Improving Remote Desktopping through Adaptive Record/Replay

Shehbaz Jaffer [*]
NetApp Inc., Bangalore
shehbaz.jaffer@netapp.com

Piyus Kedia [†]
Microsoft Research, Bangalore
piyuskedia@gmail.com

Sorav Bansal
IIT Delhi
sbansal@cse.iitd.ernet.in

Abstract

Accessing the *display* of a computer remotely, is popularly called “remote desktopping[1]”. Remote desktopping software installs at both the user-facing client computer and the remote server computer; it simulates user’s input events at server, and streams the corresponding display changes to client, thus providing an illusion to the user of controlling the remote machine using local input devices (e.g., keyboard/mouse). Many such remote desktopping tools are widely used.

We show that if the remote server is a virtual machine (VM) and the client is reasonably powerful (e.g., current laptop and desktop grade hardware), VM deterministic replay capabilities can be used adaptively to significantly reduce the network bandwidth consumption and server-side CPU utilization of a remote desktopping tool. We implement these optimizations in a tool based on Qemu/KVM virtualization platform and VNC remote desktopping platform. Our tool reduces VNC’s network bandwidth consumption by up to 9x and server-side CPU utilization by up to 56% for popular graphics-intensive applications. On the flip side, our techniques consume higher CPU/memory/disk resources at the client. The effect of our optimizations on user-perceived latency is negligible.

Categories and Subject Descriptors D.4.9 [*Operating Systems*]: Systems Programs and Utilities; C.2.4 [*Distributed Systems*]: Client/Server; I.3.4 [*Computer Graphics*]: Graphics Utilities

Keywords Virtualization, Record/Replay, Deterministic Replay, Remote Desktop, Cloud Computing, Virtual Desktop Infrastructure

[*] This work was done while the author was pursuing his Masters degree at IIT Delhi.

[†] This work was done while the author was pursuing his Ph.D. degree at IIT Delhi.

[1] We use the term “remote desktopping” instead of “remote desktop” to avoid confusion with a commercial product with the same name

VEE ’15, March 14–15, 2015, Istanbul, Turkey.
Copyright © 2015 ACM 978-1-4503-3450-1/15/03…$15.00.
http://dx.doi.org/10.1145/2731186.2731193

1. Introduction

“Remote Desktopping”, the capability to *control* a computer remotely, is used for several applications including remote access (e.g., accessing office computer at home), collaboration (e.g., multiple users working on the same screen), technical support (e.g., support staff accessing user’s machine), security surveillance (e.g., security personnel inspecting user machines), etc. Such tools and applications are popular and widespread. Some well-known remote desktopping tools are VNC, Windows Remote Desktop, TeamViewer, LogMeIn, Chrome Remote Desktop Extension, etc.

Virtualization and cloud computing has further increased the relevance of remote desktopping. Often, a server is implemented as a virtual machine (VM) and a remote desktopping client is used to access it. Another promising cloud computing paradigm, called “Virtual Desktop Infrastructure” (VDI), advocates running user desktops as VMs on a consolidated server and allows user access through remote desktopping tools. VDI has manageability, security, and cost advantages over traditional desktop computing, and improvements in remote desktopping directly impact the practicability of VDI.

All current remote desktopping tools work by streaming the display from the remote server to the user’s client computer. The network bandwidth requirement of this display stream is usually the Achilles’ heel of these tools. Although these tools use image/video compression, viewing high resolution videos or playing graphics-intensive games over a remote desktopping session is often impractical, due to network bandwidth constraints. Further, these tools require significant CPU usage at the server to compress and stream the display buffer. Our work alleviates these limitations.

We optimize remote desktopping tools to consume less bandwidth and server-side CPU while accessing a *virtual* remote server (i.e., the remote server is running as a VM) at the expense of higher client-side CPU consumption. (In Section 5, we discuss some remote desktopping applications where this tradeoff is favourable). We propose the use of VM Record/Replay technology to achieve this objective. In our scheme, instead of streaming the *screen* of the server, we stream the *state* of the server to client. The state of the server is captured using VM Record/Replay technology and consists of an *initial snapshot* and a *record log*.

To implement remote desktopping using VM record/replay, the server's hypervisor begins *recording* the state of the server VM by snapshotting it, and then continuously generates a record log. The client's hypervisor *replays* the server VM using its initial snapshot and the generated record log. The initial snapshot is made available to the client through demand paging over the network. The generated record log is streamed continuously from server to client. This allows the client to reconstruct server's state (and hence its screen) continuously over time. All keyboard/mouse input at the client is simulated at the server to provide the illusion of a remote desktopping session.

The aforementioned mechanism is useful because for many important applications, the size of the streamed record log (including the partial transfer of the initial snapshot using demand paging) is often less than the size of the streamed screen contents. In this paper, we demonstrate this (somewhat counter-intuitive) result through many experiments.

Our work provides evidence of the practicability of using VM record/replay technology to reduce network bandwidth consumption and server-side CPU usage of remote desktopping tools. We have implemented our techniques in a tool based on Qemu/KVM [12] and VNC [18]. We achieve up to 9x bandwidth improvements and 56% improvement in server's CPU utilization over current tools for display-intensive applications, while consuming 30% more CPU at the client. We think this is a welcome tradeoff in most cases. Further, as we discuss in Section 5, the replay-based remote desktopping approach is stateful, i.e., repeat remote desktopping sessions consume less network bandwidth than the first session, as we avoid repeat transfers of the same state.

There are two limitations to our work. Firstly, our current implementation works only for compatible client and server machines, i.e., the client machine should support all CPU features of the server machine for it to be able to replay the server's execution. In general, this limitation can be addressed in either of the two ways: (1) by using a more advanced virtual machine monitor that can tide over slight differences in CPUs (using binary translation for example) and thus can replay across slightly divergent CPUs, or (2) by configuring the virtual machine to assume a less-featured virtual-CPU so it is compatible with both client and server[2].

Our second limitation is that our current implementation works only for uniprocessor VMs. Multiprocessor VM record/replay is less efficient, and while recent work promises significant advances, this still remains an open problem. For this paper, we posit that many (perhaps most) workloads are run on uniprocessor VMs (e.g., in cloud), and hence they can immediately benefit from our ideas.

[2] Because successive processor generations are usually backward compatible, the vCPU need only be equivalent to either the server or the client CPU, whichever is older

The paper is organized as follows: Section 2 provides background on streaming VM deterministic replay and discusses our remote desktopping scheme, Section 3 discusses the challenges in our design and their solutions, Section 4 discusses our implementation, Section 5 discusses our experiments and results, Section 6 discusses related work, and Section 7 concludes.

