DeskBench: Flexible Virtual Desktop Benchmarking Toolkit

Junghwan Rhee
Department of Computer Science
Purdue University
West Lafayette, IN, 47907, USA
rhee@purdue.edu

Andrzej Kochut, Kirk Beaty
IBM T.J. Watson Research Center
19 Skyline Drive
Hawthorne, NY, 10532, USA
{akochut,kirkbeaty}@us.ibm.com

Abstract—**The thin-client computing model has been recently regaining popularity in a new form known as the virtual desktop. That is where the desktop is hosted on a virtualized platform. Even though the interest in this computing paradigm is broad there are relatively few tools and methods for benchmarking virtual client infrastructures. We believe that developing such tools and approaches is crucial for the future success of virtual client deployments and also for objective evaluation of existing and new algorithms, communication protocols, and technologies.**

We present *DeskBench*, a virtual desktop benchmarking tool, that allows for fast and easy creation of benchmarks by simple recording of the user's activity. It also allows for replaying the recorded actions in a synchronized manner at maximum possible speeds without compromising the correctness of the replay. The proposed approach relies only on the basic primitives of mouse and keyboard events as well as screen region updates which are common in window manager systems. We have implemented a prototype of the system and also conducted a series of experiments measuring responsiveness of virtual machine based desktops under various load conditions and network latencies. The experiments illustrate the flexibility and accuracy of the proposed method and also give some interesting insights into the scalability of virtual machine based desktops.

I. Introduction

The way users interact with applications has gone through several phases of development over the last few decades. When computing systems first adopted interactive user input/output devices the users interacted with their applications using text terminals connected to central computers, a good example being the IBM 3270 text terminal [5]. An advent and popularization of personal computers in the early 1980s marked the beginning of desktop computing as we know it today. In this contemporary model the execution of the operating system and applications happens on the end user device itself. The recent years, however, have witnessed reinvigorated interest in the centralization of end user computing. Technologies such as Citrix [4], Windows Terminal Services [8], or most recently virtual machine technology such as Kernel Virtual Machines [1], Xen [19], or VMWare ESX [10] are on the fast path to change the desktop computing landscape back to the model of a terminal connected to a central server. Figure 1 illustrates the virtual desktop model. End-user devices consist of a monitor, keyboard, mouse, and disk-less "thin-client" computer. The execution of the desktop operating system as well as applications takes place on remote servers, either within dedicated virtual machines or shared services sessions. This recent transition has been enabled by several developments of underlying technologies. The first one is ubiquitous high-speed networking access that allows effective responsiveness even at significant distances. The throughput of connections is high with fiber-optic cabling for even the "last-mile" connections to private homes and small offices. Even though basic propagation latency will remain an obstacle for long distance desktop usage the connections of up to a few hundred miles are sufficiently responsive to allow for a smooth working experience. The second important advancement is the virtualization technology. It decouples users from physical resources thus making the management of the infrastructure much more flexible and arguably more cost-effective. Virtualization can be applied at both the application level (e.g., [12]) and the operating system level (e.g., [1], [19], [10]). The latter makes it possible to run multiple instances of desktop operating system on a single physical server while maintaining full isolation and security.

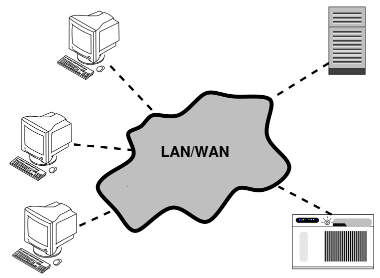


Fig. 1. Overview of virtual desktop architecture. Users access their desktops and applications using "thin-client" devices which relay only the keyboard and mouse events and screen updates over the local or wide-area network.

Desktop virtualization, while offering many advantages, is still a new and emerging technology with many research challenges. One such challenge is the efficient and flexible benchmarking of virtual client systems. This benchmarking is crucial for capacity planning of virtual desktop deployments, testing and comparing the performance of hardware and software components, and also for validating and evaluating algorithms and protocols. There are a large number of established benchmarks both for servers and traditional desktops. The prominent example is the suite of database and transaction processing benchmarks defined by Transaction Processing Performance Council [9]. It contains a specified set of applications, database schema, data and programs that can be used to exercise the system at its full capacity and at the same time precisely measure transaction throughput and latency.

We believe it is important to develop a similar level of maturity in testing and performance benchmarking of virtual desktop systems. However, it is a much more challenging task than traditional desktop or server benchmarking. There are three main reasons why this is the case.

First, replaying a sequence of user actions at a high speed (which is required to test the systems performance under high load conditions) is difficult. Before replaying the next action in a sequence the benchmarking program has to make sure that the effects of the prior action have been properly reflected so that the state of the system is correct for invocation of the next action. An example of this situation is a benchmark consisting of three actions: opening an editor window, typing a letter in this window, and closing the window. Observe, that replaying the action of typing a letter can be initiated only after the window is fully opened. If it is not the case, not only will the measured response time of the action be incorrectly computed but also the benchmark playback can enter an inconsistent state with the keystroke representing the letter being directed to the desktop rather than the editor window. This situation gets worse when the system operates under high load conditions because the delay between the completion of actions can stretch significantly. Yet this is exactly the scenario which we want to benchmark.

