ABSTRACT
Organizations that operate in the e-commerce domain typically have hundreds of web-based applications that have to be updated when new software or stacks are launched. This requires extensive testing to maintain quality of service, while minimizing downtime. In this research, we propose a structured method for classifying applications and testing new stacks for each type of application thoroughly before deploying all applications on the new stack. The method uses an emulation-simulation approach and would contribute to the organization’s leadership in providing high quality service to its customers while eliminating unnecessary expenditure on excess testing and maintaining multiple stacks, resulting in significant cost savings.

**KEYWORDS:** Web-based applications, software updates, reference architecture, service risk, emulation-simulation framework

**INTRODUCTION**

Web-based applications are central to delivering services at many small and large organizations. They provide a window through which customers can connect with organizations online, to take care of various needs, such as paying bills, managing accounts, and booking tickets. At the organizational level, there are typically several individual web-based applications. A medium size company can have more than 250 applications while large companies can have upwards of 3000-5000. Multiple applications are needed to perform a business transaction like paying your bill, setting up a registration, creating a travel itinerary. These transactions can range from less than one second to days with multiple batch, online or manual activities. Organizations deal with the task of managing Enterprise systems, which are complex interconnected set of individual applications sitting on myriad set of infrastructure. The infrastructure has multiple components like operating system, system utilities, application utilities and connectivity programs. During each application performing its part in a transaction, these components can be called up to support the application’s activity. These components are usually made by different vendors and could have compatibility issues. These components are upgraded from time to time and these upgrades might also cause compatibility issues, since all components do not have the same upgrade cycle.

Organizations have reference architectures (or stacks) that are put together by the IT departments to standardize the components into a few bundles for application development teams to use in their applications. This leads to standardization and a level of abstraction for development teams. Once a stack is created with a version of components then applications start to consume the stack or reference model. Over time, the components will be upgraded or patches provided, which will lead to newer versions of the stack needing to be created. There are usually major and minor releases of the stack and usually can follow a 1-3 year cycle for a major release and a 3-6 month cycle for a minor release.
When one or more components in a stack are upgraded, the current practice involves testing the stacks at a high level leading to some risk being passed onto the applications. This is due to testing done with very highly simplistic or unreal code samples or snippets. Reference architecture teams do not have the time or the wherewithal to test all possible applications on the new stack. They might not even choose a set of real applications as due to the interdependencies the applications they have to setup all other system dependent applications with their corresponding stacks and data. So they setup some simplistic code or applications that might not have any relation to the real-world behavior of applications. So when application teams put their applications on these stacks, there are usually issues that slowly reduce over time but significant amount of time is lost. Hence, application teams do not want to take the risk and sometimes wait to upgrade their applications to the newer stack. Also applications want to perform more detailed regression testing of their application with the new stack to reduce the risk.

As applications teams do not want to upgrade frequently and are not comfortable with the risk associated with automatically upgrading the applications to the new reference models, there could be multiple versions of the stacks in the environment causing many different variations. This causes additional issues since support teams have to be aware of many different versions and patch levels before identifying and resolving customer complaints. It could also cause applications on older versions to not being able to leverage newer versions of the methods with additional functionality. Application upgrade cycles are sometimes expensive and cannot be done with other application changes, as the degree of change could be significant due to the infrequency of upgrades. It goes from standard upgrade to a major revision or in some cases a redesign.

The presence of multiple versions of the stack and the need to support all versions causes cost overruns and potentially, a lower service quality since older versions of the reference architecture cannot leverage all the advantages of the new reference architectures. The purpose of this research is to develop a method to deploy the applications on the new stacks in an efficient manner. This research was motivated by the presence of this problem in the technology division of a Fortune 100 financial services firm. Hatzakis et al. (2010) point out the increased use of online transactions in the financial services industry and explain how quality of service, measured by response time as well as correctness of retrieved information, is a very important aspect of service delivery. Quality of service depends on several factors, including, capacity of resources (servers in this case) and the correctness and compatibility of content available on the servers. A typical Web-based application system consists of Web servers, application servers, various external servers (for example, databases and mainframe computers) and Web-application logic that controls the flow and sequence of customer requests through all the elements of the system. The application server is central to any Web-based application as it serves as the traffic controller and controls the flow of information between the end user and across the several tiers in the network. The different kinds of applications reside on the application server. At the financial services firm we were studying, since the majority of applications were financial applications, the application teams did not want to risk automatic upgrades of the applications to the new reference models without extensive testing. Hence
there were multiple versions of the stacks in the environment, causing several different variations.

