How to Estimate Web Application Concurrent Events Using Either Load Generator Data or Queueing Model Results

James Brady

September 1, 2023

How to Estimate Web Application Concurrent Events Using Either Load Generator Data or Queueing Model Results

James F Brady
Computing System Capacity and Performance Specialist
JmsFBrdy@gmail.com

Some of the most sought-after web application load testing results are the concurrent GET/POST event quantities used to dimension internal system resources such as process thread pools and input queue buffers. The concurrency information desired is the proportion of time there are 0, 1, 2, etc. web events resident in the system under test. When these proportions are not available from target system measurements, they can be approximated with data produced by the load generator and estimated using queueing models. The data analysis and queueing model tools that support the ideas presented were developed by this author and contained in two of his GitHub repositories, web-generator-toolkit2 and QueState, as free open-source downloads.

1.0 Introduction

One of the many questions load testing professionals are asked is, "how many concurrent users will the new web application support?" This question normally refers to the number of user requests that can simultaneously be resident in the System Under Test (SUT) while maintaining acceptable service levels. Often, there are no direct measurements being performed in the SUT to draw a picture of concurrency, but load tool query statistics as well as queueing model state probabilities can be used to estimate concurrency levels. The load tool approach taken here combines query timestamps and round-trip response times to produce user request concurrency statistics. The queueing model state probabilities are typically useful as a concurrency estimator for applications that are in the early stages of design or development before measurements are possible.

The web-generator-toolkit2 (Toolkit2) data analysis software on GitHub is this author's approach to concurrency distribution approximation using load tool data. It provides the statistical count and proportion of time there are "N" user requests in the system being tested and it does this for specific query types as well as the total number of requests outstanding. The example used in this document to illustrate the ideas presented is the one contained in the Toolkit2 demo.

The queueing model state probabilities are produced with the QueState package on GitHub. This repository contains eight queueing models and is somewhat unique in that it lists state probabilities along with the usual performance statistics like average waiting time in system, i.e., mean response time. The state probabilities produced by these computer programs are a modeling estimator of the request concurrency distribution.

The discussion begins in Section 2 with a pictorial representation of concurrent events within the context of a load testing environment using the Toolkit2 demo as a basis. Section 3 expands upon this demo example with additional load test specifics used to step through the concurrency calculations performed by the analysis program. This section also summarizes the concurrency related results that program produces. This summary information leads to the Section 4 queueing model discussion where Toolkit2 demo traffic parameters are applied to two candidate models with a graphical comparison made between the model results and demo statistics. Section 5 wraps up the discussion with some conclusions and summary remarks.

2.0 Load Generator View of Concurrent Events

Figure 1 shows a load testing environment containing a traffic generator on the left and the System Under Test on the right with the two of them communicating over Ethernet. The load generator is configured with 200 virtual user threads operating in closed loops offering web traffic to the SUT containing 8 CPU cores. There are nine GET/POST web events initiated by the traffic generator currently being processed in the SUT.

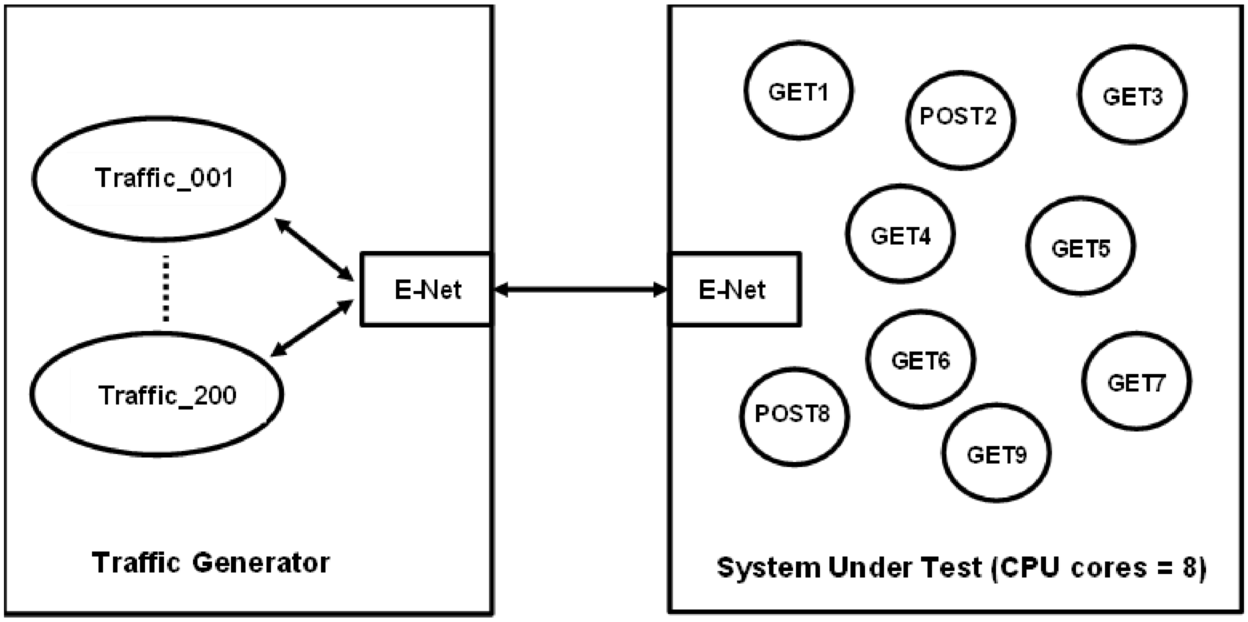


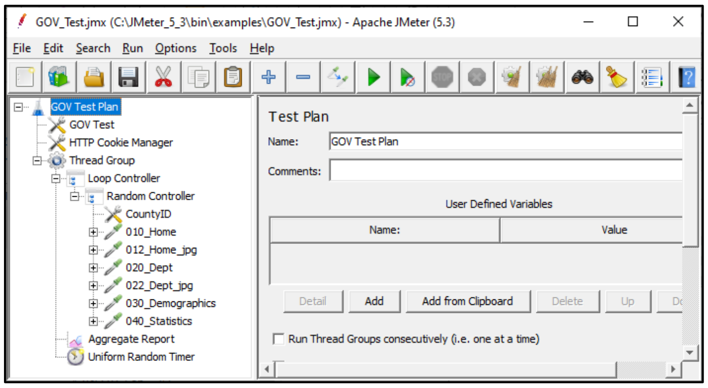
Figure 1: Load Generator and System Under Test (SUT) Topology

