Perf Cells: A case study in achieving clean performance test results

Vivek Venkatachalam (vivekven@microsoft.com)
Shirley Tan (shirleyt@microsoft.com)
Marcelo Nery dos Santos (marsant@microsoft.com)

Abstract

A key indicator of the quality of a software product is its responsiveness and performance, as this enables users to get tasks done quickly and efficiently. Test teams realize this and therefore typically set aside time and resources for performance testing. Unfortunately, the effort inevitably runs into the dreaded "high unexplainable variability in performance test results". Since fixing this is hard to do, a typical workaround is to raise the acceptable loss threshold to ensure only large performance trigger corrective action. However, this causes a "death-by-a-thousand-cuts" effect where numerous small performance losses (that are all real) are allowed into the product. By the time the product is ready to ship, these small performance losses have accumulated and the product exhibits poor performance with no easy fix in sight.

How does one get close to the ideal of a performance test system that can reliably detect even small performance losses on a build-over-build basis while minimizing false positives? This paper describes our attempt to tackle this problem while running performance tests for Microsoft Lync.

The bulk of the variation in performance results when testing a distributed system is typically due to network traffic variations and varying load on the servers. Our approach to eliminate this variability was to design a system where the performance test runs on a virtualized distributed system, that we call a Perf Cell. This is essentially a single physical machine that houses all the individual components on separate virtual machines connected via a virtual network and otherwise isolated from all external networks. In the paper, we present our design and implementation as well as results that indicate that variability is indeed reduced. We believe this paper would be useful reading for engineers responsible for designing, implementing or running a performance engineering system.

Biography

Vivek Venkatachalam is a software test lead on the Microsoft Lync team. He joined Microsoft in 2003 and worked on the Messenger Server test team before moving to the Lync client team in 2007. He is passionate about working on innovative techniques to tackle software testing problems.

Marcelo Nery dos Santos is a software engineer on the Microsoft Lync test team. He has primarily worked on tools to help test performance for the Lync product. His main interests are working on innovative tools and approaches for enabling more efficient testing.

Shirley Tan is a software engineer on the Microsoft Lync test team and recently completed 5 years at Microsoft. Apart from her work on performance testing, she is interested in Model Based Testing and Code Churn Analysis.

1. Introduction

As competing software products grow increasingly similar in terms of functionality and feature set, aspects such as performance, security, accessibility etc. are what set them apart. Performance is arguably among the more important of these aspects. Given two software products that provide a comparable set of features, customers will tend to prefer the product that allows them to accomplish their tasks in the quickest and most efficient manner. This implies that software teams should pay careful attention to the performance of their products to maximize customer satisfaction. The best way to do this is to incorporate a formal performance engineering process into the product development life cycle and to keep making improvements continually to both the process and the product.

A performance engineering system, in our opinion, is not really very different from any other engineering quality control system. At its core, a well-designed performance engineering system allows the product team to a) prevent unintended performance losses from creeping into the product by monitoring daily builds for degradations in performance and b) reliably verify that any fixes made actually fix the problem. A key obstacle one faces when designing/implementing/running this system is that the performance of the product under test could exhibit variations due to factors external to the product (e.g. state of the system under test, state of the network etc.). We propose the term “noise” to collectively refer to all such factors because they mask the “signal” that we care about (performance changes directly attributable to specific changes in the product code). Noise in the engineering system is a big problem because by masking the signal it becomes difficult to distinguish real performance problems in the product code from artificial/transient problems caused by these external factors. As members of the Microsoft Lync client performance test team, the authors directly encountered this problem during the Lync 2010 ship cycle and this paper is essentially a firsthand account of our experience with this problem and our attempt to solve it using virtual machines.