Second, virtual workstation performance is considerably more difficult to define than that of a database server, application server, or traditional desktop. This is because we are interested not only in the timing of events, but also in the quality of the display updates that the user perceives. Some remote access protocols, such as VNC [18], RDP [6], or ICA [17], may drop screen updates that do not arrive in time to be played back, thus even though the timing of the action's execution is acceptable, the resulting output quality is not. Similar problems affect the audio quality.

Third, most protocols used by virtual workstations to communicate with the end-user devices are either proprietary or at best open but with very little documentation. Thus engineering a benchmarking system that is applicable to a variety of protocols and applications is a challenge.

We present *DeskBench*, a flexible and accurate desktop benchmarking system capable of replaying and timing previously recorded user actions. The novelty lies in using only the basic primitives of keyboard and mouse events and frame buffer region updates to record and time the replay of user actions with arbitrary speed. The tool works in two phases: recording of user actions and replaying them against a test system. The recording of user actions stores the keyboard and mouse events together with inter-arrival time information. Moreover, at important points of the execution which are required to synchronize the replay (such as the moment of opening the editor window in the example described above) the user recording the benchmark can mark the current screen state as a synchronization element. The synchronization element represents the state of the desktop's screen image at that instance. During the replaying phase the keyboard and mouse events are sent to the desktop application. In order to deal with the replay synchronization problem the subsequent mouse and keyboard events are appropriately delayed until the state of the screen reaches the expected synchronization point. The decision whether the screen is in an expected state can be either fully deterministic or "fuzzy" based on the image similarity metrics we have developed. We believe that *DeskBench* has the following advantages over the other approaches described in the literature:

- The recording of user actions is made easy by having the user simply perform the actions he would normally do and only marking (with a simple keypress) the synchronization points at instances when the replaying sequence needs to be (potentially) delayed to maintain synchronization.
- Replaying can be done at arbitrary speeds because of the self-synchronizing mechanism being based on actual screen state. Thus the system can be used not only to measure the responsiveness of an almost idle system but also to generate high load. Further, the responsiveness measurements are accurate and account for the total time between the first request and the last screen element update.
- During the recording and replaying of the user actions we only use basic window system primitives (keyboard, mouse, and screen updates) thus *DeskBench* can be implemented at the windows manager level with no application-level modifications. This makes our approach generic and applicable to wide variety of test scenarios.

Probably the closest work to ours is the *VNCPlay* [20] which captures and replays keyboard and mouse events within RealVNC player [18]. Our approach differs by allowing for more general artifacts (covering the whole screen) as well as provides for "fuzzy" matching making it more robust in adapting to not fully deterministic desktops. Moreover, *DeskBench* is application and protocol agnostic because it can be implemented within the window manager library. Another work close to ours is *Slow Motion Benchmarking* [13] which captures and replays the user actions at the network layer. It is also generic in a sense of being application agnostic. However, our approach allows replaying the actions at maximum speeds and is not restricted to slow motion. Thus *DeskBench* can be used to simulate high levels of load using the same actions that are used to time the responsiveness of the system. Moreover, it is easy to create large libraries of user actions, we refer to as artifacts, by simply recording user's activity. Broader discussion of how our work compares to other approaches (including window manager API scripting) can be found in Section IV.

Roadmap

The remainder of this paper is organized as follows: Section II presents the *DeskBench* approach and describes the prototype implementation. Section III relates the results of the evaluation of the tool as well as insights into the scalability of virtual machine based virtual desktop systems. Section IV

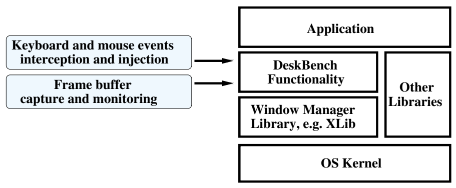

Application
Keyboard and mouse events interception and injection
Frame buffer capture and monitoring
DeskBench Functionality
Window Manager Library, e.g. XLib
Other Libraries
OS Kernel

Fig. 2. *DeskBench* architecture. The functionality of the system can be implemented within the window manager software or as an independent layer between the application and the window manager library (as a shim). The primitives that need to be intercepted and injected are common throughout all major window managers both the open source and proprietary.

discusses the related work. Finally, Section V concludes and outlines our future research plans.

II. Desktop Benchmarking Toolkit

Today there are well accepted benchmarks for producing various types of server workloads. Notably, the Transaction Processing Performance Council (TPC) [9] provides a well established suite of server workload benchmarks, including TPC-App for application server and web services, TPC-C/TPC-E for on-line transaction processing against a database, and TPC-H for ad-hoc decision support. TPC defines a set of functional requirements for each benchmark, and then vendors can use proprietary or open systems to implement the benchmark. Our research goal is to develop both the tools which allow for desktop workload benchmarks to be created and driven as well as to define standardized benchmarks for client workloads such as the definition of the categories of workloads (for example, administrative worker, and application developer). In the following sub-sections we introduce the approach and our prototype implementation of desktop benchmarking.