The concurrency information of interest is a list containing the fraction of the time the SUT is processing 0, 1, 2, ... 9, …, etc. web events. How can query/response data collected by the load generator be used to estimate the SUT's concurrency distribution? As a means of describing the steps needed to produce concurrency statistics from this data, the topological setup of Figure 1 is expanded into the load testing specifics of Figure 2.

3.0 Concurrency Calculation Setup

Figure 2 contains the Toolkit2 demo list of web events being tested on the left and the layout of the JMeter load testing script on the right. Six events are being queried to obtain state government statistics with a single group containing 200 virtual user threads. Threads randomly select an event from the list each cycle (think + response) and draw their think times from a uniform random timer. This is the same example load testing setup as used in [4.]

GOV Web Site – Web Page GET / POST Events	
Web Page Name	**Purpose**
010_Home	Home Page
012_Home_jpg	Background Image
020_Dept	Department Information
022_Dept_jpg	Department Image
030_Demographics	Demographic Information
040_Statistics	Summary Statistics


GOV_Test.jmx (C:\JMeter_5_3\bin\examples\GOV_Test.jmx) - Apache JMeter (5.3)

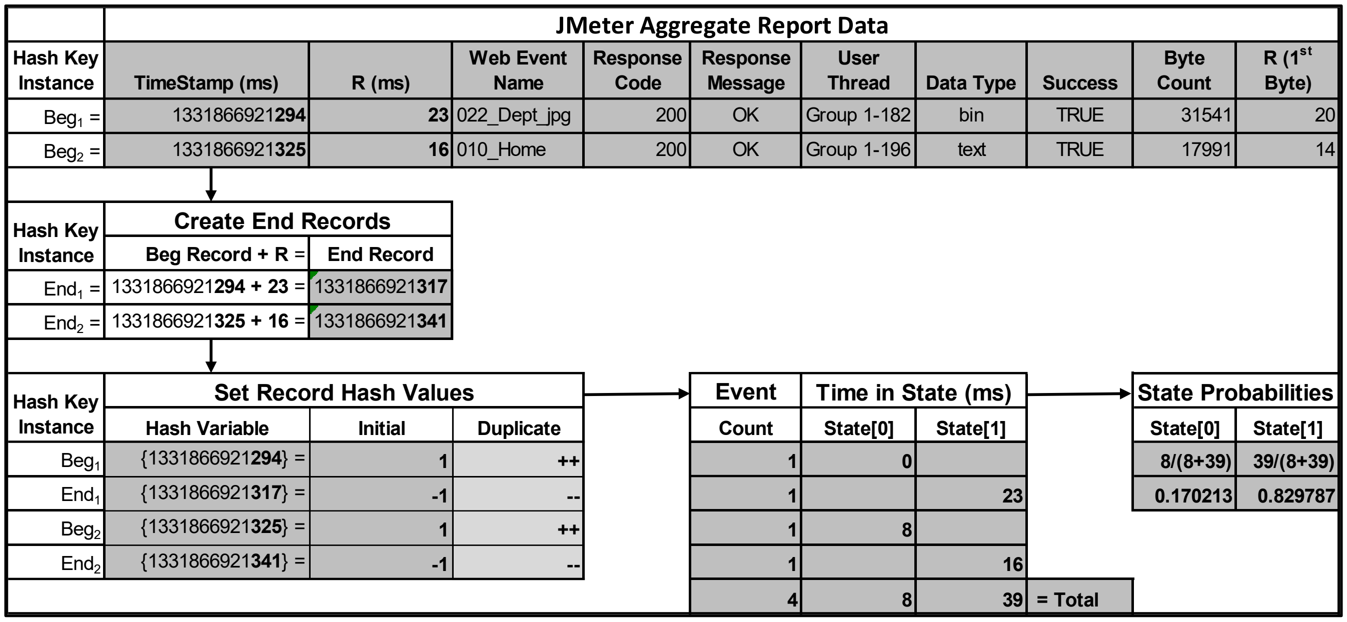
Name: GOV Test Plan
Comments:
[ ] Run Thread Groups consecutively (i.e. one at a time)

Figure 2: Web Page Events and JMeter Load Testing Script

The test run is 25-minutes in duration with the first 2 and last 3 minutes excluded from the analysis to minimize the impact of startup and shutdown time transients. The resulting JMeter Aggregate Report output file, 2000_AggRpt_120_1199.csv, contains 92,984 query/response records. The first two rows of that file are listed at the top of Figure 3 where the timestamp and response time fields are the two left most gray columns. The TimeStamp (ms) column is the launch time of the event in Unix time expressed in milliseconds and the R (ms) is the elapsed milliseconds required for the response to be returned. The remainder of this figure uses these two records to illustrate how concurrency calculations are performed within the entire file.

3.1 Concurrency Calculation Methodology

The first step implemented in the Toolkit2 software is to create End timestamp records by adding the response time to a set of Beg records. The next step is to generate a list of Beg and End time hash records where a previously unset Beg record hash is assigned a value of 1 and an unset End record hash is given a value of -1. Duplicate Beg hash values are incremented by one and End hash occurrences are decremented by one. When the complete list is processed, each timestamp has a single hash number assigned to it that represents its state change contribution. The timestamp list is then processed in sorted timestamp order.