We start off by giving the reader a brief background on the Lync product and an overview of the performance engineering process we followed during the Lync development life cycle. Section 2 also serves as a high-level introduction on the key things to consider while setting up a performance engineering process for the benefit of readers unfamiliar with this domain. In section 3, we delve into the problems we faced because of the unexplained variations in our performance results and the repercussions of those problems. In sections 4 and 5, we then describe our solution and present details on the design and implementation of our solution to minimize noise in the performance engineering system by using virtual machines. Finally in sections 5 and 6, we present our results from this solution, key takeaways and future improvements.

2. Lync and the performance engineering process

This section will describe the multiple phases of our performance process and bring some considerations on the initial performance tests that were executed.

2.1. Lync – a brief overview

Microsoft Lync[1] is an enterprise communication and collaboration system that offers users a rich set of features (instant messaging, user presence, voice/video calls and conferencing, content sharing among many other features). It comprises of a set of server components (collectively known as the Lync server topology) and a set of clients that run on multiple platforms (Windows, Mac, Web client amongst others) and connect to and communicate with the server components to provide this rich functionality. The authors were on the team responsible for verifying the performance aspects of the flagship Lync client,

[1] http://lync.microsoft.com/

the client running on the Windows platform and all future references to Lync in this paper imply this specific Lync client.

As should be clear from this description, the Lync system is a distributed system and the majority of the functionality of the Lync client involves communication with the server components and through them, communication with other Lync clients if needed. This causes some unique issues while attempting to test the performance characteristics of any one instance of the Lync client, which is discussed later in the paper. With this basic understanding of the Lync product in place, we now give the reader a basic overview of the performance engineering process we followed.

2.2. Performance Engineering Process Phases

2.2.1. Planning

During the planning phase, we identified the set of key scenarios that we wanted to focus on improving, as well as the key metric we wanted to track. There are many performance related metrics one can track (elapsed time, CPU usage, disk usage, network usage, memory usage, power usage etc.) but in the interests of optimizing the cost/benefit ratio of our effort, we decided at the very outset to focus on the elapsed time metric, since that made the most sense for a real-time application such as Lync.

Picking the elapsed time metric does not mean that we ignored all other metrics. It is just that our performance exit criteria for the release were based on whether or not we met our elapsed time goals for the set of scenarios we had identified. The performance scenarios and goals themselves were defined based on feedback from customers, the real world user environment and extensive meetings with the members of the product team that owned designing and implementing those specific scenarios.

Some examples of scenarios and associated metrics that we decided to target are:

- Time it takes for a user to be signed in from the time the user clicks the “Sign In” button.
- Time it takes for a call to be connected from the time the user accepts an incoming call.
- Time it takes for a desktop sharing session to be setup from the time that the user clicks the “Start Sharing” button.

In addition to defining the scenarios and associated goals, we also defined a range of user models which covered things like what the hardware specifications should be for client machine used to run these tests, what software environment these machines had (operating system version, 32-bit vs. 64-bit etc.), and external considerations such as the kind of network connection (corporate intranet, broadband DSL/Cable, dial-up modem) etc. Once again, to optimize the cost benefit ratio of our effort, we decided to focus all our tests on a very specific user model (user on specific kind of laptop, connected to the corporate intranet on high-speed Ethernet etc.). This also served the purpose of defining a consistent environment in which we would run our tests and eliminate variability in performance results due to these factors. We still ran performance tests for other user models as part of sanity testing but the majority of our tests were run using the identified target user model.

2.2.2. Product Source Code Instrumentation

Once the scenarios were defined and appropriate goals set for them, the product source code was instrumented using the Event Tracing for Windows (ETW) Toolkit[2]. The APIs were used to insert ETW

[2] A system that provides application programmers the ability to start and stop event tracing sessions, instrument an application to provide trace events, and consume trace events. http://msdn.microsoft.com/en-us/library/bb968803(v=VS.85).aspx

events in appropriate locations for each of the target scenarios. For example, instrumenting the SignIn scenario meant inserting code (using the ETW API) to:

- Fire an ETW event called SignInStarted in the very beginning of the method that handled the SignIn button click.