A. DeskBench approach

DeskBench provides recording and playback capability of client user actions, keyboard and mouse events, by intercepting then later injecting the events while monitoring the updates to the application's screen frame buffer to detect event completion. The architecture of the system and how it integrates within the software stack of the client machine is depicted in Figure 2. The lightweight software layer intercepts the function calls between the application and window manager library. In case of X Windows [16] system it is the XLib library, but the same functionality can be implemented for Microsoft Windows platform by implementing this functionality for its window manager. *DeskBench* operates in one of two modes: recording and replay.

Recording phase: During the recording phase all of the keyboard and mouse-click events that are generated by the window manager and passed to the application are recorded. Optionally, the timing of actual user think-times, i.e. the delays between actions, can be also measured and recorded. At each instance that may require enforcing the synchronization of the replay processing the user can denote this in the recorded artifact by pressing a defined key sequence (e.g. prtScr key). An example of such a synchronization point is following a "double-click" event to open a window/application, the next action has to be delayed until the screen fully refreshes. In this case the tool user signals the synchronization point after the screen is fully refreshed. This way during the replay phase *DeskBench* will delay the issuing of the next action until the screen reaches the proper state. A synchronization point represents a screen state that is the logical end of a set of events that either is a necessary point to reach before proceeding with subsequent actions, or is a point that the tool user wants to mark for measured execution time. These are used by the play back mechanism to monitor the screen images for completion, after which play back can continue on to the next set of recorded events from the artifact. For each synchronization point, there is one or more hash codes stored with it in the artifact. The hash codes represent an MD5 hash[14] of the screen image buffer that is expected at the completion of the corresponding event element of the artifact being played. By limiting the checking of hash codes to the synchronization points in the artifacts it greatly reduces the CPU needed to process the hashes and thereby the amount of processing needed to monitor the event processing.

Replaying phase: When replaying *DeskBench* processes each event found in the given artifact file in order and injects these into the window manager. Interleaved with event injection is monitoring of the returned screen updates so as to be able to detect event completion at each of the recorded synchronization points. This playback process is depicted in Figure 3. The horizontal line represents time. Short vertical black dashed lines represent requests sent from the client to the desktop running on the server. Short vertical dash-dotted lines (in red) represent screen updates arriving from the server. High vertical solid black lines represent synchronization points when the observed screen states are compared with the expected one until a match occurs, signaling event completion. Assume that the prior artifact finished at time t_0 (which is the time instance of the arrival of the last update associated with that artifact). Periodic screen checking occurs at t_1. The procedure used at the synchronization point to determine event completion is to first compare the hash code of the the observed screen with those defined in the artifact for the current synchronization point. If there is no match, the "fuzzy" comparison initiates and proceeds as described in the next paragraph. When a match is detected, the system delays the execution for the duration representing the configured user think time (equal to $t_2 - t_0$ in Figure 3). Observe that since some time have already elapsed, due to the lag between the arrival of the last response of the prior artifact and the recognition of the synchronization point (t_1). Thus the actual wait time since t_1 is the desired think time reduced by $t_1 - t_0$. After the think time elapses the next action of the artifact is executed at t_2. It is followed by all actions that do not require synchronization as well as responses arriving from the server. After the last event prior to a synchronization point is played, several checks of the screen state can occur but result in a non-match, therefore the system keeps waiting for the arrival of the proper screen update. Finally, the last required response arrives from the server at t_3 and is identified by the system at t_4. If the

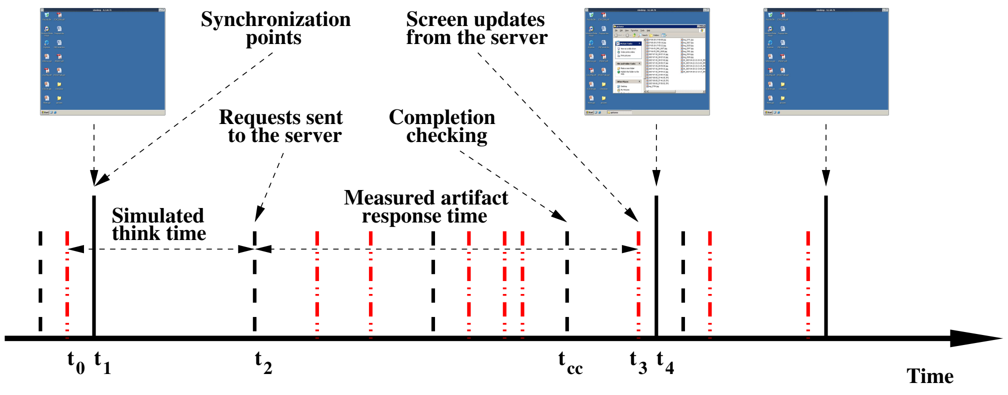


Fig. 3. *DeskBench* replaying process. The artifact starts at t_2 after the required think time. Throughout the period between t_2 and t_3 keyboard and mouse requests are sent to the server (black vertical dashed lines) and the screen updates arrive from the server (vertical red dot-dashed lines). After sending last request (at t_{cc}) system starts checking incoming updates. The final update arrives at t_3 and is recognized by the system at t_4. At this instance *DeskBench*, computes the artifact's response time ($t_3 - t_2$), and initiates new think time.

