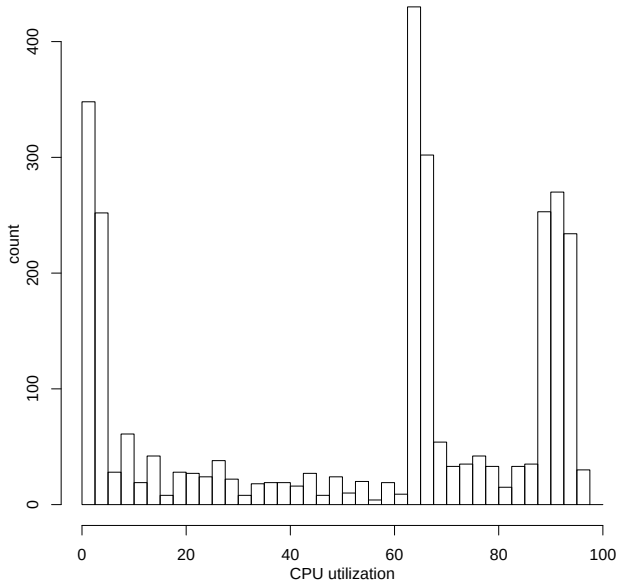


Figure 12: Histogram of 15-second CPU Utilization in Constant-work, Concurrency Control, Manual Mode scenario

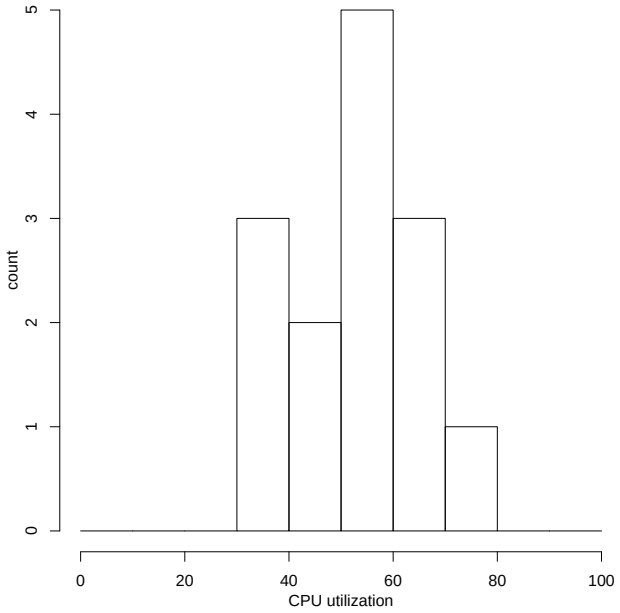


Figure 13: Histogram of 25-minute CPU Utilization in Constant-work, Concurrency Control, Manual Mode scenario

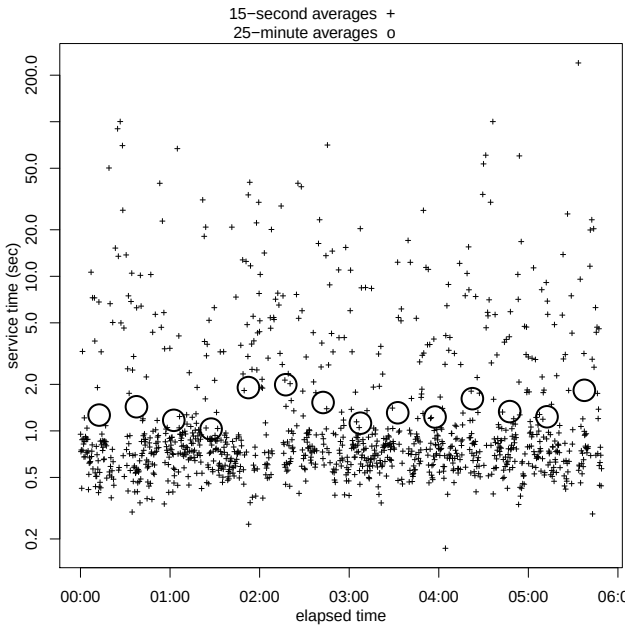


Figure 14: Average Service Time in Constant-work, Concurrency Control, Manual Mode scenario

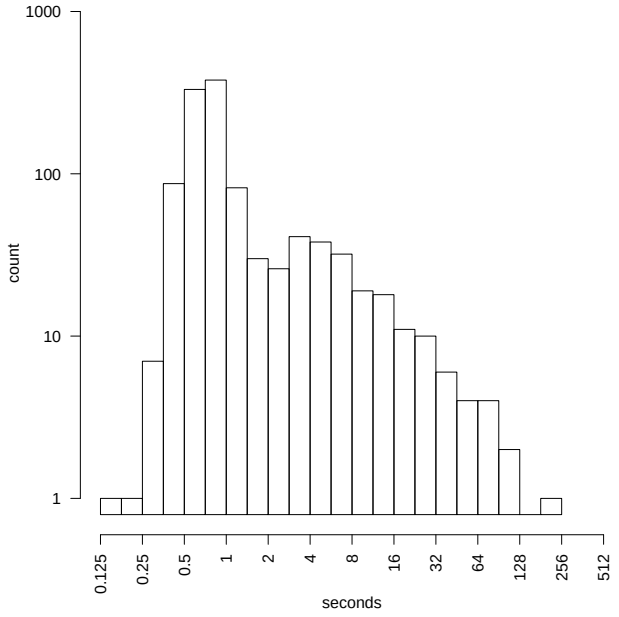


Figure 15: Histogram of 15-second Average Service Time in Constant-work, Concurrency Control, Manual Mode scenario

shows a histogram of 15-second CPU utilization averages from the entire run after warmup. The CPU utilization shows a lot of variation, showing that this mechanism is not robust in this scenario. There are three clusters of values corresponding to the number of long requests being served: around 90% CPU when no long requests are being served, around 67% when one is being served, and under 5% when two or more are being served. Figure 14 shows the time series of the outer service times, averaged over the same 15-second and 25-minute intervals as the CPU utilization; it shows quite a lot of variation. Figure 15 shows a histogram of the 15-second service time averages for the entire run after warmup. This distribution has a long tail, for the same reason as in the constant-ratio scenario.

5.2.2 Constant-Work, Manual Rate Control Results

We ran the constant-work scenario for three hours, with the rate limit set to 1.733 req/sec, collecting the usual kinds of statistics. The average throughput was about 1.7 requests/sec, thus expecting about 1 long request per minute. Figure 16 shows time series of the outer server's CPU utilization, and figure 17 shows a histogram of the 15-second averages. The alternating high-low pattern is an artifact of rounding: the original data are report as whole numbers of percent for 15-second intervals, and the histogram bins are $2\frac{1}{2}$ percent wide. All the 25-minute averages of CPU utilization were in the range of 60–70%. This distribution is much tighter than those of figures 7 and 12, but is broader than the one for concurrency control of the constant-ratio workload because dispatch rate control does not compensate for variation in service time. Figure 18 shows time series of outer service time averages, and figure 19 shows a histogram of 15-second averages, showing that there was indeed significant variation in this traffic. As in the constant-ratio rate

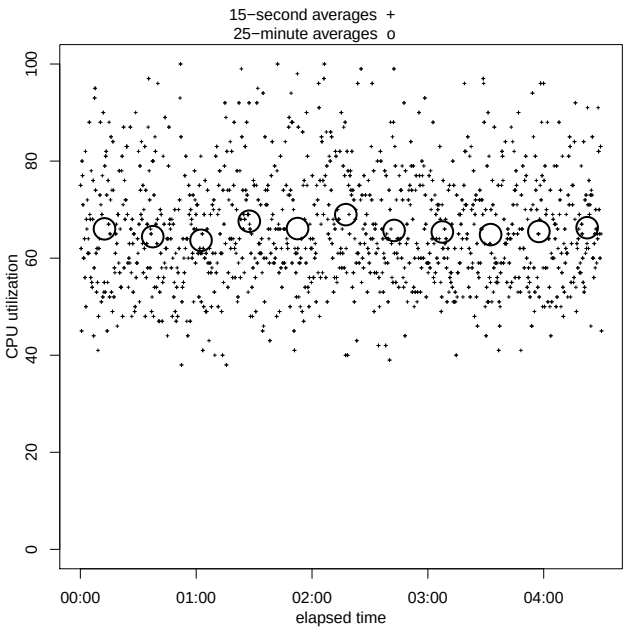


Figure 16: CPU Utilization in Constant-work, Rate Control, Manual mode scenario

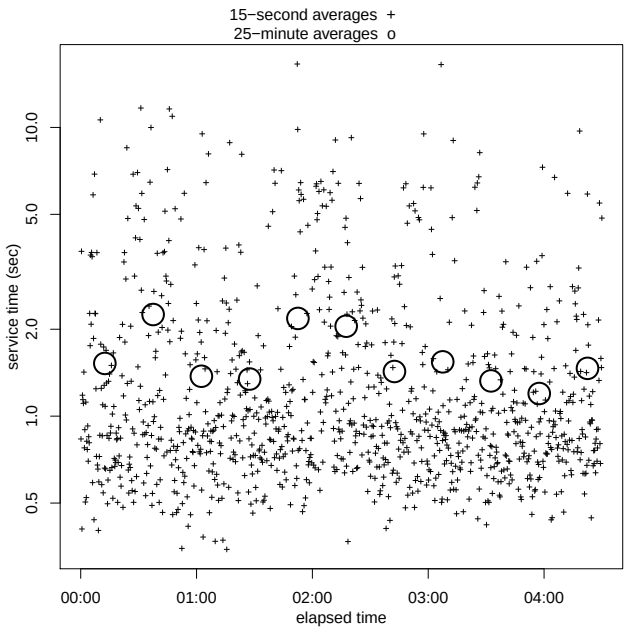


Figure 18: Service Time averages in Constant-work, Rate Control, Manual mode scenario

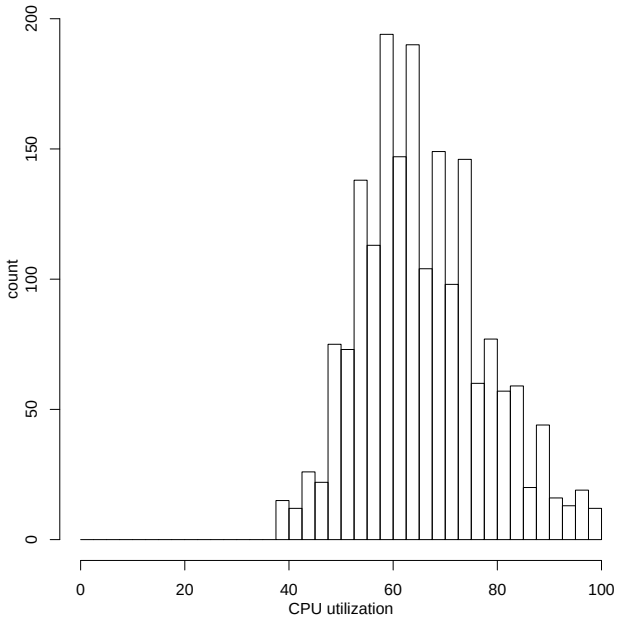


Figure 17: Histogram of 15-second Averages of CPU Utilization in Constant-work, Rate Control, Manual mode scenario

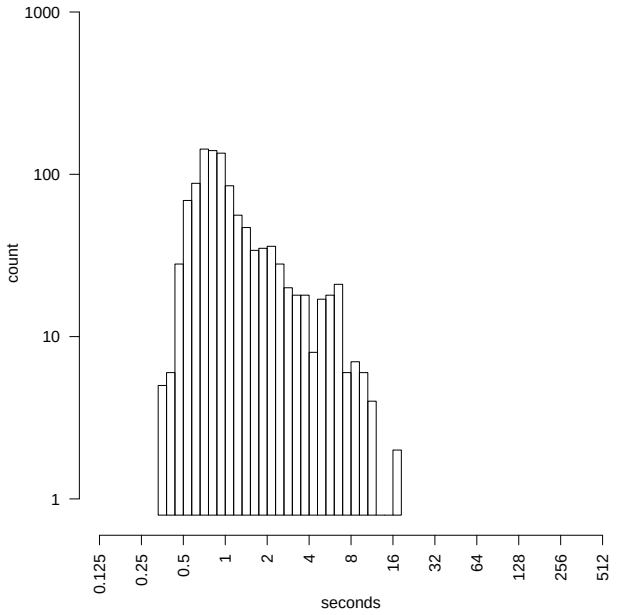


Figure 19: Histogram of 15-second averages of Service Time in Constant-work, Rate Control, Manual mode scenario

	concurrency gateway	rate gateway
constant ratio	61.73237 ± 3.113405	60.02592 ± 24.86503
constant work	54.25371 ± 34.35370	65.87227 ± 11.96114

Table 1: Mean+-Standard Deviation of 15-second averages of CPU utilization in manual control

control case, this distribution does not have a long tail.

5.3 Summary of Manual Control

Table 1 summarizes the results of the manual control experiments, showing mean and sample standard deviation for 15-second averages of CPU utilization. Figure 20 summarizes the results using histograms of 15-second averages of CPU utilization.

Figure 21 summarizes the results of the manual control experiments, using histograms of 25-minute averages of CPU utilization.

6. RESULTS IN AUTOMATIC CONTROL

The manual control experiments showed that each dispatch mechanism provides robust overload protection in one scenario and not the other, but the different mechanisms are better for different scenarios. We now turn to the question of what happens if a policy agent periodically adjusts the dispatch parameters.

Because the modeling depends on the statistics, which in turn depend on the length of the sampling period, we now double the number of scenarios. For each of the constant-work and constant-ratio cases we now test two statistics periods: one with an average of about one long request per period and one with an average of about 100 long requests per period.

6.1 Constant-Ratio, Automatic Mode, ∼1 Longs Per Cycle Results

We first consider some experiments with rather short control cycles, in which there is an average of about 1 long request per control cycle. The policy agent adjusts the gateway's control setting once per minute.

6.1.1 Constant-Ratio, Automatic Concurrency Control, ∼1 Longs Per Cycle Results

We ran the constant-ratio scenario with an average of roughly one long request per control cycle, for several hours. The CPU utilization target was set to 62%, which corresponds with the long-term average utilization seen under manual control with a concurrency limit of 2. Figure 22 shows the 15-second and 25-minute averages of CPU utilization on the outer server over the course of two and a half characteristic hours. Figure 23 shows a histogram of the 15-second CPU averages on the outer server, for the whole trial excluding the startup transient. The CPU utilization was not well managed: it was rarely near its target, and was often near 100%. Compare with figures 33 et. seq. Figure 25 shows the 15-second and 25-minute averages of the service times over the same 2.5 characteristic hours. Figure 26 shows a histogram of 15-second averages of service times, for the whole

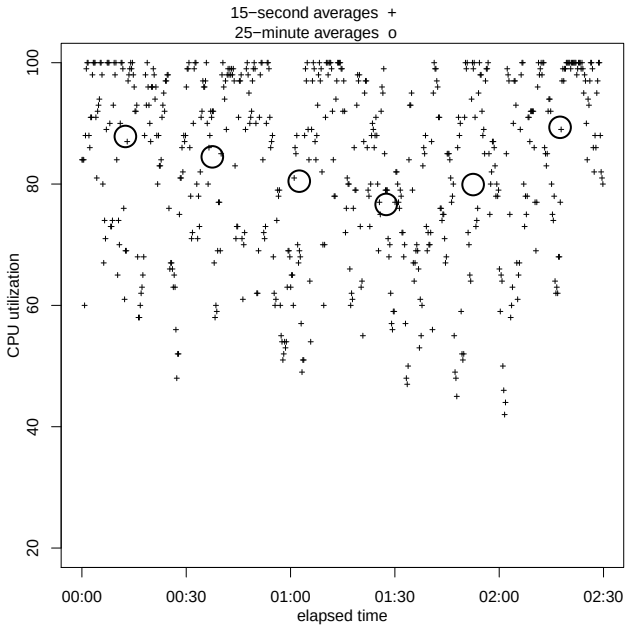


Figure 22: CPU Utilization in Constant-ratio, Concurrency Control, ∼1 longs/cycle, Automatic Mode scenario

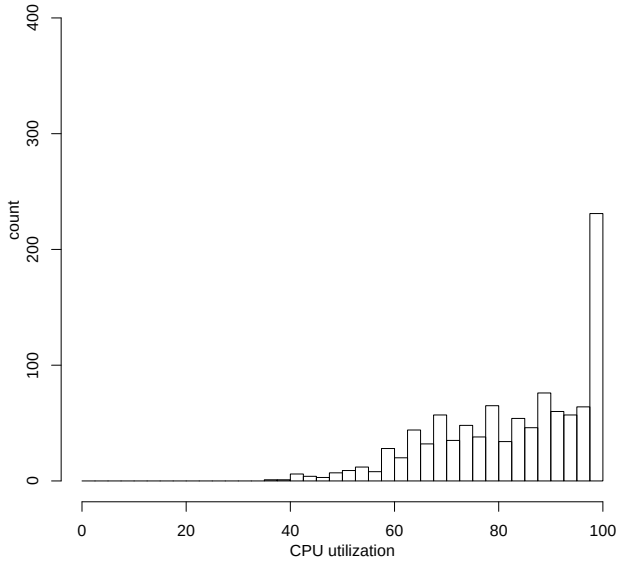


Figure 23: Histogram of 15-second average CPU Utilization in Constant-ratio, Concurrency Control, ∼1 longs/cycle, Automatic Mode scenario