Hash Key Instance	JMeter Aggregate Report Data									
	TimeStamp (ms)	R (ms)	Web Event Name	Response Code	Response Message	User Thread	Data Type	Success	Byte Count	R (1st Byte)
Beg_1 =	1331866921**294**	**23**	022_Dept_jpg	200	OK	Group 1-182	bin	TRUE	31541	20
Beg_2 =	1331866921**325**	**16**	010_Home	200	OK	Group 1-196	text	TRUE	17991	14

Hash Key Instance	Create End Records	
	Beg Record + R =	End Record
End_1 =	1331866921**294 + 23** =	1331866921**317**
End_2 =	1331866921**325 + 16** =	1331866921**341**

Hash Key Instance	Set Record Hash Values		
	Hash Variable	Initial	Duplicate
Beg_1	{1331866921**294**} =	1	++
End_1	{1331866921**317**} =	-1	--
Beg_2	{1331866921**325**} =	1	++
End_2	{1331866921**341**} =	-1	--

Event	Time in State (ms)		
Count	State[0]	State[1]	
1	0		
1		23	
1	8		
1		16	
4	8	39	= Total

State Probabilities	
State[0]	State[1]
8/(8+39)	39/(8+39)
0.170213	0.829787

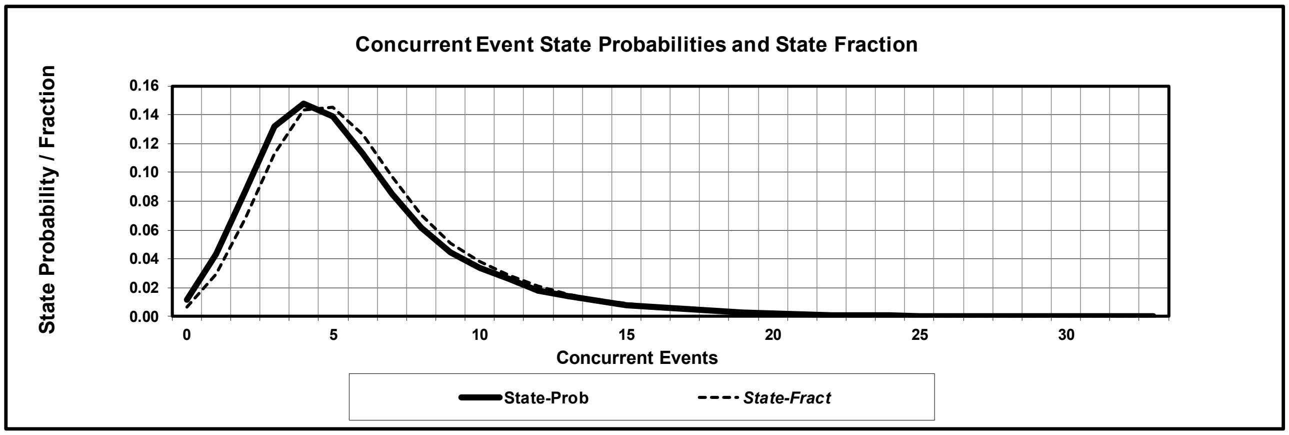
Figure 3: Concurrent Query Calculations Using Load Generator data

The time in a particular State is calculated by subtracting the current timestamp from the next one in ascending order. These calculated values are summed for each State with the example in Figure 3 yielding two events containing a total time in State [1] = 39 milliseconds. This methodology produces a list of States [ ], i.e., [0], [1], [2], etc., with each element of the list including an event count and a sum of milliseconds. The millisecond sums are divided by the total milliseconds for the test, yielding the proportion of time in each State. This event count and proportion of time in each state array is the frequency distribution of concurrent events in the SUT from a load generator perspective. Obviously, long network latencies across the E-Net connection between the load generator and the SUT skew this estimate. See Appendix A for a concurrency function source code listing.

3.2 Concurrency Calculation Results

Figure 4 tabulates and charts Toolkit2 demo concurrency statistics for all six web events, 010_Home through 040_Statistics, from both a state probability and state fractional count perspective. The table contains a statistical “Value” row, a “State-Prob” list of empirical state probabilities, a test run “Freq-Count” for each state observed, and a “State-Frac” decimal fraction of this frequency count by state. The two sets of “Value” row mean, sdev, and var values are calculated using the formulas for grouped data. For example, the mean value for the “State-Prob” data is $mean = \sum State\ x\ StateProb = 5.81$ and the same statistic for the “State-Frac” data is $mean = \sum State\ x\ StateFrac = 6.21$. The “Value” row also lists the maximum number of concurrent queries in the SUT during the entire measurement period, $max = 33$, and the total number of state change timestamps processes, $freq\_sum = 171{,}699$.

The “State-Prob” list of empirical state probabilities across all states experienced (0 – 33) is the concurrency state probability distribution based on this set of measurements. It should be noted that these empirical state probabilities differ from the frequency count by state fraction of total values, State-Frac”, because they account for time in a state. For example, the frequency count decimal fraction for state zero is State-Fract [0] = 1091 / 171699 = 0.00635, but its empirical state probability is State-Prob [0] = 0.01181. The software logic used to calculate frequency count by state and percentage of time in each state is contained in the concurrency() function listed in Appendix A. This Perl code is included in the “web_generator_toolkit2.pl” Perl program as a component of the JFBrady/web-generator-toolkit2 repository.