match is found the response time is computed as $t_3 - t_2$ and the process repeats for the next artifact. In the event that the system waits too long and no proper screen can be matched (either exactly or via the "fuzzy" matching) *DeskBench* stops replaying and reports an out of synchronization exception.

$$\begin{aligned} \frac{Delta(A_{Expected}, A_{Obtained})}{Area(A)} &\leq T_{Expected} \\ \frac{Area(B_{Obtained})}{Area(Screen)} &\leq T_{Unexpected} \end{aligned} \qquad (1)$$

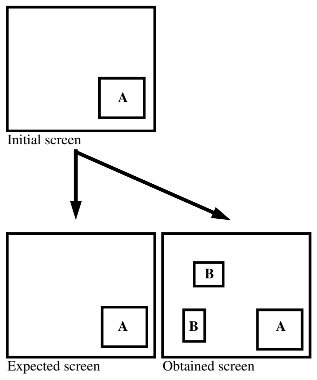
"Fuzzy" matching of screen states: An initial version of *DeskBench* required an exact match of the hash code of the current screen image to that of the hash code stored at the synchronization point of the recorded artifact. This worked adequately but required more controlled environments to run properly. For instance the smallest change of any pixels in the screen image, such as a scroll bar being even a slight bit different in length would cause the driver to not find a match and the playback would be halted. This compelled us to invent novel methods of "fuzzy" hash code comparisons that have proven beneficial by providing the desired improved robustness. The current version of the DeskBench prototype detects small differences in regions outside of the target area as non-essential in making a match, as well as allowing some small percentage of change even within the screen region of interest. This greatly increases the hit ratio for matches while maintaining high accuracy in the proper recognition of event completion. As new screen image hash codes are encountered which are not exact matches, yet considered "fuzzy" matches to those listed for the corresponding synchronization point, they are automatically added to the list of matching hash codes and recorded in the artifact file. This increases the chance for exact matches and thereby increases robustness for future replays of the same artifact.

Figure 4a illustrates this process. The changes that we expect after execution of an artifact are denoted as *A*. The figure shows them as a rectangular area, but of course in general they do not have to be shaped like this (the algorithm uses just the number of pixels changed and not the shape of the region). Additional changes that we did not expect are denoted by *B*. In such a setting the screens are deemed as matching if both inequalities are true 1, where $T_{Expected}$ and $T_{Unexpected}$ are the thresholds for making the decision and represent a fraction of pixels that is allowed to differ in the expected change region and the fraction of pixels that are allowed to differ in the remaining part of the screen, respectively. We have experimented with setting the values of these variables and comment on it in Section III. $Delta(A, B)$ for corresponding pixel sets A and B is defined as the number of pixels differing in those sets.

An example of this process is depicted in Figure 4b. The initial screen is a desktop and the expected one after the execution of a synchronized action has a windows explorer window opened and no items highlighted. The screen obtained has the window opened but in addition a file is highlighted and the icon on the desktop was moved. The system detects this fact and evaluates against the thresholds defined with Inequalities 1, determining that the screens differ too much thus aborting further execution.

Prior to our introducing "fuzzy" logic to better detect event completion, we found that small screen differences would occasionally get our replay out of synchronization. And even with the "fuzzy" logic, if the differences are large enough, this can still occur. As an aid when we first were developing the tool, we built a utility to assist in the detection of what might be only a few pixels difference between what was expected and what was received. As *DeskBench* records it saves screen images for the state of each synchronization point. Thus the utility does a pixel-level comparison of the expected screen image with the screen image of what was received and produces an image of just the areas with differences. This can then be easily viewed to find where to focus in determining what has changed, and what needs to be corrected in the recorded artifact. As stated, if the differences are small, the current enhanced "fuzzy" matching will handle these differences automatically.

Optimization of CPU consumption during the replaying process: The crucial component of CPU processing is computing the screen hash codes as the screen updates arrive. There



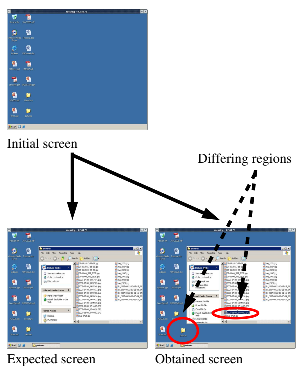


Fig. 4. Concept of "fuzzy" comparison of screens (a). An expected area to change after the execution of an artifact is denoted by *A*. Additional areas outside of *A* that have changed are denoted by B. An example of "fuzzy" screen comparison (b).

are several ways of optimizing this process. One optimization that we have implemented is limit the completion checking until the last event of a synchronization set is injected (this is the black dashed line at t_{cc} in Figure 3), which greatly reduces the amount of CPU cycles needed to perform the hash code matching necessary.

