Performance Workload Design

The goal of this paper is to show the basic principles involved in designing a workload for performance and scalability testing. We will understand how to achieve these principles in practice. We will design a simple workload to understand the performance and scalability characteristics of a web application. Our sample web application is a website consisting of 3 pages. This paper walks through the steps for designing a workload for this application.

Why Design?

Before we talk about a good workload design, first let's tackle the question of why this is important. The many load testing tools out there make light of the design process. They make claims of how easy it is to create a load test by simply clicking through a web application which is recorded and the steps played back. Throw hundreds or thousands of emulated users running that same scenario and presto - you have a workload. Of course, this method only works for web applications and even in this case there are problems with this method that are a subject of a separate Record and Playback article.

The important thing to keep in mind is that test results are only as good as the tests that are used. Software bugs are often the result of poor QA. As QA tests improve their coverage, the bugs in the application will correspondingly decrease. Workloads are "QA tests" for performance. The better the workloads, the more valuable they are in measuring and analyzing performance problems. There are several reasons why one would want to ensure a workload is well-designed:

- The workload helps in performance and scalability tests to understand bottlenecks.
- The workload can be used for capacity planning and sizing purposes.
- The workload resembles the real load on the production application thus helping better optimize the application infrastructure.

We will see how a workload can achieve these goals.

Definitions

Before we go further, let us first define some common terms.

Workload: The amount of work a system has to perform in a given time. In the performance field, a workload usually refers to the combined load placed on an application by the set of clients it services.
Benchmark: A specific workload when applied to a target application that can be standardized. A benchmark usually implies rigorous rules to ensure that results are meaningful and comparable.
Scalability: The ability of a system to continue to perform well as load is increased. Scalability is crucial for websites as they have to deal with constantly changing load and may have to handle peak loads that are several times the average.
SUT: The System Under Test (commonly referred to as SUT) is the entire infrastructure that is required to host the application being tested. For example, a typical SUT for a web application will include systems for proxy, web, application and database servers and the corresponding software infrastructure and applications running on them.
Load Generator or **Driver**: A Load Generator or Driver is a program that controls the workload submission to the SUT.
Operation: An Operation is a logical unit of work initiated by the Load Generator or Driver. An operation may consist of multiple request/response cycles between the Driver and the SUT (e.g a request for a web page with multiple embedded objects can constitute an operation). A mix of operations constitutes a workload. The term **Transaction** is also often used, especially for

database-related workloads.

Workload Design Principles

A well-designed workload should have the following characteristics :

1. Predictability
The behavior of the system while running the workload, should be predictable. This means that one should be able to determine how the workload processes requests and accesses data. This helps in analyzing performance. Predictability does not mean that the workload should perform the exact same actions in the same sequence every time (i.e. has no randomness), nor that there should be no variation over time. The graph in Figure 1 below shows processor utilization of a system running a well designed workload. As can be seen, the workload varies over time but is predictable.

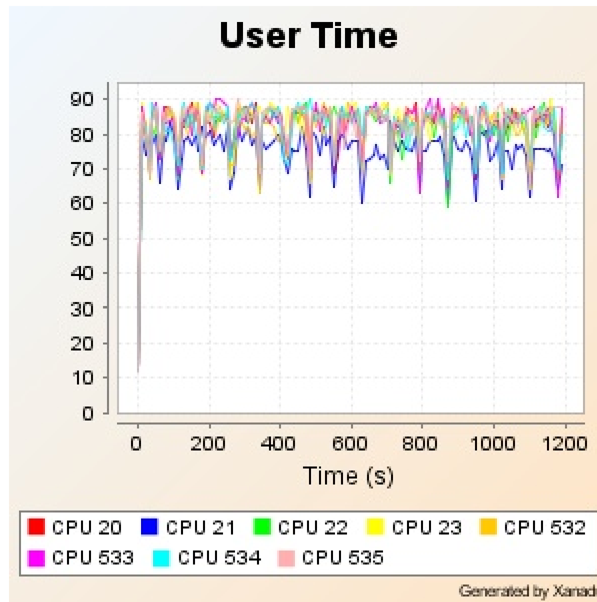

Figure 1: CPU Utilization Over Time

2. **Repeatability**
It is important that a workload produces repeatable results. If the workload is run several times in an identical fashion, it should produce results that are almost identical. Without this repeatability, performance analysis becomes difficult.

3.**Scalability**
A workload should be able to place different levels of load in order to test the scalability of the target application and infrastructure. A workload that can only generate a fixed load or one that scales in an haphazard fashion that does not resemble actual scaling in production is not very useful.

