

LOAD TESTING VIRTUALIZED SERVERS ISSUES AND GUIDELINES

James F Brady
Capacity Planner for the State Of Nevada
jfbrady@doit.nv.gov

The use of virtualization techniques for server consolidation is focusing attention on load testing as a way to insure that the newly virtualized applications efficiently share hardware resources while achieving individual service level objectives. This paper identifies some issues associated with and provides a list of guidelines for load testing transaction based virtualized servers when the goal is to understand their congestion behavior and quantify their traffic capacity. Many of the ideas presented are also applicable to non-virtualized transaction based server environments.

1. Introduction

Virtualized systems present a set of unique opportunities to those tasked with using load testing as a means of quantifying a server's traffic capacity and identifying its performance characteristics. This paper describes some of these opportunities and contains a set of testing guidelines to follow. The discussion focuses on transaction oriented applications where a large population of users is making requests of the target server using connectionless web based protocols like HTTP. The systematic approach to load testing described in this document emphasizes fundamental traffic generation principles and consistent resource utilization measurement and analysis techniques.

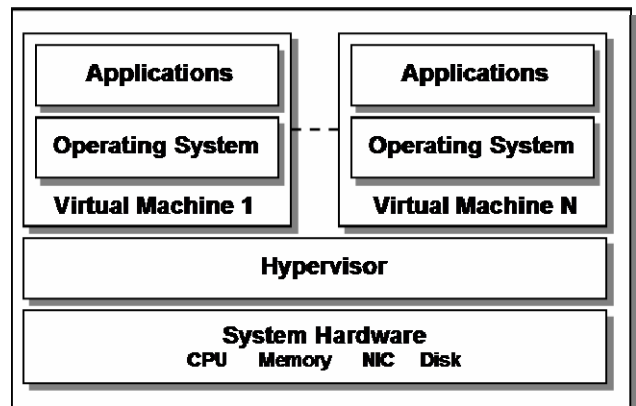
The paper begins with a description of the virtualization environment along with a discussion of its performance characteristics from a capacity determination and resource measurement perspective. This is followed by a load testing section where a traffic generation and resource consumption analysis framework is described which is designed to scale up to the traffic volume needed while maximizing the clarity of the system's congestion behavior characteristics. The virtualization performance characteristics and testing approach discussions lead to an example load test and a set of guidelines for capacity testing transaction based virtual servers. Some summary comments complete the paper.

2. Virtualization Environment

Server virtualization can be viewed from a software

and hardware perspective. Figure 1 illustrates software virtualization as it applies in today's server consolidation environment where multiple operating systems and associated applications share hardware resources which are managed by a hypervisor, sometimes referred to as the Virtual Machine Monitor [Fri07]. The environments being managed by the hypervisor, often identified as "guests", are usually Windows® or Unix/Linux operating systems and related applications.

Figure 1: Virtualized Server Environment



The performance characteristics of this systems structure differs from the standalone server environment in several ways. For example, the physical processors potentially incur three basic types of overhead due to software virtualization.

1. Hypervisor scheduler overhead.
2. Privileged instruction emulation overhead.
3. I/O driver overhead.

The hypervisor controls the scheduling of individual guest CPU resources using a form of scheduling defined by Gunther as proportional polling [Gun06]. This polling mechanism uses share allocation to manage when, and for how long, an individual guest is allowed to run, while the guest controls its process execution priority as if it were a standalone system. The lack of guest execution timing control impacts its timekeeping ability rendering time based resource utilization counters, such as CPU busy, unreliable [Fer05]. The fundamental principle to keep in mind when analyzing guest recorded performance statistics is the guest does not own the hardware or the system clock.

Guest operating system privileged instructions, such as those issued when generating I/O requests, are often emulated by the virtual machine. As Friedman [Fri07] points out for a Windows guest in a VMware ESX[®] virtual environment, the emulation routine can be quite involved and dramatically increase the number of instructions executed to perform the function. Other system impacts highlighted by Friedman are I/O driver overhead incurred when a native device driver maps a physical request into a virtualized context and potential I/O interrupt delays do to guests outnumbering the physical processors available.

Memory beyond that required by the applications is needed to support the virtual environment. Vendor estimates of the quantity needed are generally accurate and yield sufficient resources to efficiently operate the system under load.

The processing congestion behavior of the separate guests is potentially very different in the virtual environment than it is when guests operate as standalone servers. This is especially true if there are significantly more guests than physical processors assigned to them. As Brady [Bra05] observed with multiple Linux guests performing CPU intensive work using one physical processor, the guests were periodically suspended, in a “wait state”, for multiple second intervals. This behavior created response time discontinuities and inconsistencies in Linux time based resource utilization statistics not seen in standalone server environments.

There is an additional CPU resource utilization counter complexity introduced when the physical processors being software virtualized are also hardware virtualized; e.g., Hyper-Threaded. Hyper-Threading, in its simplest form, is the addition of a thread register; i.e., input queue, to an execution unit in an attempt to use idle execution unit cycles [Gun06]. The measurement problem arises because the operating system views this new queue as a second logical processor but it typically only increases throughput by 15-20%. Cockcroft [Coc06] describes the behavior this

way, “If all you are looking at is CPU %busy reported by the OS, you will see a Hyper-Threaded system perform reasonably well up to 50% busy then as a small amount of additional load is added the CPU usage shoots up and the machine saturates.”