2. Streaming VM Record/Replay

The ability to record the minimal set of non-deterministic events occurring in a VM, and then using this recorded log to replay the VM, is popularly known as VM Record/Replay [19]. For a uniprocessor VM execution, the sources of non-determinism are input I/O (e.g., network), interrupt timing, and certain non-deterministic instructions (e.g., reading the timestamp counter). A record log consists of a stream of non-deterministic events tagged with their *timestamp*. The timestamp of an event uniquely identifies the (deterministic) logical time epoch at which the event occurred. Modern x86 processors provide the capability to precisely monitor the number of branches executed (`branchcount`), and this together with the value of the current program counter (encoded by `rip` and `rcx` registers) provides a unique logical timestamp.

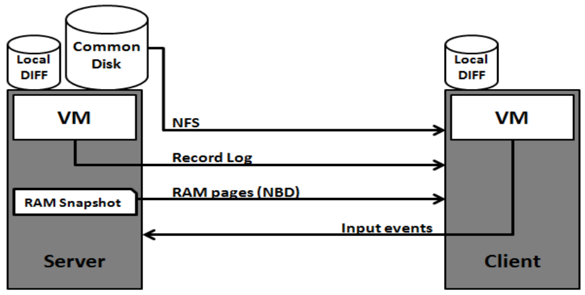

Figure 1. Architecture of record/replay-based remote desktopping

Replay involves first loading the snapshot created at the start of the recording session, and then injecting the recorded non-deterministic events into the VM at their respective time epochs. The x86 architecture provides the performance monitoring interrupt (PMI) mechanism to allow setting up of hypervisor traps at desired values of `branchcount` (somewhat imprecisely though [8]), and we use this for injecting non-deterministic events during replay.

Our proposed remote desktopping mechanism involves streaming the input keyboard/mouse events from client to server and streaming the record log from server to client. The record log contains the non-deterministic event stream at the server, and the client replays it to reconstruct the server state continuously. Figure 1 shows our solution's architecture. A sample sequence of events in this architecture involves (1) an input keyboard/mouse event being sent over network from client to server; (2) the server injecting this event into the VM; (3) the ensuing non-deterministic events getting recorded in the log which is streamed to the client; and (4) the log getting replayed at the client to provide a remote desktopping experience to the user. The first two steps (1) and (2), are common with existing remote desktopping tools. The difference is in the traffic generated from server to client — existing tools transfer server's screen contents, while we transfer server's state. We call our scheme *record/replay-based remote desktopping* to distinguish it from traditional *display-based remote desktopping*.

Notice that the log of non-deterministic events at server is generated continuously and includes events unrelated to client's input activity. For example, any network packets received at server also get recorded as non-deterministic events in the record log and are streamed to the client (so that it can reconstruct server's state). Thus, the user is provided the illusion of controlling the server and viewing its console remotely (even though the console is actually that of the client's replayed VM).

Our on-the-fly reconstruction of server state at the client using record/replay differs from previous work (e.g., Moka5 [10]) on streaming VM images over network (which we call *VM streaming*). In VM streaming, a VM's image is streamed on-demand from server to client, and this VM is executed stand-alone on the client computer with the client's environment. Hence, in VM streaming, the VM appears to the user as executing locally at client with a client-local network address. On the other hand, our remote desktopping architecture provides an illusion to the user of controlling a VM which is running on the remote server with the server's environment (e.g., the user observes a server-side network address). Also, unlike VM streaming, remote desktopping allows opening of multiple simultaneous sessions by multiple users for the same logical VM state (thus enabling collaborative applications of remote desktopping). We discuss this comparison between VM streaming and record/replay-based remote desktopping in more detail in Section 6.

Our record/replay-based solution is useful for remote desktopping *only if* record log sizes are lower than the corresponding (compressed and streamed) video framebuffer sizes generated by existing display-based remote desktopping tools. As we show in our experiments (Section 5), this is true for many display intensive workloads, and our solution is quite effective in these situations. Intuitively, the record log size is primarily a function of input I/O volume — hence, deterministic replay is likely to generate less traffic than display-transfers if the input I/O volume for a workload is less than the display activity it generates.

3. Architecture

In this section, we discuss the two primary challenges we faced (and their solutions) in making record/replay-based remote desktopping practical.

First, any streaming record/replay session involves creation and transfer of the initial VM snapshot. Because VM sizes can be huge (disks up to a few TBs, and memory up to a few 100GBs), the time and bandwidth required to create and transfer these snapshots could overshadow any improvements by our solution. In Section 3.1, we discuss fast and asynchronous creation of VM snapshot, and in Section 3.2, we discuss on-demand transfer of snapshot state to minimize network traffic.

Second, different workloads exhibit different video framebuffer (screen) activity and different non-deterministic record log activity. Depending on the workload, either display-based remote desktopping or record/replay-based remote desktopping may be the more performant option. We adaptively switch between the two options to optimize overall performance (Section 3.3).

3.1 Snapshot creation

Every streaming record/replay session (and hence, every record/replay-based remote desktopping session) requires VM snapshot creation, which involves snapshotting its CPU, memory, devices, and disk. The snapshot size of CPU and device state is small (less than 9 MBs uncompressed), and thus the dominant cost of snapshot creation and transfer is due to disk and memory.

We compare two methods of snapshotting and recording the disk. In the first method, we use standard copy-on-write snapshotting techniques (we call this method, the *snapshotted disk* method). In the second method, we do not create a disk snapshot, but instead record all disk reads in the record log, thus causing all disk reads to be replayed identically at the client (we call this method, the *output-replayed* disk method). In the snapshotted disk method, creation of copy-on-write disk snapshot is fast (sub-second), although future disk writes experience slightly higher latencies (due to extra copying). Output-replayed disks have zero snapshotting cost, but incur more network traffic if multiple reads, or reads-after-writes are made to the same disk blocks.

Snapshotting memory is more involved. Simply marking all memory copy-on-write (like disks) results in relatively higher costs of copy-on-write faults in memory. Our goal is to minimize any performance effects on server's execution due to snapshotting. To avoid user-visible slowdowns during application execution, memory snapshots are created asynchronously[3]. i.e., the VM continues to execute while a

[3] Asynchronous snapshotting could still result in user-visible slowdowns if the asynchronous thread contends for a common resource with the VM (e.g., CPU or bus bandwidth). We discuss the practical implications of this in our experiments (Section 5).

snapshotted memory copy is created in parallel. Here is a sequence of steps taken to create a snapshot asynchronously:

1. All memory pages of the VM are marked *clean*. The VM is stopped for a short duration (few milliseconds) during this step.
2. A separate snapshotting thread begins saving memory contents to a snapshot file. This file could be stored on-disk or in-memory.
3. Any concurrent modifications to memory pages by the VM, mark the corresponding pages *dirty*.
4. When the snapshotting thread finishes, the size of the set of dirty pages is compared with a threshold:
   a. If the dirty set size is below the threshold, the server is stopped and a snapshot of the pages in dirty set is taken synchronously. At this point, we have a separate copy of a complete memory snapshot at server.
   b. If the dirty set size is greater than the threshold, steps 1-4 are repeated.