Workload Design Steps

Listed below are the steps we will follow to design our example workload :

1. Design the application
2. Define the metrics

3. Design the load
4. Define scaling rules
5. Design the load generator

1. Design the application

This step may sound confusing - after all, the whole purpose of performance testing is to test an already existing application (or one in development before it goes live). So let's clarify what is involved in this activity.

a) Define the actors and use cases. The use cases help define the operations of the workload. In a complex workload, the different actors may have different use cases e.g. a salesperson entering an order and an accountant generating a report.
For our sample workload, we have a single type of actor, the "user" of the website and we have 3 use cases, namely a request for the 3 web pages.
b) Define the operations. In the simplest case (as in our example), each use case will map to an operation. It is important to keep the following things in mind while defining operations :
   - For a workload to be useful, the number of operations should be kept small (6-8). Too many operations can make the workload not only difficult to construct and manage, but also less useful for understanding performance. It is not necessary to simulate every single use case, but rather focus on the use cases that are most likely to occur. The lower frequency ones can be ignored or combined with others into a single operation.
   - If the application being tested is complex with multiple use cases for distinct actors, consider grouping the use cases that are likely to occur together and create separate workloads for each group.

2. Define the Metrics

In this step, we define what the workload should measure. Typical metrics include throughput, response time or number of users. However, depending on the workload, other metrics such as processor or memory utilization, i/o rates, or some other application specific metrics may make sense.
Although throughput and response times are common metrics, there are several ways in which these can be defined.

Throughput

A throughput metric measures how many operations can be processed by the SUT during a unit of time. A throughput metric can be computed using several methods :

- For a single workload driving all operations, throughput (X), can be calculated by keeping a count (N) of all the operations executed during a certain amount of time (T) as $X = N / T$. This is the simplest method.

- Use a single operation type that contributes to the throughput metric. Although all other operations will execute, they are not counted. This method should be used when the amount of work performed by the different operation types varies considerably or one particular operation is dominant. [TPCC] uses this method.

- Use a geometric mean of all different operation types. The geometric mean weights all operations equally - relative changes in short running operations have just as much effect as in long running operations. This is the recommended method if there are multiple workloads that may not directly relate to each other and a single metric is desired. [SPCP00] uses this method.

- If multiple workloads are involved, an arithmetic mean of the throughput of all the workloads can be used if the throughputs of the different workloads are related in some manner. [SPR02] used this method.

For our example, we will choose a throughput metric of the number of web requests processed per second computed as the total number of requests processed divided by the time interval in seconds.

Response Times

In most cases, a throughput metric is not very meaningful without a corresponding response time requirement or metric.

The term Response Time is typically defined as the time interval between the last byte of an operation request sent to the SUT and the first byte of the response. This definition measures the response time of the SUT. In some cases it may be desirable to define response time as the time interval between when the emulated user sends the request and receives the response. In this case, the response time will include the client-side protocol stack. For example, if the request is a https request, the time taken by client to encrypt the request, decrypt the response etc. will be included.

Response Time metrics are usually specified in terms of *average* (mean) and *90th percentile* requirements. Although, the average response time is a common measure, it may be insufficient in many cases. This is because there can be wide fluctuations in response times (ranging from 100 ms to 5000 ms say) and yet the avg. response time requirements may still be satisfied. By requiring that 90% of operations complete within a specified interval, we are assured of a more stable system. The average response time should not be greater than the 90% response time. This ensures that we do not have excessively long outlying operations. In certain cases where performance is of utmost importance, it may be required to specify a 99th percentile response time requirement.

How does one decide what the response time requirement should be ? For certain applications, the end-user SLA may dictate the requirements. For others, especially web applications, extremely low response times (less than a second or two) may be critical to attract and maintain customers. Operations that involve actual user interactions should typically be less than a few seconds whereas a complex batch job may take hours to complete. The response time requirements may have to be finalized after the workload is constructed and sample runs performed.

For our example workload, we define the 90th percentile response time for all operations to be less than 100 ms.

3. Design the Load

This is the most important step in workload design. The relevance of the workload will depend on how closely it emulates the application load in production. At the same time, it is important to realize that a test workload should only capture the most significant aspects of the live load – otherwise, it can get complicated and not be a very useful performance tool.

Load definition means defining the manner in which the different operations defined in Step 2 load the SUT. This involves deciding on the operation mix, mechanisms to generate data for the requests, and deciding on arrival rates or think times.

Arrival Rates

The rate at which requests are submitted to the SUT is called the arrival rate and the time between requests is known as the inter-arrival time. Arrival rates are used in open systems where the user population is large and unknown, as for example, in web applications. Arrival rates are typically modeled using a negative exponential distribution [TRIV82] that are truncated at some

reasonable value, say 5 times the mean.
For our example workload, we define an inter-arrival time of 1 second with a negative exponential distribution truncating at 10 seconds.