Concurrent Events (Count) {Total} - demo_2000_AggRpt_120_1199 Thursday 03/15/2012																	
	----------State Probability Statistics---------										----------State Fraction Statistics---------						
Statistic	mean	sdev	var	max	p_sum	freq_sum				Statistic	mean	sdev	var	max	p_sum	freq_sum	
Value	5.81	3.65	13.35	33	1.00000	171699				Value	6.21	3.66	13.36	33	1.00000	171699	
State	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
State-Prob	0.01181	0.04314	0.08711	0.13177	0.14784	0.13866	0.11296	0.08505	0.06141	0.04481	0.03388	0.02629	0.01817	0.01442	0.01087	0.00780	0.00621
Freq-Count	1091	4991	11664	19359	24583	24918	21655	16669	12091	8725	6538	4969	3586	2674	2037	1576	1174
State-Fract	*0.00635*	*0.02907*	*0.06793*	*0.11275*	*0.14317*	*0.14513*	*0.12612*	*0.09708*	*0.07042*	*0.05082*	*0.03808*	*0.02894*	*0.02089*	*0.01557*	*0.01186*	*0.00918*	*0.00684*
State	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
State-Prob	0.00506	0.00374	0.00252	0.00180	0.00140	0.00104	0.00074	0.00055	0.00034	0.00015	0.00020	0.00011	0.00004	0.00004	0.00005	0.00002	0.00001
Freq-Count	925	685	489	338	277	218	167	116	68	27	26	21	10	11	13	6	2
State-Fract	*0.00539*	*0.00399*	*0.00285*	*0.00197*	*0.00161*	*0.00127*	*0.00097*	*0.00068*	*0.00040*	*0.00016*	*0.00015*	*0.00012*	*0.00006*	*0.00006*	*0.00008*	*0.00003*	*0.00001*

Figure 4: Concurrent Events by Proportion of Time in State (Prob) and Frequency Count (Freq)

The Figure 4 graphs of the “State-Prob” and “State-Fract” results for this set of data are close to each other with the “State-Frac” plot shifted slightly to the right. The GitHub location of the Toolkit2 demo output file that produces the Figure 4 “State-Prob” results is provided in Appendix B under the “Figure 4: Concurrent Events” section. The “State-Frac” results are summarized in the “Concurrent Events” row of Figure 5 and expanded upon in the “Figure 5: Full Reports” section of Appendix B.

Figure 5 contains summary information for three of the Toolkit2 demo statistical reports. This figure includes results for all six web events and is the “Total” row of the full set of reports shown in Appendix B.

1. Concurrent Events
2. Response Times
3. Time Between Queries

Summary Statistics (ms) - demo_2000_AggRpt_120_1199 Thursday 03/15/2012											
label	n	tps	median	mean	sdev	cv	p90	p95	p99	min	max
Concurrent Events	171699	143.20	5	6.21	3.66	0.59	11	13	19	0	33
Response Times	92984	77.55	11	74.97	228.82	3.05	88	617	1048	3	4060
Time Between Queries	92983	77.55	9	12.89	13.22	1.03	30	39	59	0	341

Figure 5: Toolkit2 Demo Statistical Report Totals

The gray cells in Figure 5 are of interest from a concurrency perspective as well as a test traffic quality orientation. The first data row reiterates some of the “State-Fract” statistics listed in Figure 4 including, mean, sdev, max, and freq_sum (n). The second data row contains the query count, n = 92,984, the query rate, tps = 77.55 trans/sec, and response time statistics, e.g., mean = 74.97 milliseconds. The query count is the total number of data rows in the JMeter output file processed.

The last row provides “Time Between Queries” statistics with a count value, n, that is one less than the query count because the numbers are the differences between sorted order query timestamps. The coefficient of variation, cv = sdev / mean, of the “Time Between Queries” metric is an indication of request independence, a key measure of offered traffic quality. The observed value of cv = 1.03 is very close to the ideal 1.00, indicating the load generator is mimicking real-world user request timing. For a detailed discussion of request independence and its impact on test traffic quality see [4.].

4.0 Queueing Model View of Concurrent Events

The Figure 1 load testing setup can be viewed as a queueing system with 200 traffic sources (virtual user threads) and 8 servers (CPU cores) where the GET/POST circles in the SUT represent nine concurrent events. If an application's traffic flow characteristics match a queueing model's mathematical assumptions, the SUT's concurrency distribution can be approximated by the model's state probabilities.

One key assumption many common queueing models possess, including those in the QueState repository, is the request independence property of the Toolkit2 demo. With this traffic flow characteristic in common, how well do the empirical state probabilities, "State-Prob", of Figure 4 fit the appropriate queueing model's state probability distribution? If there is a reasonable match, that model may be useful as a concurrency distribution estimator for applications in the early stages of design or development.

The two queueing models illustrated in Figure 6 and Figure 7 are chosen for this comparison. The mathematical details of these two models are contained in Appendix C.

In Kendall Notation the models are:

1. M/M/c/N - Markovian arrivals and service times with c servers and fixed number of traffic sources (N)
2. M/M/c - Markovian arrivals and service with c servers and an infinite number of traffic sources

The Markovian property, the two M's in the Kendall notation, imply arrivals occur independently (first M) and service times are Negative-Exponentially distributed (second M).

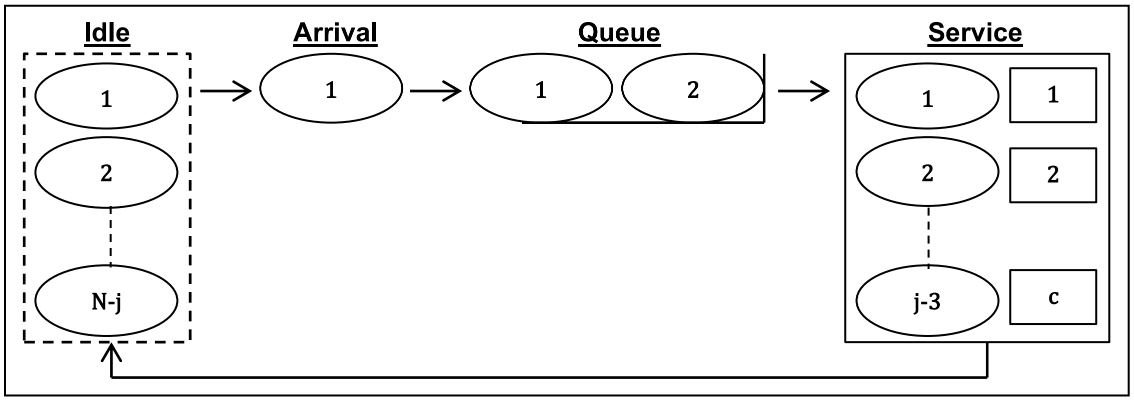


Figure 6: M/M/c/N Queueing Model

The M/M/c/N model accommodates the fact that the virtual user thread pool, i.e., group of users, is fixed and a thread in queue or service cannot be an arrival. The M/M/c model, on the other hand, assumes a dynamically changing set of users. As will be illustrated, this model difference is superfluous for a group of user threads as large as 200.

