A PERFORMANCE MODEL FOR CORE SYSTEM MONITORING FUNCTIONS

KEVIN J. MITCHELL  
CACI, Inc.  
4114 Legato Rd.  
Fairfax, VA 22033, USA  
kemitchell@caci.com

DANIEL A. MENASCÉ  
DEPT. OF COMPUTER SCIENCE, MS A45  
VOLGENAU SCHOOL OF ENGINEERING  
George Mason University  
Fairfax, VA 22030, USA  
menasce@gmu.edu

Abstract
System monitoring platforms offer advanced functionality to help IT organizations manage their resources and understand system behavior. These platforms collect, act on, and visualize data about running systems. The performance of a monitoring system under a given workload must be understood in advance in order to achieve identified performance objectives. Myriad system monitoring platforms exist for implementation by organizations, each offering distinguishing features. Most exhibit a similar system architecture consisting of a central monitoring server, a monitoring datastore, and hosts and services to monitor. Scheduling, obtaining and persisting the status of monitored hosts and services forms the primary workload placed on the monitoring system. This paper presents a closed queuing network (QN) model that predicts response time and utilization for this primary workload placed on a monitoring system. The model accounts for monitoring frequency, CPU and disk consumption at the monitoring server, and response time of the monitored system. Experiments conducted with the open source Nagios monitoring platform validate the model predictions across several workloads.

1 Introduction
System monitoring platforms offer advanced functionality to help IT organizations manage their IT resources and understand the behavior of their IT systems. These platforms collect data about running systems and chart or otherwise visualize the data. They also allow organizations to identify normal operating ranges for system measures such as utilization and notify system operators when observed values fall outside operating ranges. Finally, these platforms implement data structures tailored for the retention of time series data. All this functionality can help organizations lower maintenance costs by identifying problems earlier, and can provide valuable insight for system evolution.

Despite the compelling functionality described above, organizations need to be cautious about implementing a system monitoring platform. In almost all cases, agents must be deployed to the live production systems to be monitored, consuming limited system resources and potentially impacting performance. Organizations considering deployment should define performance objectives for the monitoring system itself, especially if it is to become a central part of the system operations and support infrastructure or if it will be used as a mechanism to collect or assess compliance with quality of service (QoS) objectives embodied in service level agreements (SLA). Consequently, the performance profile of the entire monitoring system, including the monitoring core and database in addition to the agents, must be understood in advance in order to achieve the identified performance objectives. Finally, platform implementation, licensing and maintenance carry costs that must be taken into account and balanced against the benefits of the platform from a total cost of ownership (TCO) perspective.

Myriad system monitoring platforms exist for implementation by organizations, each offering distinguishing features. Examples include Tivoli [1], System Center Operations Manager [2], Solarwinds [3], and Nagios [4]. Most exhibit a similar system architecture consisting of a central monitoring server, a monitoring datastore, and hosts and services to monitor. Scheduling, obtaining, and persisting the status of monitored hosts and services are the core functions performed by the monitoring system.

Most system monitoring platforms are highly configurable and extensible. Almost every aspect of the system, aside from the core monitoring logic and data
structure, can (and usually is) tailored by the implementing organization, and these customizations affect system performance. One approach that we could have taken in this paper would be to assume a specific set of customizations, such as a web interface or notification mechanism, and include those in the performance model. However, the conclusions of such a study would be useful only to organizations that find value in that specific set of customizations. Instead, we focus on the core system monitoring functions, as the resulting model will have greater applicability across system monitoring platforms. As a result, the model does not account for workload associated with user interfaces for monitoring management and control, or report services covering monitoring data.

The objective of the model is to understand the relationship between the performance of core monitoring functions on the monitoring server and the number of monitored hosts and services. This key relationship allows organizations to plan capacity for core monitoring based on the number of hosts and services to be monitored. We selected the Nagios open source monitoring platform to motivate and validate a performance model to achieve these objectives. However, the methodology and the model presented here can be easily ported to monitoring platforms that have a core architecture similar to that of Nagios.