An additional resource measurement complexity, mentioned by Cockcroft, is the implementation of processors that manage CPU clock speed based on utilization in an attempt to reduce their overall power consumption. Obviously, CPU utilization statistics collected in this environment are very suspect.

The impact of these virtualization characteristics varies from one implementation to the next. For example, Friedman [Fri06] indicates the open source Xen project hypervisor virtualizes its devices, allowing native device drivers installed on the guest OS to communicate directly to attached peripherals. It does this at the expense of being non-transparent to the guest OS.

Table 1 summarizes the virtualization issues just discussed by listing each specific characteristic and identifying its primary performance impact.

Table 1: Virtualization Characteristic Performance Impact Summary

Software Virtualization	
Characteristic	Primary Impact
Hypervisor scheduler O/H	workload
Hypervisor scheduler polling	guest time view
Privileged instruct emulation	workload
I/O driver Overhead	workload
I/O interrupt delays	latency
Virtual environment memory	memory size
Guest “wait state”	Latency/ guest time view
Hardware Virtualization	
Characteristic	Primary Impact
Hyper-Threading	guest/hypervisor CPU counters
Variable Clock Speed CPUs	guest/hypervisor time counters

3. Load Testing Framework

Given all the issues listed in Table 1, what is the best load testing technique for transaction oriented applications being moved from the standalone server world to a common virtualized platform? It is this author’s belief that the best approach, in the face of this much uncertainty, is to rely on fundamental principles regarding traffic generation techniques, capacity quantification approaches, and service level identification procedures.

Traffic generation is usually accomplished with a computer program running on one or more simulation computers that launch requests, measure response times, and determine if those responses are correct. There are a number of commercial and open source load testing tools available. Podelko [Pod05] provides

a list of those that are the most popular. When applied to transaction oriented applications such as web requests, these tools are generally structured to record the interactions between a user and the target environment for playback later. The interactions recorded are placed within the context of a script in a manner that simulates an end user sequence of transaction events. These “virtual user” scripts are set up to contain fixed or uniformly distributed think times between transactions. An example virtual user traffic mix containing think time values is shown in Table 2. This virtual user session mix scenario is Web based with transactions consisting of a login followed by queries and/or updates and a logout. A think time delay is imposed between each step in the five virtual user scripts shown.

Table 2: Virtual User Traffic Mix

Transaction	Delay Seconds	Vuser	Vuser	Vuser	Vuser	Vuser	Total
		Type 1	Type 2	Type 3	Type 4	Type 5	
		10%	15%	40%	10%	25%	100%
LOGIN	10	1	1	1	1	1	100%
Query1	15	1		1		1	75%
Query2	10		1		1	1	50%
Update1	20	1					10%
Update2	15		1				15%
LOGOUT	10	1	1	1	1	1	100%
Script Time		55	45	35	30	45	40.5

There are a lot of things to consider when setting up and executing a load test. Podelko [Pod06] provides a concise process flow diagram which walks through requirements collection, load definition, creation of test assets, running the tests, and analyzing results. He also identifies a feedback mechanism between the test run and analysis steps.

Three methodological steps come to mind when applying this process flow to the virtualized transaction processing environment for the most meaningful testing results to be obtained.

1. Insure a real world traffic pattern is produced.
2. Structure each test as a series of incrementally increasing traffic volumes.
3. Maintain a consistent traffic mix for all traffic volumes.

This list is useful when load testing any target system. It is especially important for the virtual environment where guest operating system time based counters are unreliable and a systematic approach to traffic generation is needed.

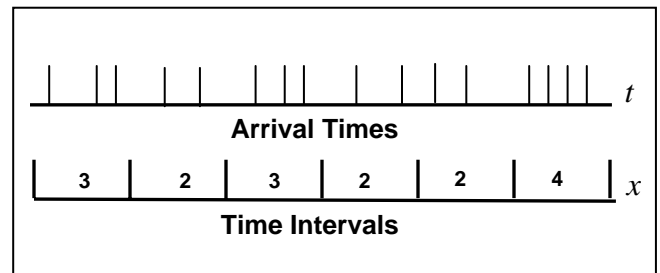
The most common real world traffic pattern for transaction processing is request independence or random arrivals. This model assumes that the population of traffic sources is large and there is no coercion or interaction between these sources when they press the enter key, making each of their actions independent events. This is a powerful observation and is a fundamental assumption of most queuing models.

This random arrivals environment is referred to in the literature as a Poisson process and is quantified by two formulas from probability theory. The times between arrivals (t) are negative exponentially distributed and the number of arrivals in non-overlapping constant length intervals is Poisson distributed.

Figure 2, is a pictorial representation of a random arrivals pattern where the arrival times, t , are represented by the non-equally spaced vertical bars and the frequency counts, x , indicate the arrivals within the equally spaced intervals shown. Mathematically, $f(t)$ is described by the negative exponential probability density function, Equation 3.1, which has a mean inter-arrival time of $\frac{1}{\lambda}$. The number

of arrivals (x) in constant length intervals is specified by the Poisson probability distribution function, shown as Equation 3.2, which has a mean number of arrivals per interval of μ .