- Fire an ETW event called SignInCompleted at the very end of the method that updated the UI to indicate to the user that she was now signed in.

When this code then executes, the ETW system fires an event that contains in its payload, a very precise timestamp for when the code was executed. The event payloads can also include state information specific to the event. This user-level payload is optional but can be used to uniquely identify events based on this data. The ETW system is highly optimized and therefore the instrumentation itself adds no overhead to the performance of the product. In fact, if there are no active ETW consumers listening for these events (the normal situation on a customer's machine) then the ETW system does nothing and no event is fired.

2.2.3. Test Design and Execution

This involved driving one or more Lync clients to exercise the performance scenario. We started by running these tests manually while building up the capability to run these tests in an automated fashion. First, we needed to create a mechanism to automate UI actions on an individual Lync client. Once we had built this, we then needed to design a distributed test harness that could enable us to run UI actions on multiple Lync clients in a synchronized manner. We next had to make a decision on the number of times to run each scenario. Ideally we would have liked to run each scenario 20 to 30 times to make sure summary statistics such as averages and standard deviations were meaningful. On the other hand we had to consider the fact that we needed the entire set of tests to run within a reasonable period of time. After careful deliberation and some experimentation we decided to run each scenario 5 times. Each time a scenario was run, the associated ETW events would fire because the performance test harness explicitly setup an ETW consumer to listen for these events. For example, the UI automation for the SignIn scenario would click the SignIn button and wait for the client to be signed in, which in turn would cause the associated ETW events to be fired and saved in a log file. At this point the log file essentially contained a list of entries, each of which contained information about an event such as the timestamp at which these events were fired. In addition to the ETW log files, we also collected the product log files to aid in investigation of any problems that came up.

2.2.4. Log processing

At this stage, all scenarios had been run and relevant log files been collected. This involved parsing each of the ETW log files generated in the above step matching start and stop events for each scenario, and calculating elapsed time by computing the difference between the timestamps of the stop and start events and then dumped into a database.

2.2.5. Analysis and Reporting

Summary views of this data were then exposed through a Silverlight[3] based interactive reporting portal so that stakeholders could easily view this information and stay informed on the overall status of the Lync client as it pertained to performance scenarios and associated goals. The engineering team would use this reporting portal to quickly identify scenarios whose average elapsed times were above stated goals for the latest set of test runs and start a thorough investigation into the problem. This investigation involved collecting and poring through associated log files from all the components to try and understand

[3] Silverlight is a Microsoft development platform for creating engaging user experiences. http://www.silverlight.net/

where the delays appeared to be occurring and why. Once we had narrowed down the cause via investigation of the logs, the normal test process would then be followed. The test team would file a bug to track the issue, assign to the appropriate developer, who would then design and implement a fix, and assign it back to the tester. Finally the tester would verify that the problem had been fixed in subsequent test runs and close the associated bug.

3. Overview of the problem

Since our primary goal was to track the overall response time faced by the user in a realistic environment, we had made the choice early on to run these tests against a topology that was actively used by members of the Lync team for their day-to-day work. Prior to release of the product externally, this was the best way for us to get performance data in a realistic setting. These tests were run approximately once a week on the latest build and the data from these tests was used to get a sense of how that build fared with respect to meeting the goals set for each scenario. As described earlier, each week we would go through the report from the most recent test run and identify scenarios that were not meeting the target goals and start investigations into each one of these.