Another implemented optimization is to compute screen hashes not at each screen update received but at predefined intervals. The interval can not be too large because that would slow down the initiation of next artifact, that is the lag time after the last response arrives from the server (in our example shown in Figure 3 these lag times are $t_1 - t_0$ and $t_4 - t_3$). In our prototype (which is described in Subsection II-D) we used the interval value of 250ms which we found sufficiently large to result in very low CPU utilization and also responsive enough so the extra delay between artifacts is small.

Another possible optimization is to subdivide the screen into subregions and then compute and store hash codes for each of them separately. This way the hash code computation can be done on each arrival of screen updates (because it affects only a small part of the screen).

B. Measuring the screen update quality

Multiple applications, especially the ones rendering the video frames with high frequency, can finish the processing in time but with poor quality to the user. The reason is that most of the screen updates sent by the application running on the server will get discarded because of the lack of resources (either too slow of a network connection or overcommitment of the server CPU). We want to detect and quantify this deterioration in quality. In order to accomplish this the measurement of the number of changes of a given screen region can be computed during the replay of a given artifact on a lightly loaded server with very low latency and a high bandwidth network connection to the client. This value constitutes a comparison baseline. Later during the replay of the artifact under normal conditions the number of region refreshes can be compared with that baseline to get an objective measurement. A similar approach was applied to the volume of network traffic associated with the artifact in slow motion benchmarking [13].

C. Required preparation for DeskBench execution

It would be remiss to not point out that a fairly deterministic state of the desktop is required to be maintained from recording to playback. For instance to maintain a common desktop with icons arranged in the same order, we use roaming profiles for Microsoft Windows clients so that the same user id running on different hosts will appear the same. Further we turn off attributes of the graphical user interface which would cause differences in the screen images, examples include removing the clock from the task bar and turning off the blinking cursor. With the enhancements to use fuzzy matching and support for multiple hash codes many of these non-deterministic aspects of the user desktop can be properly handled without special preparations.

Options for batch testing: DeskBench makes it easy to create and save recorded artifacts that can then be individually re-played or included in a playlist for automated, scripted playback. Configuration settings allow for artifacts named in playlists to be run in random or deterministic order, to be run once or repetitively, and to have user think-time delays included between execution of the elements of the artifacts. Specifically think-time delays can be set to be fixed, exponentially random, or to the actual timing from that of the recording. *DeskBench* runs with low execution overhead thus permitting many simulated clients to be ran at the same time from a single machine. This is a key advantage as it permits us to realize our goal of efficiently driving realistic client workloads without unwanted effects from the execution of the driver program itself.

Results of a DeskBench *run:* There are two outputs of DeskBench that are of interest: automated execution of user desktop workloads resulting in loading the tested system, and the measurement of execution times for the desktop events of interest. For the latter, when DeskBench plays back artifacts it captures precise timing for each of the groups of events represented by a synchronization point, and reports these for use in subsequent performance analysis.

D. Prototype Implementation

As a proof of concept, we have developed a prototype implementation of *DeskBench*. We have implemented it with the XLib library in Linux OS and are using the "rdesktop" program as a test application. "rdesktop" is an open-source client for Windows NT Terminal Server and Windows 2000/2003 Terminal Services, which is capable of natively speaking Remote Desktop Protocol (RDP) [6] and currently runs on most UNIX based platforms using the X Windows System to present the client Windows desktop [2]. Because of that our prototype may not yet be fully generic since it was not tested with other applications that may potentially exercise XLib functions we did not handle.

In case of the XLib library the following calls are responsible for the basic windowing operations that have to be modified and intercepted/injected:

- XNextEvent() which is used by applications to process the windowing events such as keyboard, mouse, and window visibility notifications, etc.
- XPutImage() that applications use to update a screen region
- XCopyArea() used for retrieving data from the cache

However, as discussed in the prior section the method has no inherent dependencies on any application specific constructs or metrics and can be applied to other operating systems and window managers too.

III. Performance Studies

In order to illustrate the flexibility and applicability of DeskBench to performance evaluation of virtual workstations we have conducted a series of experiments on our testbed. The artifacts used in the experiments represent a subset of actions of a personal computer user. However, they are not exhaustive and the results presented should not be treated as definitive capacity estimates for a given user class but rather as illustrations of applicability and flexibility of our benchmarking tool.

The experiments are designed to test two aspects of desktop virtualization: (1) scalability of virtual machine hypervisors to the number of concurrent sessions and the intensity of workload applied, and (2) sensitivity of the virtual desktop to network latencies. The first aspect is crucial for capacity planning of back-end hosting resources. Good understanding of actual responsiveness as perceived by the end-user is the key metric that needs to be used in deciding what levels of concurrency (number of desktop virtual machines per physical server) should be used. The second aspect, impact of network latency, lets us reason about the distances between the server hosting the virtual machine and the end-user's device so as to not noticeably deteriorate the user's experience.