   This threshold is chosen such that the server stops (in Step 4a) for less than a few milliseconds. In our experiments, this threshold was set to 2048 pages.

If this procedure does not converge in thirty iterations, we simply give up and disable record/replay-based remote desktopping.

Notice that this snapshotting procedure could take long (10-100 seconds), but because it is executed asynchronously, this time is largely hidden from the user. The user only observes a potential stall of a few milliseconds when the server is stopped in Step 1 at initialization time and in Step 4 at convergence-check time. Because the client can begin replay only after the server has finished snapshotting, we resort to display-based remote desktopping till then (for a continuous user experience). On completion of snapshot creation and its transfer from server to client, our tool switches from display-based remote desktopping to record/replay-based remote desktopping seamlessly (the full discussion of this switching is in Section 3.3). If the snapshotting procedure did not converge, we continue with display-based remote desktopping as before.

3.2 Snapshot transfer

Transfer of snapshotted state is done *on-demand* to minimize user-perceived latency and network traffic. All register and device snapshot state is transferred from server to client at the start of the streaming record/replay session.

For snapshotted disks, the disk snapshot is transferred on-demand from server to client over NFS [13]. Only those snapshotted disk blocks are transferred which are ever accessed at the client. Also, because the snapshot file is read-only, repeat accesses by client to the same disk block do not result in repeat network traffic.

Transfer of snapshot's memory state is performed on-demand at page granularity (4KB). Demand paging is implemented at client by modifying the VM's extended page tables, and only the pages touched by client are transferred over network. Because typical working set sizes of VMs are much smaller than their total memory allocations, demand paging saves significant network bandwidth and reduces user-perceived latency.

3.3 Adaptive switching

The application workload characteristics determine the amount of video framebuffer activity and the amount of non-determinism generated in the system. For example, execution of a graphics-intensive game involves large screen activity, but involves relatively smaller non-determinism (most non-determinism is only due to interrupt timing and keyboard/mouse inputs). On the other hand, an application involving large network downloads (e.g., P2P file sharing) results in large input non-determinism (due to a stream of incoming network packets) and relatively smaller screen activity. Thus, display-based remote desktopping is likely to be the more performant alternative for applications of the latter type, while record/replay-based remote desktopping perform better for the former type of applications. We use a dynamic adaptive mechanism to switch between the two modes depending on current workload behavior.

The adaption logic runs at both client and server. A remote desktopping session always starts in *display mode* (i.e., traditional display-based remote desktopping) and switches to *record/replay mode* (i.e., record/replay based remote desktopping), if required. The client decides the appropriate mode, and informs the server. On receiving a request to switch from display to record/replay mode, the server begins snapshotting asynchronously. Till snapshotting is complete, the client and server run in display mode, and together switch to record/replay mode after the snapshot is taken. If the client again decides to switch back to display mode (from record/replay mode), it simply stops replay and spawns a display mode client (e.g., VNC client) to connect to the display mode server (e.g., VNC server).

This scheme works well for low-latency (< 1ms) networks; the transition from display mode to record/replay mode is seamless and translates to a blip of only 100-300 milliseconds (not noticeable by human user) at the client. With higher network latencies ($5 - 100$ms), the switchover from display to record/replay mode causes a longer human-visible stall at the user, as we discuss below [4].

A client that has freshly switched from display to record/replay mode generates many demand paging faults for memory (cold misses), and each fault results in a network round trip. The net effect is a client that is slower than the server in the first few seconds of the switchover. This client-side slow-

[4] Transitions in the other direction (record/replay to display mode) are always quick ($< 50ms$).

down is more for higher latency networks. A slow client causes record log traffic to get queued at the server, effectively making the client appear chronologically behind the server (resulting in overall poor user experience).

To deal with this problem, we introduce a transient *dual mode*, where the tool runs in both display and record/replay modes simultaneously. While switching from display to record/replay mode, the client runs in dual mode for up to 200 seconds after snapshotting is complete. In dual mode, the client computer streams-in both display and record/replay traffic, while the user facing screen displays the output of only the display mode client. This allows the record/replay client to "catch up" with the server, while keeping the user oblivious of this background switch. We consider a client to have *caught up* with the server if it reaches the end of the streamed record log network buffer (i.e., the client and the server are at the same execution epoch). This strategy of using a transient dual mode works well for both low-latency and high-latency networks. If the client does not catch up with the server in the 200 seconds of dual mode, the tool reverts back to display mode. 200 seconds are usually enough for a record/replay client to catch up (if it can), while still keeping dual mode overheads low. Typical catch-up times are lower and depend on the network latencies and workloads (see experiments in Section 5). The dual mode results in more network and CPU overhead during switch; this cost is recovered if the switch is successful and saves network and CPU over a longer duration.

The dual mode provides a convenient and simple switching strategy: if the client catches up during dual mode, the switch (from display to record/replay mode) is upheld, else it is cancelled. This end-to-end decision strategy holistically captures many system characteristics like network parameters, client-side resource capabilities, and workload behavior automatically. An attempted switch that gets cancelled in dual mode (due to the client not catching up in 200 seconds) results in wasted overhead, and so it is important to avoid failed switching attempts. As we discuss later, the switching decision from display to dual mode is based on traffic measurements and estimates. We also disallow a consecutive switching attempt within 20 minutes of a failed switch.

We use a similar end-to-end strategy for switching back from record/replay to display mode. If during record/replay mode, we find that the record log buffer has grown more than a threshold, we initiate a switch back to the display mode. In our current implementation, we switch back if the client falls behind the server by more than 500ms (we measure time difference by counting the number of in-flight timer interrupt entries). As we discuss next, we also switch from record/replay to display mode if traffic measurements and estimates suggest a possible benefit by doing so.

We measure and estimate network traffic in display and record/replay modes to aid the switching decision. In display mode, the record/replay mode traffic is estimated by measuring the amount of non-deterministic input at the server VM. In record/replay mode, the display mode traffic is estimated by capturing the VGA virtual device activity. Both these measurements are lightweight ($< 1\%$ overhead) and are implemented in the hypervisor at the server, and are communicated to the client. We have tested our estimation routines on a variety of workloads, and they provide accurate estimates of traffic in each mode.