The rest of the paper is organized as follows. Section 2 discusses the general architecture of most system monitoring platforms. The next section presents an overview of the architecture of the Nagios monitoring system. Section 4 presents a brief overview of closed queuing network (QN) models and the following section presents the specific closed QN model used to model and predict the performance of the core functionality of the Nagios system. The results obtained by using this model in an experimental environmental are discussed in section 6. The paper presents some concluding remarks in section 7.

2 Generic System Monitoring Platform Architecture

Figure 1 illustrates the architecture of a generic system monitoring platform. Users interact with a monitoring server through a user interface, usually web-based. They configure hosts and services to monitor, monitoring frequency, and notification mechanisms. Users also view monitoring results through the interface.

The monitoring server schedules and executes local programs or embedded routines that check the status of monitored hosts and services. For many checks, an agent runs on the monitored host or service to provide information not accessible remotely, and the monitoring server simply interfaces with the agent. A monitoring platform may also support an agent “push” of status from monitored hosts and services to the monitoring server. The monitoring datastore holds configuration, schedule and monitoring results data. It often spans a series of files and may involve a relational database and a specialized “round-robin” or “circular buffer” datastore to hold monitoring results in a bounded amount of disk space.

3 Nagios Architecture Overview

The primary functions of Nagios are to check the status of hosts and services at regular intervals, and communicate the status to users. A host is a device with an IP address accessible from Nagios that should be checked by Nagios. A service is a specific capability on a host that can be checked remotely by Nagios. A check is a specific instance of Nagios polling or querying a host or service to determine the current status. An interval is the desired time (in minutes, perhaps fractional) between checks performed by Nagios. Checks can be active or passive. With active checks, Nagios proactively schedules and executes a check on the monitoring server that determines the status of a host or service and persists the result to a datastore on disk. This processing maps to the core system monitoring functions that are the focus of this paper. On the other hand, a remote host or service submits a check to Nagios under the passive check paradigm. Intuitively, a passive check consumes fewer resources on the monitoring server than an active check, because the status determination and scheduling are all performed on the remote host and not the central monitoring server. This study covers performance of active, not passive checks.

In this context, the number of hosts and services, coupled with the check interval, and the runtime status of the monitored hosts and services, drives the overall intensity of monitoring check load on the Nagios system. Nagios contains three important mechanisms that regulate check load and therefore affect the performance of the system. First, Nagios schedules checks in an intelligent fashion, spreading them evenly over the average check interval to smooth load on the monitoring host. Second, it also interleaves checks across monitored hosts to minimize load spikes on monitored hosts. Finally, Nagios includes a mechanism to set a maximum number of concurrent checks, providing admission control on the monitoring server if so desired.

A Nagios system consists of three main components: the Nagios daemon that performs core monitoring logic, a set of configured programs such as plugins and event handlers run by the Nagios daemon to interact with external programs and users, and external programs such as a web interface that process Nagios daemon output and submit control commands to the Nagios daemon. Figure 2 illustrates these main components and interactions of a Nagios system [5].
Figure 1: Generic System Monitoring Platform Architecture

Figure 2: Nagios System Diagram (see [5])
4 Closed Queuing Network Models

A closed queuing network (QN) models a system of \( K \) devices that handle the load of \( N \) concurrent requests. As each request completes, a new request starts in its place, keeping the total number of requests in the system equal to \( N \). This leads to the characterization of the system as "closed", because requests never leave or enter. For the purpose of this paper, a device can be load-independent (queued with a processing time that is independent of the number of customers in the queue) or delay (not queued, or alternatively, infinite capacity) [7]. The \( N \) requests may be partitioned into \( R \) classes, each containing \( N_r \) requests, where each class \( r \) requires a (potentially) distinct service (processing) time \( D_{i,r} \) at device \( i \) \((r = 1, \ldots, R \) and \( i = 1, \ldots, K)\). The service times \( D_{i,r} \) are assumed to be exponentially distributed, although this condition can be relaxed.