A. Experimental setup

The setup for our experiments is presented in Figure 5. We

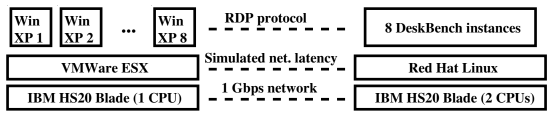


Fig. 5. Logical view of the testbed environment.

have used two IBM HS20 Blades. The first runs ESX Server (version 3.0.2) hosting virtual machines running the Microsoft Windows XP operating system. These virtual machines represent desktop workstations running within a virtualized desktop environment. This blade is equipped with 1 dual core Xeon CPU (3.2GHZ dual core). The second blade runs Red Hat Enterprise Linux AS 3 and executes the DeskBench program. This blade has 2 Xeon CPUs (each 3.2GHZ dual core). The blades are within the same blade center thus networking latency is practically zero, but for network latency experiments we have introduced simulated delay using the Linux *tc* [15] utility. The protocol used to access workstations from client programs was RDP v. 5 [6].

B. Artifacts used in the experiments

In order to demonstrate flexibility and applicability of the DeskBench tool we have created 7 artifacts representing a typical interaction of end-users with their desktops. The artifacts exercise multiple GUI functionalities as well as requiring the operating system to load files, and to utilize both the network and storage. The artifacts we have used are:

- **Acrobat:** Open an Adobe Acrobat document and browse the pages. Synchronization elements are after the opening, following each page change, and at the close of the application.
- **Folder:** Open a folder within the local file system. Highlight and unhighlight files and directories. Change directories and quit the Microsoft Windows Explorer. The synchronization elements are after each action.
- **MS Word:** Load large document with text and graphics, browsing the document and type a few words. Synchronization elements are located after each operation except of scrolling pages down.
- **Picture:** Open several pictures (with resolution 3200x2400) and zoom in on some of them. Synchronization is after opening each picture and following each of the application closing operations.
- **Powerpoint:** Open a Microsoft Powerpoint presentation file and scroll through the pages. Synchronization elements are after opening the Powerpoint and after displaying each of the pages.
- **Web browser:** Open a web browser and visit websites. Synchronization elements are after opening the application and then after loading each of the websites.

Creation of the above artifacts requires only a few minutes of work. The "fuzzy" matching of screen images provides for extra flexibility when dealing with applications that are not fully deterministic or would require significant configuration effort to make them deterministic. An example might be a web browser artifact where the website we have visited changes a little bit with each visit, (e.g. contains a visit counter and the date of last visit). With the exact matching technique this website could not be benchmarked but the threshold based extensions handle this case well.

C. Discussion of the experimental results

We have performed two groups of experiments designed to quantify the quality of a user's desktop experience. The first group was focused on measuring the responsiveness of artifacts described above with increased contention for server resources. The second group focused on quantifying the impact of network latency (and thus RDP protocol slowdown) on the responsiveness of the artifacts.

Effects of the hypervisor contention: In each round of these experiments we run DeskBench clients against the set of test virtual machines that are hosted on one HS20 Blade. The number of active sessions (i.e. virtual machines being driven) varied between 1 and 8. Each desktop session replayed the artifacts described above 20 times in random order to

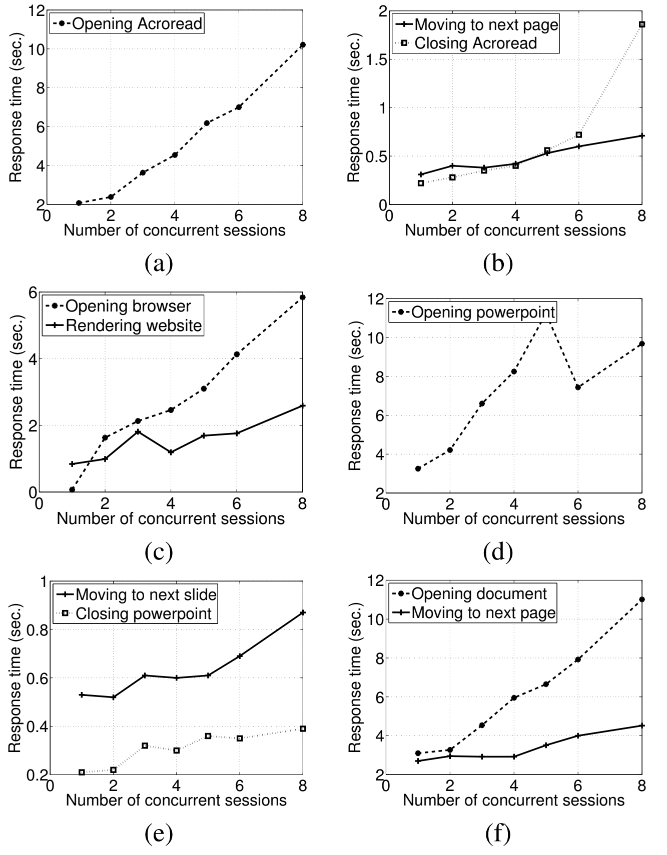


Fig. 6. Responsiveness of opening the Acrobat Reader (a), Acrobat operations (b), Internet Explorer (c), Powerpoint (d,e), and word processor (f) as a function of number of VMs on a PM.