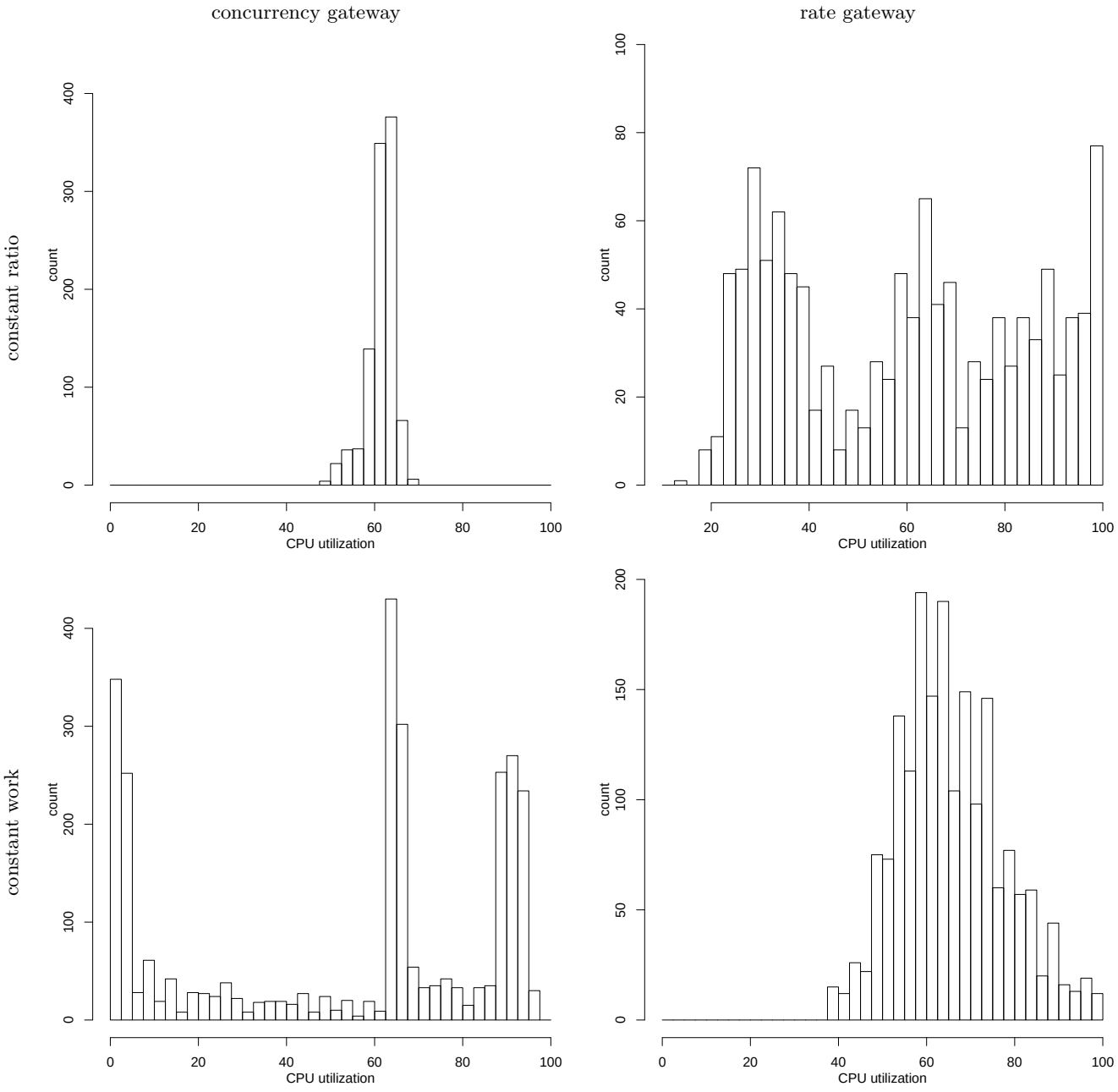


Figure 20: Histograms of 15-second averages of CPU utilization in manual control

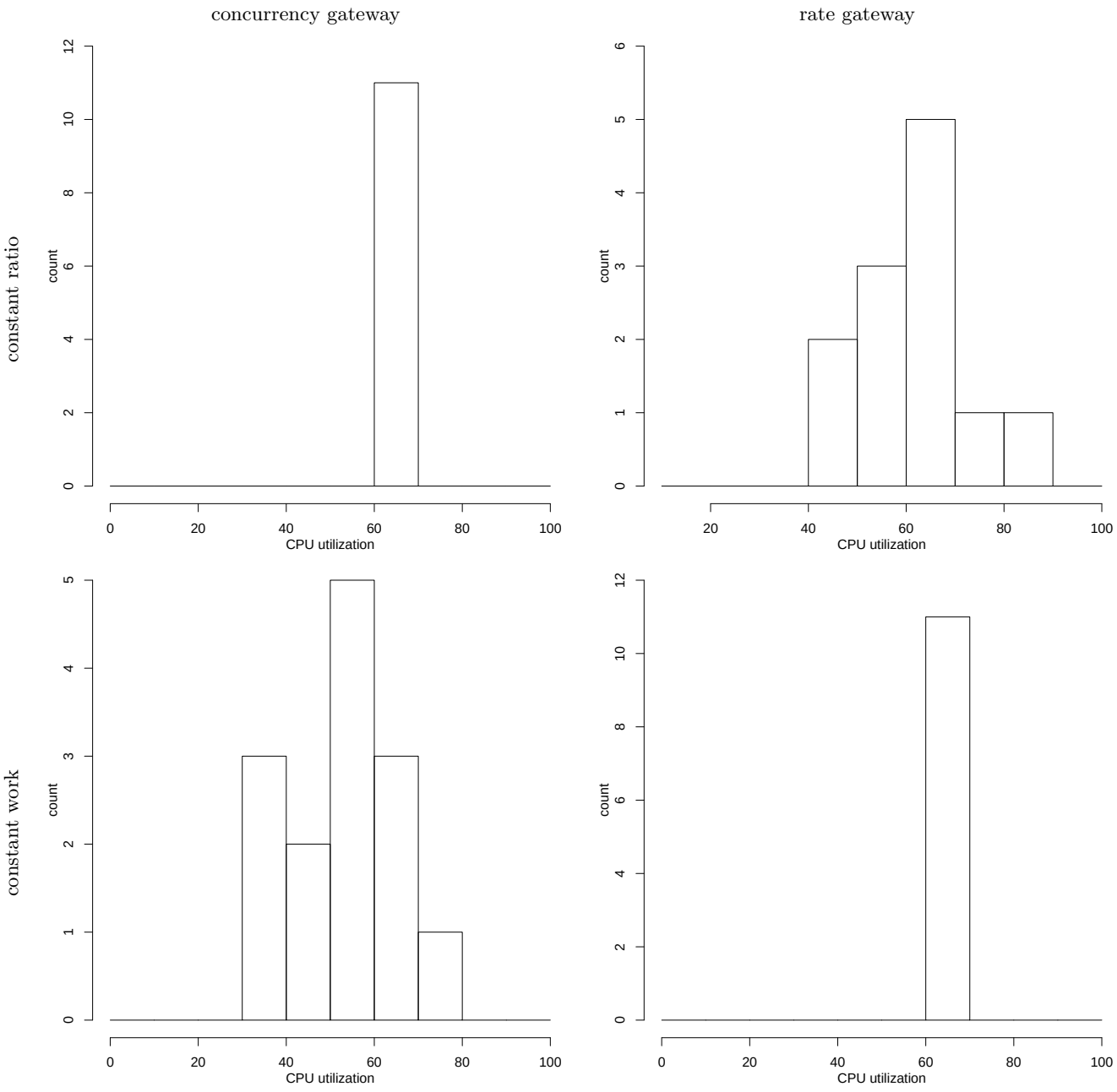


Figure 21: Histograms of 25-minute averages of CPU utilization in manual control

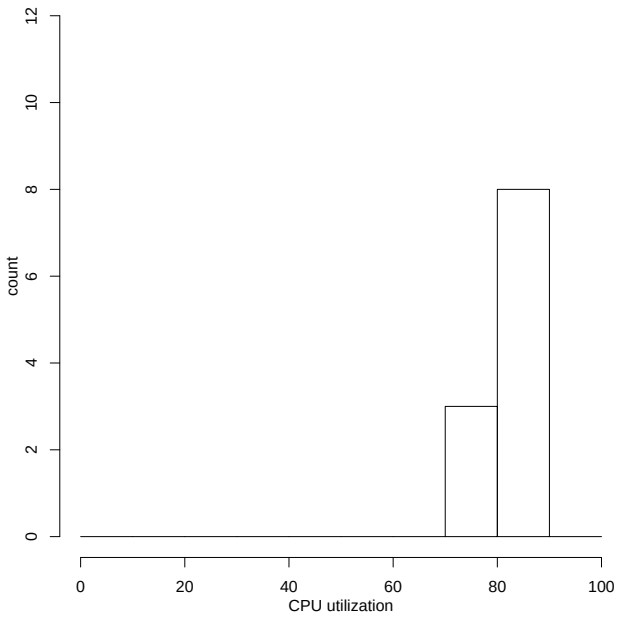


Figure 24: Histogram of 25-minute average CPU Utilization in Constant-ratio, Concurrency Control, ∼1 longs/cycle, Automatic Mode scenario

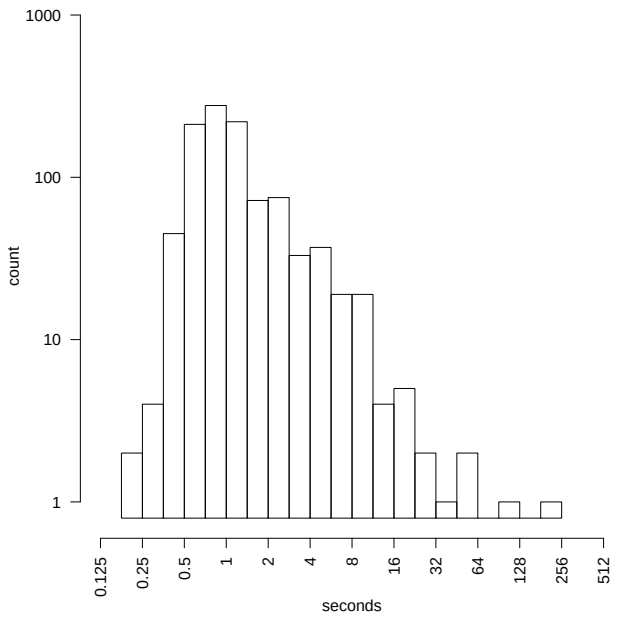


Figure 26: Histogram of 15-second-average Service Time in Constant-ratio, Concurrency Control, ∼1 longs/cycle, Automatic Mode scenario

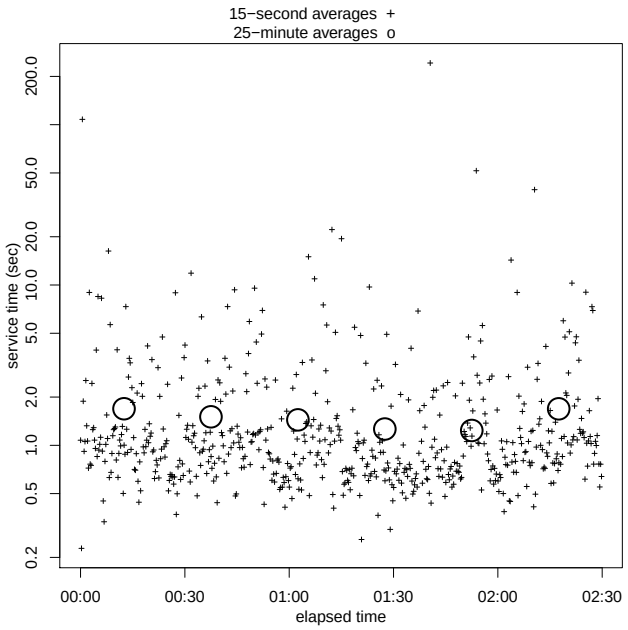


Figure 25: Average Service Time in Constant-ratio, Concurrency Control, ∼1 longs/cycle, Automatic Mode scenario

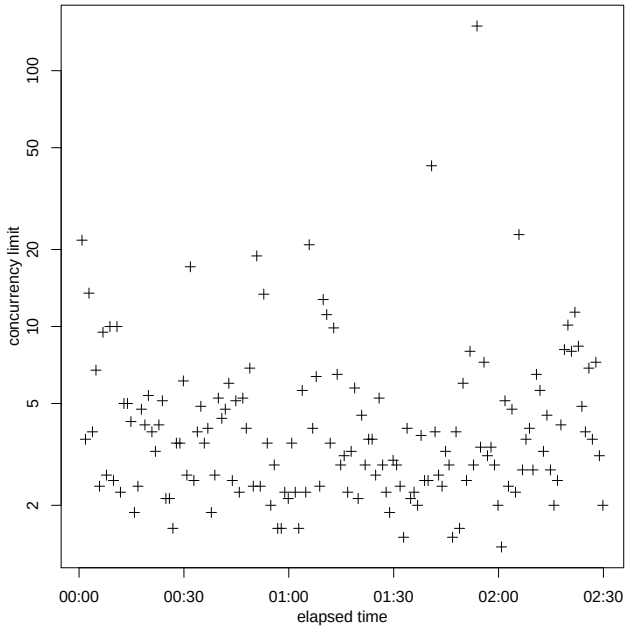


Figure 27: Gateway Control Setting in Constant-ratio, Concurrency Control, ∼1 longs/cycle, Automatic Mode scenario

trial excluding the startup transient. Figure 27 shows a time series of the control settings issued by the policy agent.

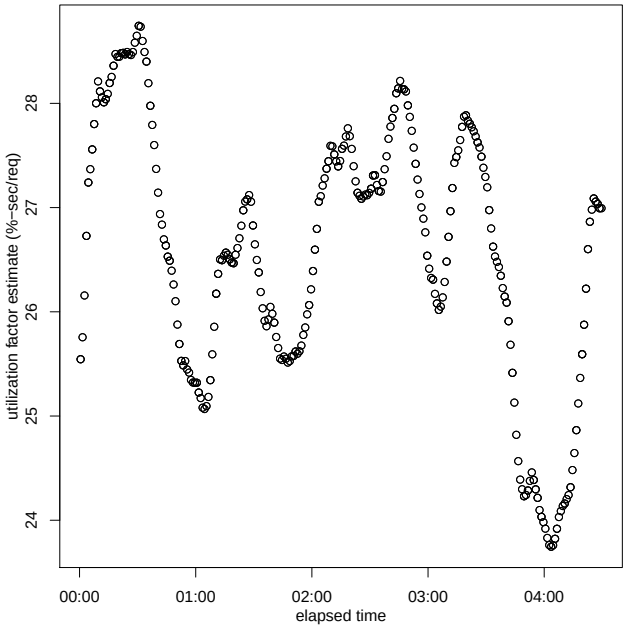


Figure 28: Utilization Factor estimates in Constant-ratio, Concurrency Control, ∼1 longs/cycle, Automatic Mode scenario

In this trial the on-line estimates of the utilization factor were consistently low. By comparing smoothed throughput and utilization we estimated that factor to be about 37 %-sec/req. Figure 28 shows the time series of the estimates made on-line; they are consistently low.

We then ran the policy agent using a configured utilization factor rather than on-line estimates. Figure 29 shows a time series of CPU utilizations, figure 30 shows a histogram of 15-second averages. In this configuration the CPU utilization was more often near the target. Figure 32 shows a time series of the control settings issued by the policy agent. Using the correct utilization factor made the control settings stay closer to the value corresponding to long-term averages. Even so, most of the 15-second averages of CPU utilization were far from the target.

We did a similar experiment but with a CPU utilization target of 42%. We tested three configurations: manual control with occupancy=2, automatic control with on-line utilization factor estimates, and automatic control with off-line utilization factor estimates. The on-line utilization factor estimates were about 3/4 of the correct (off-line estimated) value. Table 2 shows the resultant sample mean and standard deviation of 15-second averages of CPU utilization; it also shows histograms.

6.1.2 *Constant-Ratio, Automatic Rate Control, ∼1 Longs Per Cycle Results*

We ran the constant-ratio scenario for several hours, collecting the same kinds of statistics as before. The CPU

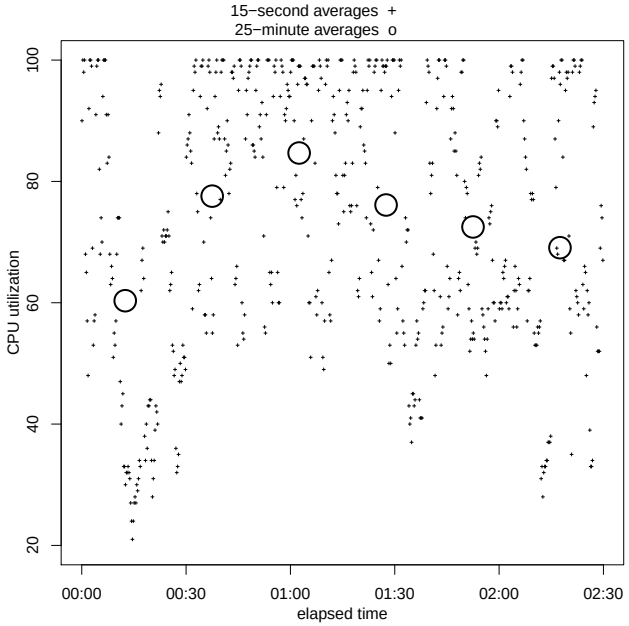


Figure 29: CPU Utilization in Constant-ratio, Concurrency Control, ∼1 longs/cycle, Automatic Mode, configured utilization factor

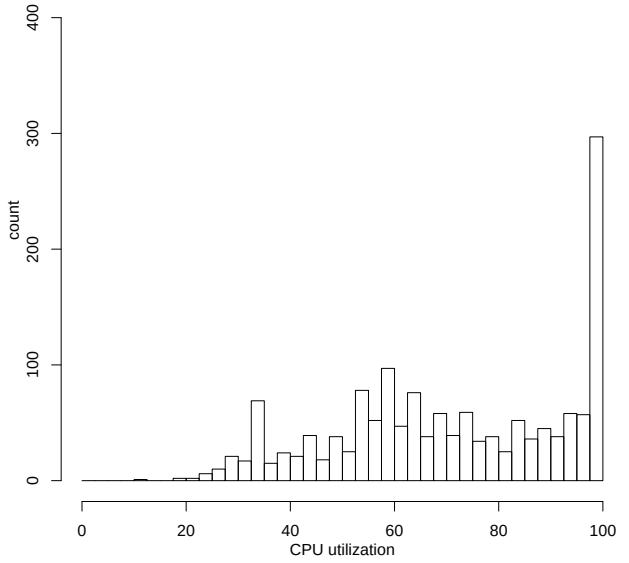
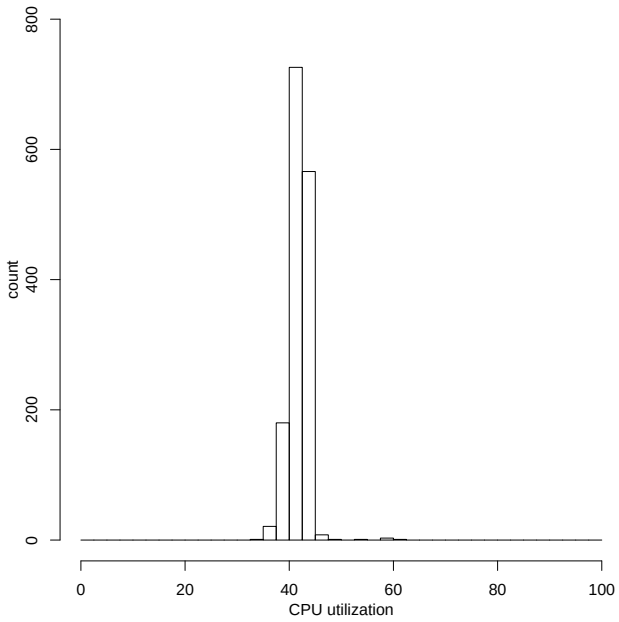
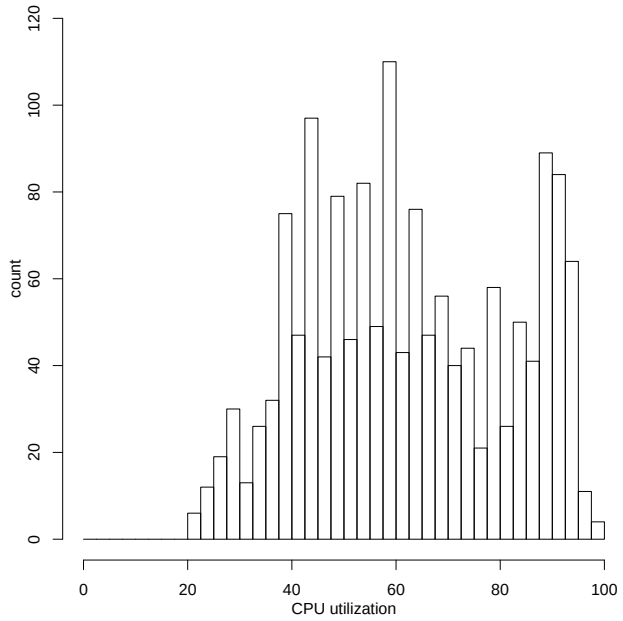


Figure 30: Histogram of 15-second average CPU Utilization in Constant-ratio, Concurrency Control, ∼1 longs/cycle, Automatic Mode, configured utilization factor

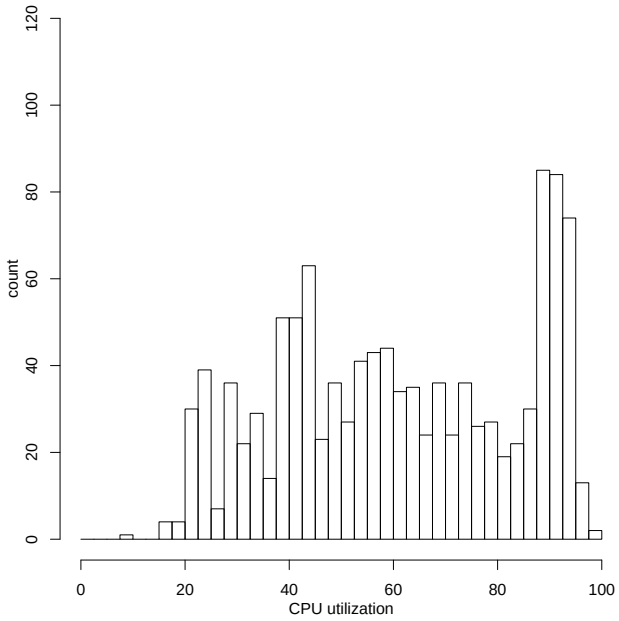
scenario	mean	std. dev.
manual	42.02984	1.779418
auto, on-line	62.50296	19.51831
auto, off-line	61.65669	22.96383



Manual mode



Automatic, on-line utilization factor estimates



Automatic, off-line utilization factor estimates

Table 2: Mean+-Standard Deviation and Histograms of 15-second averge CPU utilization with low target, constant rate, concurrency control, ∼1 longs/cycle