Provided the QN is not degenerate (e.g., the service times and number of requests result in an infinite waiting time), a closed QN will reach a steady state, allowing assessment of long-run average performance. Solving a closed QN involves determining performance metrics including average requests (per class and in total) at each device, residence time (i.e., total time spent at a device, waiting and receiving service) per class for each device, device and system level throughput per class, and device utilization (per class and in total).

Mean value analysis (MVA) [8] is an efficient and widely used algorithm for solving closed QNs. It relies on an important result in queuing theory known as the "Arrival Theorem", which states that in a closed system with \( N \) requests, a request number arriving for service at a device will "see" the long-run average number of requests at that device for a system with \( N - 1 \) requests. MVA uses this result to create a set of simple recursive equations for solving a closed QN. The algorithm starts with an empty system and then iterates, increasing \( N \) by one and leveraging the results from the preceding iteration, and terminates on reaching the desired \( N \). An approximation by Schweitzer [9] extends MVA to efficiently handle multiclass closed QNs with many classes and devices. MVA can be implemented easily in programming languages and data analysis applications. Examples include the spreadsheet formulation in [7] and a Java implementation called JMVA [10].

5 Performance Model

For convenience and without loss of generality, the model adopts the Nagios term check to represent the scheduling, obtaining and persisting status of a monitored host or service by the monitoring server. It assumes the monitoring platform does not independently generate new checks or adjust the check frequency. These parameters can change, but only explicitly through the platform's configuration interface. In this context, we define the following multiclass closed QN model:

\[
\begin{align*}
K &= \text{number of devices} \\
R &= \text{number of check classes} \\
N &= \text{total number of checks} \\
N_r &= \text{number of checks of class } r = 1, \ldots, R \\
D_{i,r} &= \text{service time of class } r \text{ checks at device } i = 1, \ldots, K
\end{align*}
\]

For our model, \( K \geq 5 \), with devices defined as follows:

- **CPU**: a load-independent device representing the CPU of the monitoring server that handles check processing. There should be at least one CPU device.
- **disk**: a load-independent device representing the disk on the monitoring server that stores check configuration and results. There should be at least one disk device.
- **check interval**: a delay device representing the configured time interval between successive executions of a single check. This is essentially a wait time between executions of a check, and models the monitoring frequency.
- **check sleep**: a delay device that models sleep time within a single check execution. Some checks actually contact the monitored host or service multiple times during a single check, potentially waiting between each contact. A ping check that leverages the standard UNIX `ping` command with default behavior is an example. This check pings the target 5 times, waiting one second after each ping but the last, resulting in 4 seconds of sleep time where resource consumption is negligible.
- **monitored system (ms)**: a delay device representing the response time of the monitored host or service, including any network transit time. This time factors into overall check execution time, but does not consume resources of the monitoring server.

Figure 3 depicts the closed QN graphically. The CPU and disk devices are load-independent, while the check interval, check sleep and monitored system (ms) devices are delay devices.

The multiclass model allows for differing service times across devices. In particular, it enables different check intervals, CPU/disk consumption, the absence/presence of check sleep time, and differences in monitored system response time. For example, one may want to check a critical e-commerce web server more frequently than an internal-facing development server. Multiple classes
Figure 3: Performance Model of the Nagios System
allows the model to accommodate these situations. Defining the classes and the associated service demands and per-class check populations (the $N_r$) is an exercise in parameter assignment to be accomplished during application of the model to a specific system monitoring implementation.

Given the above formulation, one can use the number of configured host and service checks—a number readily available to implementers—as request population input to the closed QN model. The model views each check as a persistent request that simply cycles around the system, spending (presumably) most of its time at the check interval delay device. This result has the benefit of providing the direct connection between the functional expression of the load and the input to the model, allowing for easy assignment of values to parameters.