A switch from display to dual mode is initiated if record/replay-mode traffic is estimated to be at least 75% of the current display-mode traffic over the past 20 seconds. Similarly, a switch from record/replay mode to display mode is initiated if the display-mode traffic is estimated to be at least 1.1x less than the record/replay-mode traffic over the past 20 seconds. Notice that our strategy for switching from record/replay to display mode is more aggressive than our strategy for switching in the other direction. This is justified given the higher overhead of a bad switching decision in the latter case. The thresholds (1.5x, 1.1x, 20 seconds) have been chosen based on expected workload characteristics; they capture the important optimizations for display-intensive workloads, but ensure that overheads/bad-decisions do not occur for other workloads. The system is largely indifferent to small changes (10-60%) to these thresholds.

We call the duration between a *decision* to switch (by the client) and the *actual* switch, the "switching latency". We call the time for which the server actually stops (e.g., steps 1 and 4 in Section 3.1), the "switching stall time". The switching stall time is always smaller (often much smaller) than the switching latency.

The switching latency from display mode to record/replay mode is dominated by the time taken to snapshot the server's VM, and could be up to tens of seconds. The corresponding switching stall time is typically smaller (less than 0.35 seconds in all our experiments). The switching latency from record/replay mode to display mode is much lower and is always less than 10-50 milliseconds, as it only involves spawning a display mode client.

4. Implementation

We have implemented our remote desktopping tool [1] using Linux/Qemu/KVM [5], similar to previous work [19]. Record/replay of the disk controller, network card, and other devices was implemented by modifying the emulation logic of the respective devices appropriately. The record log entries have been structured to take minimal space, and no further compression is performed on them.

To implement record/replay-based remote desktopping, we extended our implementation to be able to read/write the record log from/to a network socket. The server runs the KVM process in record mode (outputting the record log to a network socket) and the client runs the KVM process in replay mode (inputting the log from the socket). If the client (consumer) is faster than the server (producer), the socket's buffers eventually empty, causing the client to wait for the server to produce more log entries. Because the server generates at least one record log entry on every timer interrupt, the client never appears "inactive" to the user. Similarly, if the server is faster than the client, the socket's buffers eventually get full. We size the send socket buffer of Server and receive socket buffer of Client to 16MB to avoid the buffer from getting full due to network/traffic variations. We switch back to display mode if the socket buffer gets full. We also switch back to display mode if the client lags behind the server by more than 500ms in record/replay mode (such lags are allowed in dual mode).

Our remote desktopping client, when running in record/replay mode, runs a full VM on the client machine and displays its console to the user. All user input keyboard/mouse activity on that console is transmitted to the server. Demand paging (for transfer of snapshotted memory) is implemented by memory-mapping a "network block device" (NBD) [9] in the guest's address space. Thus, guest's memory accesses at the client cause appropriate network transfers from the server. If using snapshotted disks, the snapshotted disk is exported over NFS from server to client (NFS server process is run on the server machine, and NFS client process is run on the client machine). NBD's read-ahead parameter was set to 256 (default value).

TightVNC [18], an efficient open-source VNC implementation, is used to implement the display mode of our remote desktopping tool. A TightVNC server process runs on the server machine; the client connects to this server to run in display mode.

5. Experiments

We performed our experiments on a server machine with Intel Xeon X5650 2.67 GHz processor, 24GB memory, and 300GB disk, and a client laptop machine with Intel Core i5 2.40GHz processor, 4GB memory and 500GB disk. In our first set of experiments, we connected the machines through a 100Mbps network (effective download bandwidth of around 10-12MBps) with 0.2ms latency. We discuss the effect of network latency and bandwidth later in this section We report comparisons with VNC — our comparisons with other display-based remote desktopping tools had similar results (see Section 6 for details).

We first discuss the performance of our record/replay implementation. Table 1 lists some workloads and the corresponding record log growth rates for different workloads. We have chosen a mix of graphics-intensive, compute-intensive, disk and network intensive applications. The runtime overhead of record/replay for all these workloads is less than 10% (mostly 1-5%). For `sleep`, replay finishes 2x faster than a normal run, because CPU's idle periods (marked by the execution of the `hlt` instruction on x86) get skipped-over during replay. In general, any workload involving CPU idle time finishes faster during replay (i.e., client executes faster than server in our remote desktopping setup).

We next compare network bandwidth usage of our remote desktopping tool with existing display-based remote desktopping tools. Figure 2 presents our network bandwidth consumption results for display-intensive applications (`video-local`, `video-stream`, `supertux`, and `tuxfootball`). For each application, we show the cumulative network bandwidth consumed over time by display-based remote desktopping (`VNC`) and record/replay-based remote desktopping (`RecRep`). We show results for two videos of 240p and 480p resolutions. With VNC, we use two compression modes: lossy (`VNC-6` for quality-level 6, and `VNC-9` for quality-level 9) and lossless (`VNC-lossless`). VNC-lossy uses JPEG compression, while VNC-lossless uses the most efficient lossless compression method[5].

Applications involving large display activity perform significantly better with record/replay-based remote desktopping. Table 3 lists the average network traffic rates for each workload produced by `VNC-6`, `VNC-lossless`, and `RecRep`. Notice that for video and game workloads, `RecRep` provides a lossless viewing experience and yet consumes significantly less bandwidth than `VNC-6`. The measurements in Figure 2 include the startup cost (i.e., the cost of snapshot transfer at the start of the remote desktopping session). The extra traffic due to snapshot transfer at startup is seen as a steep vertical jump in bandwidth consumption in the first few seconds of each curve in Figure 2. However, the slope of the `RecRep` curve is significantly lower than `VNC`. Table 3 lists the steady-state traffic rates (steady-state slope of the curve in Figure 2) for each application.

`RecRep`'s traffic consists of record log, memory demand paging traffic, and NFS traffic. For `video-local` applications, a majority of `RecRep` traffic is due to demand paging and NFS — client accesses the video stored on disk resulting in fresh memory and NFS file accesses, causing on-demand network traffic. For `video-stream`, the majority of `RecRep` traffic is due to the record log — all incoming network packets containing the video manifest as input non-determinism and get transferred from server to client as record log entries. In both cases, the traffic between server and client is proportional to the size of the original video (stored on disk or streamed over network). This is an improvement over VNC, where in comparison, video is first decompressed (to generate a screen) and then *re*-compressed (to transfer to client). The quality of this on-the-fly recompression (done by VNC) is evidently less than the quality of compression on the original video (which is a property of the video's file format, e.g., FLV, MP4, etc.).

`RecRep` performs even better for graphical games (`supertux`, `tuxfootball`) where there is low input non-determinism

[5] We set `TightVNC` parameters to minimize network traffic. `TightVNC` was the best performing VNC implementation we came across in terms of network bandwidth consumption.