Figure 2: Random Arrivals



$$f(t) = \lambda e^{-\lambda t} \quad (3.1)$$

$$p(x) = \frac{e^{-\mu} \mu^x}{x!} \quad (3.2)$$

Where:

$$\frac{1}{\lambda} = \text{mean inter-arrival time } t .$$

$$\text{for } f(t), \quad \frac{1}{\lambda} = \text{mean} = \text{std dev} = \sqrt{\text{variance}} .$$

$$\mu = \text{mean number of arrivals per time period } x$$

$$\text{for } p(x), \quad \mu = \text{mean} = \text{variance} .$$

Note that the mean of the negative exponential distribution is equal to its standard deviation and the mean of the Poisson distribution is equal to its variance. The fact that the mean inter-arrival time equals its standard deviation is often used to determine if random arrivals traffic is present.

The technique used to artificially produce negative exponentially distributed inter-arrival times is the Cumulative Distribution Function (CDF) reverse

transformation method implemented in many traffic generation and simulation software packages summarized as equation 3.3.

$$t_0 = -\frac{1}{\lambda} \ln(r_0) \quad (3.3)$$

Where:

t_0 = time until the next arrival

$\frac{1}{\lambda}$ = mean inter-arrival time

\ln = natural log ($\ln e = 1$)

r_0 = random number $0 \leq r_0 \leq 1$

For a derivation of Equation 3.3 from Equation 3.1 see [Bra04] or [Bra06a].

This author has observed random arrivals behavior first hand as a software developer and performance analyst for a prepaid calling card database system. The system contained transaction request time recording capability as part of its response time overload control mechanism. When this recording capability was exercised, the mean and standard deviation of the inter-arrival times associated with call setup and tear-down events were almost identical for hourly intervals analyzed.

Structuring each test as a set of incrementally increasing traffic volume steps provides a way to methodically quantify resource consumption as a function of traffic level and produce a profile of resource congestion behavior. Note the functional relationship is between resource consumption and traffic volume, not number of users. As Podelko [Pod06] points out, "Throughput defines load on the system. Unfortunately, quite often the number of users (concurrency) is used to define the load for interactive systems instead of throughput. Partially because that number is often easier to find, partially because it is the way load testing tools work. Without defining what each user is doing and how intensely (i.e., throughput for one user), the number of users is not a good measure of load."

This author believes that another possible reason why most load testing tools "work the way they work" is because they are designed to accommodate connection based protocols that require complex context to be saved between transaction sequences within a user session. This functionality is not needed for connectionless protocols like HTTP and can be cumbersome to work within.

The profile of resource congestion behavior is meaningful only if a consistent traffic mix is maintained throughout the set of test increments performed. Any significant change in traffic mix from increment to

increment destroys the functional relationship between resource consumption level and traffic volume. Maintaining a consistent traffic mix may be a challenge using a virtual user script tool because script startups must be managed across all traffic increments.

Consider a transaction based traffic generator whose traffic mix is set up as in Table 3. The traffic generator implemented in this illustration contains no virtual user scripts and has the advantage that the request statistics produced are in the units of work presented to the target system; i.e., transactions/sec. This transaction model can be used for traffic generation when connectionless protocols like web HTTP are being exercised because no context saving is needed between the transactions of a user session.

Table 3: Transaction Based Traffic Mix

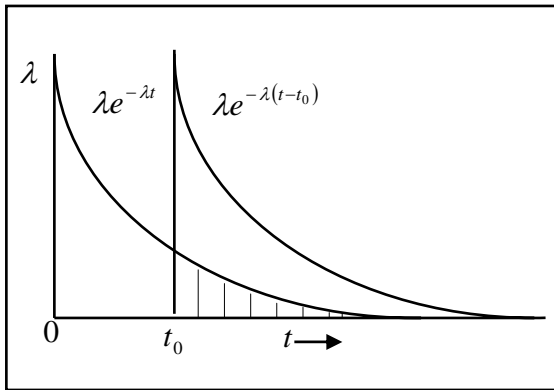
Transaction	Percent
LOGIN	Once Per Source
Query1	50%
Query2	33%
Update1	7%
Update2	10%
LOGOUT	0%
Total	100%

This traffic generator's software is designed such that each process or thread makes the same type of transaction request each time with a negative exponentially distributed (memoryless) delay between requests. If a login is required, each requesting process performs its assigned query or update repeatedly after a single successful login has been accomplished. There is little need to worry about startup and shutdown time transients, from a traffic generator perspective, because the first thing each source does is delay in a memoryless manner.

The memoryless, or time invariant, property of the negative exponential is illustrated in Figure 3 which shows the negative exponential density $\lambda e^{-\lambda t}$ at time 0. Given that t_0 seconds have elapsed, the density function for the time until the next arrival is computed by magnifying the portion of the curve to the right of t_0 (shaded) and increasing its area to unity yielding $\lambda e^{-\lambda(t-t_0)}$. The shape of this new density function is identical to the original and is only shifted in time.

As Kleinrock [Kle75] indicates, "No other density function has the property that its tail everywhere possesses the exact same shape as the entire density function." Gunther [Gun05] describes the memoryless property of the negative exponential with a numerical illustration and a counter example using the Normal probability distribution.