5.1 Application of the Model to Nagios

Applying the model to a specific system monitoring platform involves defining the $R$ classes, the request populations $N_r$, additional CPU and disk devices, and the service demands $D_{i,r}$. We applied the model to an experimental Nagios configuration to validate the parameterization and predictive capabilities of the model. This involved three activities: establishing an experimental environment, monitoring the experimental load, and analyzing the results to determine model parameter values.

5.1.1 Experimental Environment

To determine classes, estimate the check populations $N_r$, and estimate the service demands $D_{i,r}$, we established a representative environment for monitoring under Nagios and conducted a targeted and controlled experiment in that environment. The environment includes three servers: one Nagios monitoring server and two monitored hosts. All servers run Linux. Figure 4 depicts the structure and configuration of the experimental environment.

We implemented each server as a VMware virtual machine (VM) running under VMware Player under Microsoft Windows 7 Professional 64-bit and configured each VM with one CPU and 1 GB RAM. The monitoring server has a 10 GB disk and each monitored host has a 5 GB disk. All of this ran on a laptop sufficiently powered (quad-core, 8 GB RAM) to handle this load easily.

In this environment, we configured Nagios Core and Nagios NRPE to allow monitoring of the two monitored hosts from the monitoring server. Nagios NRPE is a plugin and an associated agent process that allows Nagios to remotely check the internals of a monitored host such as CPU utilization and disk space. Specifically, the configuration contains the following:

- 40 hosts, consisting of 20 "replicas" of each of the two actual monitored hosts. This is a convenient feature of Nagios - one can define multiple hosts with the same IP address but different names.

- Each host has eight service checks. Four monitor public services (ping, HTTP, FTP and SSH) of the monitored host and the other four check private services (current users, current CPU utilization, disk space on the root partition, and current number of processes) via NRPE. This results in 320 distinct services to be monitored by Nagios.

- A host check interval of 30 seconds and a service check interval of 15 seconds.

The above configuration results in five devices as described in the baseline model. Devices 1, …, 5 correspond to the CPU, disk, cl (check interval), cs (check sleep) and ms (monitored system) devices. The configuration also produces two classes ($R = 2$) to accommodate the differing check intervals. Letting “hc” denote “host checks" and “sc” denote “service checks", we have $N_1 = N_{hc} = 40, N_2 = N_{sc} = 320, N = N_{hc} + N_{sc} = 360, D_{ci, hc} = 30 \text{ sec}, D_{ci, sc} = 15 \text{ sec}$.

5.1.2 Monitoring

We monitored the monitoring server under the configured load for $\tau = 15$ minutes (900 seconds) to capture CPU utilization ($U_{cpu}$), disk (sda) utilization ($U_{disk}$) and the number of checks executed ($C_0$). These measures allow for the derivation of $D_{cpu,r}$ and $D_{disk,r}$ via the service demand law $D = U/X$ [7] where $X$ = overall throughput. The monitoring also captured the response time (called the execution time in Nagios) of each check to allow estimation of the response time of the monitored system.

We employed four separate tools to capture these measures. The command:

```
iostat -kx sda 60 16 > iostat.txt
```

captured CPU and disk (sda) utilization at a 60 second interval for 16 repetitions. This number of repetitions are necessary because iostat immediately emits output representing the utilizations since system boot. The second tool employed was:

```
 pidstat -urd -h 60 15 > pidstat.txt
```

The pidstat tool captures process level CPU and disk usage, to allow for analysis below the overall level if needed.