Benchmark	Description	Log growth rate (KBps)	
		snapshotted disk	output-replayed disk
`video-local-` `(240, 360, 480)`	Viewing a video stored locally on the VM at 240p, 360p, and 480p resp.	59, 63, 64 resp.	79, 85, 86
`video-stream-` `(240, 360, 480)`	Viewing a video streamed from a LAN server at 240p, 360p, and 480p resp.	131, 208, 272 resp.	140, 221, 267
`emptyloop`	A process running a compute-intensive loop	32.3	36.3
`forkbomb`	Forks 1 million processes, each exiting gracefully	65.8	89.2
`cp`	Copies 100MB file within the same directory on ext3	10.6	58.3
`inet`	Receives data over TCP socket in 4-byte chunks	793	791
`onet`	Sends data over TCP socket in 4-byte chunks	714	732
`sleep`	Calls `sleep(10)` (idles for 10 seconds)	1.7	2.9
`iscp`	Copies 100MB from host to guest using `scp`	269.5	282.8
`lincompile`	Builds the Linux kernel from source	42.7	45.6
`supertux`	Playing a graphical game called `supertux`	48.6	82.75
`tuxfootball`	Playing a graphical game called `tuxfootball`	148.5	171.2

Table 1. Benchmark description and record log growth rates

Benchmarks	Server CPU Usage			Client CPU Usage		
	RecRep	VNC-lossless	VNC-6	RecRep	VNC-lossless	VNC-6
`video-local 240p`	16.21	39.14	36.12	54.36	17.97	18.30
`video-local 360p`	28.10	63.82	62.62	63.00	22.89	26.10
`video-local 480p`	34.90	85.40	79.87	71.33	24.33	27.50
`video-stream 240p`	17.17	36.82	35.96	65.13	18.41	19.13
`video-stream 360p`	33.93	70.95	64.88	72.03	23.62	27.97
`video-stream 480p`	37.05	81.57	80.17	73.89	23.33	28.32

Table 2. Average CPU usage at server and client for VNC and RecRep with snapshotted disk. Similar results are obtained with output-replayed disk .

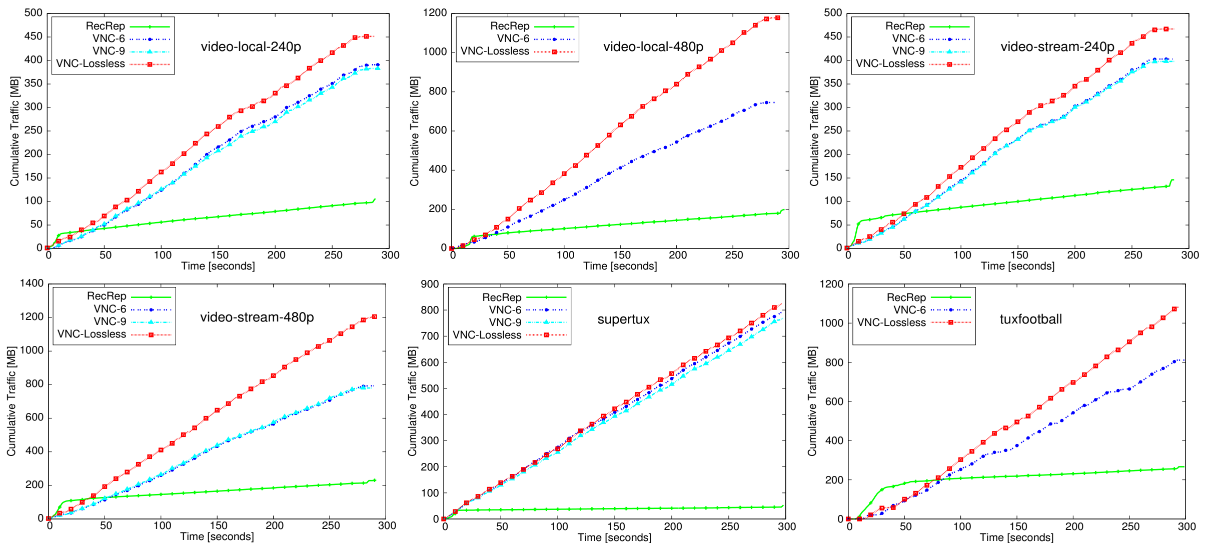

Figure 2. Comparisons on network bandwidth consumption for display-intensive applications. These results were obtained on a 100Mbps, 0.2ms latency network between client and server.

Application	VNC-lossless	VNC-6	RecRep			
			record log	demand paging	Snapshotted	total
`video-local 240p`	1867	1414	151	80	38	268
`video-local 360p`	3444	2390	157	113	107	377
`video-local 480p`	4605	2937	168	201	155	524
`video-stream 240p`	1763	1157	245	16	0	262
`video-stream 360p`	4030	2598	319	12	0	332
`video-stream 480p`	4688	2905	391	11	0	403
`supertux`	2421	1918	217	55	0	271
`tuxfootball`	4112	2941	301	464	0	767
`lincompile`	23	22	133	177	14	325
`video-B-after-A 480p`	4688	2905	158.51	159.27	155.12	470.41
`video-A-repeat 480p`	4688	2905	165.23	0	0.14	165.37

Table 3. Average network bandwidth rates with snapshotted disk (KBps). The measurements with output-replayed disks are similar .

and high screen activity. Most graphics are generated using local computation, and thus `RecRep` generates minimal traffic for `supertux` (92.8% reduction over VNC). With `RecRep`, the game's computation happens locally at client — this improves network and server CPU utilization at the expense of client CPU usage.

The initial traffic due to snapshot transfer is approximately 41MB, 47MB and 37MB for `video-local-480p`, `video-stream-480p`, and `supertux` respectively. This initial snapshot transfer size depends on the memory and disk footprints of the application. This traffic is compensated by reduction in steady-state traffic growth rates, and we break even at 28s, 21s, and 52s respectively for the three applications. Further, unlike display-based remote desktopping, record/replay-based remote desktopping has the advantage of being *stateful*. i.e., state once transferred from server to client can be reused without repeat transfers over the network. We next performed an experiment where the same video was played twice in the same session. To further distinguish application traffic (due to video-player) from data traffic (due to the played video), we also played another video in that session. Initially, all videos and applications were stored on disk (at server) when the client initiated a fresh remote desktopping session. Figure 3 plots the network usage results. The corresponding traffic rates are also presented in Table 3 (labels `video-B-after-A`, and `video-A-repeat`). Interestingly, compared to `video-local-480`, the total traffic rates for `video-B-after-A` and `video-A-repeat` are 10.30% and 68.51% lower. `video-B-after-A` requires only the video (and not the video-player) to be transferred from server to client; while `video-A-repeat` requires no new state transfer (only execution information is transferred). These improvements were seen both for snapshotted and output-replayed disks (the video gets cached in memory after first run). Because of our fetch-only-on-demand optimizations, the amount of `RecRep` traffic is largely independent of VM's memory/disk size, and only depends on the workload's memory/disk footprints.