We propose a robust method where real applications are tested on the new stacks. At the same time, we do not recommend exhaustive testing. We recommend an initial analysis of existing applications to classify them into broad groups based on system behavior. Research (Malek et al., 2012; Aleti et al., 2013) suggests that applications behave differently and their behavior depends on several factors such as heap size, connectivity, cpu, wait times, and memory, to name a few. Our early data analysis (data from the financial services company) showed that the behavior of an application can be characterized based on whether the application is memory intensive, CPU intensive or a combination (memory and CPU intensive) and other individual parameters play a much less important role in characterizing it’s behavior. If we can test 2-3 applications of each type extensively under various scenarios (slowdowns, high demand, breakdowns, etc.), we would have a clear picture of system behavior for each type of application. With this initial testing, we can establish baseline standards for performance on the existing stacks.

When new stacks are brought online by the reference architecture teams, we can now test only one or two applications of each type. If the performance metrics with the new stacks match the ones established with the old stack with an acceptable confidence level, all applications below a pre-determined criticality level can be automatically upgraded to the latest stack. The applications that have been tagged as “critical” during the initial calibration phase can be tested individually before deployment on the new stacks. This method would significantly reduce cost of deployment on the new stacks, and the downtime as well. A key component of the proposed method is an emulation-simulation framework to establish baseline performance standards on existing stacks for all the applications and classify them into broad categories.

The rest of the paper is organized as follows. First, we present an overview of the related literature and identify the gap in the literature addressed by this research. We next provide a detailed description of the application mapping and emulation testing procedure. The following section describes the discrete event simulation model in detail, presents results, validates our model, and discusses the managerial implications. The final section concludes with directions for future research.

LITERATURE REVIEW

There are two classes of research on performance modeling for Web-based services. One stream of research focuses on maintaining the hardware and software required to keep the Web-based services updated, while the other focuses on Web-server performance from a capacity planning perspective. The first stream of research usually is focused more on Computer Science. Miedes and Munoz-Escoi (2012) present a survey about automatic software updating. This stream of research focuses on tools developed to automatically update software
on distributed systems. Our research moves a step back in the chain of decision making and helps to decide which class of applications should be updated together.

The second stream of research presents a class of management science models for studying the performance of web-based applications. One class of models (Menascé, 2002; Cao, Anderson, Nyberg, & Kihl, 2003; Liu, Heo, Sha, & Zhu, 2006) describes Web-based applications as a single tier architecture. They model the Web server performance at a high level, and are useful for identifying performance trade-offs making higher level server sizing and allocation decisions. A second class of models model Web-based systems on a lower level and address many of the system interactions directly. One important system interaction is called resource locking. Resource locking occurs in multi-tier architectures. When an upstream server resource is kept occupied and waiting while it is waiting for a response from a downstream resource, it is referred to as resource locking. It is quite different from blocking and has a big impact on system performance and hence several aspects such as capacity planning. Reeser and Hariharan (2002), Urgoankar et al. (2005), Ramesh and Perros (2000a, 2000b) and Mohan et al. (2014) have addressed resource locking directly while modeling the performance of Web-based systems.

This research builds on some of the work from the second stream of research. We are utilizing a simulation model to replicate the performance of a Web-based application under several scenarios in order to characterize its performance. This characterization helps determine an efficient procedure for deploying applications on new software stacks.