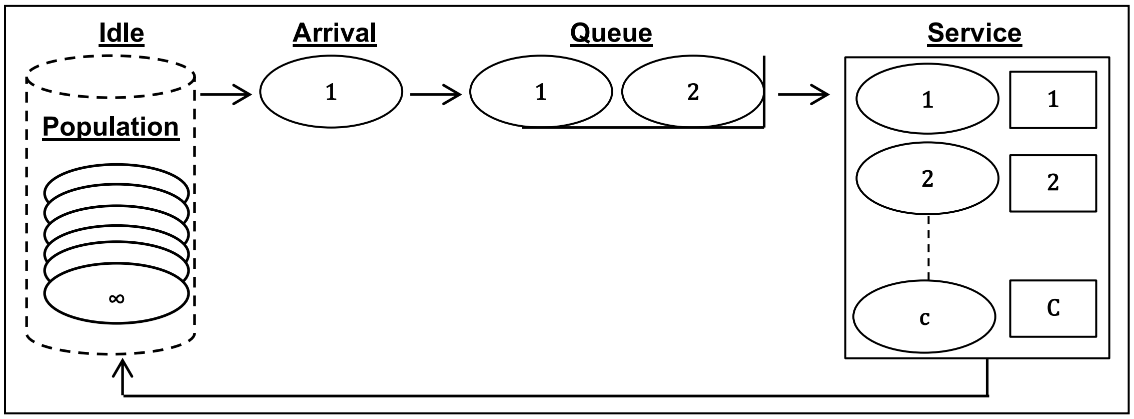


Figure 7: M/M/c Queueing Model

Figure 8 lists the queueing model configuration parameters and their relationship to the information contained in previous figures. Figure1 identifies the number of servers, c = 8, and traffic sources, N = 200, while Figure 4 includes the mean number of concurrent events, λ/μ = 5.81, and Figure 5 provides the query arrival rate, λ = 77.55 tps. This information is entered into the two models to produce their QueState demo state probabilities. Appendix C contains GitHub repository execution logistics for these results.

Queueing Model Input Parameters				
Parameter	**Symbol**	**Value**	**Description**	**Source**
Servers	c	8	Number Of Servers	CPU Cores in SUT
Sources	N	200	Number of Traffic Sources	Load Generator Threads
Traffic	λ/μ	5.81	Intended Offered Load (Erlangs)	Concurrent Event Count Mean
Arrival Rate	λ	77.55	GET / POST Arrival Rate	tps from Response Time Report
Service Rate	μ	13.34	GET / POST Service Rate / Server	Calculated From λ and λ/μ

Figure 8: Queueing Model Input Parameters

Figure 9 provides a comparison of the queueing model state probabilities (red triangles and green line) with the load test proportion of time in state empirical state probabilities (black solid line) where "Probability" is plotted as a function of "Concurrent Events".

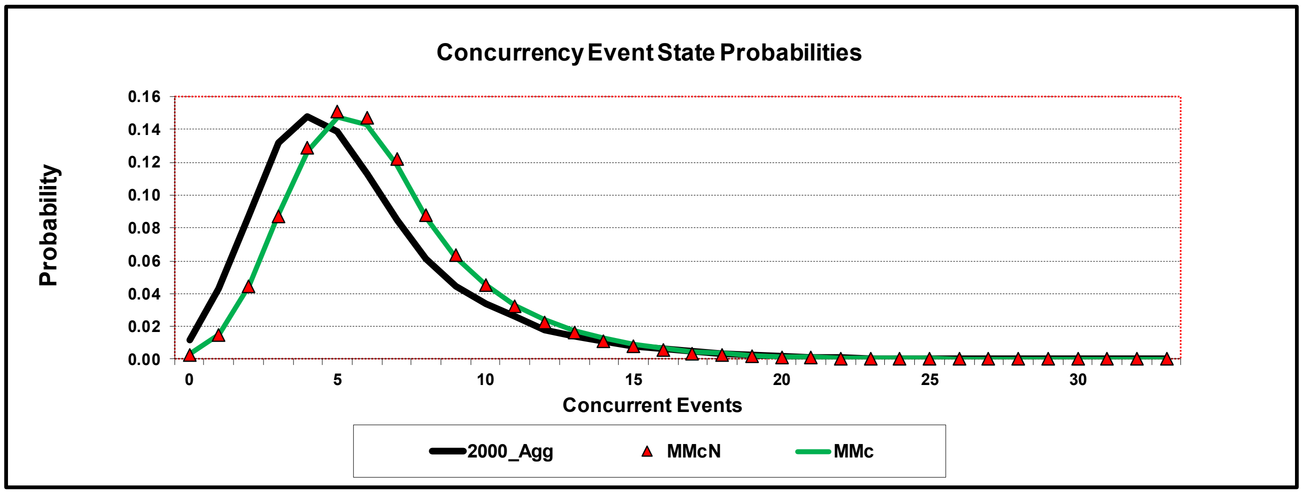


Table A	User Request Concurrency - 2000_Agg Test				
Statistic	**mean**	**sdev**	**var**	**max**	**p_sum**	**f_sum**
Value	5.81	3.65	13.35	33	1.000000	171699