Think Times

In benchmarks that model applications involving direct user interaction, it may make sense to model think times instead of arrival rates, the idea being that the user takes some time to respond to what is being presented to him. Think times can vary based on the type of operation. If an operation presents lots of data, it might take the user more time to respond. Think times are also typically modeled using a negative exponential distribution with some targeted mean. The distribution is truncated at a fairly large value, say 5-10 times the mean.

Operation Mix

We need to decide on the frequency with which each of the operations will be exercised, the goal being to execute operations that test most frequently used code paths. This is often expressed as a operation mix% for each type of operation, the total adding up to 100%. The mix for any one type of operation should not be very small (say < 1%) as it may require a large number of samples to achieve that probability.

Mixes can be specified in different ways :

- ***Flat Mix***: The flat mix is the simplest and is used when the operations are independent of each other and each operation has the same probability of occurrence. The mix selects one of the operations at random.
- ***Flat Sequence Mix***: This mix specifies a set of a sequence of operations. For example, one set could be the sequence {operation1, operation2} and another could be {operation1, operation3} with a probability assigned to each set. This is often referred to as scenarios.
- ***Matrix Mix***: Also called a ***Transition Mix***, this mix describes the transition probabilities in a Morkov model [Menascè04] as a right stochastic matrix. This must be a square matrix. Each row represents a probability vector for transitioning from an operation (row operations index) to an operation (column operations index). Most web applications would typically use a matrix mix as transitions between various pages can be represented by a state diagram.

For our example workload, we choose a matrix mix with the following probabilities :

From	*To* *Page 1*	*To* *Page 2*	*To* *Page 3*
Page 1	0.00%	80.00%	20.00%
Page 2	20.00%	39.00%	41.00%
Page 3	60.00%	19.00%	21.00%

Operation Data

Depending on the operation, various pieces of data may have to be generated as input to the request. It is essential to vary the data used in order to test realistic scenarios and also vary the load on the SUT. For example, if an operation accesses a varying number of items ranging from [1..100], it is a good idea to test the entire range instead of always selecting a fixed number of items. It is also good practice to inject a small number of errors by using invalid data as input to

the operations. This may catch performance problems in error handling.

Generating data can be problematic. If tests are to be conducted on a copy of the production data set, then the workload developer must know the possible values for all of the input fields. For large data sets, this can become unwieldy and a great deal of energy may be spent in extracting the data, loading the data into the test harness, ensure it's accuracy etc.

Another approach is to use synthetic data. By using algorithms to generate the initial data in the database as well as data generated by the workload, it is easier to keep things in sync and at the same time test interesting scenarios. The most common method of generating synthetic data is by the use of random number generators. Random number generators can be used effectively to simulate various types of access patterns, either uniform or non-uniform random. A rule of thumb: Any time a random number is generated, it should be selected from a distribution.

Uniform Random

This is the most commonly used method of generating random data. A piece of data is selected at random from a uniform distribution with a given mean. Operating systems and languages provide support for various random number generators. However, these not only vary in quality and performance, but also in the randomness of the data they provide. Some good random number generators are available in [PARK88] and [MARS91].

Non-uniform Random

In the real world, data access is usually not uniform. Enterprises don't have all customers ordering the same number of items and typically some items are more popular than others. Many applications rely on such non-uniform accesses to cache data that is heavily used. In order to test the performance of caching algorithms and emulate real use cases, it may be important for a workload to simulate such non-uniform access.

A well known non-uniform random number algorithm is specified by the Transaction Processing Council© as :

NURand(A, x, y) = ((random(0,A) | random(x,y)) % (y – x + 1)) + x

where :
A is a constant chosen according to the size of the range [x..y].
The value chosen for A determines the frequency of the hotspots and as such it is a good idea to model this function and understand its characteristics. See [TPCC] to understand how it uses this to generate non-uniform data.

4. Define Scaling Rules

Complex workloads need a means to scale the workload depending on the actual hardware they are deployed on. Often times, scaling is done by increasing (or decreasing) the number of emulated users. Although this will cause the load on the SUT to change, scaling the load often needs to go way beyond that.

Linear Scaling

In this model, everything is scaled in a linear fashion. If the workload simulates users performing operations that access data, then the number of users and the amount of data accessed are both scaled, usually using a single scaling factor or rate. The advantage to this model is that it provides a means of predicting performance and scalability (assuming that the SUT exhibits stable, scalable

behavior). This type of scaling especially makes sense, if the workload is to be used for sizing purposes. It also represents many real-world situations. However, note that even though load may be added linearly, there may actually be non-linear increases in the work performed by the SUT, limiting system scalability. This could be due to increased contention for various resources resulting in serialization on mutexes, etc.

There are many instances when all the data is not scaled by the same scale factor. Consider an online store that sells *x* different products and has *y* customers. As business grows and the number of customers increase, the types of products sold by the store will increase at a much lower rate. A website that aggregates news from several different news feeds will see growth as the number of customers it serves increase, requiring it to perhaps fetch more/different categories of news, but the number of news feeds remains constant.