Figure 3: Memoryless Property of the Negative Exponential Distribution

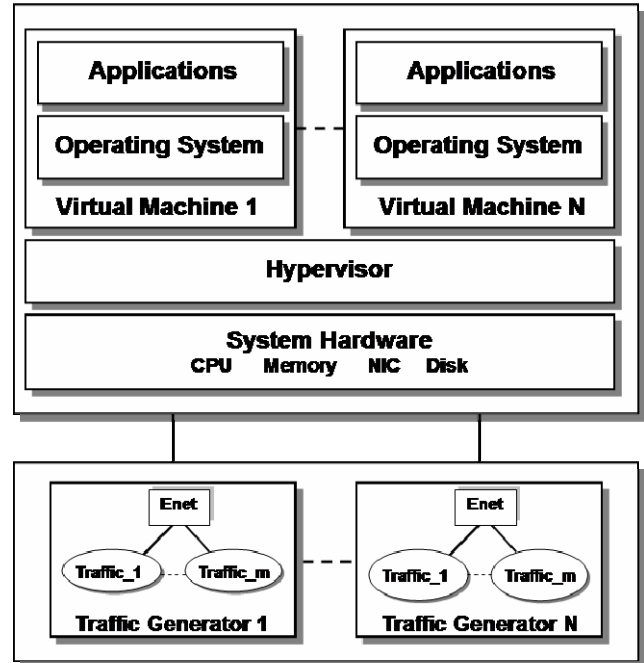


Whether a traffic generator is virtual script or transaction based, statistical feedback regarding the quality of its request launch pattern is important. The test results obtained from a traffic generator that does not provide inter-arrival statistics or whose inter-arrival mean and standard deviation are available but not close to each other should be scrutinized very carefully.

The virtual user script traffic generator represented by Table 2 is unlikely to produce a random arrivals traffic pattern when the think time delay seconds are fixed values, because the scripts will potentially synchronize during the traffic run. When “random” think times are selected within the virtual user script environment, a minimum and maximum delay interval is usually specified. This specification is inconsistent with the negative exponential distribution which contains one parameter, the mean inter-arrival time.

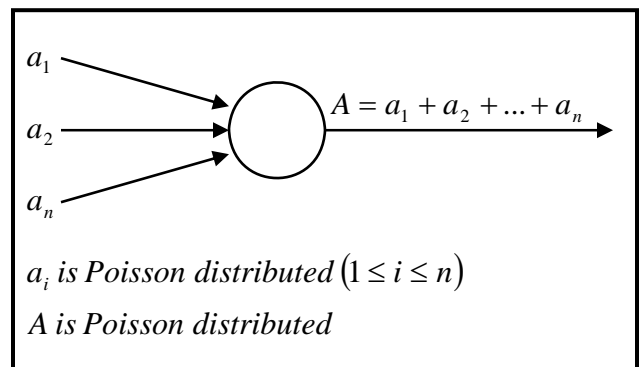
The topology in Figure 4 illustrates traffic generation in the virtual environment using the transaction based traffic generator exemplified by Table 3. The flow of transactions in this situation occurs in the following way. The target system receives a random arrivals transaction request initiated by one of the “N” traffic generator’s “m” traffic processes. This request traverses the communications interface (typically Ethernet) to the virtualized System Hardware. The Hypervisor schedules the guest to run using its proportional polling mechanism. When the application software within the guest executes, a response is generated which traverses through the virtual environment to the requesting traffic generator process. The traffic process analyzes the response to determine if it is correct and records the transaction response time.

Figure 4: Traffic Generation Topology



From an arrival pattern perspective, it does not matter if the “N x m” traffic sources are producing traffic from one generator or the “N” generators represented in Figure 4. As Gunther [Gun05] points out, “the merging of two Poisson streams produces a single Poisson stream with intensity equal to the sum of their intensities.” Giffin [Gif78] showed this to be true for an arbitrary number of Poisson streams by performing their convolution using transform techniques. This stream merging property is illustrated in Figure 5 where each a_i is a Poisson distributed random variable whose sum, A , the total arrivals per unit time, is also Poisson distributed with a mean equal to the sum of the a_i means.

Figure 5: Combining Poisson Streams

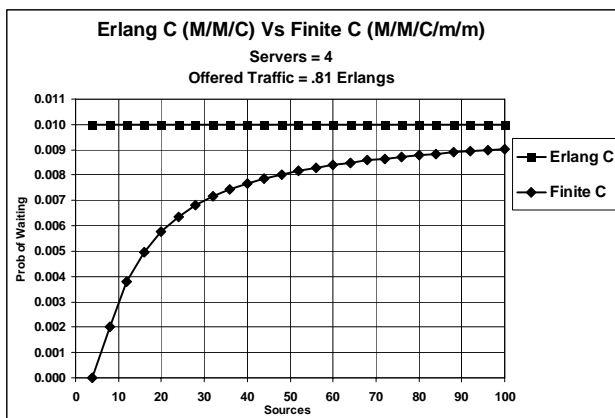


The arrival pattern may not depend on how many Poisson streams exist but it does depend on the size of “N x m.” This number must be large enough to be assumed infinite if the Poisson assumption is to

hold true. How many sources are needed for a closed finite source queuing system (M/M/C/m/m) like the traffic generator in Figure 4 to operate as if it were an open, infinite source, system (M/M/C)?