Over time, we noticed that a large percentage of these investigations led to dead ends because there was nothing obvious about what the cause was. To make things worse, every once in a while, a scenario that was previously above its goal would automatically get back within its goal in subsequent test runs without any relevant code changes having been made in the interim. Gradually, it became clear that there were unexplained variations in the elapsed time results for a given scenario across test runs and this was making it difficult to distinguish real performance degradation due to code changes introduced in a new build from random variation in the results. While the causes of these were unknown, it was clear that we could no longer use our process to detect small performance degradations reliably. We were forced to ignore any small performance losses that the data was indicating because it took a non-trivial amount of developer/test engineering effort to investigate every such issue and frequently these investigations could not pinpoint any specific cause. In each such case, we were then left with no choice but to conclude that the performance degradations were transient issues caused by noise in the system and not due to a real issue in the product. Essentially, we were spending critical resources in wild goose chases while at the same time management was slowly starting to lose trust in the ability of our performance engineering system to accurately evaluate performance of the Lync client and to find real issues in the product.

Due to the time and resource crunch during the ship cycle, we could not afford to spin up a separate process to get to the bottom of the issue. So we chose the next best solution, which was to make the decision to consciously focus only on obviously large degradations that stood out. The good news was we managed to ship a high quality Lync with this process but we knew we needed a better mechanism going forward.

4. Our Solution- The Perf Cell

After Lync was released to manufacturing, we conducted many post-mortem and brainstorming sessions to discuss how to improve the overall performance test process with special emphasis on identifying and eliminating sources of variability in our engineering system. Over these sessions, one key issue soon became clear. We were attempting to achieve inherently incongruent goals by running a single set of tests to determine both that:

a) There were no performance degradations since the last tested build;
b) The product was meeting goals against real world topologies.

We had attempted this kill-two-birds-with-one-stone strategy by running all our tests against a real-world topology which was a reasonable solution for the latter but a terrible solution for the former because of the inherent variability in a real-world system.

We also realized that detecting and fixing performance degradations as soon as they were introduced was the best way to ensure that the product would meet its goals at the end of the product cycle. This was crucial since performance is lost not by a small number of large losses (which are usually easy to detect and fix) but by a large number of small losses that build up over time (which are usually difficult to detect and also harder to fix). Once we had internalized this, a couple more brainstorm sessions helped us arrive at the innovative concept of what we call a Perf Cell.

A Perf Cell is essentially a self-contained virtual test topology that contains all the necessary server and client components, all running as virtual machines (using Hyper-V[4]) on a single physical machine.

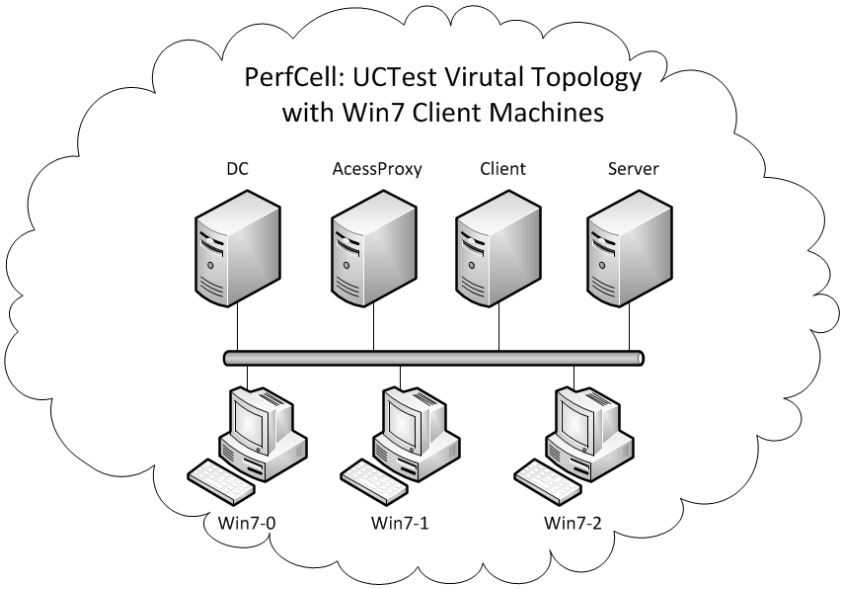

Figure 1: Perf Cell Virtual Topology Diagram

Given a physical machine that meets the necessary minimum requirements (Memory >=12GB), the Perf Cell topology is generated automatically by a script that sets up the whole system starting with bare metal. The test network is completely isolated from other networks and is completely self-contained since all the necessary server components are available on it. On average, with this script in hand the deployment time for building a fresh Perf Cell from scratch was about 4 hours and once we had the basic Perf Cell deployed, updating existing components in place was much faster. So overall, we were already seeing advantages in terms of the time and resources it took to setup an entire performance test lab.