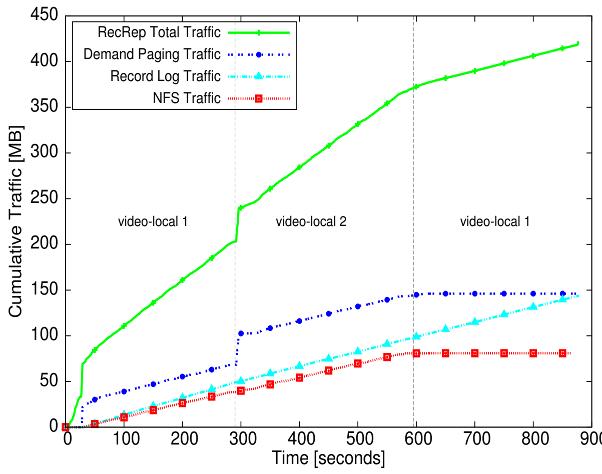

Figure 3. Network bandwidth consumption during a session involving three local video playbacks. During the first 60 seconds, video A is played; the next 60 seconds, video B is played using the same video player; and in the last 60 seconds, video A is played again. These results were obtained on a 100Mbps, 0.2ms latency network between client and server.

For other applications which do not exhibit rapid screen activity however, display-based tools (`VNC`) perform better. Our tool adaptively switches between the two approaches based on current workload behavior. We next discuss experiments involving multiple switches between display and record/replay modes. We ran a sequence of applications in the following order (we call this experiment the "sequence experiment"):

1. 5 minutes of `video-local` 360p
2. 100 seconds of `idle`

3. 5 minutes of `video-stream` 360p
4. 12MB file `download` over LAN
5. 300 seconds of playing `supertux`

Figure 4 plots the network bandwidth consumption of our remote desktopping tool over time, and compares it with `VNC`. Our tool starts in display mode (our display mode has identical performance as `VNC-6`) and switches to record/replay mode for `video-local`, `video-stream`, and `supertux`. It switches back to display mode for `idle` and `download`. The back-and-forth switching between modes is fast. The time between the decision to switch and the actual switch (switching latency) from record/replay mode to display mode is less than 10 milliseconds. The switching latency from display mode to record/replay mode was less than 10 seconds for all such switches in this experiment. The corresponding switching stall times ranged between 0.31 seconds and 0.35 seconds. Around half of this switching stall time (around 0.16 seconds) is due to the synchronous transfer of the last few memory pages (Step 4a in Section 3.1). The rest of the stall time is due to transfer of disk state metadata (to implement copy-on-write) and other processing at server to create a full snapshot. Switching latency is independent of VM's memory/disk size, and only depends on the workload's memory access pattern. Switching stall time is independent of VM size and workload's characteristics and only depends on the underlying hardware speed.

We next discuss comparisons on resource utilization at server and client. Table 2 lists average client/server CPU utilization for each workload. Figure 4 also plots the server-side CPU utilization for the sequence experiment over time. For applications involving small display activity (`idle` and `download`), both `VNC` and our tool (which switches to display mode) cause small CPU overheads. For video applications (`video-local` and `video-stream`), `VNC` consumes more CPU than `RecRep`. `VNC` requires more CPU to encode and compress the video framebuffer before streaming it to the client. On the other hand, `RecRep`'s network stream is lighter and thus requires fewer CPU cycles. Overall, our tool consumes up to 56% less CPU on the server for video playback, streaming, and gaming applications. Our tool consumes less CPU on the server than on the client (even though they are executing the same instructions) for two reasons: first, unlike the client, the server does not execute the code to render the screen on its own display; second, replay is slightly more expensive (around 10% for CPU-intensive workloads) than record as it also involves instruction-level singlestepping for interrupt replay, as discussed in Section 4.

Figure 4 also plots the client's CPU utilization for the sequence experiment over time. At client, our tool in record/replay mode consumes significantly more CPU than `VNC`. `VNC` is only required to decode the network packets and produce a screen. On the other hand, our tool runs a full VM in replay mode at client, resulting in higher CPU usage. This is the downside of our approach, and we discuss this tradeoff in Section 6.

We next measured keystroke latency while running in record/replay mode. We recorded the time elapsed between a client's keypress and the appearance of the corresponding character at client screen. This latency was between 10-70ms in all our experiments, ensuring a fast and comfortable user experience. (Any extra latency added by network will be common to both display-based and record/replay-based tools). Like other display-based remote desktopping tools, our tool supports concurrent access by multiple clients to one server. Even in collaborative settings, the network and CPU performance comparisons with display-based tools remain similar.

We next discuss the effect of network latency and bandwidth on our record/replay based scheme. Figure 5 plots the network traffic generated by our scheme on 0.2ms and 100ms networks for different bandwidths (10Mbps-1000Mbps) for `video-360p` application. For comparison, we also plot the traffic generated by VNC on a 100Mbps network in these graphs. To better understand the behavior of our tool, we disabled the limit on dual mode duration in this experiment, i.e., dual mode would go on till the client reaches the end of record log network buffer, and is not limited to 200 seconds.

Our scheme is more sensitive than VNC to network latency and bandwidth changes, as network parameters directly impact the switching time and the time the system remains in dual mode. The steady state bandwidth consumption of our tool remains largely independent of the network parameters. Slower networks (with higher latencies and lower bandwidths) cause our system to take longer to switch to steady-state record/replay operation, and thus cause higher traffic generation (longer time in dual mode). The overall comparison still remains favorable however, with break-even points reached at around 78s, 89s, and 195s for 0.2ms, 10ms, and 100ms latency networks respectively.

For bandwidths less than 30Mbps, the dual mode never finishes as the client is never able to catch up. For this experiment, dual mode bandwidth requirements of our tool are around 25Mbps, which are not met by a 20Mbps network. In contrast, VNC's bandwidth requirement for this experiment is only 18Mbps. Hence, our scheme has a drawback of requiring slightly higher network capacity during dual mode. After dual mode finishes, the bandwidth requirement of our scheme is much lower (3Mbps). If dual mode can be avoided (for example, by manual switch to record/replay mode), our scheme can run on lower capacity networks also.