APPLICATION MAPPING AND EMULATION TESTING

The initial challenge in developing a re-usable model of application server performance is the need for empirical data on how this central tier in the architecture performs when upstream loads and downstream delays vary under different patterns. To address this challenge the team developed a two-step approach (Figure 4) using emulation to first characterize application server behavior under production-like conditions followed by development of a general purpose simulation that could determine resource requirements and performance impacts without the excessive cost and time required for emulation testing.

A typical web-based application consists of web servers (WS), application servers (AS), and various external servers (ES, for example, databases and mainframe computers) and application logic that controls the flow of customers’ requests through all the elements of the system. Web servers transmit user requests to the application servers where the application software translates each request into transactions, transmits the transactions, collects responses and adds promotion or personalization to complete the overall eBusiness request. This requires memory caching, parallel and sequential operations, and resource locking. A representative application architecture is shown in Figure 5.
Figure 4: Emulation-Simulation framework

Figure 5: Architecture of a typical web-based application
Resource locking is a unique behavior that occurs in this middle tier of multi-tiered service oriented architectures – the application server must consume and hold resources such as memory and connections when requests arrive and while waiting for external services to complete processing and respond. This introduces interaction and queuing effects resulting in non-linear consumption patterns of computing resources that are not readily characterized (Mohan et al. 2010, 2014). To gather empirical data on how application servers behave when loads and service delays vary, the team developed a portable emulation testing capability that relied on two key elements: 1) a load server that could generate controlled, statistically varying user loads at increasing stress levels, and 2) a stub server that could emulate external service responses with controlled delays. These areas are depicted graphically in Figure 6 as shaded overlays on a typical call flow between the tiers for a single business function. The reusable, object-oriented load generator and stub server software we developed to perform emulation testing could support multiple test threads and the scheduling of loads and external service delays based on arrival and delay patterns that mirror a production environment.

Configuring this testing environment and subsequent simulation model required a representative sequence of calls to the external services to be mapped. Application mapping included: 1) identification of representative business functions to be modeled, 2) mapping the sequence of services required for completion of each business function and 3) identification of the corresponding external servers for each of these services. The numbered background in Figure 6 shows a typical service-call map that sequences how the emulation environment will operate using actual logic from the live application software. This figure shows the specific sequence of steps that occur when a user interacts with the application via the browser and web-server, and in turn how the applicationserver interacts with the external services that retrieve information and interact with systems of record for service transaction processing. The square blocks surrounding each system element organize the input and output arrows for graphical clarity. Even a very simple use-case including a field entry and mouse click to look up customer data generates this type of complex calling sequence. The mappings were derived based on expert knowledge of the particular application under study and provided the blueprint for structuring a discrete event simulation.

Figure 6: Emulation testing framework
The input data necessary for emulation tests and simulation modeling are the response times and success rates for the different external servers and the demand arrival pattern at the application server. To characterize these factors, production logs from the live application were obtained providing sufficient data to develop statistically valid representations of system behavior. An example analysis for characterizing one external service is provided in the appendix. Similar analyses were performed for each external service and for the user load on the production system to identify the delay and load patterns and their current nominal levels. Next the team scaled up the loads and delays for a designed set of experiments to characterize server consumption under varying environmental conditions. Initial stress tests were conducted to characterize the maximum load and service delay levels that the test server could handle before becoming critically overloaded. Then full factorial experiments were performed over a range of operating levels to characterize the performance and consumption footprint of representative business functions under varying conditions.

The emulation environment was able to provide representative measurements of application server performance under controlled conditions because the actual production application server software was running on a complete production software and representative hardware stack. Only the loads coming from the web server and the external service responses were emulated; duplicate production http:// calls and .xml files were sent and received respectively by the load and stub servers – to the application server they looked exactly like production interactions. Network bandwidth and delays were also tested and found to have negligible impact on the time to respond. There was one factor that we could not entirely control, and that was the time delay for authentication of a valid user account at each user transaction. For security reasons this authentication could not be bypassed, adding a potential source of error for our testing, but coordination with the owners of the authentication servers allowed the team to test during periods of light use, minimizing errors from this effect.