The third tool is nagiosstats and is part of Nagios itself. It can be classified into the program analyzer category of monitors. It provides numerous aggregate statistics about load and performance of the Nagios system, including check and external command
counts over 1, 5, 15, and 60 minute time intervals, and average execution (i.e., residence or elapsed) times and check latencies. We ran the following command at the start of the monitoring to capture nagiostats output after 15 minutes:

```
sleep 900; \usr\local\nagios\bin\nagiostats > nagiostats.txt
```

The fourth tool is a check performance data logging mechanism within Nagios. Through simple configuration settings, one can produce customized logs of host and service check execution, including individual execution (response) times, latencies and results. This tool can also log so-called “performance data” provided by the plugin performing the check. This performance data is about the host or service monitored by Nagios, not the check itself. Unfortunately, this facility does not allow for logging of CPU time or I/Os consumed per check. If it did, one might be able to use that data to derive more accurate CPU or disk service demands by partitioning checks into classes via clustering or other data analysis techniques.

Check performance logs consume a significant amount of disk space, so we employed standard UNIX FIFOs (via the mkfifo command) to facilitate persistence of data only during the measurement interval. Specifically, we configured Nagios to log performance data to FIFOs, and then used the UNIX commands `timeout` and `cat` to persist the entries written by Nagios to the FIFO over the measurement interval to a file for later analysis. The general format of the capture command is:

```
timeout (measurement interval) cat < (name of FIFO) > (capture file name)
```

During experiments, we found that the results from nagiostats and check performance data logging were not consistent. Specifically, we found that the number of service checks reported by nagiostats over a 15 minute interval was never equal to the number of service executions logged by the performance data facility over the same time period. We identified the following three sources for this inconsistency:

- Plugins that do not provide performance data. Nagios would not log the check execution if any value to be logged was empty or otherwise undefined. This would cause the number of checks logged to be below the number reported by nagiostats. We applied a source code patch to the check_procs plugin to enable that plugin to return performance data.

- In another case, we found that the check_ssh plugin did not return performance data when the check failed, and therefore Nagios did not write those checks to the log. As in the previous cases, this would cause the performance data log to underestimate the number of actual checks executed. We could not find a patch to address this problem, so decided not to log check performance data.

- A third contributor is the apparent behavior of Nagios to write performance data to logs asynchronously from the execution of the check itself. As a result, at the beginning of a measurement interval one might see “old” checks. Conversely, one might “miss” checks executed at the end of the measure-
5.1.3 Analysis of Monitoring Results

We imported the 15 (all of the 16 except the first) one-minute CPU and disk utilization samples from `iostat` into Excel using Copy/Paste and “Text to Columns” operations, and then averaged the 15 samples. This produced \( U_{cpu} = 0.2025 \) and \( U_{disk} = 0.0470 \). The `nagios` output reported 1,153 host checks and 21,540 service checks, resulting in \( C_0 = 1153 + 21540 = 22693 \). This results in an observed throughput of \( \lambda = X_0 / \tau = 22693 / 900 = 25.21 \) checks/sec.

As mentioned earlier in this report, Nagios does not currently record and log CPU time per check and disk I/O time per check. As a result, one must rely on the overall utilizations observed during the measurement period to derive the service demands \( D_i \), for each class \( r \) at the CPU and disk devices. Additionally, considering just the overall utilization at only the CPU and disk devices, service demands for these devices across all classes will be identical. Accordingly, the workload model defines \( D_{cpu,r} = U_{cpu} / X_0 = 0.2025 / 25.21 = 0.00803 \) sec for any class \( r \) and \( D_{disk,r} = U_{disk} / X_0 = 0.0470 / 25.21 = 0.00186 \) sec for any class \( r \).

The model also requires the specification of the service demands for the check sleep and monitored system devices, \( D_{cs,r} \) and \( D_{ms,r} \) respectively. The check performance data provided insight into these service demands. We imported the check performance data into Excel and calculated descriptive statistics (see Table 1) for the execution times of the \( C_0 = 22693 \) checks observed during the measurement interval. The statistics indicate variability in the data, evidenced by the coefficient of variation (CV) value of 2.005, and the fact that 75% of the values lie at or below 0.167, yet the maximum is 4.155. This variability indicates multiple classes may be required to accurately estimate the remaining service demands.