Figure 6 plots service level (probability of waiting) as a function of traffic sources for the two models with offered traffic and server quantity held constant. In the infinite source situation, the probability of waiting is constant for all sources but in the finite source case service levels asymptotically approach this constant as the number of sources increase to around 100. Note there is no waiting in the finite source situation if the number of sources is less than or equal to the number of servers, i.e., 4.

Figure 6: Finite Source Convergence to Infinite Source Model



4. Example Load Test

The focus of the traffic generation side of this load testing effort is to produce a representative and consistent traffic mix delivered with a real world timing pattern. In the simple transaction web environment being discussed, the target system does not perceive individual users performing a particular sequence of transactions but transactions of a specific mix arriving in a random arrivals fashion.

The following example is intended to illustrate this concept while highlighting where virtualization needs special consideration. Figure 7 is an example screen from a run of the transaction based traffic generator. This finite source traffic generator, developed by this author, is a Perl script that implements the Perl LWP library (Library for WWW in Perl) in a multi-process mode and uses a free tool to record transaction sequence information for playback. This particular example depicts a thirty minute run where good, bad, and late response statistics are reported every 100 seconds for both GET and POST web queries. There are 200 GET and 200 POST traffic sources producing requests randomly at an aggregate rate of a little over

87 transactions per second. Some late events are being recorded at this transaction rate for the three second threshold specified.

Figure 7: Traffic Generator Run

```
Linux Fri May 11 00:47:02 2007, 200 GET and 200 POST source(s), 3000 ms late
-----GET-----
Time      sent  good  bad  late
0:47:02   9809  4901  0    93
0:48:42   8934  4535  0    70
0:50:22   8772  4281  0    62
0:52:02   8763  4338  0    74
0:53:42   8935  4446  0    72
0:55:22   8708  4359  0    66
0:57:02   8778  4315  0    55
0:58:42   8733  4360  0    60
1:00:22   8724  4350  0    68
1:02:02   8842  4442  0    67
1:03:42   8785  4256  0    73
1:05:22   8973  4528  0    69
1:07:02   8840  4438  0    61
1:08:42   8921  4478  0    74
1:10:22   8893  4475  0    62
1:12:02   8892  4472  0    62
1:13:42   8892  4442  0    75
Total     151194 75416 0    1163
-----POST-----
Time      sent  good  bad  late
0:47:02   9809  4908  0    72
0:48:42   8934  4399  0    73
0:50:22   8772  4491  0    76
0:52:02   8763  4425  0    70
0:53:42   8935  4489  0    71
0:55:22   8708  4349  0    71
0:57:02   8778  4463  0    66
0:58:42   8733  4373  0    72
1:00:22   8724  4374  0    72
1:02:02   8842  4400  0    71
1:03:42   8785  4529  0    74
1:05:22   8973  4445  0    89
1:07:02   8840  4402  0    83
1:08:42   8921  4443  0    78
1:10:22   8893  4418  0    69
1:12:02   8892  4420  0    71
1:13:42   8892  4450  0    76
Total     151194 75778 0    1254

Caught a SIGALRM signal -- shutting down
Fri May 11 01:15:37 2007
```

The load test numerical results report in Figure 8 shows arrival, response time, CPU, Disk I/O, and Packet rate statistics for the nine traffic runs listed. Resource consumption levels are sampled every fifteen seconds on the target system during each of the half-hour runs performed. Traffic run number 9 of Figure 8 maps to the Figure 7 run and shows an inter-arrival time mean and standard deviation of 11 and 12 milliseconds, respectively, indicating that a random arrivals, or memoryless, request pattern is being offered to the target system.

Figure 8: Load Test Results – 400 Sources

Arrival, Response Time, CPU, Disk I/O and Packets - 400 Sources									
run	tps	Inter-arrival times(ms)		Response times(ms)		CPU %	Disk Packets		rs/s
		mean	sdev	mean	p95		rw/s	rs/s	
1	0.00	0	0	88	257	0	0	0	0
2	8.56	116	117	88	257	8	43	416	
3	26.36	37	38	117	343	24	104	1318	
4	39.16	25	26	154	454	37	140	1894	
5	59.26	16	17	202	603	55	192	2799	
6	67.55	14	15	241	722	62	218	3148	
7	74.53	13	14	289	863	68	249	3510	
8	82.86	11	12	470	1411	76	276	3926	
9	87.41	11	12	725	2163	80	292	4108	

Support for the assertion that merged Poisson streams yield a Poisson stream; i.e., Figure 5, is provided in the inter-arrival time statistics of Figure 9. This figure contains these statistics for 100 traffic sources selected from the 400 traffic source environment of Figure 8. The inter-arrival time mean and standard deviation for this one-fourth slice of each run are statistically equal, indicating negative exponential inter-arrival times. So the 400 traffic source environment can be viewed as a Poisson stream composed of four 100 traffic source Poisson streams. Carrying the argument one step

farther, a number of traffic generators, each with 400 traffic sources, could be merged to produce a single Poisson stream.