**Emulation Testing Results**
To evaluate the emulation test results, data was captured both at the test load generating server and at the target application server. At the load server we were able to observe the load level over time, response durations, and the success or failure of each business function. At the application server we used monitoring agents to capture the processing levels and consumption of key server resources including the percentage of consumed CPU and the consumption of heap memory. Correlating the data from both servers on a time scale enabled empirical correlation between the varying consumption levels and response times with loads and latencies. This provided benchmarks that we would later match with a discrete event simulation. An example of the observed performance is shown in Figure 7 where the response durations are seen to improve as the external service delays decrease; however as loads increase to critical levels the system stability degrades with increasing service time variability and service failures.

Figure 7: Load server sees improving response times as external service delays shorten, yet failures (seen in red) increase as loads reach critical levels

Results from the initial tests determined loading levels that stress the system and degrade performance, and identified boundary conditions for sizing. These tests increased user counts with each user generating sequential requests at 100 millisecond intervals as soon as the prior request is complete. The 100 ms interval is necessary to allow resources to recycle, avoiding a race condition. The effective user count was increased in increments of 80 until the load was
sufficient to cause significant service outages. This occurred at the 240 user-thread level. The test results shown in Figures 5 used fixed delays for the stubbed services to more clearly discern the impact of increasing the external service delays. These fixed delays decrease at 1 second intervals from 6 seconds to 3 seconds for all external services.

Identification of the Bottleneck Resource

The potential resource bottleneck parameters tracked at the application server level included CPU, Java Virtual Machine (JVM) heap memory, swap space, and free memory. For the particular application functionality being tested, CPU was identified as the bottleneck resource (See Figure 8). The application under study was not very memory intensive because the application returns only small amounts of data to the user. Heap memory was also significantly consumed but was not a bottleneck, reaching a peak of 50 – 60% of the configured memory allotment. It was important to understand this result in the context of the test server configuration so that simulation results could be scaled for a production environment.

Tests with Distributed Loads and Latencies

Once initial stress tests were complete, tests were run with statistically distributed loads and both fixed and exponentially distributed external latencies. Because of the emulation testing approach, the load on the server is a function of both the request arrival rate and the number of pooled user accounts available to execute the requests. The tested application is used in call centers so users typically log in once at the beginning of their shift and are ready to handle incoming calls. For this reason the emulation load server first initialized a set of accounts to eliminate log-in processing as a part of the test, allowing the system to equilibrate to a production like state. A boundary setting was established for the requisite number of user accounts at which the test server could be crashed and tests were run with a user pool at this
level. From the production log data we characterized the nominal production load level currently on the system using a Weibull distribution that best fit the observed data. There were a small number of initial beta users in the call center using the new system; however it was scheduled to be rolled out to at least a thousand more. Statistical loads were generated at 500, 1000, and 1500 times the current nominal level, allowing us to capture the effect of projected loads on the system. Examples of critical resource consumption for 1500 times the nominal load and 8 second fixed delays for each service is shown in Figure 9 which includes CPU and JVM heap utilization, the two system resources determined to be critical for evaluating server requirements for the target application. At these heavily loaded settings the test server settled into a steady response pattern averaging around 40 seconds with occasional peak times between 80 and 100 seconds. Similar utilization curves for these key resources along with response time patterns were obtained for each experimental level of load and delay.

Figure 9: CPU and JVM heap consumption and total process response times at 1500 times nominal load and 8 second service delays shows how critical resources are consumed under stress and the resulting impact on performance
One chief benefit from using the emulation / simulation approach is the learning it affords for determining how resources are consumed by different types of applications and how this impacts server performance. Observing actual resource consumption and performance on a test server over a range of loads and latencies, and then simulating the application server at the same loads and latencies using different patterns for consuming resources and generating delays allowed us to test a number of different simulation approaches for their representational accuracy. This provided a powerful, empirical approach for developing and validating simulation models to predict application server performance for different types of web-based applications. Some applications are CPU intensive and others which return large data-sets, graphics or multi-media files are more memory intensive. For the call-center application in our work we were able to observe patterns exhibited by a lightweight application that processes transactions that return relatively small data-sets. This type of application was found to be CPU resource constrained and observed to consume resources in specific ways.