This design clearly provides the following key benefits:

- **Eliminate extraneous network traffic**: As noted before, all clients and server components in the Perf Cell were connected via a virtual network which was disconnected from the rest of the network. This avoids traffic on external network resources and helps minimizing network variability.
- **Ensure consistent load on the server**: This was automatic because the only load on these server components was from the test clients we were actually running.
- **Easy management of the test system:** Instead of having to maintain a network of individual machines and the network components, we really only needed a single physical machine with appropriate hardware specs and our script which could be completely automated.

[4] Hyper-V is the virtualization technology provided by Microsoft. http://www.microsoft.com/hyper-v-server/en/us/default.aspx

- **Cheap and fast deployment:** With the deployment scripts mentioned before, we could roll out a self-contained performance test environment in less than half a day using automated scripts and for the cost of one server class physical machine. This is cheaper when compared to the process of setting up a more realistic topology with many physical machines, both in terms of cost of machines and then time to deploy them. In addition to the physical machines, there are also networking issues one has to deal with to get this setup running. These benefits are possible thanks to the magic of virtualization and Hyper-V.
- **Scale Out:** This is a corollary of the above benefit. Since the only constraint is the availability of physical machines, the tests can be partitioned and run in parallel (limited only by budget available for paying for additional host machines[5]) to speed up overall execution time. In practice this meant that instead of running tests once a week against a build from a specific branch, we could run these tests daily if we wanted against a build from each branch.

5. Design of the Data Collection/Analysis/Reporting system

Once we had designed the Perf Cell, we designed and implemented a lightweight performance test harness that we could easily use to run our multi-client UI automation tests on the virtual client machines in the Perf Cell. The harness is based on Windows Communication Foundation (WCF)[6] and allows the multiple virtual machines to be remotely controlled by a process that runs on the physical machine that hosts these virtual machines. The controller process on the host machine runs a set of scripts that co-ordinate tests across all the virtual client machines hosted on that machine (win7-0, win7-1 and win7-2 in Fig 1 above). It is worth noting that a single real machine exists and besides hosting the virtual machines, it performs a double duty by also executing the controller process.

For a given scenario that needs to be exercised, the controller process launches the corresponding set of tests on each virtual machine. Each test then executes independently on its virtual machine and drives the Lync UI on that machine as needed. Synchronization between tests running on independent machines is achieved by a lightweight synchronization mechanism based on a Publish-Subscribe (1) model. Test cases on individual virtual machines are able to post messages with the controller process running on the host machine indicating that they are done executing some significant part of the test. Similarly they can indicate to the controller process that they are waiting for messages from other machines and block test execution until the controller passes on the expected message from the expected source endpoint. This mechanism ensures that every test process is autonomous and could work independently until the point where it needs to sync with the controller or with other test processes. This approach is easy to manage and the main controller only needs to start the broader test scenario and does not need to control every single step of test execution. A brief overview of the system is provided below:

1. The controller script running on the host machine reads a configuration file and launches a test case process on each client virtual machine. It passes some parameters so the actual test case binary knows which role it is playing, which test account to use, the logging directory to be used and other parameters. The controller script is also parameterized, so it is flexible enough to execute just a subset of test cases.
2. The test case process is responsible for making sure that appropriate performance counters are collected during the test execution. The system is flexible enough that any sort of data may be

[5] A host machine is the physical computer that is running one or more instances of a virtual machine.