Figure 9: Inter-arrival Times - 100 Sources

Inter-arrival times(ms)			
run	tps	mean	sdev
1	0	0	0
2	2.19	457	454
3	6.66	149	148
4	9.87	101	101
5	14.98	66	66
6	17.09	58	59
7	18.72	53	53
8	20.91	47	48
9	22.12	45	45

Often the best way to show a system's capacity characteristics is to draw a picture of resource consumption levels and response time service levels as a function of incremental increases in traffic volume. The X-Y Plots in Figure 10 are an example of such a graphical illustration when the transaction mix is held constant and traffic is added incrementally. Under these circumstances, certain resource consumption statistics tend to increase proportional to increases in traffic volume unless there is a system imbalance or bottleneck. The basic idea is that if the workload is consistent, and doubles, the resource utilization level also doubles. The three most important resources to consider with this traffic congestion characteristic are CPUs, Disks, and Enet communications devices.

The CPU % Utilization, Disk I/O rate, and Enet Packet rate graphs in Figure 10 exemplify throughput proportionality since consumption levels for all of them are a linear function of traffic rate during the runs performed. It appears from this graph that the system is approaching a CPU limitation at around 80% Utilization because both the mean and p95 response times are increasing sharply. Disk and Enet are unlikely to be causing this service level degradation since their traffic rates are both below normal saturation levels and their X-Y Plots are linear to the end.

The Figure 11 pie chart and associated table expand upon the Figure 10 CPU Utilization Graph by indicating which processes use CPU time and to what degree. This graph is created by proportioning the CPU time recorded during the last four traffic runs. It is particularly helpful to software developers for focusing their tuning efforts on processes which consume the most CPU time. There is little payoff improving the performance of a process which uses 1% of the CPU by 50% but such a performance improvement on a process that uses 37% is significant.

Figure 10: Traffic Capacity – X-Y Plots

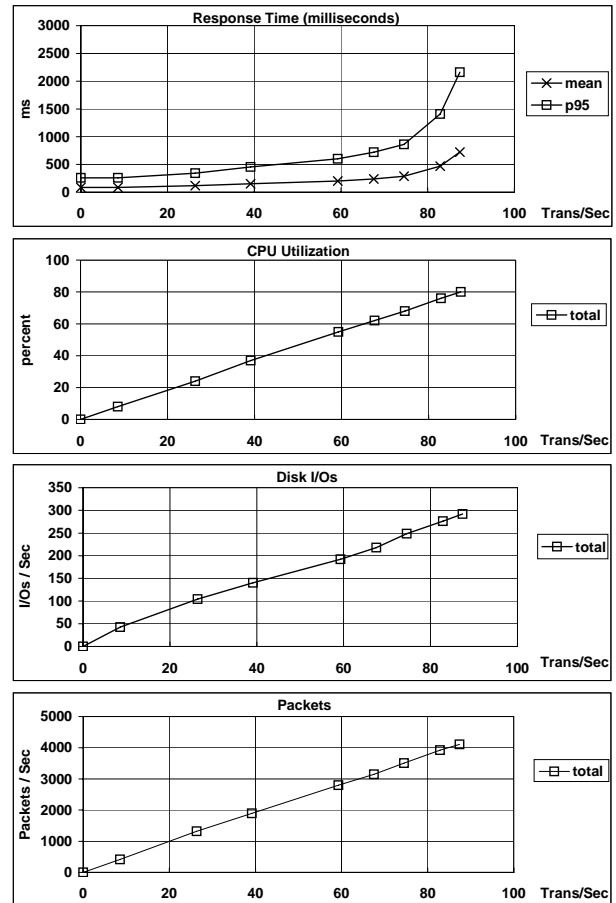
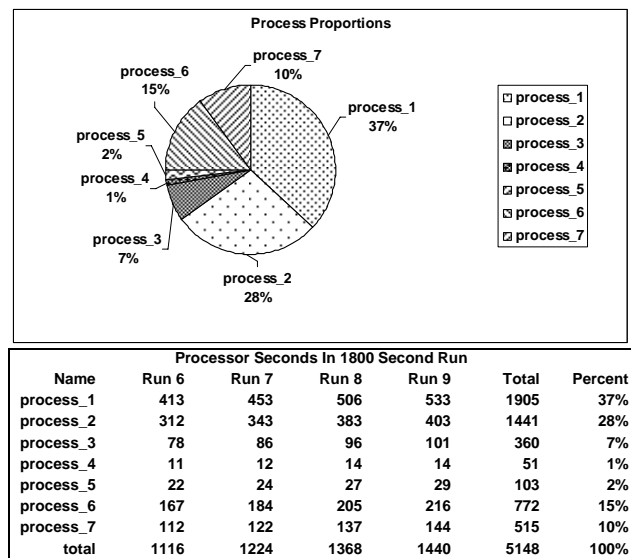


Figure 11: Process Proportions - Pie



From a resource utilization recording perspective, the most basic tool available for a Windows guest is its perfmon utility where data can be saved in either binary or csv format. Friedman [Fri02] recommends recording the data in binary and converting it to csv

with a tool like relog. This approach minimizes the potentially substantial CPU overhead generated by the process producing the performance data. The simplest Unix/Linux guest resource utilization recording technique is to run the set of stat utilities available; i.e., vmstat, netstat, iostat, mpstat, sar, and ps. This author has constructed a set of Perl scripts that collect and organize the data produced by these utilities for easy construction of Figure 10 and 11.