From the designed emulation experiments we observed the following consumption trends with increasing loads and latencies: 1) increased utilization of CPU and heap memory with increased load, and 2) increased utilization of heap memory with increased delays for external services to respond. The heap memory results confirmed trends predicted by the earlier resource locking analytical model (Mohan et al. 2009) indicating that some memory is consumed and held while waiting for external services to respond; however CPU resources appear to be seized and released more immediately at the point in time when request information is being processed. Based on the observed behavior we determined that resource consumption for our test application could be simulated using two primary consumption patterns: 1) a resource locking pattern that captures and holds resources while waiting for subsequent downstream activities to occur, and 2) an instantaneous use consumption pattern that consumes and almost immediately releases resources. These two consumption patterns are depicted in the first two rows of Figure 10. These patterns occurred in combination such that a portion of required resources are consumed and released instantaneously while additional portions are locked and unavailable until the external call or the entire service process is complete.
We also postulate that memory intensive applications that display large volumes of information or media files, or are used to upload large amounts of data will need resource consumption patterns that seize and release resources in the two patterns shown at the bottom of Figure 8. The large data return pattern would seize significant resources such as memory when the external service returns a large file, while the large data upload pattern would consume significant CPU and memory resources while streaming a large file to an external server. This second set of patterns was not applicable to our initial pilot but is planned for incorporation in the capacity planning methodology for further testing and use on memory intensive applications.

DISCRETE EVENT SIMULATION

A discrete event simulation model for analyzing the resource loading and performance of an application server was developed that mapped the application calling sequence and consumed application server resources using a blend of the resource locking and instantaneous consumption patterns outlined in Figure 10. An example of the simulated entity flow for resource locking consumption is provided in Figure 11.

Figure 8: Resource consumption patterns observed and postulated
A, B, C, and D represent discrete amounts of resource consumption.

<table>
<thead>
<tr>
<th>Resource Seize/Release Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Consumption Pattern</td>
</tr>
<tr>
<td>Resource Locking</td>
</tr>
<tr>
<td>Instantaneous Use</td>
</tr>
<tr>
<td>Large Data Returned by External Server</td>
</tr>
<tr>
<td>Large Data Upload to External Server</td>
</tr>
</tbody>
</table>

Figure 11: Resource locking logic is used to simulate a portion of total resource consumption. This pattern was found useful to generate consumption levels that closely match test data for heap memory across varying loads and service delays.
This pattern seizes resources when a request arrives, and again when external service calls are made. Then it releases resources when the external services respond, and again when the original request is complete. This generates resource consumption levels that increase both with loads and with increased external service delay times, but with increasing delay times the pattern created too steep a consumption curve to match the observed resource consumption behavior for heap memory. We found that a pure resource locking or instantaneous use model could not replicate the memory consumption observed in the test server, but that by blending the two models with a percentage of each we could tune the simulation to match the observed levels of resource consumption across the test range of loads and delays. This approach had face validity with the software engineers familiar with the application because when a request or external service call is processed some objects are persistent, while others are used briefly and destroyed, allowing garbage collection to quickly recover the memory resources. In support of this, we observed that garbage collection activity was quite high, for example averaging between 40 and 50 garbage collections per minute during the test run depicted in Figure 9. Once the correct percentage of persistent locking was determined, the simulation model consumption levels were self-tuning to the test results over the full range of loads and latencies observed in the designed emulation experiments.

The simulation approach relies on mapping an application’s call flow to develop a routing for each business function to be simulated, so we developed a general purpose capability to map and load application call routings into the simulation through a spreadsheet interface. Then the resource consumption patterns were applied and tuned to characterize the impact of loads and latencies on the consumption of CPU and memory resources. Finally the timing and performance of the simulated server was tuned to match the observed performance of the application server by running it under identical loads and latencies and adjusting the time required to acquire system resources as a function of the utilization level of CPU resources. This approach generated overall resource consumption and processing time distributions that closely matched the performance observed during testing over a range of loads and latencies,
and was a reasonable abstraction of the processing dependencies known to take place in the Java operating environment.