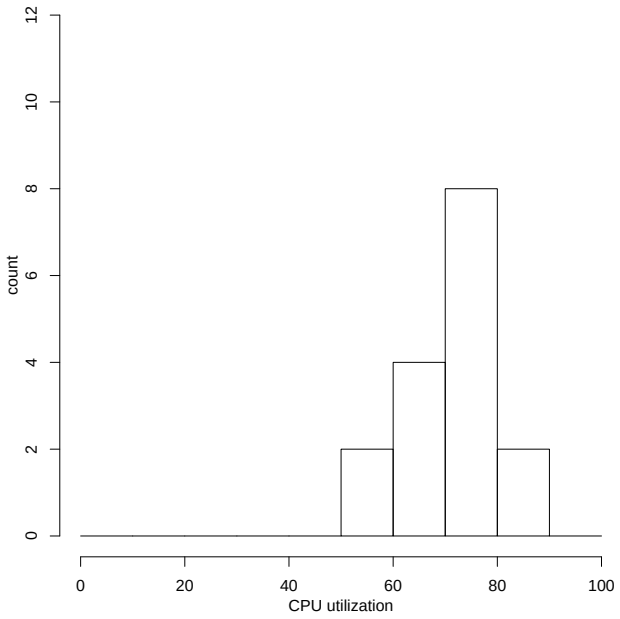


Figure 31: Histogram of 25-minute average CPU Utilization in Constant-ratio, Concurrency Control, ∼1 longs/cycle, Automatic Mode, configured utilization factor

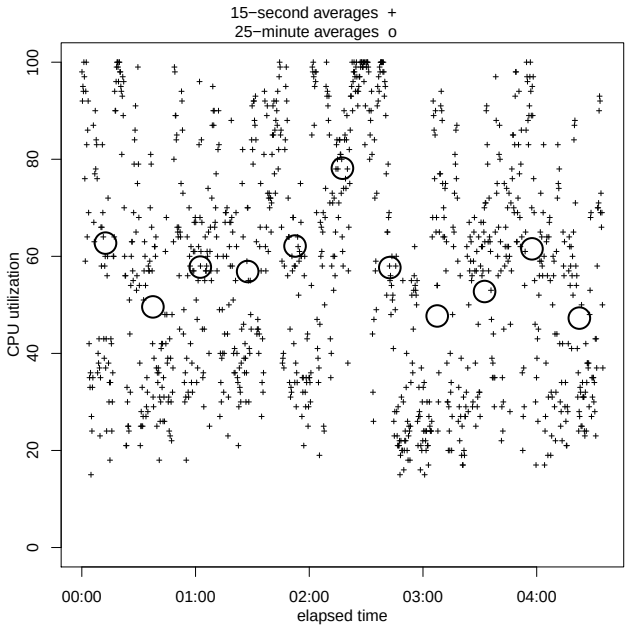


Figure 33: CPU Utilization in Constant-ratio, Rate Control, Automatic Mode, ∼1 longs/cycle scenario

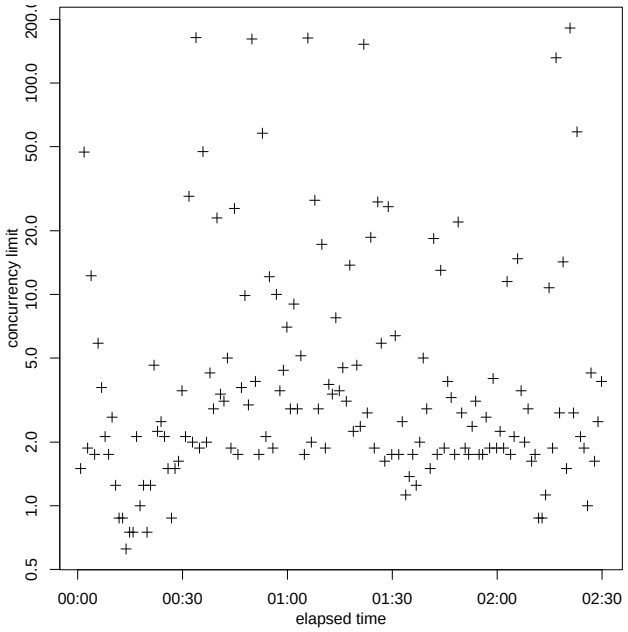


Figure 32: Gateway Control Setting in Constant-ratio, Concurrency Control, ∼1 longs/cycle, Automatic Mode, configured utilization factor

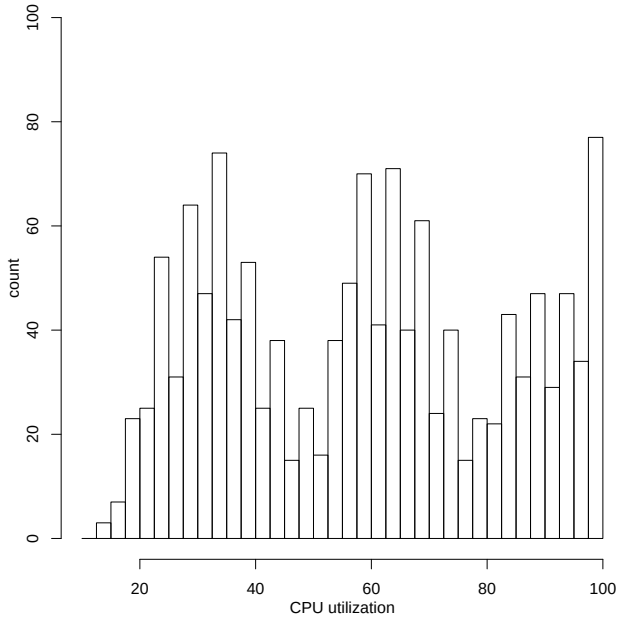


Figure 34: Histogram of 15-second CPU Utilization in Constant-ratio, Rate Mode, Automatic Mode, ∼1 longs/cycle scenario

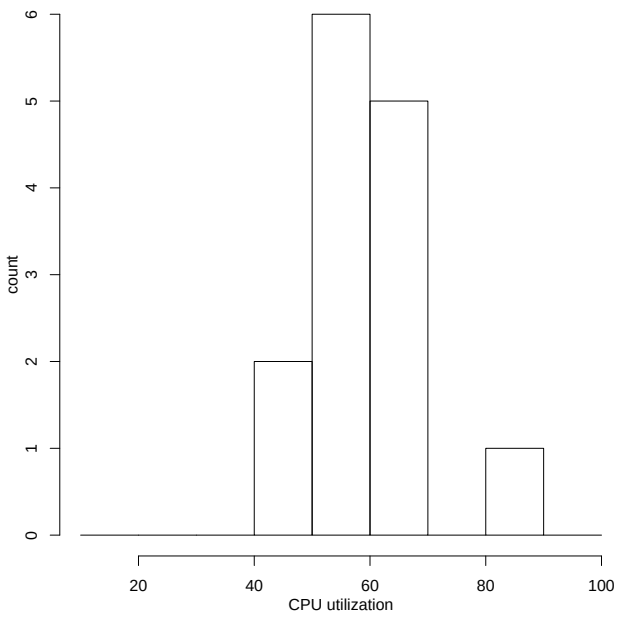


Figure 35: Histogram of 25-minute CPU Utilization in Constant-ratio, Rate Mode, Automatic Mode, ∼1 longs/cycle scenario

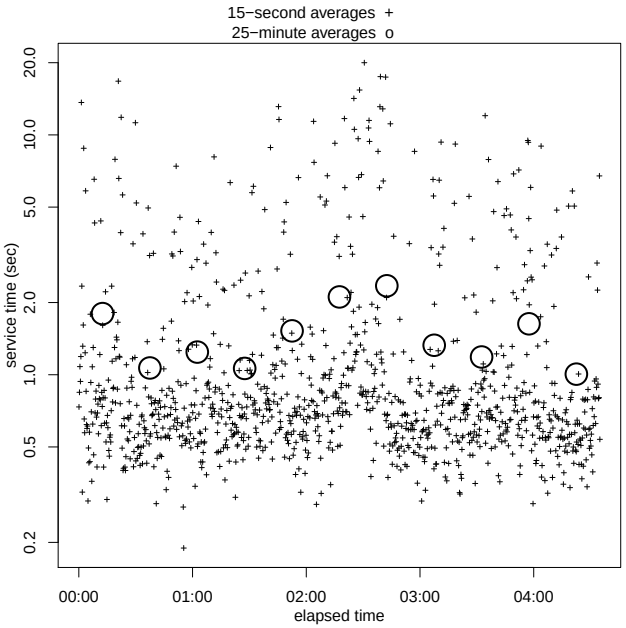


Figure 37: Average Service Time in Constant-ratio, Rate Control, Automatic Mode, ∼1 longs/cycle scenario

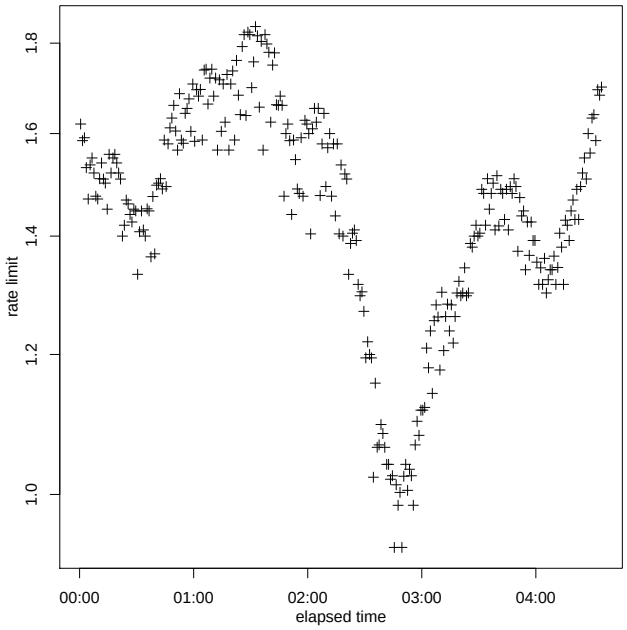


Figure 36: Gateway Control Setting in Constant-ratio, Rate Control, Automatic Mode, ∼1 longs/cycle scenario

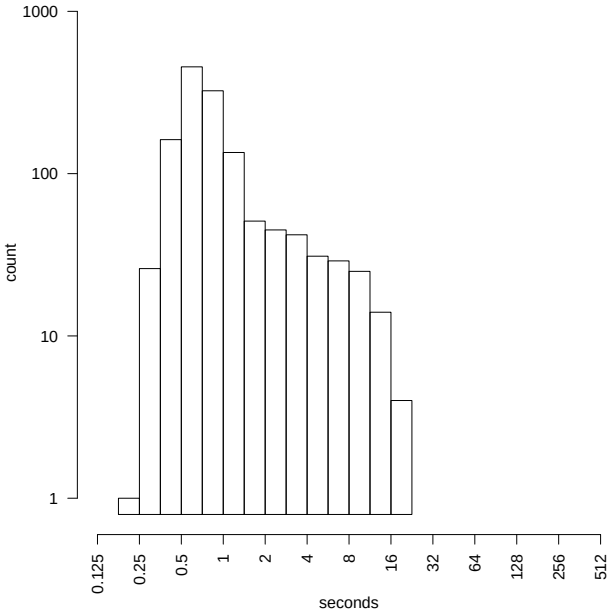


Figure 38: Histogram of 15-second Average Service Time in Constant-ratio, Rate Control, Automatic Mode, ∼1 longs/cycle scenario

utilization target was set at 64%, which corresponds with a throughput of about 1.6 req/sec. The average throughput was about 1.5 req/sec, thus expecting about 1/4 long requests per 15-second interval or about 1 per 60-second control cycle. Figure 33 shows the time series of the outer server's CPU utilization, averaged over 15 seconds and over 25 minutes, for 275 typical minutes of this run. Figure 34 shows a histogram of the 15-second averages from the whole run after warmup. These show a tri-modal distribution. This is because the CPU utilization is dominated by a state variable that takes on three values: the number of long requests being served is either zero, one, or multiple. Compare with figures 22 et. seq. Figure 37 shows the time series of the 15-second and 25-minute averages of service time, for the same 275 minutes as the CPU utilization. Figure 38 shows a histogram of the 15-second averages from the whole run after warmup. Figure 36 shows a time series of the control settings issued by the policy agent. Compared with figure 27 or even 32 we see a much better control, the setting stays much nearer the ideal value.

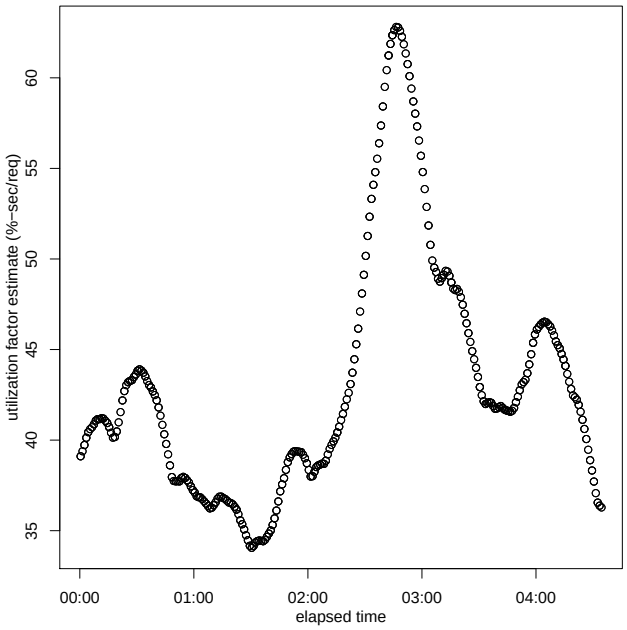


Figure 39: Utilization Factor Estimates in Constant-ratio, Rate Control, Automatic Mode, ∼1 longs/cycle scenario

Again, an identifiable source of error is in the on-line estimation of the utilization factor. Offline estimates from long-term averages give a value of about 40 %-sec/req; figure 39 shows a time series of the estimates produced on-line.

We then had the policy agent use a configured utilization factor of about 40 %-sec/req rather than the on-line estimates. Figure 40 shows a time series of the control settings issued; it stayed closer to the ideal value. Figure 41 shows a histogram of 15 second averages of CPU utilization; it still showed no great regulation. Indeed, this workload is designed to be very difficult to regulate at this time scale. Next let us look at what happens if we multiply the number of requests per control cycle by about 100.

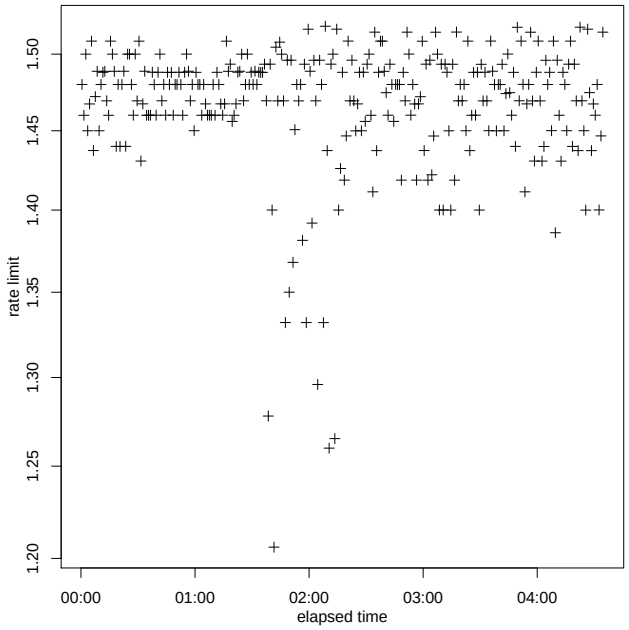


Figure 40: Gateway Control Setting in Constant-ratio, Rate Control, Automatic Mode, ∼1 longs/cycle, configured utilization factor

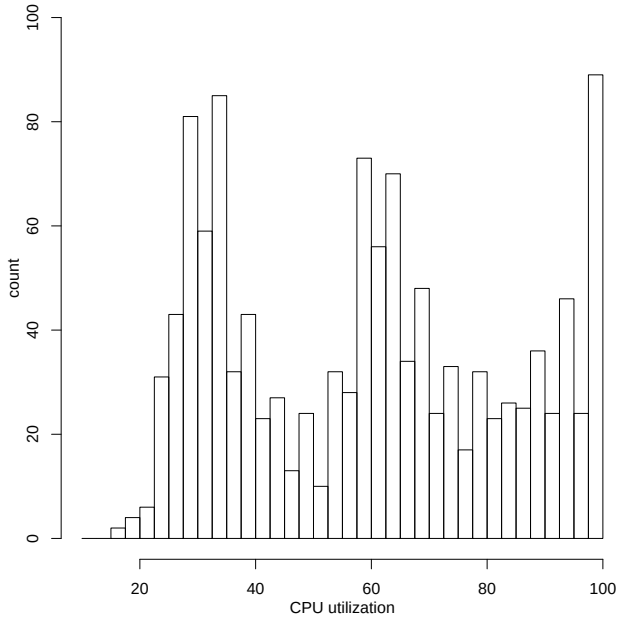


Figure 41: Histogram of 15-second CPU Utilization in Constant-ratio, Rate Mode, Automatic Mode, ∼1 longs/cycle, configured utilization factor

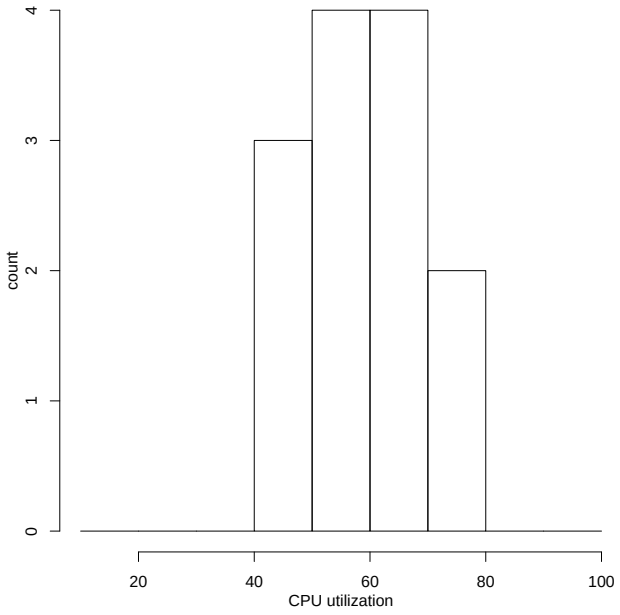


Figure 42: Histogram of 25-minute CPU Utilization in Constant-ratio, Rate Mode, Automatic Mode, ∼1 longs/cycle, configured utilization factor

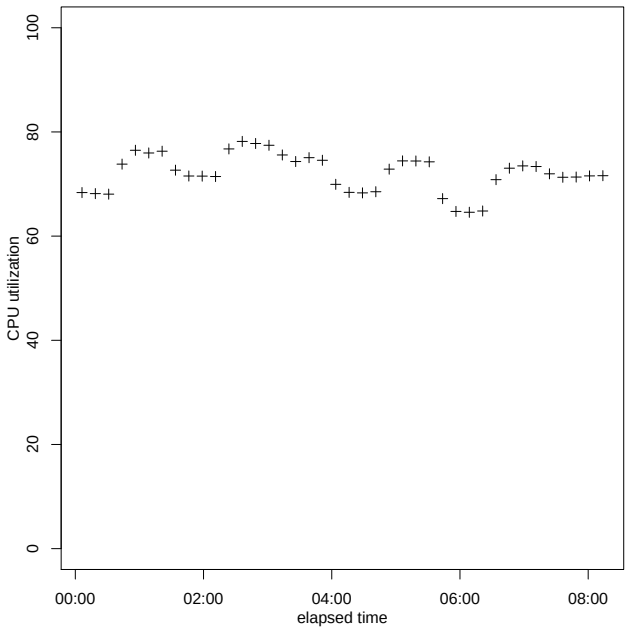


Figure 43: 12.5-Minute Averages of CPU Utilization in Constant-ratio, Concurrency Control, ∼100 longs/cycle, Automatic Mode scenario

6.2 Constant-Ratio, Automatic Mode, ∼100 Longs Per Cycle Results

Next we repeated those experiments but configured to have roughly 100 long requests per control cycle. We doubled the number of processors on the outer server (from 1 to 2) and lengthened the control cycle to 50 minutes. After this reconfiguration we repeated the manual mode trials to calibrate to the modified servers. We found that the constant-ratio workload had a long-term average CPU utilization of about 63% when the concurrency gateway was set to about 3.75 req, and when the rate gateway was set to about 3.65 req/sec.