5. Virtualization Issues

Figure 10 represents a profile of a server identifying the congestion behavior of its key system resources as traffic is increased in a systematic way. How does this traffic generation and profile analysis change under virtualization?

Unlike the standalone server, there are multiple operating system environments to measure and analyze including the hypervisor and the individual guests. The hypervisor is an important place to gather statistics because it provides an overall view of the system's resources as owner of both the system clock and the hardware. Measurements taken on individual guests yield a process level perspective and give an indication of how well resources are balanced. Therefore, Figure 10 and Figure 11 graphs are needed for the hypervisor and each guest to generate a complete system profile.

The performance properties of those virtualization characteristics listed in Table 1 whose primary impacts are workload or latency, such as hypervisor scheduler overhead or I/O interrupt delay, will be reflected in the resource consumption and response time statistics as usual. Those characteristics of virtualization that affect the guest's view of time or impact time based counters, like hypervisor scheduler polling or Hyper-Threading, create complexities regarding how data is interpreted and placed into context.

From a Figure 10 perspective, the transaction mix is the same in the virtual and non-virtual worlds with response time curves providing an indication of system congestion. Also, the linear relationship between packet rates and transaction rates should hold true. Packet rates reflect real activity in and out of the system and are a key indicator of work accomplished; i.e., throughput. Therefore, the key congestion level indicator, response time, should be relied upon and correlated with what appears to be the most reliable throughput measure in the virtual environment, packets/sec.

Disk I/O rates are also a potential indicator of throughput but are less reliable than packet rates because cache hits complicate their meaning. If the load generator randomizes over a large portion of the

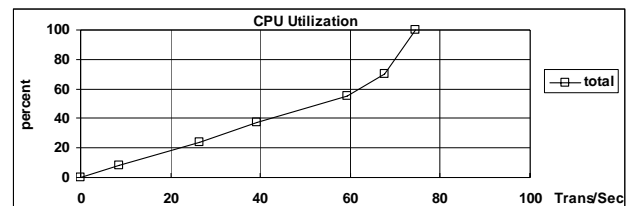
database, the Disk I/O rate should be a linear function of transaction rate with the slope of the line dependent on the cache hit ratio.

There is no X-Y plot in Figure 10 for memory because memory allocation isn't as much a function of transaction rate as it is number of running processes and associated active threads. There is additional memory needed to support the virtual environment and memory limitation can best be monitored by looking at the hypervisor's statistics for each traffic run.

As mentioned, time based counters running in a virtualized guest, like percent CPU utilization, are suspect and cannot be depended upon. Therefore, the linear CPU busy X-Y plot in Figure 10 may not exist for a specific guest. This lack of dependability also extends to the process level CPU utilization statistics shown in the Pie Chart of Figure 11. Since the hypervisor does not have visibility at the guest process level, guest Figure 11 statistics are suspect. Even though these process level statistics are of questionable integrity, the proportion of CPU used by the key processes may be reasonably representative; i.e., 37% CPU use by process_1 is likely to be more accurate and useful than process_1 consuming 533 sec of CPU time during run 9.

If the physical processors are Hyper-Threaded or they manage CPU clock speed based on utilization, it is likely that the Figure 10 straight line function will not exist for either the hypervisor or the guests. The relationship that exists, however, may be intuitive and instructive. A CPU utilization X-Y plot of a Hyper-Threaded hypervisor is likely to appear as shown in Figure 12, indicating that once the total processor utilization exceeds 50% saturation occurs rapidly.

Figure 12: Hyper-Threaded Hypervisor



It is also important to monitor the virtual environment during traffic runs for possible guest operating system wait states. These events appear in the response times recorded but wait state response times are difficult to differentiate from congestion based response times. It is best to identify these events by reviewing the hypervisor statistics available or running a wait state event identification script like the one Brady used in [Bra05].

This script saves an initial time stamp and calls a sleep function requesting "n" seconds of sleep time. When the script awakens, it saves an ending time stamp,

computes the difference between the two time stamps, and compares that value with the sleep interval requested. If the difference is greater than the tolerance value of “m” seconds, a log record is generated containing the requested and experienced sleep time. The concept behind this script is to provide a check of the hypervisor to see if it honors the sleep call wakeup request within the allotted time. An example warning message extracted from [Bra05] is shown in Figure 13.

Figure 13: Wait State Warning

```
Warning: Sleep Time is 98 sec but should be 60 sec – 17:03:08  
Warning: Sleep Time is 115 sec but should be 60 sec – 19:19:05
```

The most meaningful load testing information obtainable in a virtualized environment is the relationship between service level and work accomplished; i.e., response time and transaction rate. The second most useful information is to determine if the flow of traffic coming in and out of the system is linearly related to the work accomplished; i.e., is packet rate linearly related to transaction rate. Packet rates also provide a test setup cross-check since generator rates mirror server rates if the setup is correct.