[6] WCF is a set of Microsoft technologies that enable development of distributed systems. http://msdn.microsoft.com/en-us/netframework/aa663324

collected, such as ETW traces, system information (like CPU, memory, IO, Registry operations, etc.) or any other custom performance indicator the product may have. During the test execution, log files are created and performance results are written to the log location.

3. A completely separate log analysis process is responsible for analyzing the performance data. This approach fully decouples the test execution from the performance analysis and reporting. It also allows a simple extensible model where a handler on the analysis side can be easily developed to parse the results for any new set of performance data collected by the tests. As an example, we may have an analyzer which deals with ETW traces, another one dealing with memory dumps or an analyzer which parses custom data the test wrote during test execution.
4. The performance infrastructure is structured in a way that multiple test machines can run in parallel, writing their test results to a log store. The log analysis process checks for new data in the log store in real time and analyses new data as soon as it is available. The design of the system allows for multiple log processors to run in parallel and thus enable quicker analysis of test results.
5. The end results generated by the analyzers are stored in a database and can be accessed through a rich reporting portal built using Silverlight technology which minimizes the time it takes for new visualizations to be shown to the user. Once a set of data is retrieved, the user is able to fully interact with it until new data needs to be retrieved from the back end. The performance of the backend database was also highly considered and we currently have over a million data points being served with high performance, where we are able to filter data and show results in a few seconds.

6. Initial Results

Our preliminary results from analyzing performance data for a test run against a Perf Cell are very promising. The variation in the results appears to be drastically lower compared to variation in the results from a run against a real-world topology.

The following two charts illustrate the difference between elapsed times for scenario A when tested against a real topology and a virtual topology. The charts are essentially histograms showing the frequency distribution of elapsed time measurements for this scenario. The elapsed times for each time are slotted into a bin and the height of the bin indicates the number of elapsed times that fall in that bin (indicated by the label Measurement Count on the vertical axis) and the x-axis simply denotes the entire range of elapsed times that were noticed).

Figure 2 shows elapsed time data for tests run against a real topology and one can see that the distribution appears to be approximately log-normal (2) with a long tail (the overall range of times is broad and there is no clear clustering of data, making it hard to judge the performance for this scenario). Since there is a risk that the logs might show no clear cause for why certain test iterations show higher elapsed times, we would have to assume that this was just caused by noise in the system and skip an investigation altogether in the interests of saving engineering resources. By making this choice, we automatically run the risk of letting some real degradation get into the system. The only situation in which we would notice a real degradation is when the entire histogram was shifted to the right and there was obviously something very wrong with the performance of the scenario. That is when we would decide it was worth spending the time to investigate the problem because of the high probability of finding a real issue.