Through histograms and further analysis, we discovered that the built-in Nagios ping check exhibits the default behavior of UNIX ping, sleeping for a total of 4 seconds during execution. These checks accounted for all of the checks whose execution time exceeded 4 seconds. After separating ping and non-ping checks, we obtained the descriptive statistics shown in the last two columns of Table 1. These statistics show tighter clustering of execution times around the mean, allowing us to safely use the mean as an accurate basis for determining service demands.

Since all host checks are ping checks, we need only separate the existing “service check” class into two classes—“service check ping” (scp) and “service check non-ping” (scn)—to allow for specification of different service demands at the check sleep device. We now have \( N_1 = N_{hc} = 40, N_2 = N_{scp} = 40, N_3 = N_{scn} = 280, N = 360, D_{cs,hc} = D_{cs,scp} = 4 \) msec, \( D_{cs,scn} = 0 \).

### Table 1: Descriptive Statistics for Check Execution Times (msec)

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Combined</th>
<th>Non-ping</th>
<th>ping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean (( \mu ))</td>
<td>713</td>
<td>101</td>
<td>4045</td>
</tr>
<tr>
<td>Std. Dev. (( \sigma ))</td>
<td>1429</td>
<td>51</td>
<td>19</td>
</tr>
<tr>
<td>C.V.</td>
<td>2.005</td>
<td>0.511</td>
<td>0.005</td>
</tr>
<tr>
<td>Min.</td>
<td>23</td>
<td>23</td>
<td>4006</td>
</tr>
<tr>
<td>1st Quartile</td>
<td>59</td>
<td>55</td>
<td>4033</td>
</tr>
<tr>
<td>Median</td>
<td>112</td>
<td>92</td>
<td>4041</td>
</tr>
<tr>
<td>3rd Quartile</td>
<td>167</td>
<td>137</td>
<td>4053</td>
</tr>
<tr>
<td>Max.</td>
<td>4155</td>
<td>584</td>
<td>4155</td>
</tr>
</tbody>
</table>

The final parameter requiring estimation is the service demand per class for the monitored system delay device, or \( D_{ms,r} \). This service demand should accurately represent the response time of the monitored system to a check request. This time is included in the check execution time reported by the Nagios check performance data logging mechanism and captured during the monitoring as described above. However, the check execution time also includes the time spent using or waiting for CPU and disk resources on the monitoring server. We need to subtract this time from the mean execution time for the class to accurately estimate the \( D_{ms,r} \). This time is the sum of the check residence times at the CPU and disk, or \( R'_{cpu} + R'_{disk} \). We obtained these residence times by solving the closed QN model so far with an arbitrary value (zero) for the unknown \( D_{ms,r} \). The residence times are 9.73 msec and 1.94 msec respectively, summing to 11.67 msec.

Accordingly, each check execution time as reported by Nagios will be reduced by 11.67 msec (0.01167 sec) to arrive at the service demand at the monitored system, \( D_{ms,r} \). Check execution times for ping checks are further reduced by 4 sec each to account for the check sleep time.

All model parameters are now completely specified and are summarized in Table 2.

### 6 Results of Model Solution

Establishing bounds on maximum throughput is a good first step in using a performance model. Under our model, the heavy load bound for check throughput, which is the maximum checks/sec before saturation, will be \( 1 / D_{cpu,r} = 1 / 0.00803 = 124.533 \) checks/sec. Assuming a standard check interval of 5 minutes (300 seconds), this bound implies the monitoring server with one CPU and one disk could theoretically handle up to 124.533 * 300 = 37,359 checks in a 5-minute interval. Through iterative
solving of the closed QN, we determined this throughput upper bound occurs at approximately 4,000 hosts with 10 services monitored on each host for a total population of 44,000 checks. Additionally, the experiment conducted during this study achieved 22693/3 = 7564 checks each 5-minute period, so the theoretical upper bound is about 5 times the load achieved on a laptop computer that also happened to be running the monitored hosts, and at least some basic desktop applications. We believe this supports the validity of the model. While these numbers are indeed plausible, the modeled system would be saturated and would likely experience at least some instability at a lower number of checks.