Table B	Queueing Models					
Param	**Servers**	**Sources**	**λ/μ**	**λ**	**μ**	**p_sum**
MMcN	8	200	5.81	77.55	13.34	0.999993
MMc	8	∞	5.81	77.55	13.34	0.999922

Table C	State Probabilities																
State	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2000_Agg	0.01181	0.04314	0.08711	0.13177	0.14784	0.13866	0.11296	0.08505	0.06141	0.04481	0.03388	0.02629	0.01817	0.01442	0.01087	0.00780	0.00621
MMcN	0.00248	0.01482	0.04415	0.08723	0.12861	0.15093	0.14685	0.12184	0.08799	0.06322	0.04519	0.03213	0.02272	0.01599	0.01119	0.00779	0.00539
MMc	0.00266	0.01547	0.04496	0.08712	0.12661	0.14721	0.14263	0.11845	0.08607	0.06255	0.04545	0.03303	0.02400	0.01744	0.01267	0.00921	0.00669
State	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
2000_Agg	0.00506	0.00374	0.00252	0.00180	0.00140	0.00104	0.00074	0.00055	0.00034	0.00015	0.00020	0.00011	0.00004	0.00004	0.00005	0.00002	0.00001
MMcN	0.00371	0.00254	0.00173	0.00117	0.00079	0.00053	0.00035	0.00023	0.00015	0.00010	0.00007	0.00004	0.00003	0.00002	0.00001	0.00001	0.00000
MMc	0.00486	0.00353	0.00257	0.00187	0.00136	0.00099	0.00072	0.00052	0.00038	0.00027	0.00020	0.00015	0.00011	0.00008	0.00006	0.00004	0.00003

Figure 9: State Probability Distribution For 2000_Agg Test and Two Queueing Models

The Figure 9 graph is produced with the "State-Prob" Value row of Figure 4 as Table A where the mean value statistic in that table is the queueing model offered load, λ/μ = 5.81, in Table B. Figure 4 "State-Prob" data is listed in the 2000_Agg rows of Table C and the queueing model state probabilities are enumerated in the MMcN and MMc rows of the same table. The MMcN and MMc rows of Table C are the state probability portion of model output for the first thirty-four states (0 – 33) which consume nearly all the probability, p_sum for M/M/c/N = .999993 and M/M/c = .999922.

A comparison of the 2000_Agg empirical results with the two sets of queueing model state probabilities leads to the following observations regarding the Figure 9 illustration:

1. The queueing model state probabilities are biased to the right of the 2000_Agg test run empirical state probabilities in the higher density section of the curve but fit well in the right-hand tail where the important maximums are listed. It is unclear why this bias exists but perhaps it's because both queueing models assume Negative-Exponentially distributed service times and SUT service times are likely to have a smaller variance than this distribution implies.
2. The two queueing models (red diamonds and green line) have virtually identical state probabilities for all states shown, making them interchangeable as modeling tools when user populations are this large, i.e., 200 virtual user threads. This is important because M/M/c is a much easier model to implement than M/M/c/N. See Appendix C for the set of equations and list of assumptions associated with these two models.

The queueing model plots are certainly a close fit from a modeling perspective, making them a potentially useful tool for configuring internal system resources for applications anticipated to have the independent request property but still on the drawing board or in the early stages of development.

5.0 Summary

Load testing professionals are often asked to estimate how many concurrent requests the new application will support when there are no direct measurements available on the SUT to make that capacity determination. The approach taken here is to approximate the concurrent request pattern with either statistics produced from load tool data or queueing model results when such data is unavailable. It is hoped the Toolkit2 software combined with its Apache JMeter setup is of value when approximating concurrency with test data and the QueState queueing models are of use when preliminary estimation is required.

Because the Toolkit2 software uses data gathered by the load generator rather than recorded in the SUT, caution should be used when interpreting the results. Long or erratic network latencies between the load generator and the SUT may skew the concurrency numbers since time in SUT includes network latency.

There are other queueing model programs on the internet which generate performance metrics, but few of them compute the state probabilities needed for concurrency estimation like the QueState programs. Although concurrency estimation is the primary application in this environment, these eight models are general purpose and can be used to analyze any situation where they are a reasonable fit.

The software contained in the web-generator-toolkit2 and QueState GitHub repositories is free, downloadable, and modifiable, with all code in the "bin" directories written in Perl. It is hoped performance analysts will take advantage of these repositories and share their experiences.

Acknowledgment

This author would like to thank Xiaosong Lou [6.] for sparking my interest in user request concurrency analysis and sharing the empirical state probability calculation technique that is implemented in the Toolkit2 statistical analysis software.

References

[1.] J. F. Brady, "When Load Testing Large User Population Web Applications the Devil Is In the (Virtual) User Details," CMG *Proceedings* 2012, http://www.perfdynamics.com/Classes/Materials/Brady-CMG12.pdf

[2.] J. F. Brady, "It's Time to Retire Our 1970's User Demand Model for Transaction Computing and Adopt One That Reflects Modern Web User Traffic Flow," CMG *Proceedings*, 2014, https://jamesbrady.academia.edu/research

[3.] J. F. Brady and N. J. Gunther, "How to Emulate Web Traffic Using Standard Load Testing Tools," CMG *Proceedings*, 2016, https://arxiv.org/abs/1607.05356

[4.] J. F. Brady, "Is Your Load Generator Launching Web Requests In Bunches?," CMG *Proceedings* 2019, https://arxiv.org/abs/1809.10663

[5.] R. B. Cooper, "Introduction to Queueing Theory", Elsevier Science Publishing Co., Inc, New York, N.Y., (1984), http://www.cse.fau.edu/~bob/publications/IntroToQueueingTheory_Cooper.pdf

[6.] Xiaosong Lou, "CONCURRENT USERS: An Analytical Approach To Proper Workload Simulation," CMG *Proceedings*, 2021.