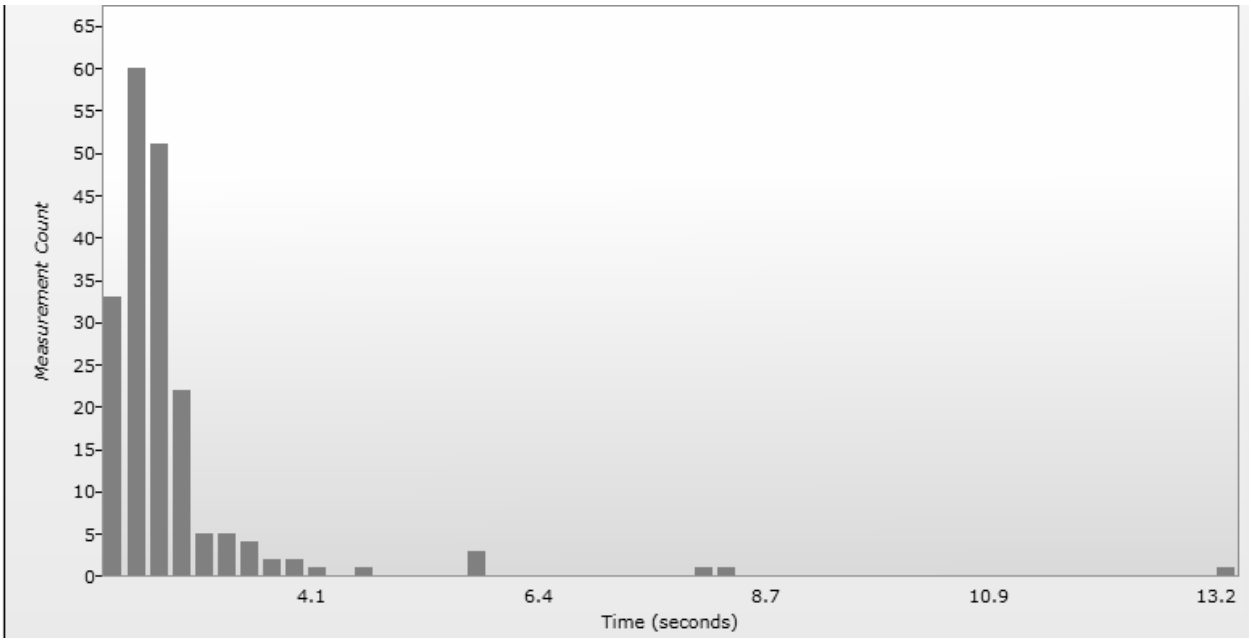

Figure 2: Histogram showing frequency distribution of scenario A's elapsed time results from performance tests run against a real topology

In contrast, in Figure 3, where a Perf Cell was used, one can see two tight clusters. The first cluster contains more than 90% of the data in the 0.695 to 0.997 range which indicates very low variability in the results, thus validating the overall design of the Perf Cell concept. We do see another cluster (approximately 6%) in the 15.57 to 15.87 range. This is a classic error condition where a given operation succeeds only after one or more retries and therefore the elapsed times are much longer. The fact that even the outliers are tightly clustered is further evidence that use of the Perf Cell is successfully eliminating most of the inherent variability in the system and increasing the power of the tests to notice even small degradations in the performance of the software. In such a system, we no longer have to make a choice to not investigate small degradations. Due to the low level of noise, we can be reasonably confident that even small degradations indicated by the data are real issues and it is worth our time getting to the bottom of the issue. This in turn means that there is a much lower chance of letting real performance degradations fall through the cracks and get into the product.