**Refining the Simulation Model**

Comparisons of initial simulation runs with the test data helped identify additional factors that were important for a simulation model to accurately represent server performance. For example, viewing the application server logs for each test showed that there is a baseline level of resource consumption occurring prior to applying any test loads. This baseline level of overhead is needed to support the application server operating system and other application and networking services. This overhead was represented in the simulation by using a disjoint network in the model that would seize resources at the start of the simulation and hold them for the duration of the run. Consumption of these baseline resources was analogous to “booting up” the server operating system, web-server and application software, and then logging in the call center users to be ready to accept customer calls.

We also noted that observed delays for end-to-end business functions were longer and more variable than could be accounted for if we considered only external service delays and nominal processing times within the application server. Application servers have to maintain a number of other processes that consume resources in addition to the specific processing of customer requests and external service calls. This requires additional mechanisms for incurring resource consumption beyond the instantaneous use and resource locking patterns derived from our initial analysis of the test data. Comparing test data to the simulation helped the team identify and model the fixed and variable processing overhead delays that need to be considered. In particular the team was able to develop and refine a dynamic mechanism to represent processing overhead that is load dependent. This was achieved by adding a dynamic resource acquisition timing delay to the model that is consumption dependent.

**Modeling Resource Acquisition Time**

The total turnaround time for a business function is dependent on the external service response times as well as the internal processing time for the application server to execute the external service calls and process their results. Each simulated external service request is assigned a success rate and response time the same as in the emulation stub server. An additional wait time is also imposed for simulated resource acquisition that is a function of the current utilization level of CPU resources in the system. A dynamic “Resource Acquisition Time” (RAT) was added to the model to provide a mechanism for adjusting the simulation to generate total delays that correspond to observed behavior. The RAT delay is an abstraction of resource processing delays caused by load-dependent processes such as garbage collection, memory management, connection pooling, and connection pool maintenance.

A RAT equation was used to tune the time taken for internal processing that the application server requires for the allocation of CPU and memory resources. Because the observed time taken by the application server to acquire resources at each stage appeared non-linearly dependent on how busy the server is we fit an exponential function of the CPU utilization level to define the average time required to acquire resources as shown in Equation (1).

\[
\text{Average RAT}_{\text{sim}} = (\text{RATParam}) e^{\text{RATParam} \times \text{CPU Utilization}}
\]  

(1)
The RAT parameters A and B are used to tune the simulation model to generate characteristic application server processing delay patterns relating the server’s average Resource Acquisition Time (RAT) to the current level of CPU utilization. This equation and values for the tuning parameters A & B were arrived at empirically to enable the simulation to closely match observed test server performance. Empirical tuning was necessary because the complexity of the internal operating system and Java Virtual Machine software precludes a purely analytical solution. Tuning starts with a best-fit curve using low and high utilization tests with fixed latencies to match the simulated total response time to observed performance. Exponentially distributed processing times for each of ten service calls and returns averaging 100 milliseconds were used at each call or return to match the minimum observed delay of one second. Since there are multiple service calls, RAT tuning requires subtracting out the summed processing and external delay times. Because there are four resource acquisitions, tuning equation (2) is used to fit the curve in Figure 12 to the observed data by adjusting A and B in equation (1).

\[
\text{Average RAT}_{\text{Observed}} = \left( \text{Observed Delay} - \sum \text{Process Time} - \sum \text{Ext Service Delay} \right) ÷ 4
\]  

(2)

Figure 12: Acquisition time for server resources is dependent on the CPU because processing time is required to make resources available. An exponential tuning equation fits the curve to observed levels of CPU utilization and response time.