avoid synchronization effects. Within each replay the delay after reaching each synchronization point was 100ms thus emulating tireless users. This is why we do not treat this group of experiments as indicative of realistic capacities but rather illustrative of the flexibility and accuracy of the DeskBench tool. In realistic conditions the load intensity is significantly smaller due to greater user think-times than we used in the experiments.

Representative results from the contention experiments are presented in Figure 6. Figure 6(a) shows the results for opening the acrobat document. The time stretches from around 2 seconds with one concurrent session up to more than 10 seconds with 8 concurrent sessions, thus representing a five-fold slowdown. Similar results are shown for other applications in our benchmark and illustrate applicability of *DeskBench*. Having larger sample runs would eliminate the spikes seen in some of the graphs. These are included to illustrate the capabilities of the tool with analysis of the results saved for a future publication.

Effects of the network latency: The second group of experiments focused on quantifying the impact of network latency on the user's desktop experience. In order to study it in a controlled manner we introduced simulated network delay using the *tc* tool that is part of modern Linux distributions. It allows for setting various queuing disciplines at the network interface level and also changing the key properties of the interface, such as transmission rate and added latency. In each experiment we run a single concurrent DeskBench client against a test virtual machine (as shown in Figure 5). We compared the experimental results of varying values of network latency introduced in the network interface of the server that was running the DeskBench program. We choose to sample the values between 0ms and 140ms because they represent the realistic range observed in networking applications. The low latency can be achieved in local setups when the back-end server is on the same LAN as the hosted virtual machines (as was the case for our experimental setup). The high RTT values (in the order of 100ms or more) are commonly observed when accessing computing systems located across the continent (stated with respect to the United States). The results of these experiments are using (a) the Notepad text editor and (b) a picture rendering program are presented in Figure 7. Time required to render and transmit the picture stretches almost 6 times representing a significant degradation in the user's experience. Similarly, keystroke response stretches approximately 3 times. Note that values for sample points with a 0ms RTT correspond to the experiments with no slowdown in the Ethernet adapter on the Linux server. In this case the actual RTT was on the order of 0.6ms.

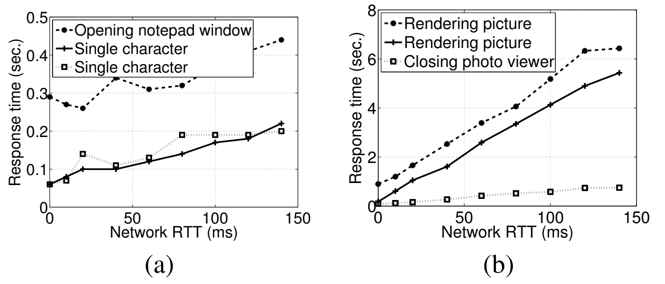


Fig. 7. Responsiveness of typing in notepad (a) and picture rendering (b) as a function of network latency.

Overhead of the benchmarking tool: While conducting the experiments we also measured the utilization of the Linux server that was running the DeskBench program. The per-session utilization of the server is about 2% which is almost identical to that of the unmodified native rdesktop client program on which our DeskBench prototype was based. In experiments with all 8 concurrent sessions the DeskBench Blade server runs at 84% idle (i.e. only 16% utilization) thus guaranteeing that the effect of any contention from driving parallel desktop clients has negligible effect on the benchmarking results. Of course experiments involving much larger numbers of client instances would require more servers running DeskBench to drive the desktop client workloads.

Choice of the "fuzzy" matching thresholds: We have experimented with the choice of two parameters for deciding whether two screens match. The adaptivity of the system increases with larger value of the parameters but may lead to unsynchronized execution. It is not a big risk though, because even if the current incorrectness is not captured it will be revealed in one of the subsequent actions which will diverge significantly to exceed even relaxed threshold values. Throughout our experimentation with the system we found that the best values of the parameters are $T_{Expected} = 0.01$ and $T_{Unexpected} = 0.001$. These values are large enough to avoid practically all “false” differences.

IV. Related Work

There are several approaches to provide remote desktop computing. The first is the shared services environment where multiple users share a single instance of the back-end operating system. In this scenario each user has a session at the server and his control over the resources are limited. He can not install or modify applications and can not modify the operating system or any libraries. Examples of such systems are Windows Terminal Server [8] and Citrix [4].