If a workload is crafted carefully, it is possible to scale various parts of the data store at different rates and yet retain its usefulness to measure system scalability. [SPECj04] is an example of a workload that uses multiple scale factors.

Example: Our sample workload scales by a rate called S_r. The number of emulated clients is S_r.

Non-linear Scaling

Although linearly scaling the benchmark makes it easier for performance analysis work, in the real world many applications scale non-linearly. Consider for example, a typical web2.0 site that allows users to tag various things. When the site is small, it will have a small number of users and a certain number of tags. As the number of users and activity increases, the number of tags does not grow linearly. At some point, the number of tags may remain constant. Such complex scaling is more difficult to model in a workload. For a realistic simulation and especially if the workload is being developed for sizing purposes, it is important to model realistic, non-linear scaling behavior. An example of such a workload can be found in [OLIO].

5. Design the Load Generator

The Load Generator or Driver implements the workload and keeps track of various performance metrics such as response times, think times and throughput. Several off-the-shelf tools are available to automate the load generation task and/or assist in developing them. Tools vary greatly in their functionality so a tool should be carefully evaluated to ensure it can generate the defined workload. Some points to keep in mind :

- If the driver is emulating multiple user connections, it is important to ensure that each emulated user has a unique connection to the SUT (http, tcp or other type of connection). No pooling of connections should occur on the driver side.

- Each emulated user should use its own random number generator, seeded with a unique value. This ensures the randomness of the generated data.

- Current time should be gathered using the most efficient and accurate method possible. This is especially important when latencies are small (in micro or milli seconds).

- When the emulated user goes to sleep to emulate think times, it is important to calibrate the sleep times as there may be delays in the operating environment's scheduling of the wake up of the driver thread.

- The driver must have good performance and be scalable – i.e. the driver should not become the bottleneck ! It is a good idea to monitor the driver system(s) while doing the test runs.

7. Other Considerations

Initial Data Store

Many servers and applications access some sort of data store, one backed by a file system, RDBMS, LDAP server or some other type of database. When designing a workload in which a data store is involved, due consideration must be given to both the initial state of this store and its state after a test run. For example, if the load needs to be run for 8 hours, the rate at which the data grows should not be such that the test configuration becomes unmanageable. Even if the workload being created only tests new records being inserted into the store, it may be important to pre-load data, as the performance of the insertions may depend on how many records already exist and the way the data is organized.
If the application performs a search on the data store based on certain value(s) in some fields, then it is important to load these fields in a manner consistent with the expected results.

Workload Model

While designing the workload, it is useful to build a model detailing the various metrics, data sizes, access patterns etc. using a spreadsheet. This helps in understanding data sizing issues (e.g. how much will the data grow if the load is run for 24 hours vs. 8 hours). Adjustments to various parameters such as operation mixes, arrival rates etc. can be easily tried out to evaluate their effects.

The model can also be used later during performance analysis to correlate measured data with what is expected.

Test Harness

Once the workload is designed and implemented, how do we use it for performance testing ? This is where a Harness comes in. Most load generator tools usually also include a harness to manage test execution. Again, functionality can vary greatly. The harness should minimally provide the following functionality:

- Run management. Queue, execute, view status and kill runs.
- Collect configuration information of the various products used.
- Automate running of various monitoring tools during a run.
- Provide a means to easily view and compare results of past runs.

Conclusion

Accurate construction of workloads is the first step in understanding performance issues. This paper highlights a methodical, step by step approach to workload development. There are many complex questions that must be answered when designing a workload. We have covered the most important issues that are applicable to a wide variety of workloads such as throughput and response time metrics, arrival rates, think times, data generation, data store sizing and workload scaling.

References

[TPCC] “TPC Benchmark C”, http://www.tpc.org/tpcc/spec/tpcc_current.pdf, pp:28, 21, 74

[SPCP00] “SPECCPU 2000 Run Rules”

[SPR02] "SPECjAppServer2002 Run Rules"

[TRIV82] Kishor Shridharbhai Trivedi, "Probability & Statistics with Reliability, Queueing, and Computer Science Applications," pp. 114-118 (1982).

[Menascè04] Daniel A. Menascè, "Performance by Design," pp.258-272 (2004).

[PARK88] S.K. Park and K.W. Miller, "Random Number Generators: Good Ones Are Hard To Find", Communications of the ACM, pp. 1192-1201 (October 1988)

[MARS91] George Marsaglia and Arif Zaman, "Toward a Universal Random Number Generator", Annals of Applied Probability, no. 3, vol. 3, pp. 462-480 (1991).

[SPECj04] "SPECjAppServer2004 Database Scaling Rules"

[OLIO] "A Web2.0 Toolkit and benchmark", http://incubator.apache.org/olio/

All Rights Reserved. See Terms and Conditions.