Analysis of observed total response times from the tests showed them to be Gamma distributed which led us to use summed exponential distributions to generate the total delays. The RAT at each of four resource acquisition events is sampled from an exponential distribution with a fluctuating mean that is pegged to the current level of CPU utilization using the exponential tuning equation in equation (1). Using CPU consumption and response time patterns observed during emulation testing to calculate the RAT parameter values allowed us to tune the simulation model for the particular server and application functionality being tested, enabling an accurate, valid simulation of the impact of load and external latency conditions on server performance. Using four distinct stages for resource acquisition as shown in Figure 10, and sampling the RAT from an exponential distribution centered on the Average RAT calculated with equation (1) produced total response time delay distributions that closely matched observed
results. Statistical tests for difference between means and examination of the response time distributions provided validation that the simulation was a reasonable abstraction for predicting system behavior. Figure 13 provides comparisons between tested vs. simulated utilization of CPU, the bottleneck resource and between the total response time distributions whose means and standard deviations vary by a second or less over the tested range of loads and latencies. Sample t-tests for difference between means at 1000x are provided in the appendix.

Figure 13: Comparisons between simulated and tested performance validate that the simulation provides an adequate abstraction of system behavior. The results for simulated vs. tested CPU utilization fall within a few percent and total response times have means and standard deviations that differ by a second or less.

<table>
<thead>
<tr>
<th>Load</th>
<th>Average CPU Utilization</th>
<th>Maximum CPU Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Latency</td>
<td>Latency</td>
</tr>
<tr>
<td></td>
<td>Nominal</td>
<td>Double</td>
</tr>
<tr>
<td>Simulation 1000x</td>
<td>40.3%</td>
<td>40.6%</td>
</tr>
<tr>
<td>Test 1000x</td>
<td>40.4%</td>
<td>38.6%</td>
</tr>
<tr>
<td>Difference</td>
<td>-0.1%</td>
<td>2.0%</td>
</tr>
<tr>
<td>Simulation 1500x</td>
<td>58.5%</td>
<td>58.4%</td>
</tr>
<tr>
<td>Test 1500x</td>
<td>57.2%</td>
<td>55.4%</td>
</tr>
<tr>
<td>Difference</td>
<td>1.3%</td>
<td>3.0%</td>
</tr>
</tbody>
</table>

SIMULATION RESULTS

Initial runs with the simulation were used to refine and tune the model leading to a representation that generated simulation results that are consistent with the testing data. In particular, we learned that CPU consumption could be modeled by combining the instantaneous consumption pattern with initial and dynamic overhead processing patterns. For heap memory, however, it was necessary to combine the resource locking and instantaneous use consumption patterns to generate consumption behavior that matched the test data. These pattern combinations were reviewed with systems engineers and determined to be reasonable, and generated patterns of resource consumption that closely match the test data without having to tune the model for each level of load and external delay.

A disjoint network is used to collect data from the simulation, averaging the consumption levels over one or two minute intervals in the same fashion as the monitoring agents running on the
application server that generated the test data. After tuning the simulation, the model results are within ~5% of the test system results for both average and peak CPU and heap memory utilization at high loads of 1,000 to 1,500 times the nominal level (x) and external service delays that vary up to double their nominal levels. Additional experimentation validated that the simulation could accurately predict the impact of spikes in load and external service slowdowns. Of particular interest was the peak utilization of the bottleneck resource, CPU. Comparison data for the maximum utilization of this constraining resource was presented in Figure 13 for load and latency levels of interest for sizing the application’s projected growth.

All of the data was collected at the financial services firm and the emulation-simulation framework was built and validated with data from the firm. The framework has since been used for making decision for deploying applications on new stacks successfully. Subsequently, the simulation model ahs also been used for making capacity allocation decisions at the firm. The research was a mutually beneficial effort, in that the firm now has a useable framework for making decisions. The authors were able to understand several aspects of performance modeling of Web-based applications and utilize discrete event simulation to significantly reduce cost of upgrades and maintenance of Web-based applications.