Another approach is to dedicate a virtual machine to the user. In this model the user’s experience is almost as flexible as on the “thick-client” desktop or laptop. Users can not only modify applications and settings of the system, but may also potentially install new libraries, modify the operating system itself and restart his virtual machine without interrupting the work of other users. This model is suitable for more demanding user classes such as knowledge workers and developers. Examples of such virtualization systems are Xen [19] and VMWare [10]. Another example of a remote desktop system, although very different from the one described above, is the X Window system [16]. It is different from the “thin-client” systems that we focus on because the actual execution of the window manager happens on the client machine. Thus instead of screen updates the server relies on higher level primitives being sent to the client.

There is little work on performance evaluation of virtual desktop systems. The closest work to ours is *VNCPlay* [20] which captures and replays keyboard and mouse events within the RealVNC client [18]. It is based on matching small screen regions around the coordinates of mouse-click events and then pairing these with expected screen updates. *VNCPlay’s* recording process is not structured and has no explicit tagging of synchronization elements thus the user has less control over which actions are timed. Our approach differs by allowing for more general artifacts (even covering the whole screen). As well it provides for “fuzzy” matching making it more robust in adapting to desktops which are not fully deterministic. Moreover, *DeskBench* is application and protocol agnostic because it can be implemented within the window manager library.

Another closely related work is *Slow Motion Benchmarking* [13]. It uses the concept of slow benchmarking that captures network traffic between the client and the server and replays it later in slow motion. The synchronization of the replay process is achieved because sufficient gaps are introduced between sending events to make sure that the server and the client are in a consistent state. This has an advantage of not requiring any execution on the client machine but can not be used to attain high replay speeds crucial for capacity benchmarking. A very interesting follow up, although not directly related to the benchmarking problem, is the virtual desktop recording capability described in [11].

Microsoft has developed a proprietary system devoted to testing the performance of Windows Terminal Services sessions. It is described in [7]. It has limited capability of synchronizing the replay by waiting for a given string to appear. Because of that this approach is not generic and can not be used with graphics display or to video replay that does not contain well defined strings in the RDP protocol stream itself. Also, it is limited to a single protocol type and architecture.

The VMWare VDI sizing document [3] bases virtual desktop benchmarking directly on the Microsoft Terminal Server Capacity and Scaling toolkit described above.

V. Conclusions and Future Work

A flexible virtual workstation benchmarking system is presented. It allows for rapid creation of artifacts representing user actions and replaying them in synchronized manner against a test system. It uses a novel approach to intercept and inject keyboard and mouse events and monitoring the screen updates in an application agnostic manner. It allows for replaying the artifacts at maximum speed thus making it possible to not only measure the responsiveness of the system precisely but also generate high load on the server. We have evaluated the system on the testbed and found it to be applicable and accurate.

Future research involves defining standard benchmarks for typical user groups, e.g., office worker or knowledge worker. Creating these may lead to establishing of virtual desktop benchmarking standard.

References

[1] Kernel Virtual Machines. http://sourceforge.net/projects/kvm.

[2] M. Chapman. rdesktop: A Remote Desktop Protocol Client. *http://www.rdesktop.org/*.

[3] VMWare Corp. VDI Server Scaling and Sizing.

[4] Citrix Corporation. Citrix Application Delivery Infrastructure. *http://www.citrix.com*.

[5] IBM Corporation. 3270 Data Stream Prgm. Ref. *http://www.elink.ibmlink.ibm.com/publications/*.

[6] Microsoft Corporation. Remote Desktop Protocol. *http://msdn2.microsoft.com/en-us/library/aa383015.aspx*.

[7] Microsoft Corporation. Windows Server 2003 Terminal Server Capacity and Scaling. *http://www.microsoft.com/windowsserver2003/*.

[8] Microsoft Corporation. Windows Terminal Services. *http://www.microsoft.com*.

[9] Transaction Processing Performance Council. TPC Benchmarks. *http://www.tpc.org/*.

[10] VMWare EMC. http://www.vmware.com.

[11] O. Laadan, R. Baratto, D. Phung, S. Potter, and J. Nieh. DejaView: A Personal Virtual Computer Recorder. *in Proceedings of the Twenty-first ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[12] Meiosys. http://www.meiosys.com.

[13] J. Nieh, S. Yang, and N. Novik. Measuring Thin-Client Performance Using Slow-Motion benchmarking. *in Proceedings of ACM Transactions on Computer Systems, Vol. 21 Npo. 1*, February 2003.

[14] R. Rivest. The MD5 Message-Digest Algorithm. *Network Working Group RFC 1321*, 1992.

[15] Linux Advanced Routing and Traffic Control. http://lartc.org/.

[16] R. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics 5,2*, 1986.

[17] Citrix Systems. Independent Computing Architecture Protocol. *http://www.citrix.com/*.

[18] Real VNC. Vnc. *http://www.realvnc.com/*.

[19] Xen. http://www.xensource.com.

[20] N. Zeldovich and R. Chandra. Interactive performance measurement with VNCPlay. *in Proceedings of FREENIX*, 2005.