6.2.1 Constant-Ratio, Automatic Concurrency Control, ∼100 Longs Per Cycle Results

We ran the constant-ratio scenario with many long requests per control cycle, for several hours. The CPU utilization target was set to 63%. The long-term average of the throughput was about 4 req/sec, which means there was an average of about 120 long requests per control cycle. Figure 43 shows the 12.5-minute averages of CPU utilization, for the whole trial excluding the startup, and figure 44 shows a histogram of those values. Figure 45 shows the 12.5-minute averages of the service times, for the whole trial excluding the startup, and figure 46 shows a histogram of those values. Figure 47 shows the control settings issued by the policy agent; they were a bit high, for the reason discussed next.

In this trial the on-line estimates of the utilization factor were consistently low. By comparing smoothed throughput and utilization we estimated that factor to be about 18.4 %-sec/req. Figure 48 shows the time series of the estimates made on-line.

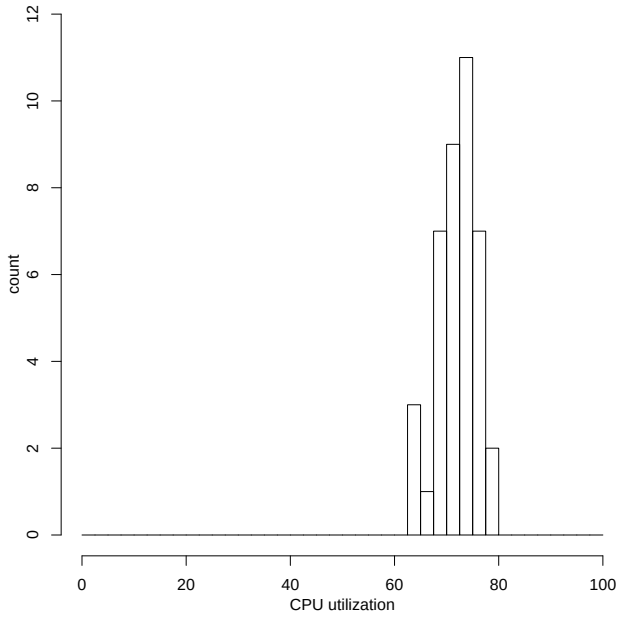


Figure 44: Histogram of 12.5-Minute Averages of CPU Utilization in Constant-ratio, Concurrency Control, ∼100 longs/cycle, Automatic Mode scenario

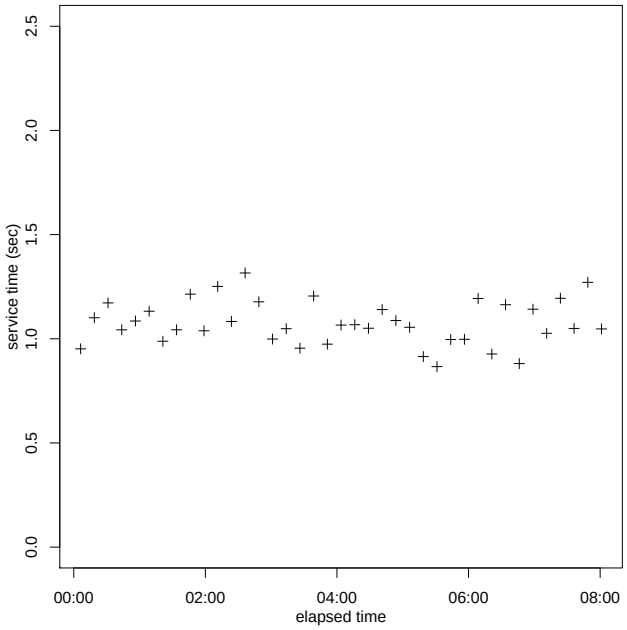


Figure 45: 12.5-Minute Averages Service Times in Constant-ratio, Concurrency Control, ∼100 longs/cycle, Automatic Mode scenario

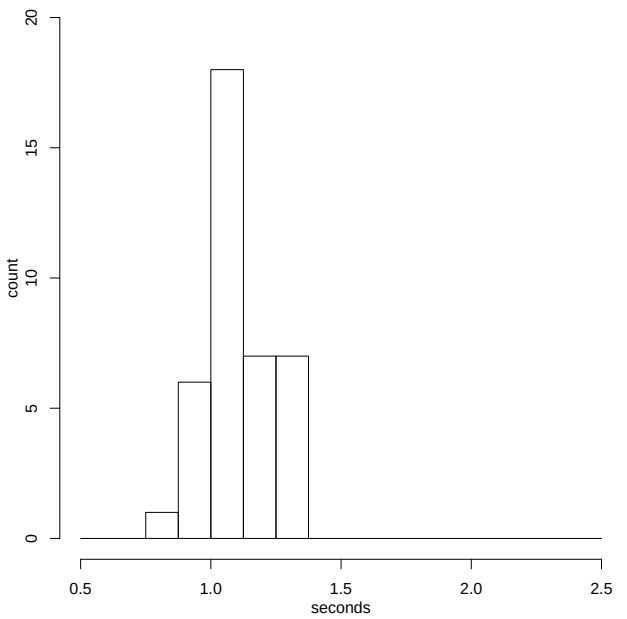


Figure 46: Histogram of 12.5-Minute Averages of Service Time in Constant-ratio, Concurrency Control, ∼100 longs/cycle, Automatic Mode scenario

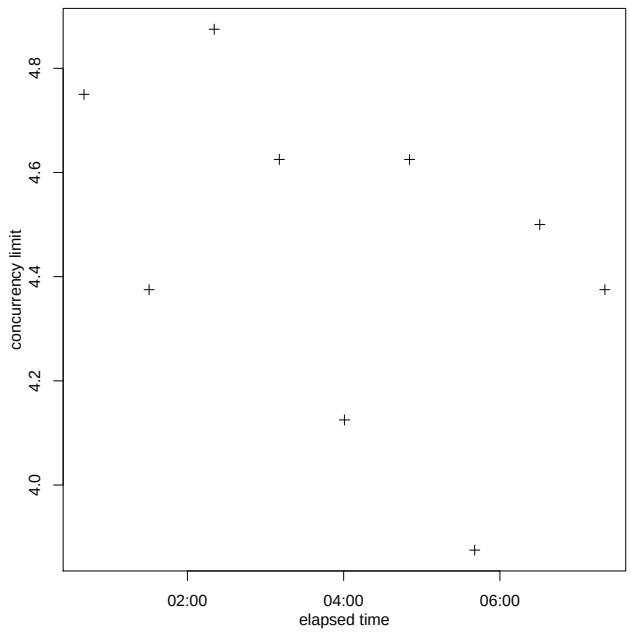


Figure 47: Gateway Control Settings in Constant-ratio, Concurrency Control, ∼100 longs/cycle, Automatic Mode scenario

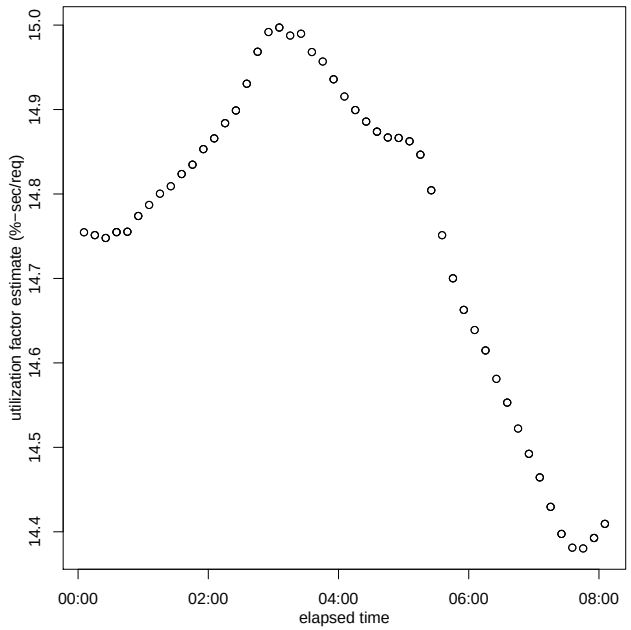


Figure 48: Utilization Factor estimates in Constant-ratio, Concurrency Control, ∼100 longs/cycle, Automatic Mode scenario

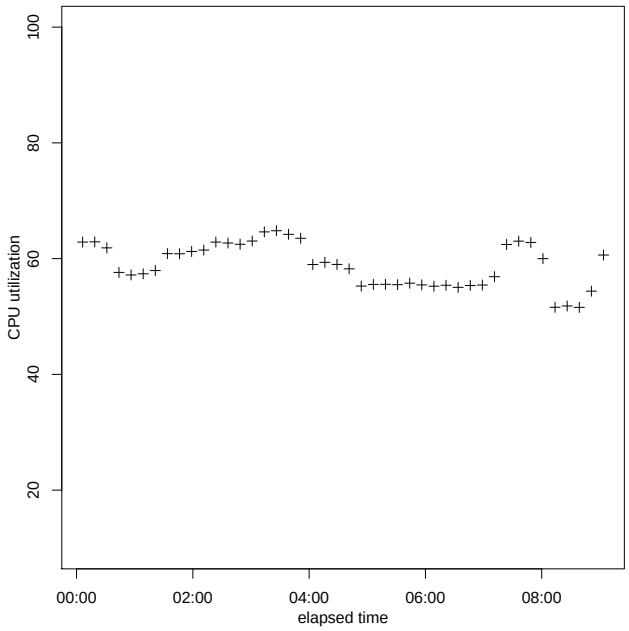


Figure 49: 12.5-Minute Averages of CPU Utilization in Constant-ratio, Concurrency Control, ∼100 longs/cycle, Automatic Mode, configured utilization factors

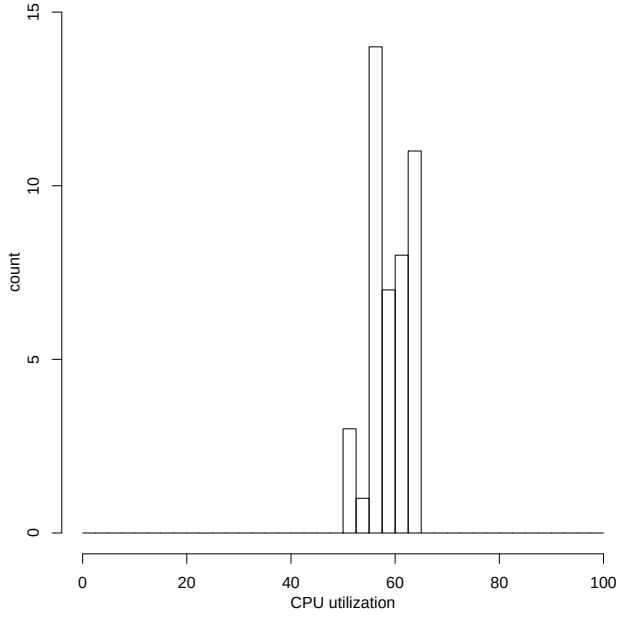


Figure 50: Histogram of 12.5-Minute Averages of CPU Utilization in Constant-ratio, Concurrency Control, ∼100 longs/cycle, Automatic Mode, configured utilization factors

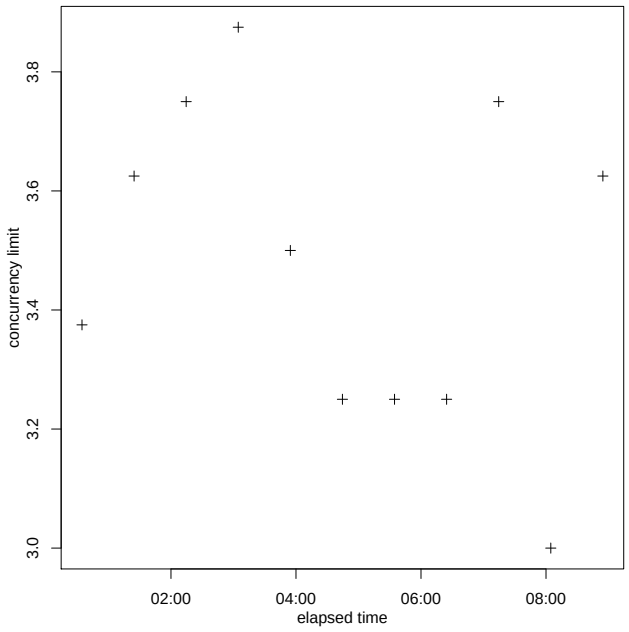


Figure 51: Gateway Control Settings in Constant-ratio, Concurrency Control, ∼100 longs/cycle, Automatic Mode, configured utilization factors

We then changed the automatic controller to use a configured utilization factor of about 18.4 %-sec/req, instead of the results of the on-line profiler. The long-term average throughput dropped to about $3\frac{1}{3}$ req/sec, which corresponds to an average of 100 long requests per control cycle. Figure 49 shows a time series of 12.5-minute averages of CPU utilization, and figure 50 shows a histogram. Figure 51 shows a time series of the control settings issued by the policy agent; these were closer to the correct value (3.75).

6.2.2 Constant-Ratio, Automatic Rate Control, ∼100 Longs Per Cycle Results

We ran the constant-ratio scenario with many long requests per control cycle, for several hours. The CPU utilization target was set to 63%. The long-term average throughput was about 3.4 req/sec, which corresponds with an average of about 102 long requests per control cycle. Figure 52 shows a time series of 12.5-minute averages of CPU utilization, for the whole trial excluding the startup, and figure 53 shows a histogram of those values. Figure 54 shows a time series of 12.5-minute averages of the service times, for the whole trial excluding the startup. Figure 55 shows a histogram of 12.5-minute averages of service times, for the whole trial excluding the startup.

In this trial the on-line estimates of the utilization factor were consistently high. By comparing smoothed throughput and utilization we estimated that factor to be about 17.2 %-sec/req. Figure 56 shows the time series of the estimates made on-line.

We then changed the automatic controller to use a configured utilization factor of 17.2 %-sec/req, instead of the re-

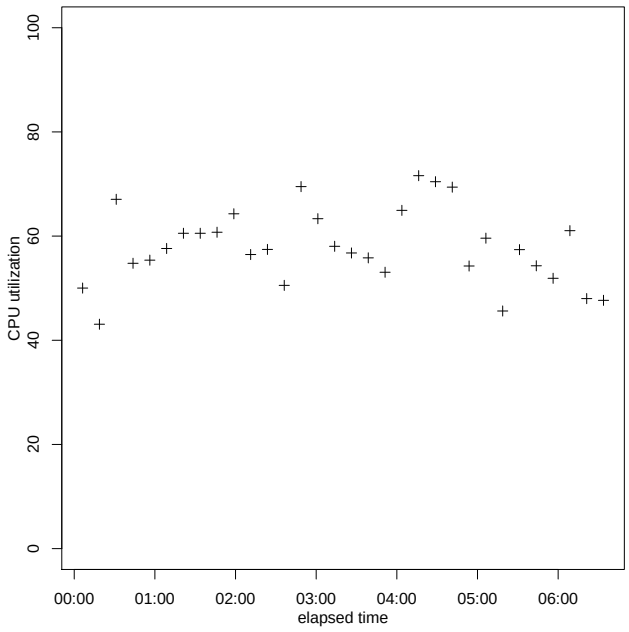


Figure 52: 12.5-Minute Averages of CPU Utilization in Constant-ratio, Rate Control, ∼100 longs/cycle, Automatic Mode scenario

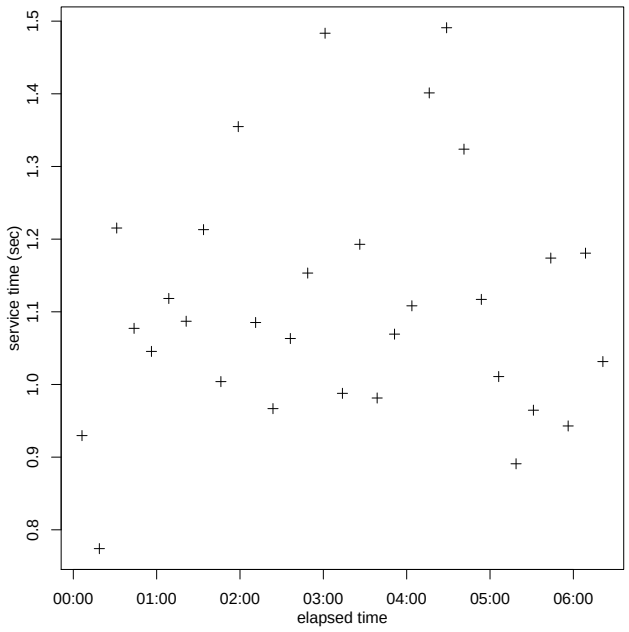


Figure 54: 12.5-Minute Averages of Service Time in Constant-ratio, Rate Control, ∼100 longs/cycle, Automatic Mode scenario

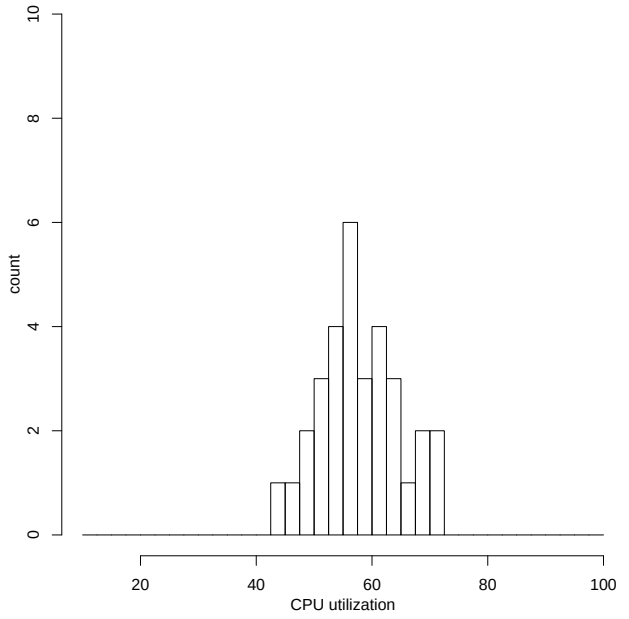


Figure 53: Histogram of 12.5-Minute Averages of CPU Utilization in Constant-ratio, Rate Control, ∼100 longs/cycle, Automatic Mode scenario

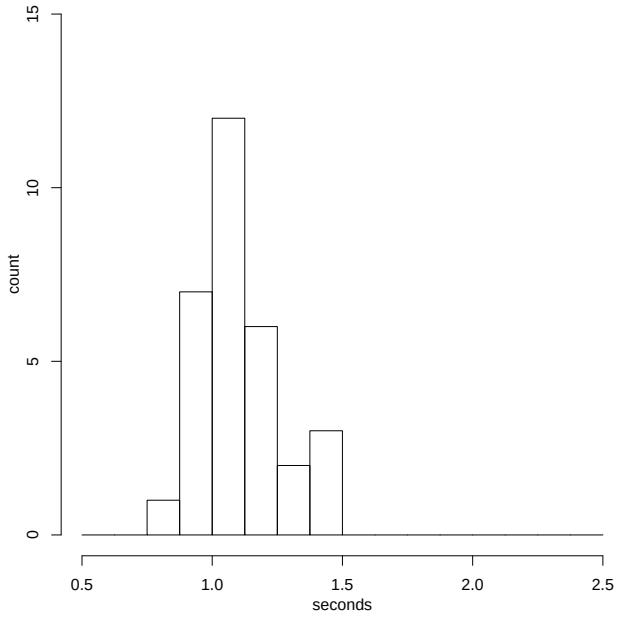


Figure 55: Histogram of 12.5-Minute Averages of Service Time in Constant-ratio, Rate Control, ∼100 longs/cycle, Automatic Mode scenario