CONCLUSION

The development of a valid model for the analysis of application server resource consumption and performance was achieved using an approach that included emulation testing and experimentation with alternative simulation constructs to arrive at a model that closely mimics real server performance. With this emulation-simulation framework, existing applications can be tested and classified as memory intensive, CPU intensive, or any other important characteristic. This classification of applications aids in automatic updates when new stacks are introduced. It would suffice to test one or a very small number of applications from each set, and fine tune stack parameters, if necessary, and upgrade all applications in the set to the new stack. Prior theoretical work had developed algorithms for analyzing and simulating resource locking but without empirical data it was not clear which resources exhibited locking behavior and what percentage of these resources was locked during external service calls versus what percentage would be recovered more immediately. We learned through this approach that for light-weight applications that return small amounts of data, CPU and a majority of heap memory resources are consumed and recovered very quickly, while a small portion (4%) of heap memory resources exhibit resource locking. The emulation/simulation approach enabled discovery of these patterns and provides a means for ongoing discovery and refinement of consumption patterns for other types of applications. The approach also provides a path for wider application in areas such as capacity planning and performance monitoring.

Our focus for continuing research is to apply this methodology across the organization to classify all the applications and develop a mapping for developing clusters of similar applications, and just as better information enables efficient supply chains to reduce unnecessary inventories; better knowledge of technology behavior will enable more efficient data centers. Based on the team’s success with developing and testing the approach, the organization has decided to formalize the process for use across a broader range of applications and technology architectures. Future work will focus on virtual server environments and further optimization of reference architectures for web-based applications.

REFERENCES


**APPENDIX**

Example analysis for one external service is provided showing that an exponential distribution can adequately represent the response times for the stubbed as well as the simulated external service. The input analyzer in Arena, a commercially available discrete event simulator was used to fit distributions to the data.

Example analysis of service execution time distribution (Validation Service)

<table>
<thead>
<tr>
<th>Data Summary</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Data Points</td>
<td>1382</td>
<td></td>
</tr>
<tr>
<td>Min Data Value</td>
<td>209</td>
<td></td>
</tr>
<tr>
<td>Max Data Value</td>
<td>45700</td>
<td></td>
</tr>
<tr>
<td>Sample Mean</td>
<td>2640</td>
<td></td>
</tr>
<tr>
<td>Sample Std Dev</td>
<td>3220</td>
<td></td>
</tr>
</tbody>
</table>

Distribution fit summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameter</th>
<th>Sq Error</th>
<th>p-value of Chi-square test</th>
<th>p-value of K-S test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponential</td>
<td>2430</td>
<td>0.0385</td>
<td>&lt; 0.005</td>
<td>&lt; 0.01</td>
</tr>
</tbody>
</table>

Similar analyses were performed for each external service and for the user load on the current production system to identify the delay and load patterns and their current nominal levels.
t Tests for Test Response Times vs Simulation Response Times

1500x nominal latency

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>Mean</th>
<th>StDev</th>
<th>SE Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test_Response_Times</td>
<td>9999</td>
<td>19.41</td>
<td>6.74</td>
<td>0.067</td>
</tr>
<tr>
<td>Simulation_Response_Time</td>
<td>12000</td>
<td>19.2</td>
<td>6.48</td>
<td>0.059</td>
</tr>
</tbody>
</table>

Estimate for difference: 0.2108
95% CI for difference: (0.0357, 0.3860)
t-Test of difference = 0 (vs not =):
t-Value = 2.36
p-Value = 0.018
DF = 21997

1000x nominal latency

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>Mean</th>
<th>StDev</th>
<th>SE Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test_Response_Times</td>
<td>10000</td>
<td>17.17</td>
<td>5.84</td>
<td>0.058</td>
</tr>
<tr>
<td>Simulation_Response_Time</td>
<td>10000</td>
<td>17.13</td>
<td>5.9</td>
<td>0.059</td>
</tr>
</tbody>
</table>

Estimate for difference: 0.0431
95% CI for difference: (-0.1195, 0.2058)
t-Test of difference = 0 (vs not =):
t-Value = 0.52
p-Value = 0.603
DF = 19998

* Both tests use Pooled Standard Deviation = 6.5997