Appendix A

Concurrency Function Source Code (web_generator_toolkit2.pl)

```
sub
concurrency
{
  my($concurrency_ref) = @_;
  ##########################################################
  # User Request Concurrency Analysis:                     #
  # - Technique Introduced To This Author By Xiaosong Lou #
  ##########################################################
  # Variable Declarations Not Shown - See GitHub Repo      #
  # Variable Declarations - Initialized                    #
  ##########################################################
  my ($currentEvents)=0;
  my ($msec_sum)=0;
  ################################
  # Read list                    #
  ################################
  foreach $record (@$concurrency_ref){
    ($beg,$rt) = split ('\,',$record);
    ################################
    # Create End Timestamp         #
    ################################
    $end = $beg + $rt;
    ##############################################
    # Run Counter For Duplicate Begin Timestamps #
    ##############################################
    if ($timestamps{$beg}){
      $timestamps{$beg}++;
    }
    else{
      $timestamps{$beg}=1;
    }
    ############################################
    # Run Counter For Duplicate End Timestamps #
    ############################################
    if ($timestamps{$end}){
      $timestamps{$end}--;
    }
    else{
      $timestamps{$end}=-1;
    }
  }
  ###############################################
  # Process timestamps in ascending order       #
  ###############################################
  foreach $timestamp (sort keys(%timestamps)){
    ###############################################
    # Calculate msec for last timestamp           #
    ###############################################
    if ($lastTimestamp){
      $msec = $timestamp-$lastTimestamp;
    }
    else{
      $msec = 0;
    }
    ################################################
    # Add msec to current histogram entry          #
    ################################################
    if ($histogram{$currentEvents}){
      ($count,$ms) = split('\,',$histogram{$currentEvents});
      $count++;
      $ms+=$msec;
      $count_ms = join ',',$count,$ms;
      $histogram{$currentEvents} = $count_ms;
    }
    else{
      $count=1;
      $count_ms = join ',',$count,$msec;
      $histogram{$currentEvents} = $count_ms;
    }
    ################################################
    # Put current event count on list              #
    ################################################
    push @concurrentEvents,$currentEvents;
    ################################################
    # Update counter variables                     #
    ################################################
    $currentEvents += $timestamps{$timestamp};
    $lastTimestamp = $timestamp;
    $msec_sum += $msec;
  }
  #####################################################
  # Create state probability array                    #
  #####################################################
  foreach $pstate_val (sort keys(%histogram){
    ($count,$ms) = split('\,',$histogram{$pstate_val});
    if ($msec_sum){
      $prob = $ms/$msec_sum;
    }
    else{
      $prob = 0;
    }
    $count_prob = join ',',$count,$prob;
    $pstate[$pstate_val] = $count_prob;
  }
  ################################################
  # Sort concurrent events list                  #
  ################################################
  foreach $concurrentEvent (@concurrentEvents){
    $events = sprintf('%08d',$concurrentEvent);
    push @concurrentEvents_out,$events;
  }
  @concurrentEvents_out = sort @concurrentEvents_out;
  #####################################################
  # Assign unobserved state probabilities zero        #
  #####################################################
  for($i=0;$i<@pstate;$i++){
    if (!$pstate[$i]){
      $pstate[$i] = join ',',0,0;
    }
    $pstate[$i] = join ',',$i,$pstate[$i];
  }
  return(\@concurrentEvents_out,\@pstate);
}
```

Appendix B

Toolkit2 Demo Output Files
(web-generator-toolkit2/demo)

Figure 4: Concurrent Events – State Probability Statistics

The GitHub location of the concurrent events for "Total" csv file produced by running the Toolkit2 demo.

1. Toolkit2 -> histograms -> concur -> demo_2000_AggRpt_120_1199_Total_concur.csv

Figure 5: Full Reports:

The GitHub locations of the three full statistical reports produced by running the Toolkit2 demo.

1. Concurrent Events – Toolkit2 -> statistics –> demo_2000_AggRpt_120_1199_20120315_concur.csv
2. Response Times – Toolkit2 -> statistics –> demo_2000_AggRpt_120_1199_20120315_agg.csv
3. Time Between Queries - Toolkit2 -> statistics –> demo_2000_AggRpt_120_1199_20120315_arr.csv

1. Concurrent Events – State Fraction Statistics

Concurrent Event Statistics (Count) - demo_2000_AggRpt_120_1199 Thursday 03/15/2012											
label	**n**	**tps**	**median**	**mean**	**sdev**	**cv**	**p90**	**p95**	**p99**	**min**	**max**
010_Home	47751	39.83	1	1.12	1.13	1.01	2	3	5	0	10
012_Home_jpg	48798	40.70	1	1.37	1.25	0.91	3	4	5	0	13
020_Dept	24719	20.62	1	0.85	0.85	1.00	2	2	4	0	7
022_Dept_jpg	24825	20.71	1	0.81	0.77	0.96	2	2	3	0	6
030_Demographics	24560	20.49	1	0.80	0.81	1.02	2	2	3	0	7
040_Statistics	12331	10.28	4	3.83	1.95	0.51	6	7	9	0	13
Total	**171699**	143.20	5	**6.21**	**3.66**	0.59	11	13	19	0	**33**

2. Response Times

Aggregate Stats [Response Time(ms)] - demo_2000_AggRpt_120_1199 Thursday 03/15/2012											
label	**n**	**tps**	**median**	**mean**	**sdev**	**cv**	**p90**	**p95**	**p99**	**min**	**max**
010_Home	24384	20.34	9	31.60	161.33	5.11	20	62	516	5	3616
012_Home_jpg	24950	20.81	20	42.23	173.89	4.12	41	65	301	13	3864
020_Dept	12489	10.42	7	34.79	184.64	5.31	19	61	1412	4	3227
022_Dept_jpg	12533	10.45	5	30.07	183.20	6.09	14	33	1298	3	3430
030_Demographics	12424	10.36	7	30.04	164.54	5.48	17	53	1220	4	3217
040_Statistics	6204	5.17	622	638.70	171.31	0.27	674	768	1038	334	4060
Total	**92984**	**77.55**	11	**74.97**	228.82	3.05	88	617	1048	3	4060