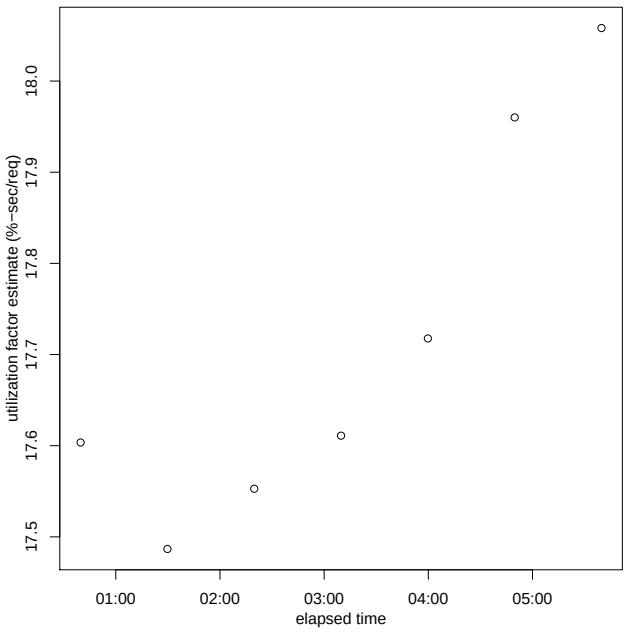


Figure 56: Utilization Factor estimates in Constant-ratio, Rate Control, ∼100 longs/cycle, Automatic Mode scenario

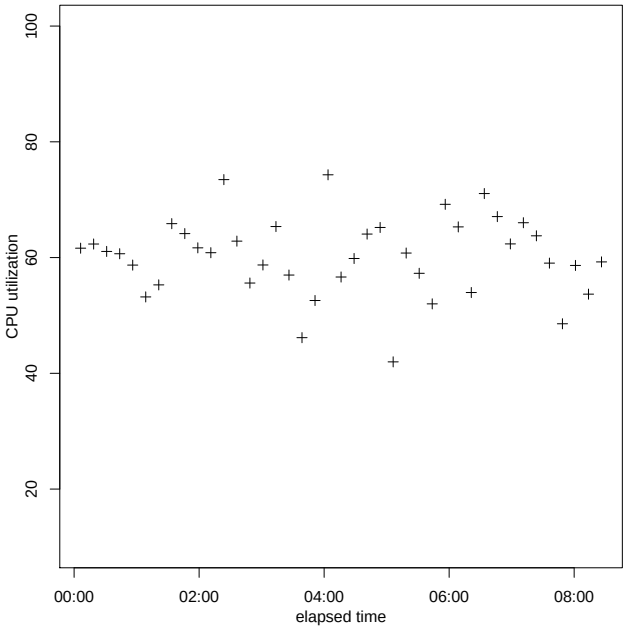


Figure 57: 12.5-Minute Averages of CPU Utilization in Constant-ratio, Rate Control, ∼100 longs/cycle, Automatic Mode, configured utilization factors

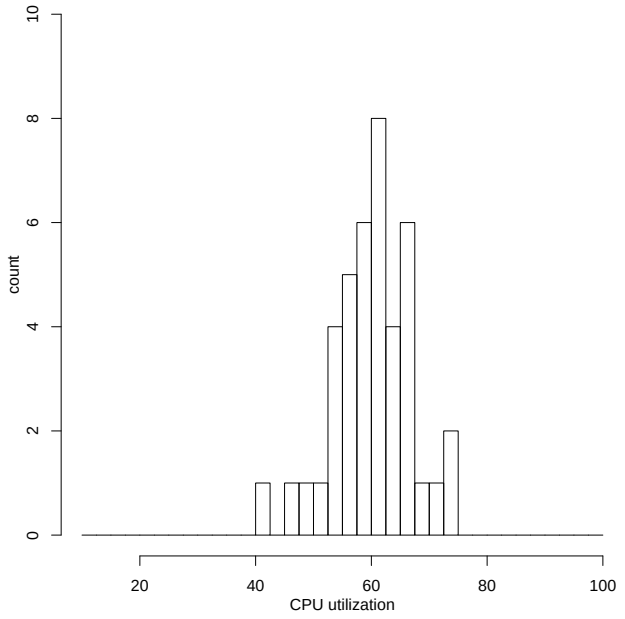


Figure 58: Histogram of 12.5-Minute Averages of CPU Utilization in Constant-ratio, Rate Control, ∼100 longs/cycle, Automatic Mode, configured utilization factors

sults of the on-line profiler. Figure 57 shows the time series of the CPU utilizations, and figure 58 shows a histogram.

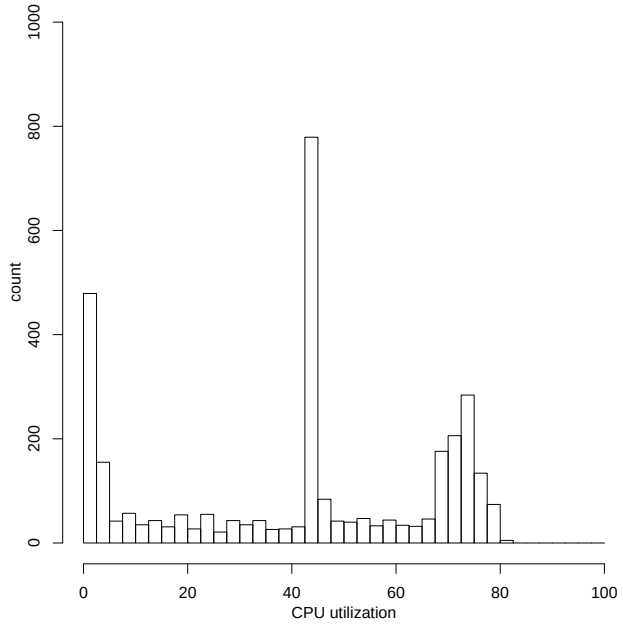
6.3 Constant-Work, Automatic Mode, ∼1 Longs Per Cycle Results

First, let us return to configurations with about one long/cycle. Again the outer server is a uniprocessor and the control cycle length is one minute.

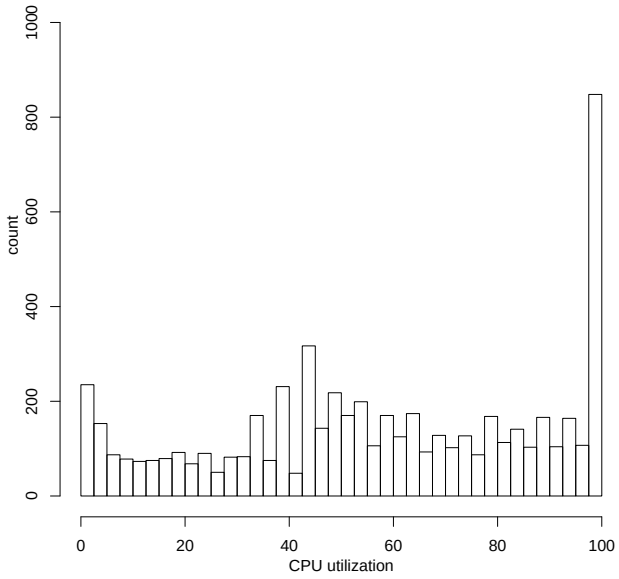
6.3.1 Constant-Work, Automatic Concurrency Control, ∼1 Longs Per Cycle Results

We ran the constant-work scenario with an average of roughly one long request per control cycle, for several hours. The CPU utilization target was set at 67%, which corresponds with a concurrency of about 2 requests. The long-term average of the throughput that actually happened was about 2 req/sec, thus an average of about 1.2 long requests per 60-second control cycle. Figure 59 shows the 15-second and 25-minute averages of CPU utilization on the outer server over the whole run excluding startup, and figure 60 shows a histogram of the 15-second CPU averages. Compare with figures 66 et. seq. Figure 62 shows a time series of the concurrency limits set by the policy agent. Figure 63 shows a time series of the on-line estimates of the utilization factor; later we took long-term averages and they indicated a utilization factor of about 38 %-sec/req, which is in the upper range of the on-line estimates. Figure 64 shows the 15-second and 25-minute averages of the service times for the whole trial excluding startup, and figure 65 shows a histogram of 15-second averages of service times.

We then repeated experiments with the constant work sce-



Manual mode



Automatic, on-line utilization factor estimates

Table 4: Histograms of 15-second averge CPU utilization with low target, constant work, concurrency control, ∼1 longs/cycle

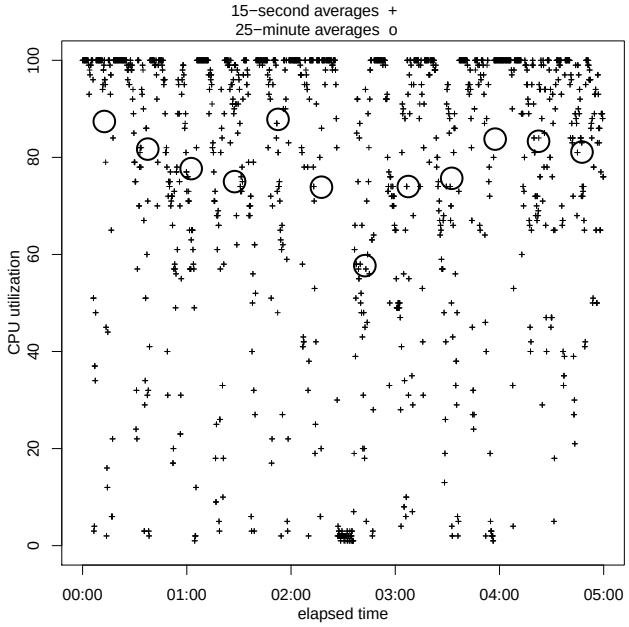


Figure 59: CPU Utilization in Constant-work, Concurrency Control, ∼1 longs/cycle, Automatic Mode scenario

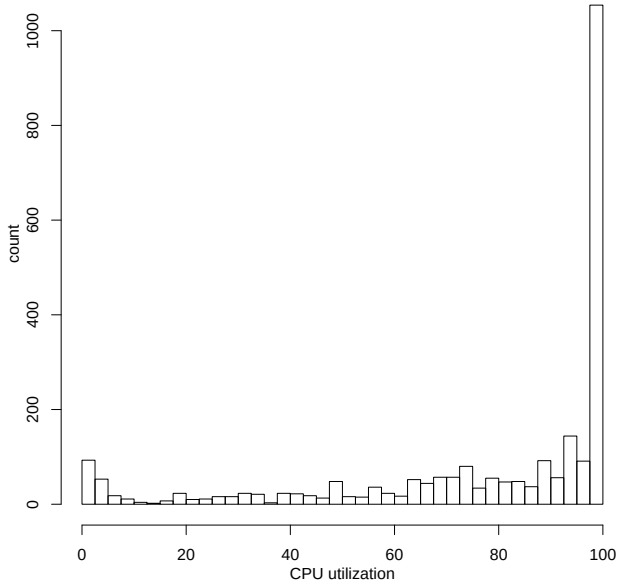


Figure 60: Histogram of 15-second average CPU Utilization in Constant-work, Concurrency Control, ∼1 longs/cycle, Automatic Mode scenario

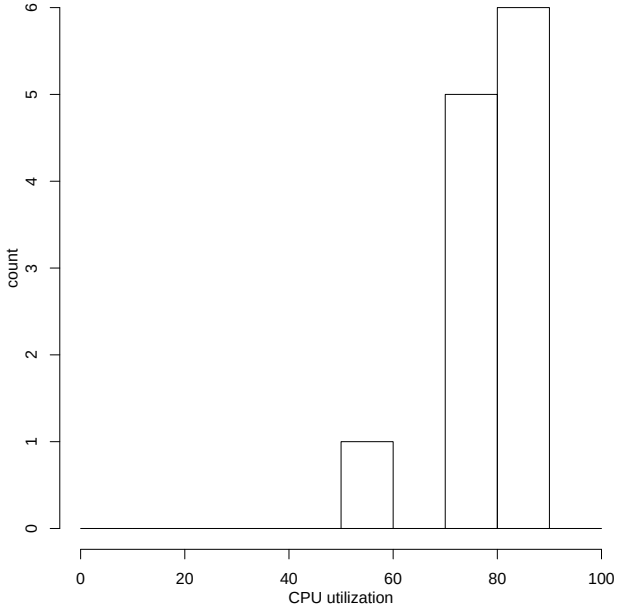


Figure 61: Histogram of 25-minute average CPU Utilization in Constant-work, Concurrency Control, ∼1 longs/cycle, Automatic Mode scenario

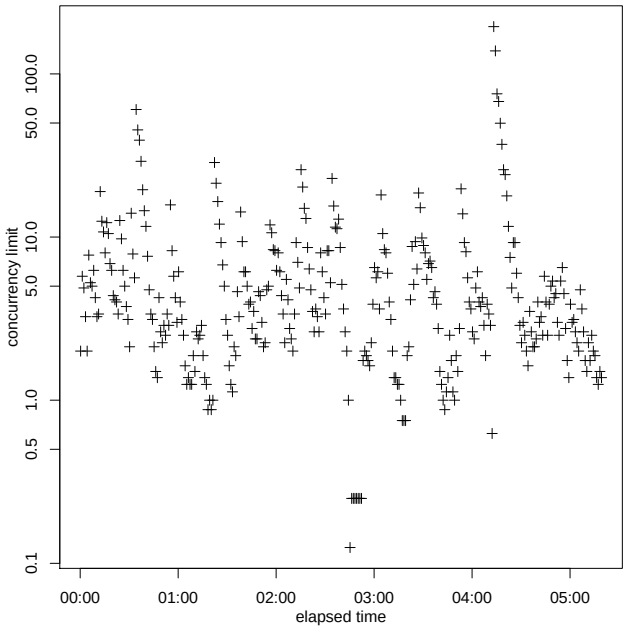


Figure 62: Gateway Control Settings in Constant-work, Concurrency Control, ∼1 longs/cycle, Automatic Mode scenario

scenario	mean	std. dev.
manual	41.43995	26.28481
auto, on-line	57.88514	30.60786

Table 3: Mean+-Standard Deviation of 15-second averge CPU utilization with low target, constant work, concurrency control, ∼1 longs/cycle

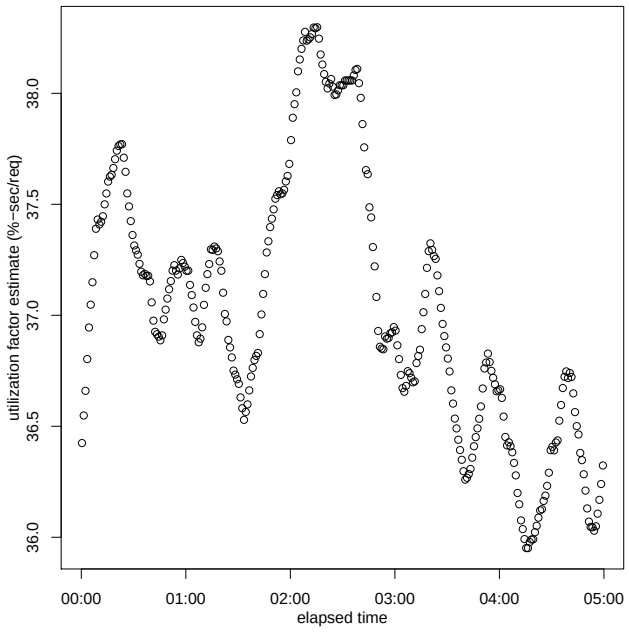


Figure 63: Utilization Factor Estimates in Constant-work, Concurrency Control, ∼1 longs/cycle, Automatic Mode scenario

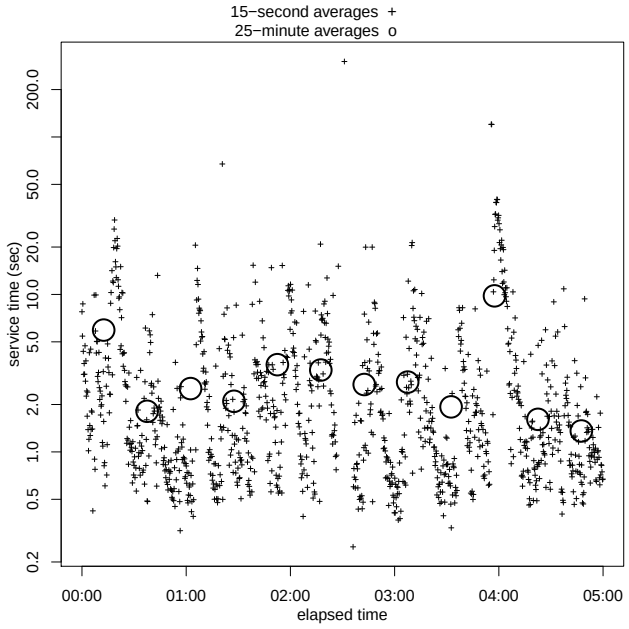


Figure 64: Average Service Time in Constant-work, Concurrency Control, ∼1 longs/cycle, Automatic Mode scenario

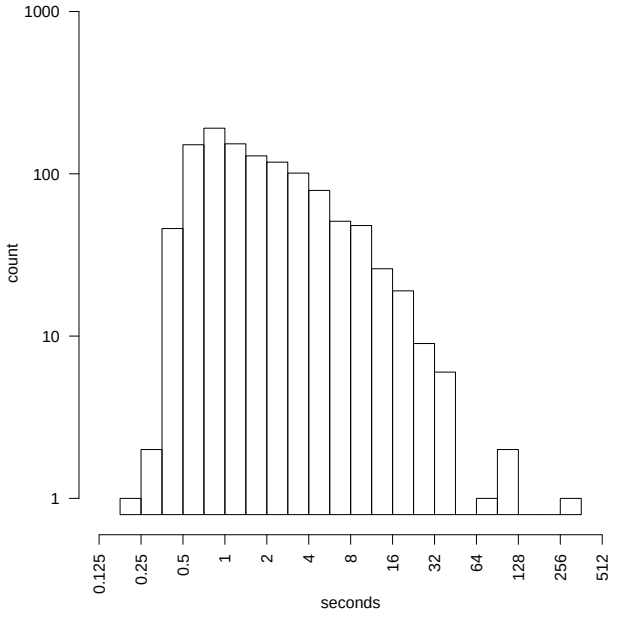


Figure 65: Histogram of 15-second-average Service Time in Constant-work, Concurrency Control, ∼1 longs/cycle, Automatic Mode scenario

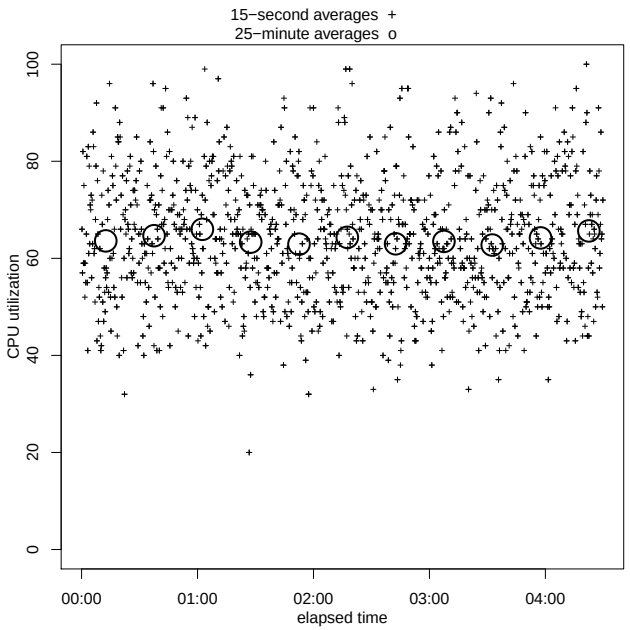


Figure 66: CPU Utilization in Constant-work, Rate Control, ∼1 longs/cycle, Automatic Mode scenario

nario, but used a CPU utilization target of 44%. Table 3 shows the resulting sample mean and standard deviation, while table 4 shows histograms.

6.3.2 Constant-Work, Automatic Rate Control, ∼1 Longs Per Cycle Results

We ran the constant-work scenario an average of one long request per control cycle, for several hours. The CPU utilization target was set at 67%. The long-term average of the throughput was about 1.7 req/sec, corresponding to an average of about 1 long request per control cycle. Figure 66 shows the 15-second and 15-minute averages of CPU utilization on the outer server over three characteristic hours, and figure 67 shows a histogram of the 15-second CPU averages on the outer server, for the whole trial excluding the startup transient; all the 25-minute averages of CPU utilization were in the 60–70% range. Compare with figures 59 et. seq. Figure 69 shows the on-line estimates of the utilization factor; we later looked at long-term averages and found they indicate a utilization factor of about 38 %-sec/req — which makes a utilization of 67% correspond with a throughput of about 1.8 req/sec. Figure 68 shows the rate limits issued by the policy agent. Figure 70 shows the 15-second and 15-minute averages of the service times for the same three characteristic hours, and figure 71 shows a histogram of 15-second averages of service times, for the whole trial excluding the startup transient.

