Live Wide-Area Migration of Virtual Machines Including Local Persistent State

Robert Bradford, Evangelos Kotsovinos, Anja Feldmann, Harald Schiöberg

Deutsche Telekom Laboratories

{firstname.lastname}@telekom.de

Abstract

So far virtual machine (VM) migration has focused on transferring the run-time memory state of the VMs in local area networks (LAN). However, for wide-area network (WAN) migration it is crucial to not just transfer the VMs image but also transfer its local persistent state (its file system) and its on-going network connections. In this paper we address both: by combining a block-level solution with pre-copying and write throttling we show that we can transfer an entire running web server, including its local persistent state, with minimal disruption — three seconds in the LAN and 68 seconds in the WAN); by combining dynDNS with tunneling, existing connections can continue transparently while new ones are redirected to the new network location. Thus we show experimentally that by combining well-known techniques in a novel manner we can provide system support for migrating virtual execution environments in the wide area.

Categories and Subject Descriptors D.4.0 [*Software*]: Operating Systems

General Terms Design, Performance, Experimentation, Measurement

Keywords Live Virtual Machine Migration, Storage, Persistent State, Wide Area, Network Redirection

1. Introduction

VM migration technologies [8] have so far focused only on capturing and transferring the *run-time, in-memory state* of a VM in a LAN, but not its *local persistent state* — a file system used by the VM, which is stored on a local storage device. Handling the latter is necessary [8, 31], among other reasons to support *wide-area* migration, where common network-attached storage, accessible by the source and destination servers, is not available. Furthermore, local storage presents availability, performance, security, and privacy advantages, and is used in practice in a variety of cases — including storing swap files, databases, and local on-disk caches. Also, VM migration across WANs involves a change in the VM's IP address, which breaks existing network associations, e.g. TCP connections, and thus can neutralise the advantages of live migration.

VEE'07, June 13–15, 2007, San Diego, California, USA.

Our contribution is the design and evaluation of a system that enables live migration of VMs that a) use local storage, and b) have open network connections, without severely disrupting their live services, even across the Internet. Our system pre-copies local persistent state — transfers it from the source to the destination while the VM operates on the source host. During this transfer it employs a user-level block device to record and forward any write accesses to the destination, to ensure consistency. A lightweight yet effective temporary network redirection scheme is combined with a quick announcement of the new addresses via dynDNS to allow the VM's open network connections to continue even after migration and to ensure that new connections are made seamlessly to its new IP address.

Our system has been implemented as part of the XenoServer platform [9] and builds upon the Xen [2] facility for memory migration. For robustness and practicality we chose to use standard techniques wherever possible, such as pre-copying, write throttling, and IP tunneling. An extensive experimental evaluation highlights that our system offers an effective solution, featuring:

- **Live migration.** The VM continues to run while transferring its memory and local persistent state.
- **Consistency.** When the VM is started on the destination after migration, its file system is consistent and identical to the one the VM was using on the source.
- **Minimal service disruption.** Migration does not significantly degrade the performance of services running in the VM, as perceived by their users — three seconds for a running web server providing dynamic content to 250 simultaneous clients in the local area, and several orders of magnitude lower than using freeze-and-copy approaches in the wide area.
- **Transparency.** Services running in the migrated VM do not need to be migration-aware in any way, and can be out-of-the-box. The VM's open network connections remain alive, even after an IP address change, and new connections are seamlessly redirected to the new IP address at the destination.

The functionality of our system cannot be achieved with any of the common workarounds for the absence of local persistent state migration: prohibiting VMs from using local storage by mandating that they exclusively use network-attached storage does not support VMs running services that require local storage, nor does it facilitate migration in the wide area. Distributed file systems [12], low-level data transfer protocols [11], and Copy-on-Write-based overlaying [14] techniques require that VMs have their files permanently stored on a remote server, and often present privacy and security challenges. Approaches based on on-demand fetching of persistent state from the source host [26, 15] suffer from complicated residual dependencies.

The rest of the paper is structured as follows. In Section 2 we explain our design choices and architecture. In Section 3 we describe the implementation of our system. In Section 4 we present the experiments undertaken and results obtained, demonstrating that it provides an effective solution. In Section 5 we discuss related work. Finally, in Section 6 we conclude and outline future work.

2. System design

A migration client, running at the source, communicates with a migration daemon, running at the destination, to orchestrate the transparent migration of the VM, its local persistent state, and its network connections.

2.1 Operating environment

Our system is designed to operate between source and destination hosts that use Xen, and are connected by a TCP/IP network. The VM to be migrated is unprivileged, and runs XenLinux — a patched Linux kernel that runs on Xen and includes the frontend block device driver described in Section 3.1. The storage VM is also a XenLinux one and uses *block tap* [33] to export block devices into the VM to be migrated. These block devices are *file-backed*, that is to say the contents of the block device are stored in an ordinary file on the file system of the storage VM. The term *disk image* is used in the rest of this paper to refer to this file. The VM uses these block devices for storing its root file system and swap files.