3. Time Between Queries

Inter-arrival Summary Statistics (ms) - demo_2000_AggRpt_120_1199 Thursday 03/15/2012											
label	**n**	**tps**	**median**	**mean**	**sdev**	**cv**	**p90**	**p95**	**p99**	**min**	**max**
010_Home	24383	20.34	34	49.17	49.55	1.01	113	148	228	0	533
012_Home_jpg	24949	20.81	33	48.06	48.66	1.01	111	145	227	0	527
020_Dept	12488	10.42	67	95.99	94.58	0.99	219	285	432	0	903
022_Dept_jpg	12532	10.45	66	95.67	95.74	1.00	221	288	437	0	871
030_Demographics	12423	10.36	67	96.51	97.87	1.01	224	295	450	0	990
040_Statistics	6203	5.17	133	193.24	196.31	1.02	445	579	910	0	1847
Total	**92983**	77.55	9	12.89	13.22	**1.03**	30	39	59	0	341

Appendix C

Queueing Model State Probabilities
(QueState/demo)

Figure 8: State Probability Input Parameters

The GitHub location of the state probability files produced by running the QueState demo.

1. QueState – MMcN -> MMcN_QueState_Results -> MMcN_2000_AggRpt_results_yyyymmddhhmmss.csv
2. QueState – MMc -> MMc_QueState_Results -> MMc_2000_AggRpt_results_yyyymmddhhmmss.csv

Queueing Model Formulas

Below are the state probability formulas for the two queueing models as depicted in [5]. The symbol in parenthesis to the right of each variable definition is the QueState repository term for that variable.

M/M/c/N Model State Probabilities:

Key assumptions:
Source - the number of traffic sources is finite.
Arrival - arrivals occur quasi-randomly and have full access to all the servers.
Queue - blocked arrivals are delayed.
Service - service times are Negative-Exponentially distributed.

Quasi-random arrivals imply no coercion exists between sources when making requests but the arrival rate changes as sources go in and out of the idle state, accounting for the fact that a source in queue or service cannot be an arrival. In this closed source system, the outside observer's and arriving customer's view differ but happily they are related in a simple way. The outside observer's n-Source state probability distribution, Eq. C1.1 and Eq. C1.2, is equal to the arriving customer's [n-1]-Source state probability distribution.

$$P_j[n] = \begin{cases} \binom{n}{j} \hat{a}^j P_0[n] & (j = 1,2,\dots,s-1) \\ \frac{n!}{(n-j)!s!s^{j-s}} \hat{a}^j P_0[n] & (j = s, s+1, \dots, n) \end{cases} \qquad (C1.1)$$

$Where\ P_0[n]\ is\ given$ by

$$P_0[n] = \left[\sum_{k=0}^{s-1} \binom{n}{k} \hat{a}^k + \sum_{k=s}^{n} \frac{n!}{(n-k)!\,s!\,s^{k-s}} \hat{a}^k\right]^{-1} \qquad (C1.2)$$

$Where$:
$n = number\ of\ traffic\ sources$ (N)
$\hat{a} = offered\ load\ per\ idle\ source\ (ai\_src)$
$s = number\ of\ servers$ (c)
$P_j[n] = state\ probability\ that\ j\ of\ the\ n\ sources\ are\ in\ the\ system\ (P[j])$
$P_0[n] = state\ probability\ that\ all\ n\ sources\ are\ idle\ (P[0])$

$The\ offered\ load\ per\ idle\ source, \hat{a}, is\ approximated\ by$:

$$\hat{a} = \frac{a_{src}}{1 - a_{src}} \qquad (C1.3)$$

$Where$:
$a_{src} = intended\ offered\ load\ per\ source\ (a\_src)$

$$a_{src} = \frac{\lambda/\mu}{n} \qquad (C1.4)$$

$Where$:
$\lambda = intended\ arrival\ rate\ (lamda)$
$\mu = service\ rate\ (mu)$

Appendix C - Continued

M/M/c Model State Probabilities:

Key assumptions:

Source - the number of traffic sources is infinite.
Arrival - arrivals occur randomly at a constant rate and have full access to all servers.
Queue - blocked arrivals are delayed.
Service - service times are Negative-Exponentially distributed.

Random arrivals imply no coercion exists between sources when making requests and arrivals are drawn from an infinite population at a constant rate. It is the well-known Poisson process where times between arrivals are Negative-Exponentially distributed and the number of arrivals in constant length intervals, are Poisson distributed. Unlike the closed source model, the outside observer and arriving customer share the same view of the system possessing the state probabilities shown in Eq. C1.5 and Eq. C1.6.

$$P_j = \begin{cases} \frac{a^j}{j!} P_0 & (j = 1,2,\dots,s-1) \\ \frac{a^j}{s! s^{j-s}} P_0 & (j = s, s+1, \dots) \end{cases} \tag{C1.5}$$

$Where\ P_0\ is\ given\ by$

$$P_0 = \left[\sum_{k=0}^{s-1} \frac{a^k}{k!} + \frac{a^s}{s!\,(1 - a/s)} \right]^{-1} \qquad (0 \le a < s) \tag{C1.6}$$

$Where$:
$a = offered\ load\ (a)$
$s = number\ of\ servers$ (c)
$P_j = state\ probability\ that\ j\ customers\ are\ in\ system\ (P[j])$
$P_0 = state\ probability\ the\ system\ is\ empty\ (P[0])$

$$a = \lambda / \mu \tag{C1.7}$$

$Where$:
$\lambda = arrival\ rate\ (lamda)$
$\mu = service\ rate\ (mu)$