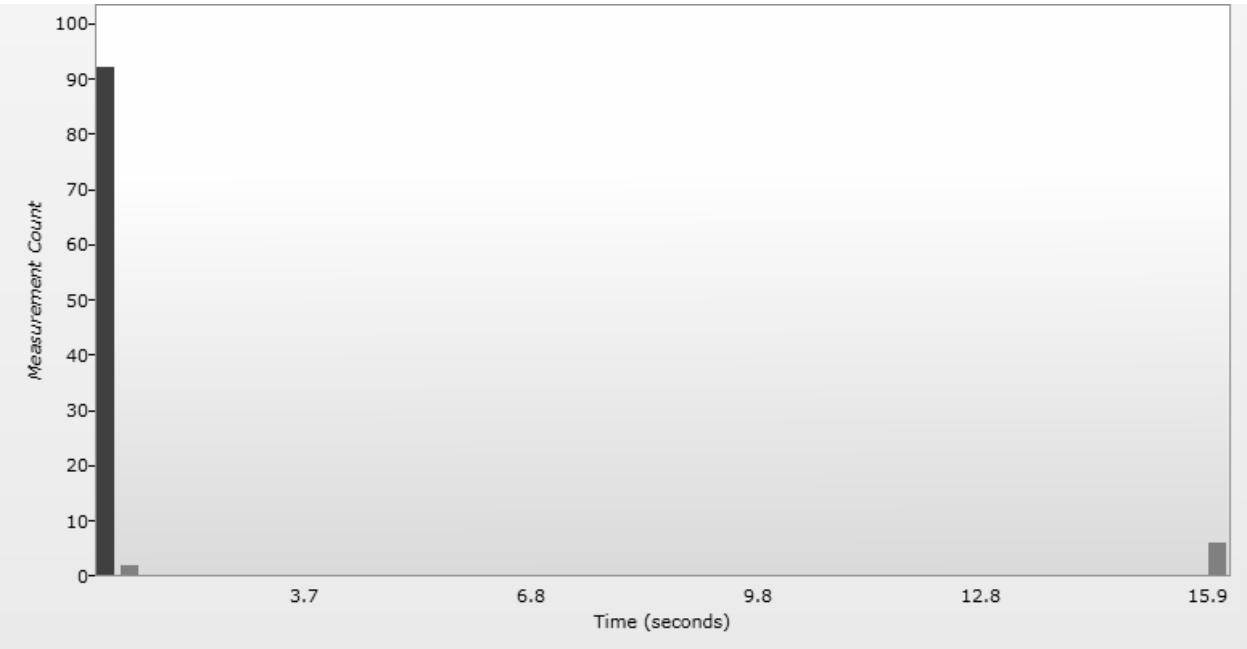

Figure 3: Histogram showing frequency distribution of scenario A's elapsed time results from performance tests run against a Perf Cell

As we can see from the above figures, the elapsed times that we observed from tests run against the Perf Cell exhibit much less variation enabling us to more clearly see real aberrations that merit further investigation. This helps us to increase the sensitivity of our tests in catching even relatively small performance degradations and help keep the product quality high throughout the cycle.

7. Conclusion

Hopefully the reader now has a general understanding of the things to consider when designing a performance engineering system. In our opinion, the most important takeaway is that the system needs to be engineered very carefully to eliminate as many sources of noise i.e. test interference as possible. As always, this needs to be balanced by the needs of the team and the resources available to contribute to this system.

Based on the results we have seen thus far, we believe the Perf Cell concept is perfect for running performance tests that are sensitive enough to catch even relatively small degradations in performance across different builds of a software product. We believe that using a Perf-cell based design is especially effective for testing distributed systems because:

- It eliminates extraneous network traffic from the system (all traffic between client components and server components is on the virtual network which is isolated from the rest of the world) which is frequently a cause for transient performance degradations that are not associated with any changes in the product.

- It greatly reduces (if not outright eliminates) variability caused by varying response times from the server during the test since the only traffic to the server components is the traffic sent by the clients actually in the test and the server is always in a known steady state.

- It allows for a relative efficient way to scale out the test effort by partitioning the tests into natural groups (for example, voice call tests, video call tests, instant messaging tests etc.) that can all be run in parallel on their own Perf Cells. With the increasing availability of computing resources in

the cloud, this easily lends itself to the possibility of spinning up Perf Cells in the cloud on demand for the duration of a test run and tearing them down after the run.

- It allows for relatively easy modifications to yield more improvements. One improvement that we are considering is the ability to test two adjacent builds simultaneously instead of over separate days. Instead of running tests against say, build A on Mon and build A+1 on Tue, we could increase the number of clients available in the perf cell (by simply modifying the script appropriately) and test build A and build A+1 simultaneously. This will allow us to further reduce noise by elimination noise caused by temporal factors (e.g. the state of the system on Mon at the time of the test is likely to be slightly different from the state of the system on Tue and by running tests against the builds from Mon and Tue simultaneously on Tue, we eliminate variability in results caused by this difference). By doing this, we expect to eliminate temporal variability and further increase the power of our performance tests to detect the smallest of performance degradations.

References

1. Publish/subscribe. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Publish/subscribe.

2. Lognormal Distribution . *NIST/SEMATECH e-Handbook of Statistical Methods.* [Online] http://www.itl.nist.gov/div898/handbook/eda/section3/eda3669.htm.