6. Load Testing Guidelines

The above observations, in conjunction with the comments made in the earlier sections, lead to the following list of virtual system load testing guidelines.

1. Present traffic to the target system in a real world fashion (random arrivals) and be sure there is a random arrivals traffic pattern cross-check in place.
2. Ramp up traffic in increments with measurements being performed during the steady-state portion of each time period.
3. Maintain a consistent traffic mix across all traffic rate increments. The focus is bottleneck identification and inconsistent behavior isolation.
4. Randomize requests across database rows to defeat the caches and reasonably represent real world disk activity.
5. Collect resource consumption data from both the hypervisor and individual guest perspective. Collecting data on the guest is important because the resource consumption characteristics of its application processes and threads are not available at the hypervisor level.
6. Run a wait state script on each guest to determine if the guest went into a prolonged wait state, rendering its counters for that time period suspect.
7. Create X-Y Plots of resource utilization Vs Transaction rate as shown in Figure 10.

8. Construct process level pie charts, as illustrated in Figure 11, at the hypervisor and guest level. The guest process level CPU times are suspect in the virtual environment but the proportion of CPU used by the high running processes, reflected in the pie chart, is a useful measure of relative CPU resource consumption.
9. Use the fewest possible transaction types that provide a representative mix. This relates to load test manageability, especially in a virtualized world, where multiple application environments need to be tested at the same time.
10. Perform initial load tests connected directly to the server complex; e.g., Figure 4. This allows a balanced and tuned target environment to be established before network connection complexities are introduced.
11. Real-time monitor each traffic run, as in Figure 7, so that target system slow downs and failures perceived by the load generators can be easily correlated in time with resource consumption indicators.
12. Record resource consumption levels on the traffic generators. These statistics can be used to insure the generators are not overloaded causing a distorted transaction launch pattern and producing an inaccurate set of response time statistics.
13. Cross-check the packet rates recorded by the traffic generators with those measured on the target system. The two rates should mirror each other if the load generating environment is setup correctly.

7. Summary

Virtualized systems have some unique characteristics that impact service levels, throughput, and system resource consumption counter behavior. These properties require a systematic approach to load testing where a conceptually correct traffic pattern is offered to the target system using a consistent transaction mix with traffic applied incrementally.

The goal of the traffic generation technique presented is to mimic the traffic offered in the server's production environment from a request timing, scope, and volume perspective. The objective of the target system measurement and analysis procedure is to apply a consistent traffic mix to the server in a way that makes the functional relationship between resource consumption and traffic volume explicit. This functional relationship provides valuable insight into the congestion characteristics of the system and the behavior of both hypervisor and guest resource utilization counters.

8. References

[Bra04] J. Brady, "Traffic Generation Concepts – Random Arrivals," www.perfdynamics.com, Classes, Supplements, 2004.

[Bra05] J. Brady, "Virtualization and CPU wait times in a Linux guest environment," *CMG Journal*, 116:3-8, Fall 2005.

[Bra06a] J. Brady, "Traffic generation and Unix/Linux system traffic capacity analysis," *CMG Journal*, 117:12-20, spring 2006.

[Bra06b] J. Brady, "Traffic capacity testing a web environment with transaction based tools," *CMG Proceedings 2006*.

[Coc06] A. Cockcroft, "Utilization is Virtually useless as a metric!" *CMG Proceedings 2006*.

[Fer05] G. Fernando, "To V or not to V: A practical guide to virtualization," *CMG Proceedings 2005*.

[Fri02] M. Friedman and O. Pentakalos, *Windows 2000 Performance Guide*, O'Reilly & Associates, Sebastopol, CA, 2002. p. 86-97.

[Fri06] M. Friedman, "The reality of virtualization for Windows Servers," *CMG Proceedings 2006*.

[Fri07] M. Friedman and S. Marksamer, "A Realistic Assessment of the Performance of Windows Guest Virtual Machines," *CMG MeasureIT*, March 2007.

[Gif78] W.C. Giffin, *Queueing: Basic Theory and Applications*, Grid, Inc, Columbus, Ohio, 1978, p. 226-227.

[Gun05] N. Gunther, *Analyzing Computer System Performance with Perl::PDQ*, Springer-Verlag, Berlin Heidelberg, 2005, p. 122-123 and p. 391-395.

[Gun06] N. Gunther, "The virtualization spectrum from hyperthreads to grids," *CMG Proceedings 2006*.

[Gun07] N. Gunther, *Guerrilla Capacity Planning*, Springer-Verlag, Berlin Heidelberg, 2007.

[Kle75] L. Kleinrock, *Queueing Systems Volume 1*, John Wiley & Sons, New York, N.Y., 1975, p. 66-67.

[Mar06] S. Marksamer and P. Weilnau, "Real world adventures in server virtualization," *CMG Proceedings 2006*.

[Pod05] A. Podelko, "Workload generation: does one approach fit all?" *CMG Proceedings 2005*.

[Pod06] A. Podelko, "Load testing: points to ponder," *CMG Proceedings 2006*.

9. Trademarks

Windows is a registered trademark of Microsoft Corporation in the U.S. and certain other countries.

VMware ESX is a registered trademark of EMC Corporation in the U.S. and certain other countries.