Example Applications

Our experiments demonstrate that for applications where inputs are smaller than outputs, it is better to ship the inputs and reconstruct the outputs, thus saving shipping costs. VM record/replay is an enabling technology of this principle for

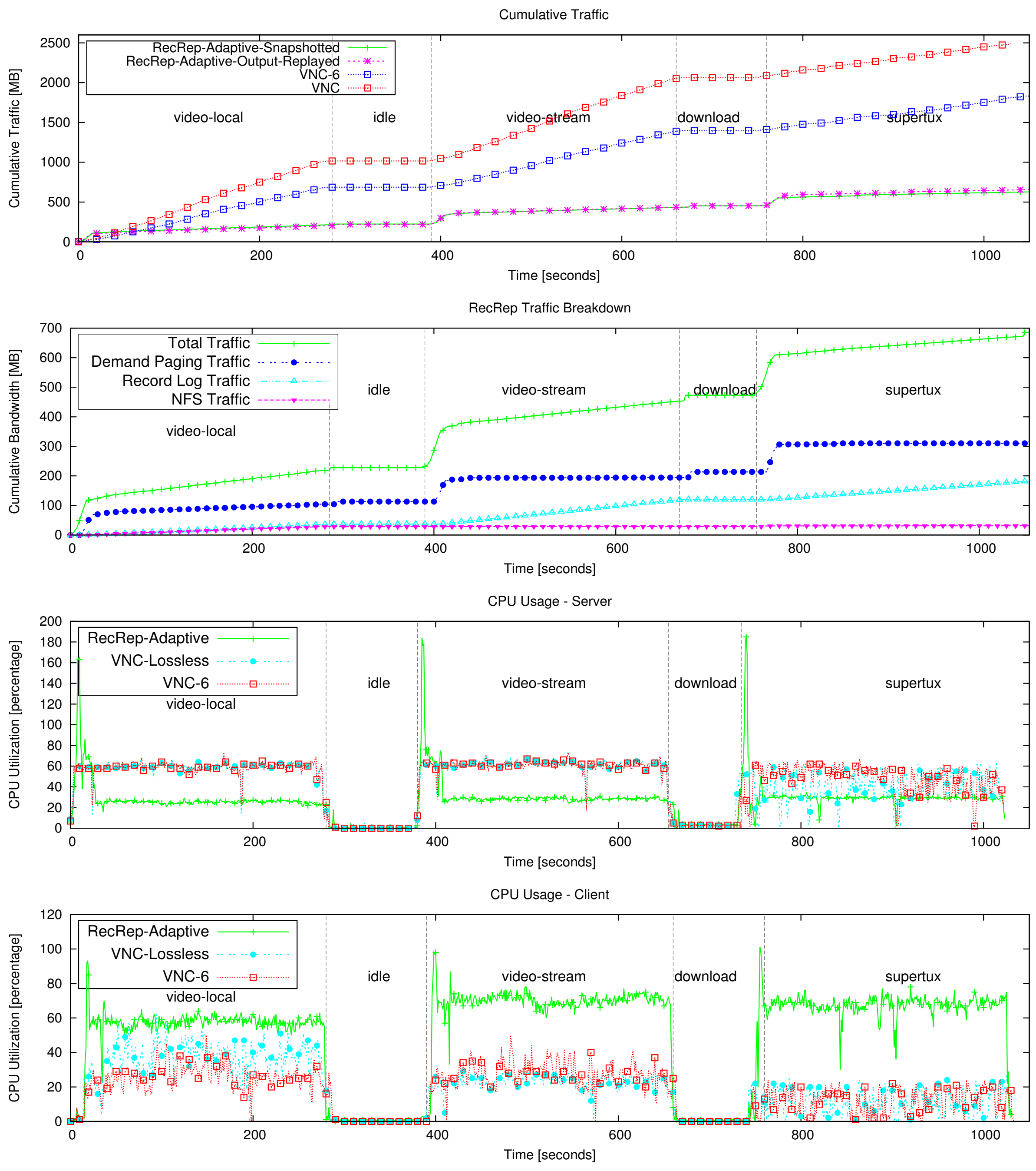

Figure 4. Running a sequence of applications (sequence experiment) over a 0.2ms latency, 100Mbps network.

remote desktopping tools. We discuss some intended applications of our tool:

Desktop-in-cloud: Being able to maintain your desktop state in the cloud has well-known advantages of cost, platform-independence, mobility, flexibility, maintenance, security, backups, among others. Remote desktopping is a natural choice for accessing a desktop in cloud. The closest alternative to this desktop-in-cloud paradigm is the use of cloud for storage (let's call this a *storage cloud*). In a storage cloud, applications are run locally on client, and the cloud maintains the data store. For display-intensive (and non-input-intensive) applications (e.g., game, video), a storage cloud offers a natural advantage over desktop-in-cloud. On the other hand, a storage cloud performs poorly if an application requires processing of lots of disk data (e.g., `grep`), as all data needs to be transported to client for computa-

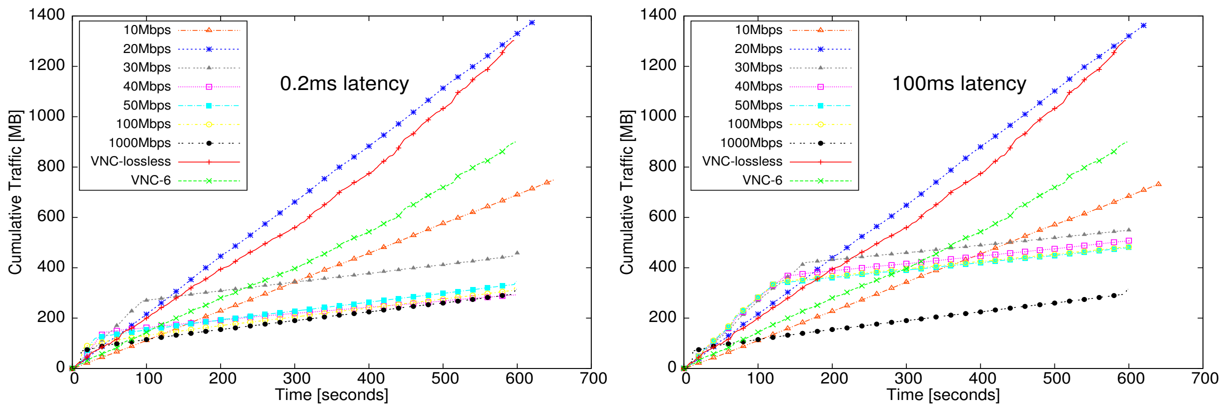

Figure 5. Cumulative Traffic consumed for varying bandwidth and varying latencies for `video-360p`.

tion. In a desktop cloud, such applications run locally on cloud and only their output needs to be shipped to the user-facing client. Our work allows the best of both worlds; input-intensive workloads like `grep` can run in display mode, while display-intensive workloads can run in record/replay mode, providing best performance in each case.