We solved the model using the “ClosedQN.xls” spreadsheet provided as a companion to [7]. As an initial validation of the model, and especially the service demand derivations, we solved the model using the check populations ($N_r$) from the experimental environment and compared the results to the observed values. Table 3 presents the model outputs (predictions) and experimental results (actuals). As shown, the model accurately predicted response times. For throughput and utilization, the model predicted values for both that were 13% lower than observed. Since both the utilization and throughput were consistently lower, we believe the model accurately predicted these measures as well.

The next step was to solve the model with varying check populations and validate the solution with experimental results. We chose one decreased population and one increased population. For the decreased population, we reduced each per-class population by 50% and repeated the experiment as outlined above. The service demands remained unchanged in the model. Table 4 compares the model output with experimental results. The experiment followed the same approach as outlined above (15 minute observation), but we doubled the host and service check intervals, to 1 minute and 30 seconds respectively, to motivate Nagios to create an overall load of 50% of the original. As shown in the table, Nagios executed more checks than expected, and accordingly throughput and the utilizations of the CPU and disk were higher than expected, although not in the same proportion relative to the model outputs. Response times were consistent between the model and observed results. Overall, we believe the results lend credibility to the model.

For an increased rate, we chose to reinstate the original check intervals, but increased the number of hosts and services by 25% from 40 to 50 and 320 to 400 respectively. We solved a closed QN using these increased populations, with the service demands unchanged from the original case. We then replicated the hosts and services in the Nagios configuration and reinstated the check intervals. Table 5 compares the model output to the results of the experiment, which ran for 15 minutes as in the other experiments. It shows a better agreement for number of checks, arrival rate and disk utilization between the model and the experiment than in the reduced rate case. The experiment reported a CPU utilization 2% higher than the value predicted by the model, reflecting an “error” of approximately 4.5%. This error is pretty good, especially considering the model presented here is an initial model that would go through refinements over time as part of an overall performance engineering methodology.

As a final step, we solved the model for varying check populations up to 43,200, which is very close to the upper bound. Figure 5 is a plot of utilizations produced by the model solution by check population. It shows the bounding role of the CPU in the overall load handled by a Nagios system.

7 Conclusions

We believe our results provide an excellent example of real-world performance analysis. Our conclusions are:

- Performance modeling techniques such as closed QN models and workload characterization can be used to accurately model core monitoring system functions.
- A system of resources that affect check response time, but are not a primary area of interest, can be modeled effectively as a delay device. Delay devices can also be used to effectively model monitoring frequency and check sleep time encountered in monitoring systems.
- In the Nagios monitoring platform, active check throughput is bounded by CPU, so organizations should ensure adequate CPU is available. Additionally, a simple Nagios monitoring server should scale to handle a large number of checks.

As mentioned in Introduction, Nagios is one option among many system monitoring platforms. The model may be applied to other system monitoring platforms using a simple approach. First, understand the platform architecture and how it implements the core system monitoring functions, and compare the findings to
## Table 3: Model vs. Observed - Baseline Case