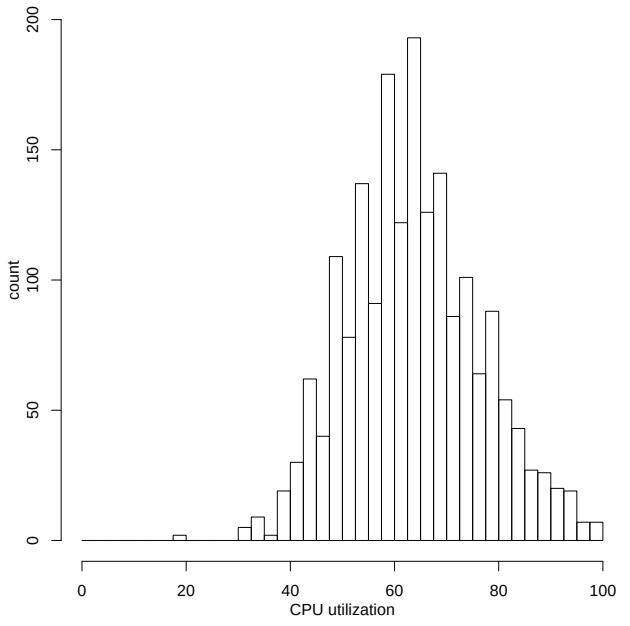


Figure 67: Histogram of 15-second average CPU Utilization in Constant-work, Rate Control, ∼1 longs/cycle, Automatic Mode scenario

6.4 Constant-Work, Automatic Mode, ∼100 Longs Per Cycle results

Now for roughly a hundred longs/cycle. Again, the outer server has two processors and the control cycle length is 50 minutes.

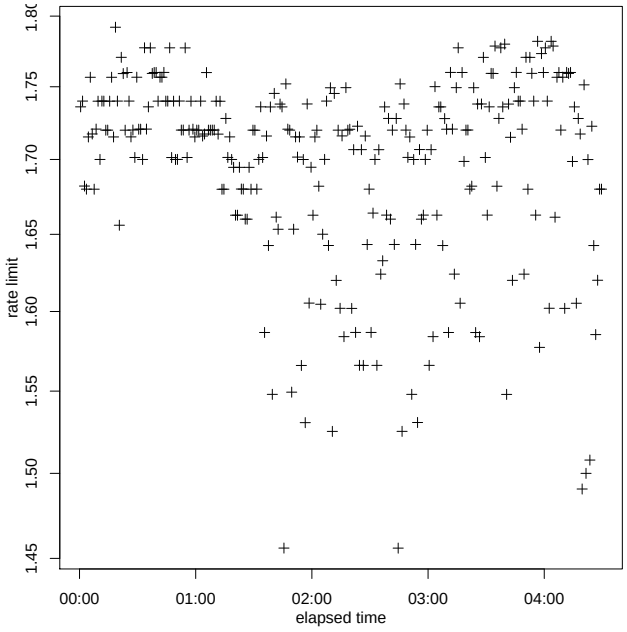


Figure 68: Gateway Control Settings in Constant-work, Rate Control, ∼1 longs/cycle, Automatic Mode scenario

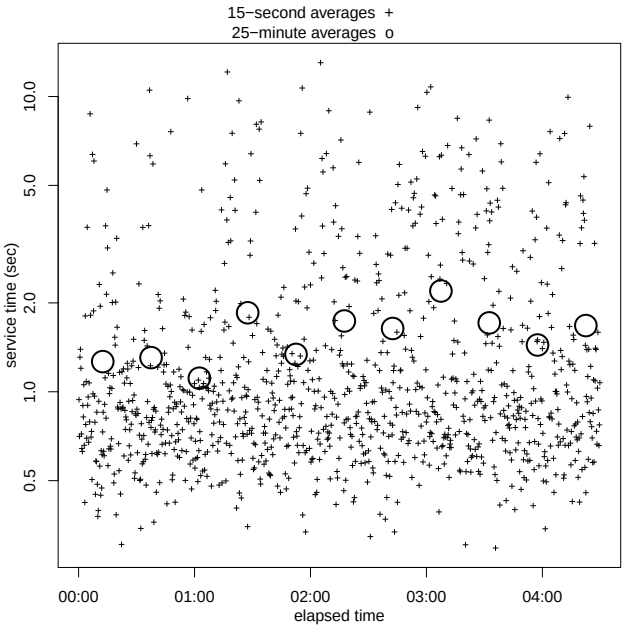


Figure 70: Average Service Time in Constant-work, Rate Control, ∼1 longs/cycle, Automatic Mode scenario

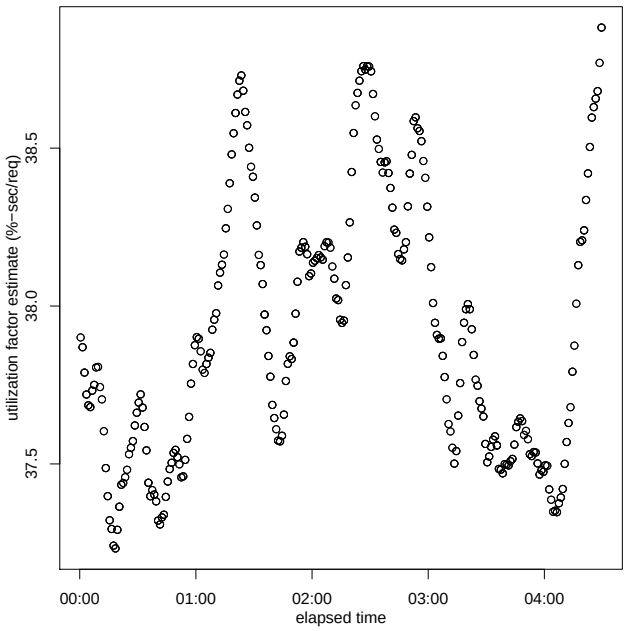


Figure 69: Utilization Factor Estimates in Constant-work, Rate Control, ∼1 longs/cycle, Automatic Mode scenario

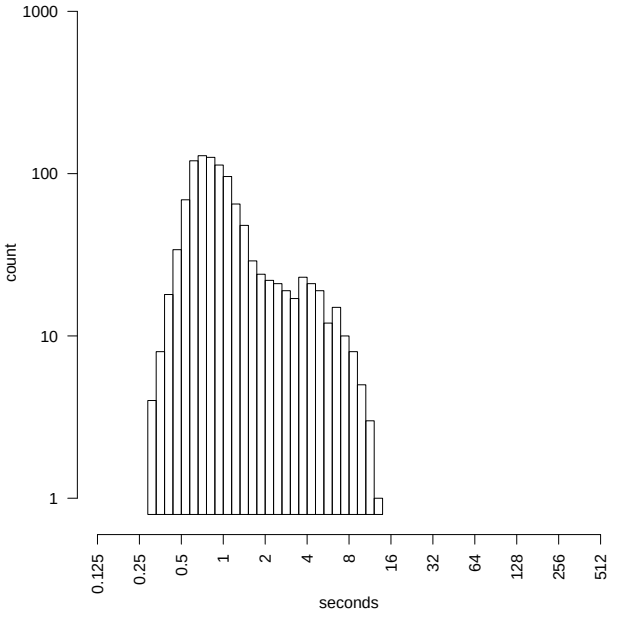


Figure 71: Histogram of 15-second-average Service Time in Constant-work, Rate Control, ∼1 longs/cycle, Automatic Mode scenario

6.4.1 Constant-Work, Automatic Concurrency Control, ∼100 Longs Per Cycle Results

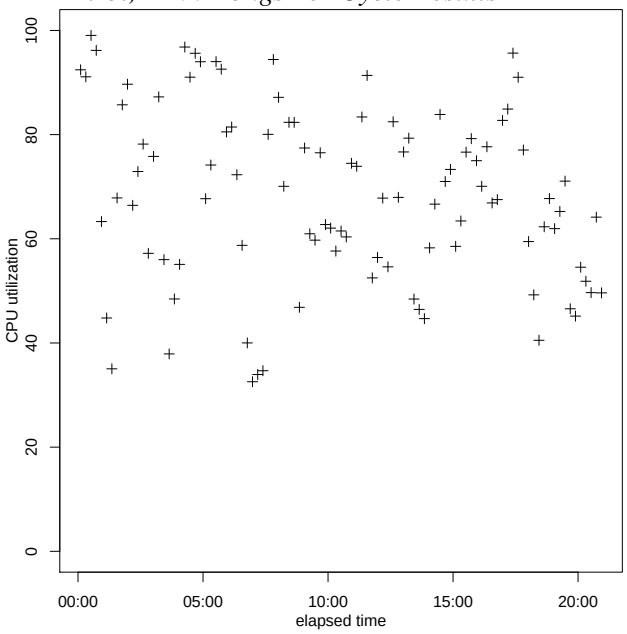


Figure 72: 12.5-Minute Averages of CPU Utilization in Constant-work, Concurrency Control, ∼100 longs/cycle, Automatic Mode scenario

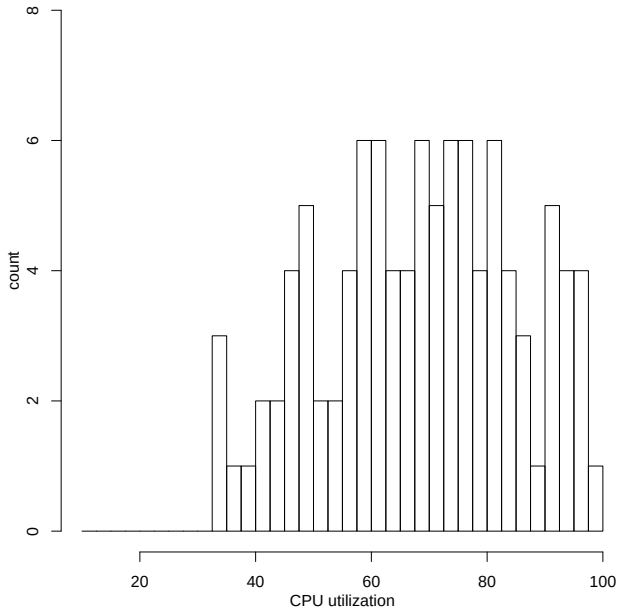


Figure 73: Histogram of 12.5-Minute Averages of CPU Utilization in Constant-work, Concurrency Control, ∼100 longs/cycle, Automatic Mode scenario

We calibrated the constant-work scenario in the many long per cycle configuration in manual mode, finding that, in long-term averages, a throughput of 3.66 req/sec corresponds with a CPU utilization of 67.8%.

We ran the constant-work scenario with an average of 100 long requests per control cycle, for several hours. The CPU utilization target was set to 67%. Figure 72 shows the 12.5-minute averages of CPU utilization, for the whole trial excluding the startup. Figure 73 shows a histogram of the 12.5-minute CPU averages on the outer server, for the whole trial excluding the startup. Figure 74 shows a time series of the gateway control settings from the policy agent. Figure 75 shows a time series of the on-line estimates of the utilization factor; we later look at long-term averages and estimated the factor was about 18.5 %-sec/req. Figure 76 shows the 12.5-minute averages of the service times, for the whole trial excluding the startup. Figure 77 shows a histogram of 12.5-minute averages of service times, for the whole trial excluding the startup.

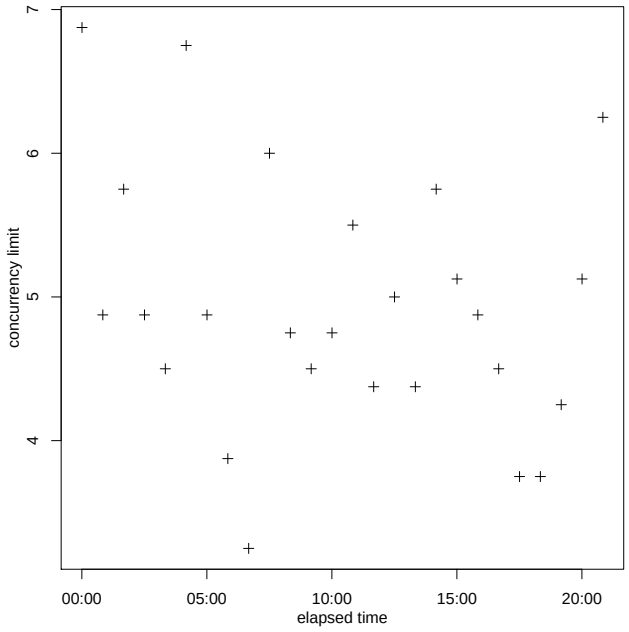


Figure 74: Gateway Control Settings in Constant-work, Concurrency Control, ∼100 longs/cycle, Automatic Mode scenario

6.4.2 Constant-Work, Automatic Rate Control, ∼100 Longs Per Cycle Results

We calibrated the constant-work scenario in the many long per cycle configuration in manual mode, finding that, in long-term averages, a throughput of 3.38 req/sec corresponds with a CPU utilization of 67%.

We ran the constant-work scenario with an average of 100 long requests per control cycle, for several hours. The CPU utilization target was set to 67%. Figure 78 shows the 12.5-minute averages of CPU utilization, for the whole trial ex-

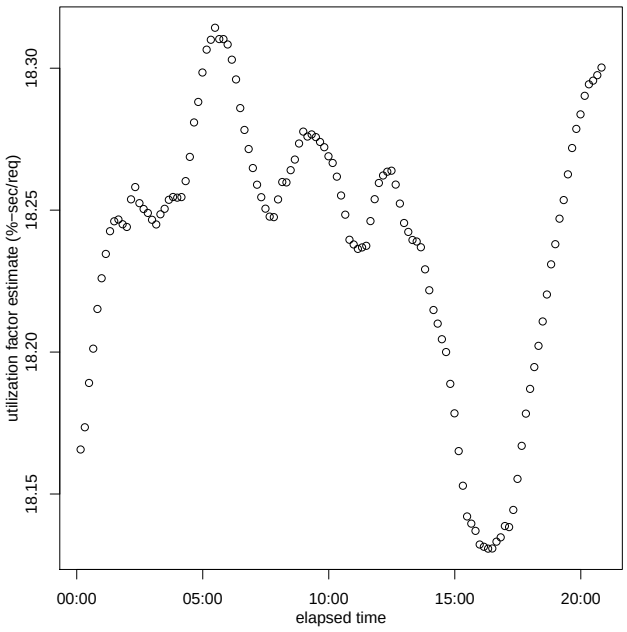


Figure 75: Utilization Factor Estimates in Constant-work, Concurrency Control, ∼100 longs/cycle, Automatic Mode scenario

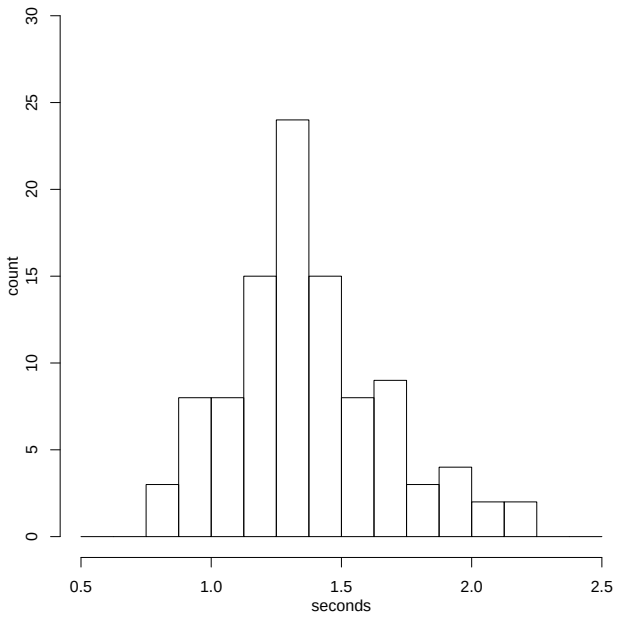


Figure 77: Histogram of 12.5-Minute Averages of Service Time in Constant-work, Concurrency Control, ∼100 longs/cycle, Automatic Mode scenario

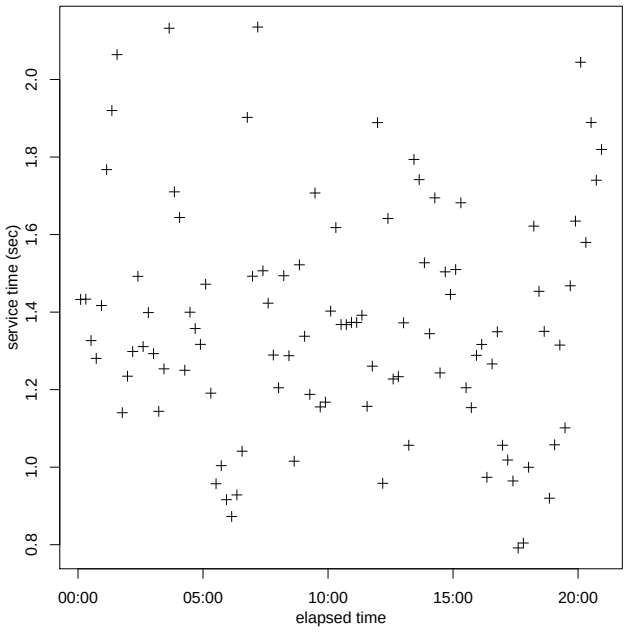


Figure 76: 12.5-Minute Averages of Service Time in Constant-work, Concurrency Control, ∼100 longs/cycle, Automatic Mode scenario

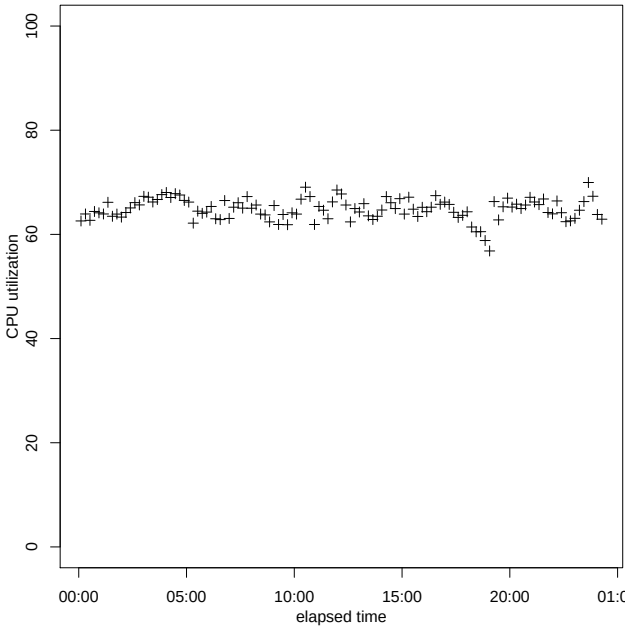


Figure 78: 12.5-Minute Averages of CPU Utilization in Constant-work, Rate Control, ∼100 longs/cycle, Automatic Mode scenario

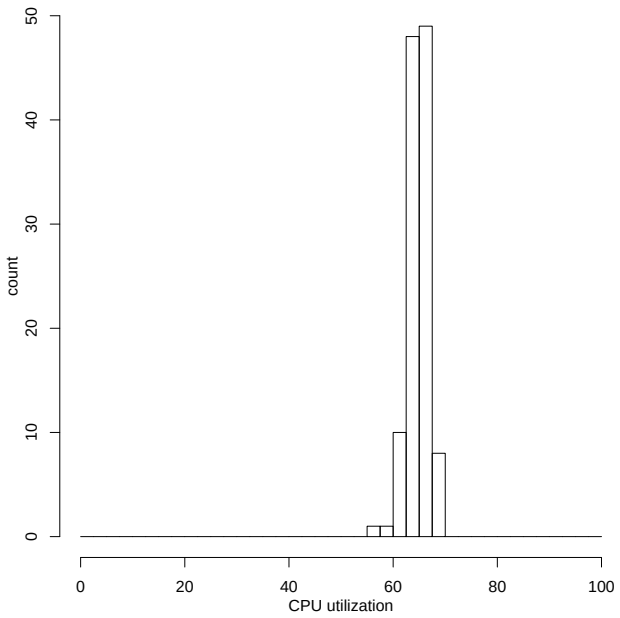


Figure 79: Histogram of 12.5-Minute Averages of CPU Utilization in Constant-work, Rate Control, ∼100 longs/cycle, Automatic Mode scenario