Network-level access control, **License Restrictions**: Often, organizations enforce network-level access control policies, whereby the server computers are allowed greater network visibility than client computers. In these situations, remote desktopping may be the only way to access applications depending on such network access. If such applications are also display-intensive (e.g., network games), our record/replay approach can help significantly. Similar situations can also arise due to license restrictions on an application (e.g., it can only run on one computer).

There remains a more fundamental question challenging our approach: if somebody has a client capable of running a workload, why will he/she use remote desktopping to run it? The alternative, of course, is to run the workload locally. We think that local installation and execution of an application entails its own challenges and usability headaches, and does not provide the mobility, inter-operability, maintenance, security and consolidation advantages of cloud computing. Our remote desktopping scheme allows the performance of local execution while retaining the benefits of "execution in the cloud".

6. Related Work

Previous efforts on improving remote desktopping technology have centered around designing the right abstractions and protocols for exchanging display information between the client and the server. While windowing systems like X [15] and NX [11] abstract display activities at the application layer, other tools like SLIM [16] and THINC [4] abstract at the device driver layer. Similarly, VNC [18] abstracts at the framebuffer layer. In our experiments with various tools including RDP [2], PCoIP [3], and X [7, 15], we found that VNC's bandwidth consumption for video/graphical workloads is less than or comparable to these tools. VNC relaxes the video quality (frame rate, lossy JPEGs) aggressively for better compression. Similar comparisons reported in [4, 16] confirm this observation. Our experiments show that record/replay-based remote desktopping is capable of even outperforming VNC-lossy on network bandwidth, while still retaining full video quality. Because input non-determinism is recorded at the VM-level, our approach works generally across all OS/Application stacks.

While the use of record/replay for remote desktopping is novel, the central observation that using deterministic replay to generate machine state can be cheaper than transmitting it explicitly has previously been made in [6, 14] in the context of fault tolerance. The work closest to ours in our objectives is perhaps VM streaming (e.g., Moka5 [10], VMware VM Streaming [17]), where VMs stored at server are fetched *on-demand* to client. In VM streaming models, multiple clients may "checkout" the same VM simultaneously and consistency across parallel updates is maintained using a revision control system. Like us, these systems also use demand paging and copy-on-write optimizations to reduce network traffic. But, there are two important differences between our models:

First, in the VM streaming model, the server only acts as a VM repository, and the client accesses a VM and image and runs it locally and independently. This implies that the user observes client-local device environment inside the VM (e.g., the VM can only access client-visible network hosts). Our remote desktopping model, on the other hand, provides the illusion of operating a VM running on the server, with the server's environment (e.g., the VM will have a server-side IP address).

Second, in the VM streaming model, multiple clients may run the same VM image concurrently. At the time of "committing" the clients' updates back to the server, the user may be required to perform appropriate conflict resolution. In our remote desktopping model, only one master copy of the VM runs at server (with clients simply mimicking the master copy), thus alleviating such consistency problems. Notice that our model still provides concurrent access from multiple clients by serializing concurrent keyboard/mouse inputs (from concurrent clients) at server.

Because we run a full VM at client, the use of the term "remote desktopping" is deceptive. We use this term for lack of a better alternative, and for its resemblance in user experience to existing remote desktopping tools. Another term to describe our model could be "online VM streaming". Our approach trades off client resources for better utilization of network and server resources. Most studies show that user-facing client computers are under-utilized, and our work uses this under-utilized resource to improve network and server-side CPU utilization. Decentralizing resource consumption is usually a good thing in cloud computing environments.

If the client is untrusted, our approach to stream the memory pages from server to client, can potentially increase security risks, as data that was not supposed to be visible at the client (e.g., pages belonging to another user's process memory), may now become visible. A potential solution is to use trusted software at the client (e.g., through digital signatures). Characterizing the security risks of our scheme requires more work, and we do not address these issues in this paper.

Finally, we have evaluated our ideas only for uniprocessor VMs due to the unavailability of a complete and general method to deterministically replay a multiprocessor VM. There has been significant recent work on deterministic VM multiprocessor replay and we hope that a solution in this space can lend more generality to our optimizations. Our uniprocessor-based adaptive remote desktopping tool is publicly available for download.

7. Conclusion

We present a new method of remotely accessing a server VM using record/replay, and discuss its performance. Record/replay based approach outperforms traditional display based approaches by up to 9x in network bandwidth utilization for display-heavy workloads. We adaptively switch between record/replay and display modes to provide high resource utilization for all types of workloads. Overall, we show that our scheme holds promise, especially for cloud computing environments.

Acknowledgments

We thank Ankur Vijay, Anuj Kalia, Rohan Sharma and Shivam Handa for working on this tool at different stages of its development. The third author thanks NetApp for their support through the Faculty Fellowship program.

References

[1] Our prototype implementation source code. `http://www.cse.iitd.ernet.in/~sbansal/rdrr`.

[2] Remote desktop protocol. Microsoft KB Article Q186607.

[3] Teradici's PC-over-IP. `http://www.teradici.com/pcoip-technology`.

[4] Ricardo A. Baratto, Leonard N. Kim, and Jason Nieh. Thinc: A virtual display architecture for thin-client computing. In *SOSP '05*.

[5] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX ATC, FREENIX Track*, 2005.

[6] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *SOSP '95*.

[7] Citrix independent computing architecture. `http://receiver.citrix.com`.

[8] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 121–130, New York, NY, USA, 2008. ACM.

[9] P.T.A. Marin Lopez and Arturo Gracia Ares. The network block device. In *Linux Journal*, volume 2000.

[10] MokaFive Inc. `http://www.mokafive.com`.

[11] Nomachine NX. `http://www.nomachine.com`.

[12] Qemu: open source processor emulator. `http://fabrice.bellard.free.fr/qemu/`.

[13] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of sun network filesystem. In *USENIX ATC, 1985*.

[14] Daniel J. Scales, Mike Nelson, and Ganesh Venkitachalam. The design of a practical system for fault-tolerant virtual machines. *Operating Systems Review*, 44(4):30–39, 2010.

[15] Robert W. Scheifler and Jim Gettys. The x window system. *ACM Trans. Graph.*, 5(2), 1986.

[16] Brian K. Schmidt, Monica S. Lam, and J. Duane Northcutt. The interactive performance of slim: A stateless, thin-client architecture. In *SOSP '99*.

[17] VMware Workstation 9.0. `www.vmware.com/products/workstation/`.

[18] XtightVNC-Viewer: Virtual network computing client software for X. `http://packages.ubuntu.com/hardy/xtightvncviewer`.

[19] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, and Boris Weissman. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS*, volume 3, 2007.