<table>
<thead>
<tr>
<th>Measure</th>
<th>Model</th>
<th>Observed</th>
<th>Relative Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Checks</td>
<td>20,400</td>
<td>22,693</td>
<td>10.10%</td>
</tr>
<tr>
<td>Throughput (checks/sec)</td>
<td>21.875</td>
<td>25.2144</td>
<td>13.47%</td>
</tr>
<tr>
<td>Response Time - Host (sec)</td>
<td>4.045</td>
<td>4.045</td>
<td>0%</td>
</tr>
<tr>
<td>Response Time - Service ping (sec)</td>
<td>4.045</td>
<td>4.045</td>
<td>0%</td>
</tr>
<tr>
<td>Response Time - Service non-ping (ms)</td>
<td>101</td>
<td>101</td>
<td>0%</td>
</tr>
<tr>
<td>CPU Utilization (%)</td>
<td>17.52</td>
<td>20.25</td>
<td>13.48%</td>
</tr>
<tr>
<td>Disk Utilization (%)</td>
<td>4.06</td>
<td>4.7</td>
<td>13.57%</td>
</tr>
</tbody>
</table>

## Table 4: Model vs. Observed - Load Reduced by 50%

<table>
<thead>
<tr>
<th>Measure</th>
<th>Model</th>
<th>Observed</th>
<th>Relative Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Checks</td>
<td>10,200</td>
<td>11,219</td>
<td>9.08%</td>
</tr>
<tr>
<td>Throughput (checks/sec)</td>
<td>11.1019</td>
<td>12.4656</td>
<td>10.94%</td>
</tr>
<tr>
<td>Response Time - Host (sec)</td>
<td>4.044</td>
<td>4.05</td>
<td>0.15%</td>
</tr>
<tr>
<td>Response Time - Service ping (sec)</td>
<td>4.044</td>
<td>4.05</td>
<td>0.15%</td>
</tr>
<tr>
<td>Response Time - Service non-ping (ms)</td>
<td>100</td>
<td>102</td>
<td>1.96%</td>
</tr>
<tr>
<td>CPU Utilization (%)</td>
<td>8.915</td>
<td>11.63</td>
<td>23.32%</td>
</tr>
<tr>
<td>Disk Utilization (%)</td>
<td>2.065</td>
<td>3.15</td>
<td>34.35%</td>
</tr>
</tbody>
</table>

## Table 5: Model vs. Observed - Load Increased by 25%

<table>
<thead>
<tr>
<th>Measure</th>
<th>Model</th>
<th>Observed</th>
<th>Relative Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Checks</td>
<td>25,500</td>
<td>25,287</td>
<td>0.84%</td>
</tr>
<tr>
<td>Throughput (checks/sec)</td>
<td>27.2709</td>
<td>28.0967</td>
<td>2.94%</td>
</tr>
<tr>
<td>Response Time - Host (sec)</td>
<td>4.046</td>
<td>4.05</td>
<td>0.20%</td>
</tr>
<tr>
<td>Response Time - Service ping (sec)</td>
<td>4.046</td>
<td>4.05</td>
<td>0%</td>
</tr>
<tr>
<td>Response Time - Service non-ping (ms)</td>
<td>101</td>
<td>103</td>
<td>1.94%</td>
</tr>
<tr>
<td>CPU Utilization (%)</td>
<td>21.898</td>
<td>23.88</td>
<td>8.28%</td>
</tr>
<tr>
<td>Disk Utilization (%)</td>
<td>5.072</td>
<td>5.77</td>
<td>12.10%</td>
</tr>
</tbody>
</table>
the generic platform architecture presented earlier. Understand any differences and note for the next step. Second, conduct a controlled experiment to identify classes and associated service demands at the CPU and disk resources. These should be different for different platforms, but the model will cover most situations one might encounter. For example, not all platforms implement a check that sleeps during execution (like ping). For such a case, all service demands at the check sleep device will be zero. Note that this is a change to the model parameters but not the model itself. On the other hand, if the platform splits core processing across distinct CPU and disk resources, one would need to adjust the model by adding CPU and disk devices to model those resources. Third, adjust the model as necessary to reflect the identified classes CPU/disk devices, load the service demand parameters, and solve the model assuming the workload from the experiment. Compare the model and experimental results to validate the model. Fourth, adjust the model and parameters to reflect the expected production workload and a reasonable starting point for CPU and disk capacity, re-solve the model, and interpret the results.

References