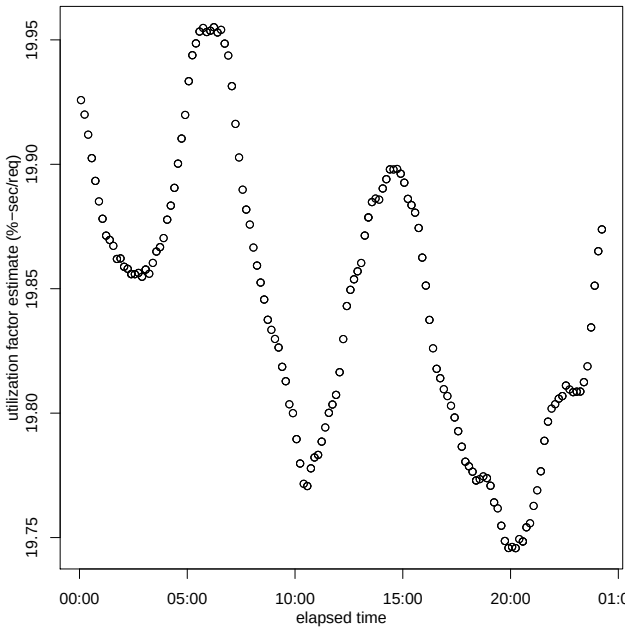


Figure 81: Utilization Factor Estimates in Constant-work, Rate Control, ∼100 longs/cycle, Automatic Mode scenario

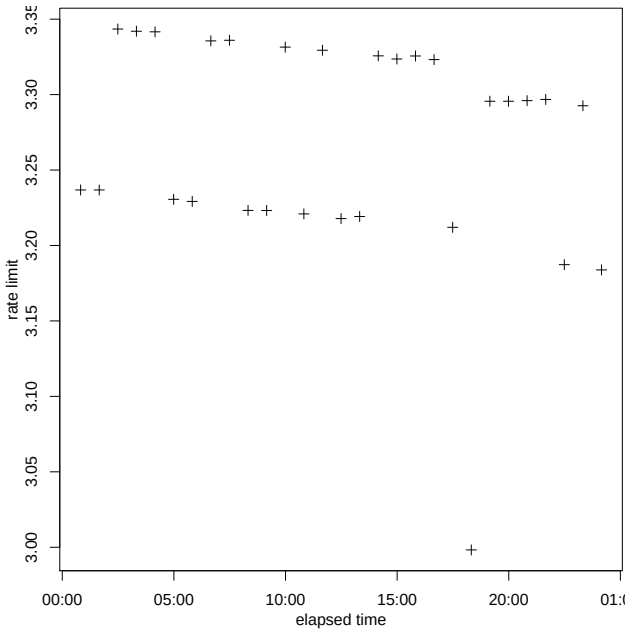


Figure 80: Gateway Control Settings in Constant-work, Rate Control, ∼100 longs/cycle, Automatic Mode scenario

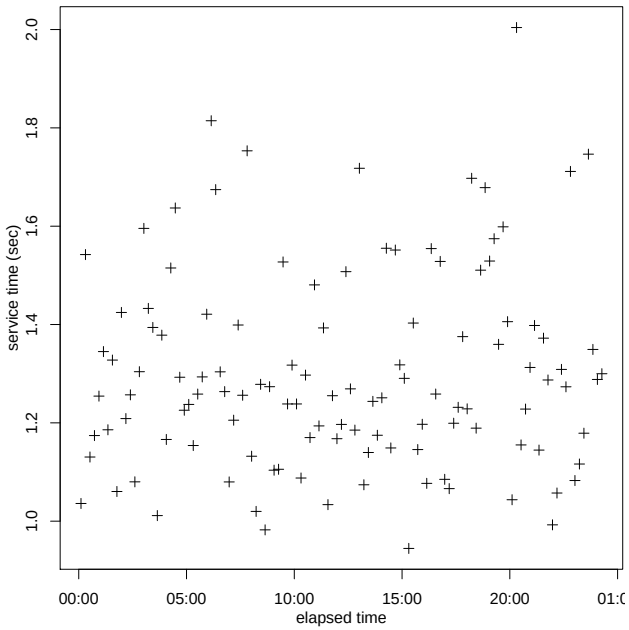


Figure 82: 12.5-Minute Averages of Service Time in Constant-work, Rate Control, ∼100 longs/cycle, Automatic Mode scenario

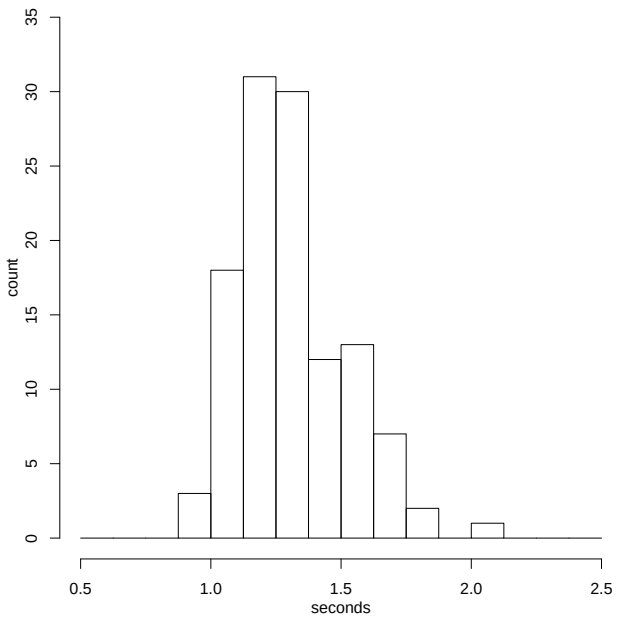


Figure 83: Histogram of 12.5-Minute Averages of Service Time in Constant-work, Rate Control, ∼100 longs/cycle, Automatic Mode scenario

	concurrency gateway	rate gateway
constant ratio	83.11346 ± 14.46747	58.79613 ± 24.30999
constant work	78.24217 ± 29.43880	63.96396 ± 12.70281

Table 5: Mean+-Standard Deviation of 15-second averge CPU utilization in automatic control, on-line utilization factor estimates, ∼1 longs/cycle

cluding the startup. Figure 79 shows a histogram of the 12.5-minute CPU averages on the outer server, for the whole trial excluding the startup. Figure 80 shows a time series of the gateway control settings from the policy agent. Figure 81 shows a time series of the on-line estimates of the utilization factor; we later look at long-term averages and estimated the factor was about 19.6 %-sec/req. Figure 82 shows the 12.5-minute averages of the service times, for the whole trial excluding the startup. Figure 83 shows a histogram of 12.5-minute averages of service times, for the whole trial excluding the startup.

6.5 Summary of Automatic Control

Table 5 summarizes the comparison of CPU utilization results in the case of automatic control, roughly a hundred longs/cycle, on-line utilization factor estimates via sample mean and standard deviation. Figure 84 summarizes the comparison via histograms of 15-second averages. Figure 85

	concurrency gateway	rate gateway
constant ratio	71.41645 ± 22.14784	59.43745 ± 23.99228

Table 6: Mean+-Standard Deviation of 15-second averge CPU utilization in automatic control, configured utilization factor estimates, ∼1 longs/cycle

	concurrency gateway	rate gateway
constant ratio	72.13367 ± 4.057711	58.13689 ± 18.25958
constant work	68.26586 ± 27.10929	64.83647 ± 8.972858

Table 7: Mean+-Standard Deviation of 15-second averge CPU utilization in automatic control, on-line utilization factor estimates, ∼100 longs/cycle

	concurrency gateway	rate gateway
constant ratio	58.87767 ± 4.119689	60.17795 ± 17.41581

Table 8: Sample Mean and Standard Deviation of 15-second averge CPU utilization in automatic control, configured utilization factor estimates, ∼100 longs/cycle

compares time series of the control settings.

Figure 86 summarizes the comparison in the case of automatic control, roughly one longs/cycle, on-line utilization factor estimates via histograms of 25-minute averages.

Table 6 summarizes the comparison in the case of automatic control, roughly one longs/cycle, configured utilization factor estimates via sample mean and standard deviation of 15-second averages. Figure 87 summarizes the comparison via histograms of 15-second averages.

Figure 88 summarizes the comparison in the case of automatic control, roughly one longs/cycle, configured utilization factor estimates via histograms of 25-minute averages.

Table 7 summarizes the comparison in the case of automatic control, roughly a hundred longs/cycle, on-line utilization factor estimates via sample mean and standard deviation of 15-second averages. Figure 89 summarizes the comparison via histograms of 12.5-minute averages. Figure 90 compares the time series of the control settings.

Table 8 summarizes the comparison in the case of automatic control, roughly a hundred longs/cycle, configured utilization factor estimates via sample mean and standard deviation of 15-second averages of CPU utilization. Figure 91 summarizes the comparison in the case of automatic control, roughly a hundred longs/cycle, configured utilization factor estimates via histograms of 12.5-minute averages.

7. CONCLUSION

We first summarize the results of this study of very problematic workloads, and then make some general recommendations. Finally we note some possible further developments.

As summarized in Table 1, Figure 20, and Figure 21, the two gateway mechanisms are simply different in a manual context: each is better for a different workload. However, there are more consistent results in automatic contexts. The rate-based gateway and automatic controller working together produced more accurate control than the occupancy-based gateway and automatic controller. On the other hand, if the highest priority is simply to avoid underload, the occupancy-based gateway and automatic controller usually did a better job of this — at the expense of causing overload more often.

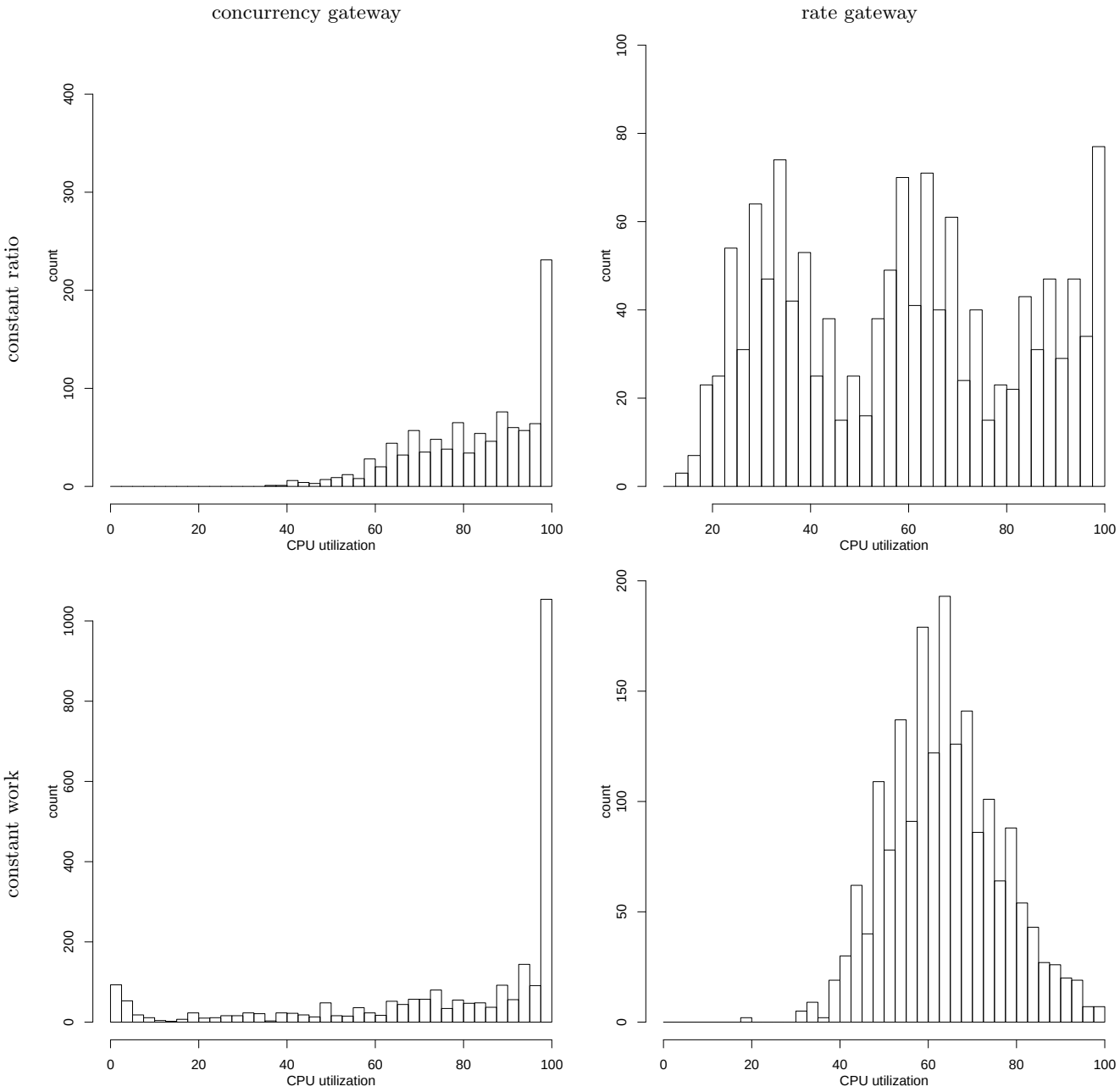


Figure 84: Histograms of 15-second averge CPU utilization in automatic control, on-line utilization factor estimates, ∼1 longs/cycle

If	Then
the workload does not include extreme variance	either gateway will work well
the extreme variance, if any, has little variation in the work/time ratio	the concurrency gateway works best
the exteme variance, if any, has little variation in work/request	the rate gateway works best
every kind of exterme variance might appear in the workload	the rate gateway is least risky

Table 9: Management depends on (what you know about) your workload

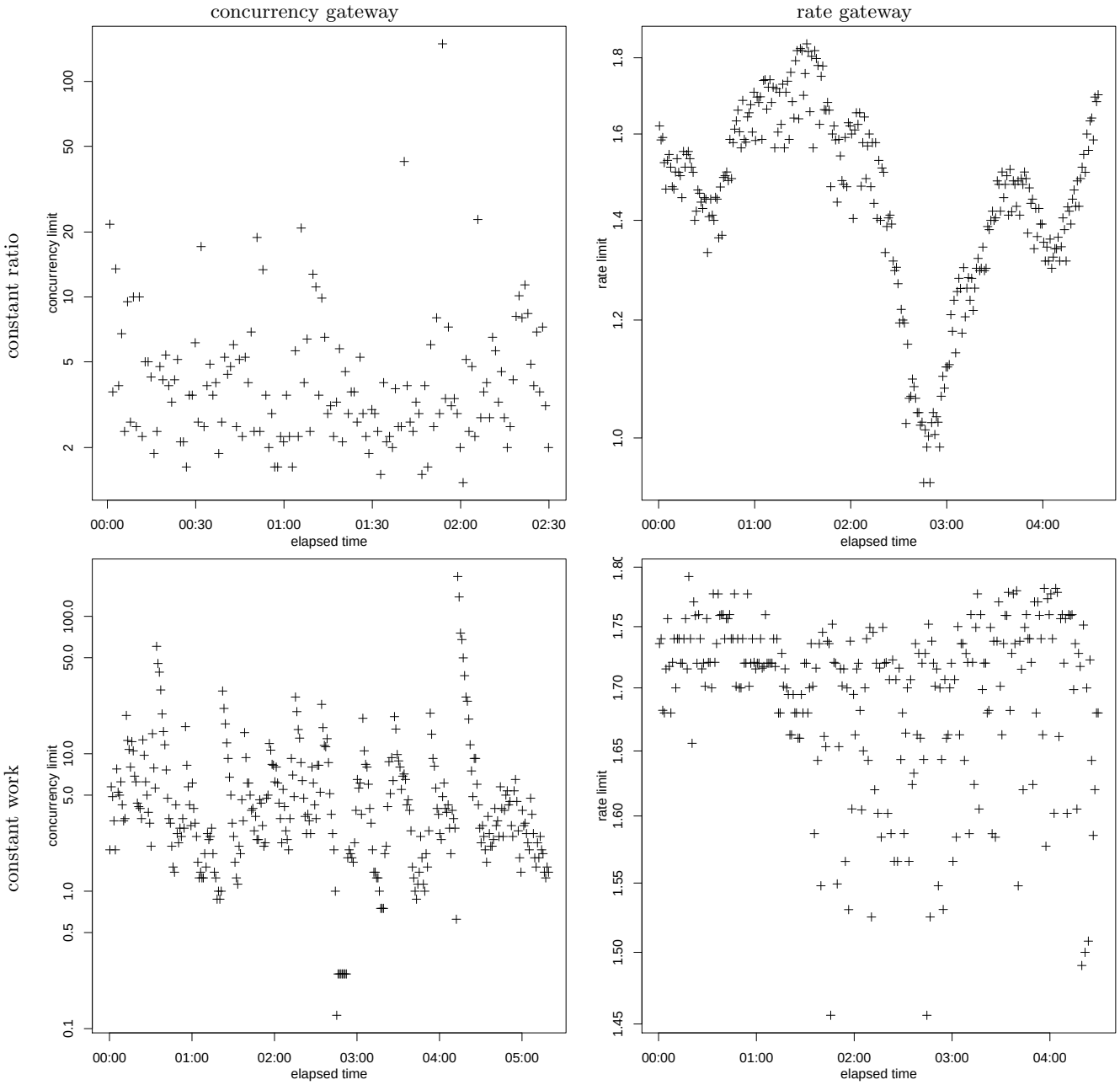


Figure 85: Time series of control settings in automatic control, on-line utilization factor estimates, ∼1 longs/cycle

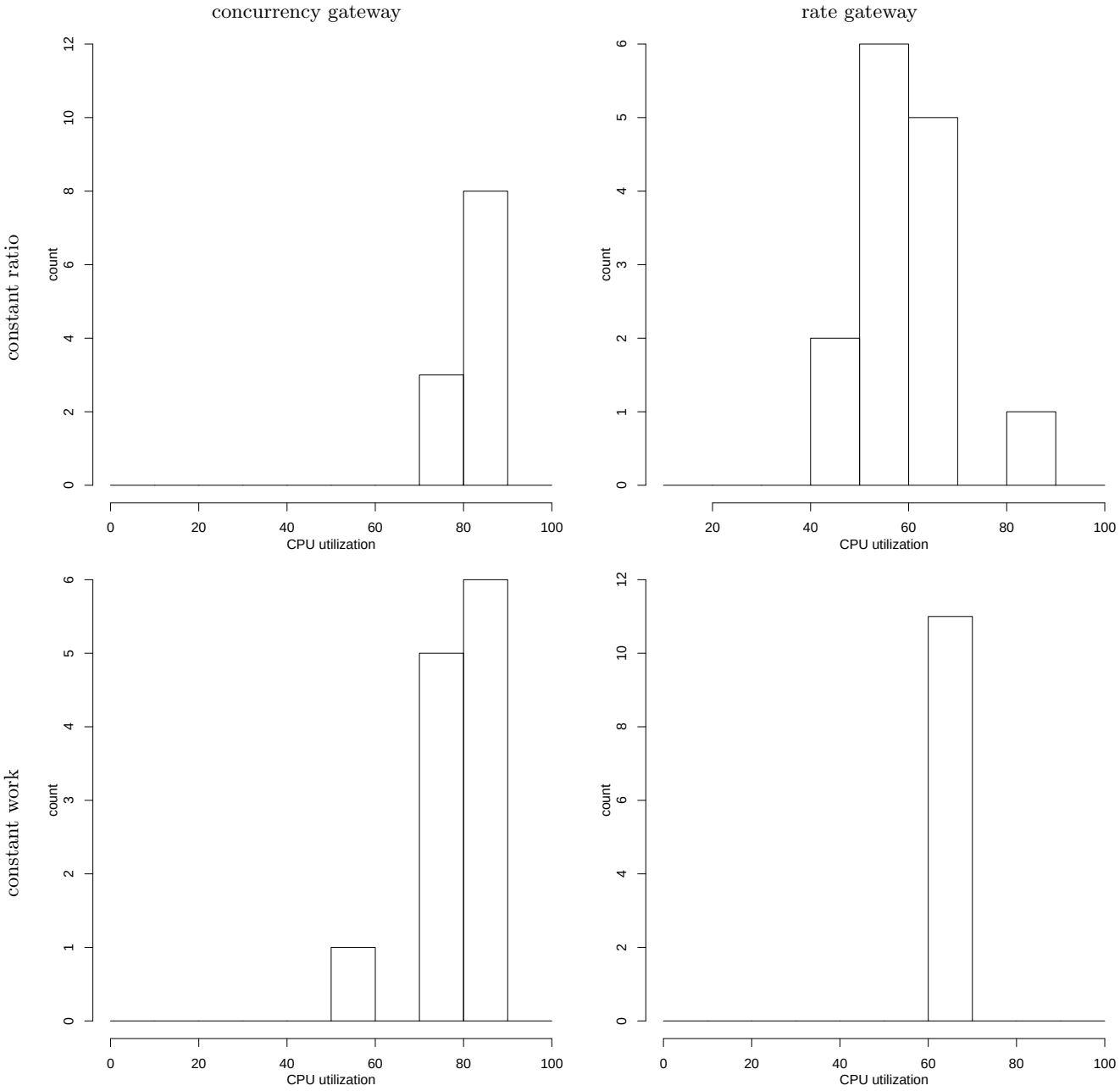


Figure 86: Histograms of 25-minute averge CPU utilization in automatic control, on-line utilization factor estimates, ∼1 longs/cycle

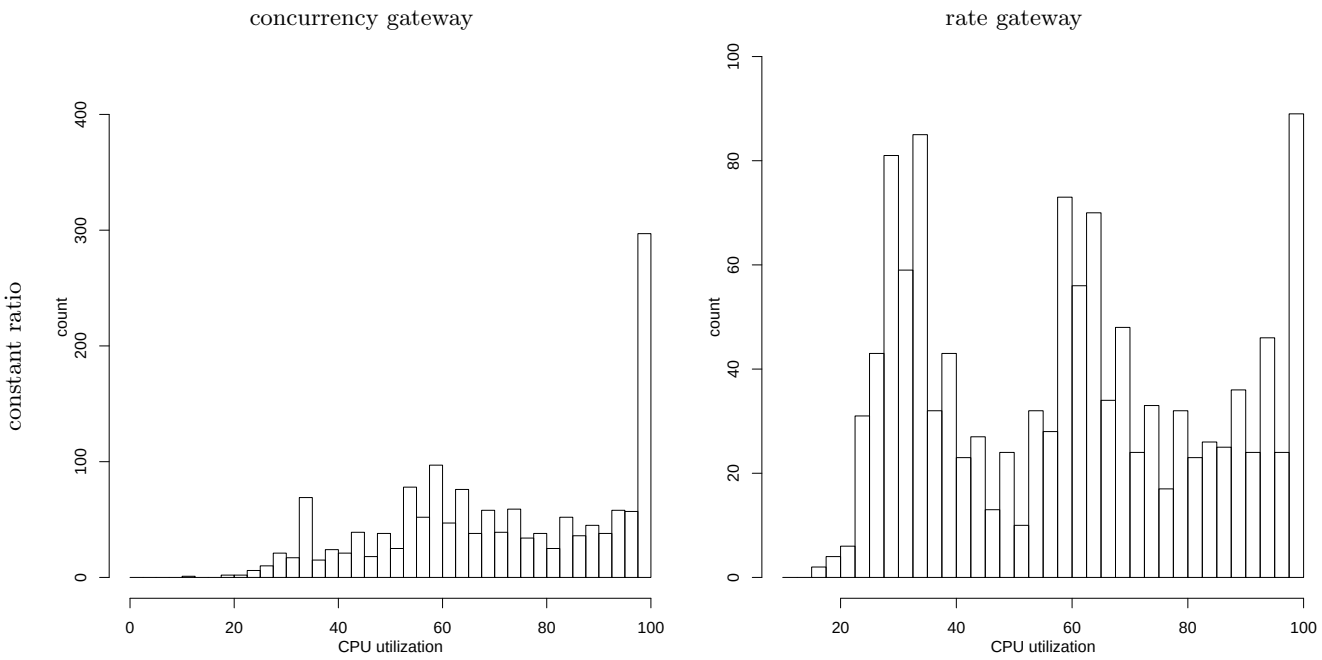


Figure 87: Histograms of 15-second averge CPU utilization in automatic control, configured utilization factor estimates, ∼1 longs/cycle

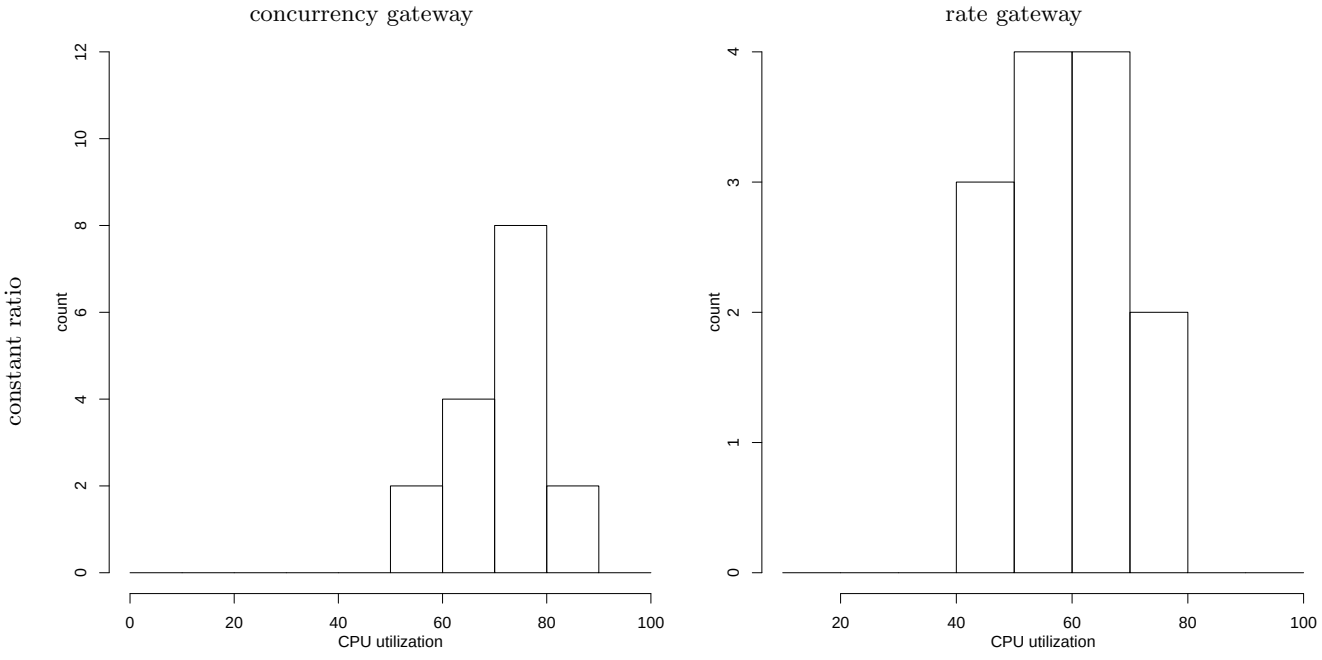


Figure 88: Histograms of 25-minute averge CPU utilization in automatic control, configured utilization factor estimates, ∼1 longs/cycle

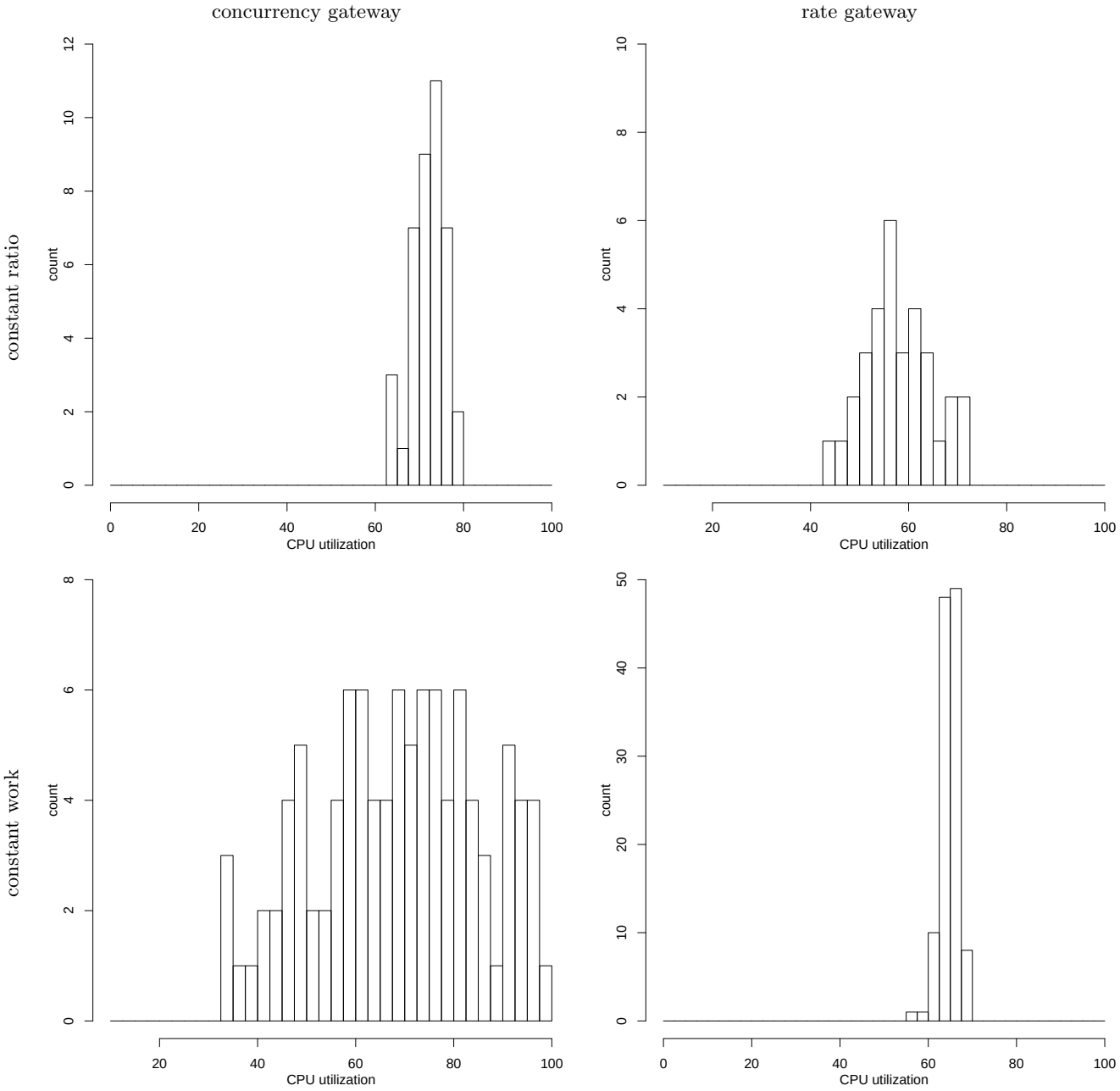


Figure 89: Histograms of 12.5-minute averge CPU utilization in automatic control, on-line utilization factor estimates, ∼100 longs/cycle

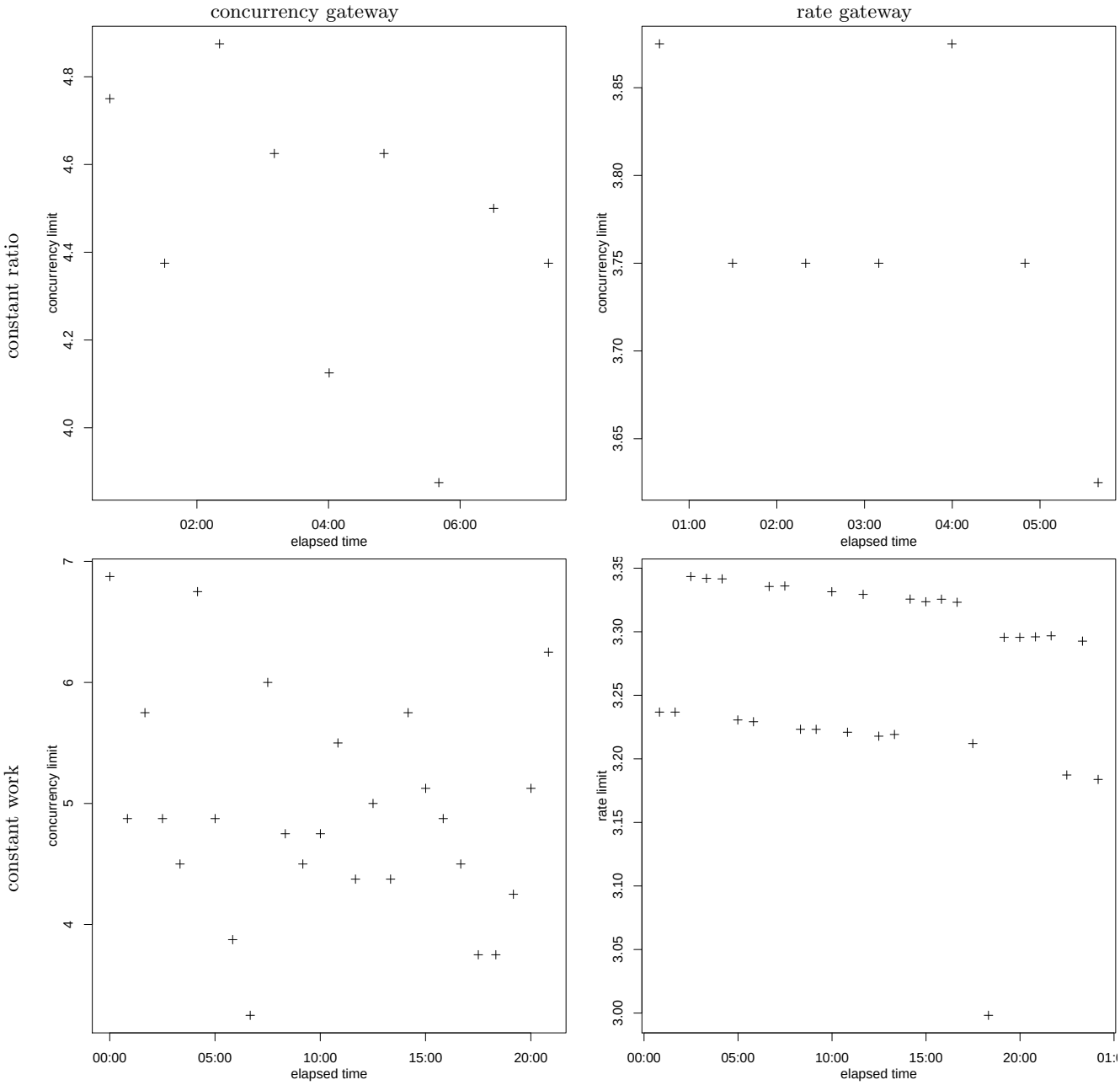


Figure 90: Time series of 12.5-minute averge CPU utilization in automatic control, on-line utilization factor estimates, ∼100 longs/cycle

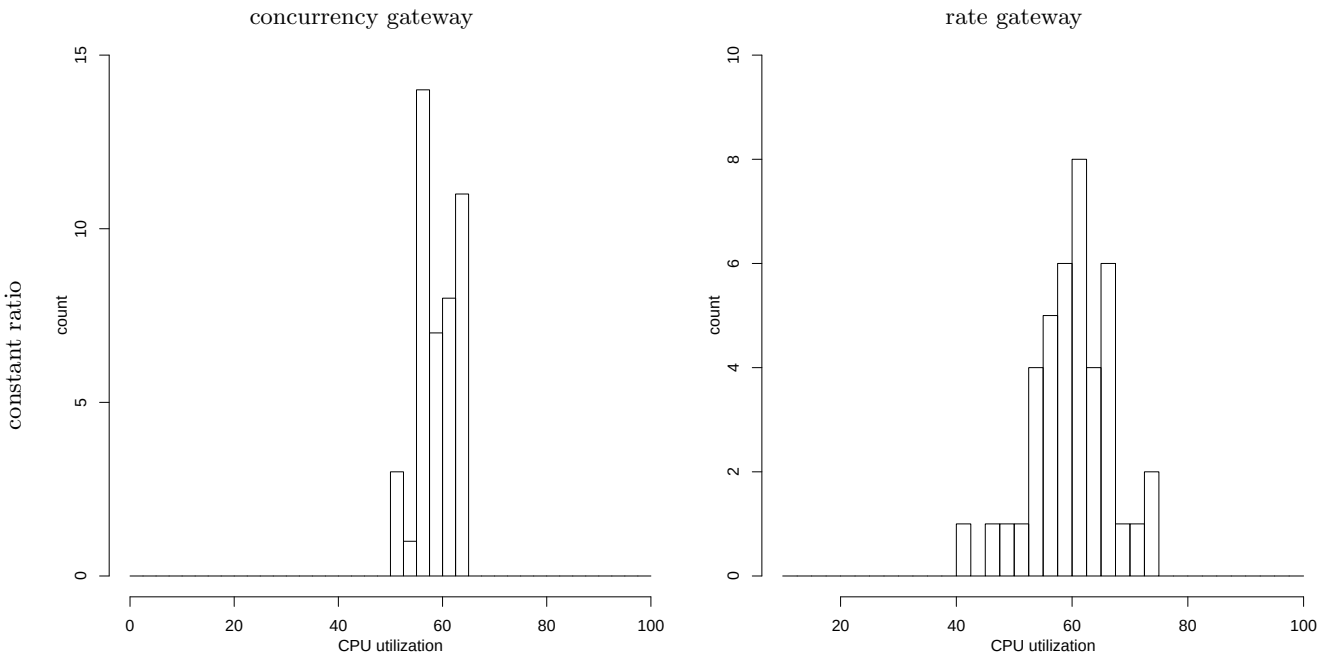


Figure 91: Histograms of 12.5-minute averge CPU utilization in automatic control, configured utilization factor estimates, ∼100 longs/cycle

Now for recommendations. Since multi-class settings are more realistic, we will suppose the objective is accuracy rather than avoiding underload. Table 9 summarizes our recommendations. Although the data are not exhibited here, we find (as you would expect) that when the workload does not have extreme variation both gateways work fairly well. If you know your workload might include extreme variation in service time but will not include high variation in the work/time ratio then the concurrency controlling gateway will give the best results. On the other hand, if you know your workload may include extreme variation in service time but not in work/request then the rate controlling gateway gives the best results. If the workload could include any sort of extreme variation, then the rate controlling gateway is the safest bet; it might not produce the best results in some cases, but it avoids the worst results in all cases.

The fact that different gateways handle different surprises better suggests a hybrid technique might be valuable. In the single-class setting, is there a way to automatically and dynamically determine which dispatching mechanism would work better, and put that into effect? A multi-class setting adds additional complexities — at least when (e.g., as in WebSphere VE [7]) the multi-class management is not simply several independent copies of the single-class management. In this situation, even if the decision for each class is static, how can the two styles of management be mixed in one multi-class solution?

8. REFERENCES

[1] Google AppEngine. http://code.google.com/appengine/.

[2] Microsoft Azure Platform. http://www.microsoft.com/windowsazure/.

[3] D. Bertsekas and R. Gallager. *Data Networks*. Prentice–Hall, 1987.

[4] J. Edward G. Coffman and P. J. Denning. *Operating Systems Theory*. Prentice–Hall, 1973.

[5] S. Elnikety, E. Nahum, J. Tracey, and W. Zwaenepoel. A method for transparent admission control and request scheduling in e-commerce web sites. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 276–286, New York, NY, USA, 2004. ACM.

[6] E. Gelenbe and G. Pujolle. *Introduction to Queueing Networks*. John Wiley & Sons, 1987.

[7] IBM. *WebSphere Virtual Enterprise*. http://www.ibm.com/software/webservers/appserv/extend/virtualenterprise/, 2009.

[8] T. Ibraki and N. Katoh. *Resource Allocation Problems*. The MIT Press, 1988.

[9] V. Kanodia and E. Knightly. Multi-class latency-bounded web services. In *Eighth International Workshop on Quality of Service (IWQoS 2000)*, pages 231–239. IEEE, June 2000.

[10] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance*. Prentice–Hall, 1984.

[11] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi. Cpu demand for web serving: Measurement analysis and dynamic estimation. *Performance Evaluation*, 65(6–7):531–553, June 2008.

[12] G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef. Performance management for cluster based web services. *IEEE Journal on Selected Areas in Communications*, 23(12), December 2005.

[13] B. Urgaonkar, G. Pacifici, M. Spreitzer, and A. Tantawi. Analytic modeling of multitier internet

applications. *ACM Trans. Web*, 1(1), May 2007.

[14] B. Urgaonkar and P. Shenoy. Cataclysm: policing extreme overloads in internet applications. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 740–749, New York, NY, USA, 2005. ACM.