2.2 Design requirements

The design of our system is driven by the following four challenging requirements.

Live migration Since it is necessary that the migrated virtual machine is not stopped during the migration, our system performs the transfer of the VM's local persistent state in parallel to the continuing operation of the VM.

Consistency To enable the VM to continue operating — and even modify the source file system — while the file system migration is in progress, and at the same time ensure consistency before the VM is restarted at the destination, our system *intercepts* all writes made through the block device at the source. These are applied to the source file system and also sent to the destination host where are later applied to the destination file system.

Our system ensures consistency and prevents *write-after-write* hazards such as *lost updates* (when older writes replace newer ones) in three ways: Firstly, we use TCP for reliable in-order delivery of network packets. Secondly, changes made at the source are not applied at the destination until the transfer of the disk image is complete. Finally, these changes are applied at the destination in the order in which they occurred.

The consistency requirements of our system are subtly different from those of distributed block devices, such as DRBD [25]. Since these are built for long-term mirroring, they need to ensure that the block devices at each end are consistent at all times. In contrast, our system is designed for one-off migration. Therefore, it only needs to ensure that the disk image at the destination is consistent with the one at the source at one point in time: when the migrated VM starts executing at the destination.

Minimal service disruption VM migration can disrupt services — from a user's perspective — by incurring a *high VM downtime*. Our system reduces this downtime by supporting live migration of local persistent state, integrated with the iterative Xen memory-only live migration solution. The temporary network redirection mechanisms we have implemented are simple and lightweight and ensure a minimal downtime. Disruption may also be caused by *contention on resources* such as network bandwidth, processor cycles, and local disk throughput, between the service and the migration process. Our system addresses this by combining *rate limiting* of the migration process with *write throttling*, which is described in Section 3.2.

Migration of local persistent state can be done a) on-demand, where blocks are fetched from the source only when they are requested by applications running in the VM at the destination, or b) using pre-copying, where all state is copied to the destination before the migration is completed. Pre-copy-based migration takes longer to complete, whereas on-demand-fetching results to a longer period of perceived performance degradation by applications running in the migrated VM. We chose the former, as minimising downtime and performance degradation is more important than reducing the total migration time in the application cases we consider. In many situations, and largely so in XenoServer usage cases, migrations can be *planned* — for instance when done for maintenance or hardware upgrades.

Transparency To ensure that out-of-the-box services can be migrated without requiring reprogramming or reconfiguration, and minimise the number of changes or special software modules required, we chose to implement our facility at the block level — instead of the file system level. It does not require that VMs use a certain type of custom file system, and enables migrating any type of file system used by the VM, including swap. At the same time, we avoid the complexity of dealing with the sharing semantics that exist at the file system level.

Our network redirection framework enhances transparency by keeping open connections alive after migration when the VM's IP address changes. It enables a seamless switchover to the VM's new IP address — new requests are automatically redirected to the new address. Thus we do not require the applications to use specific protocols that have built-in redirections [22, 16, 27, 29]. Nor do we require packet rewriting [7], which may cause inconsistencies in the TCP migrated in-memory state that relies on the old IP address. Nor do we need complex and costly software or hardware setups such as load balancers (e.g. Cisco LocalDirector).

2.3 Architecture

Our system operates in the following stages, as shown in Figure 1. First, the *initialisation* stage sets up the ends of the migration process and handles logistics. It then starts the *bulk transfer* stage, which pre-copies the disk image of the VM to the destination whilst the VM continues to run. Then, our system invokes the *Xen migration* interfaces for the incremental migration of the VM's runtime state to the destination, again without stopping the VM.

As the VM continues to run at the source during both the bulk transfer and Xen live migration stages, we need to ensure that any changes to its disk images are forwarded to the destination and applied to the version there. Thus during both stages we *intercept* write operations that occur on the source file system, and generate *deltas* — communication units that consist of the written data, the location of the write on the disk, and the size of the written data. We then *record* deltas in a queue at the destination host for later application to the disk images there. If the rate at which the VM is performing write accesses is too high for the migration to finish — which is limited by the available network bandwidth — we use *write throttling* to slow down the VM sufficiently for the migration to proceed.

After the bulk transfer stage is finished, and in parallel to the Xen live migration stage, our system enters the stage of *application of deltas*, applying each delta enqueued at the destination to the disk image there. For consistency, any new writes that occur during this stage are queued on the destination host and once again applied in order. Towards the completion of the Xen live migration stage the

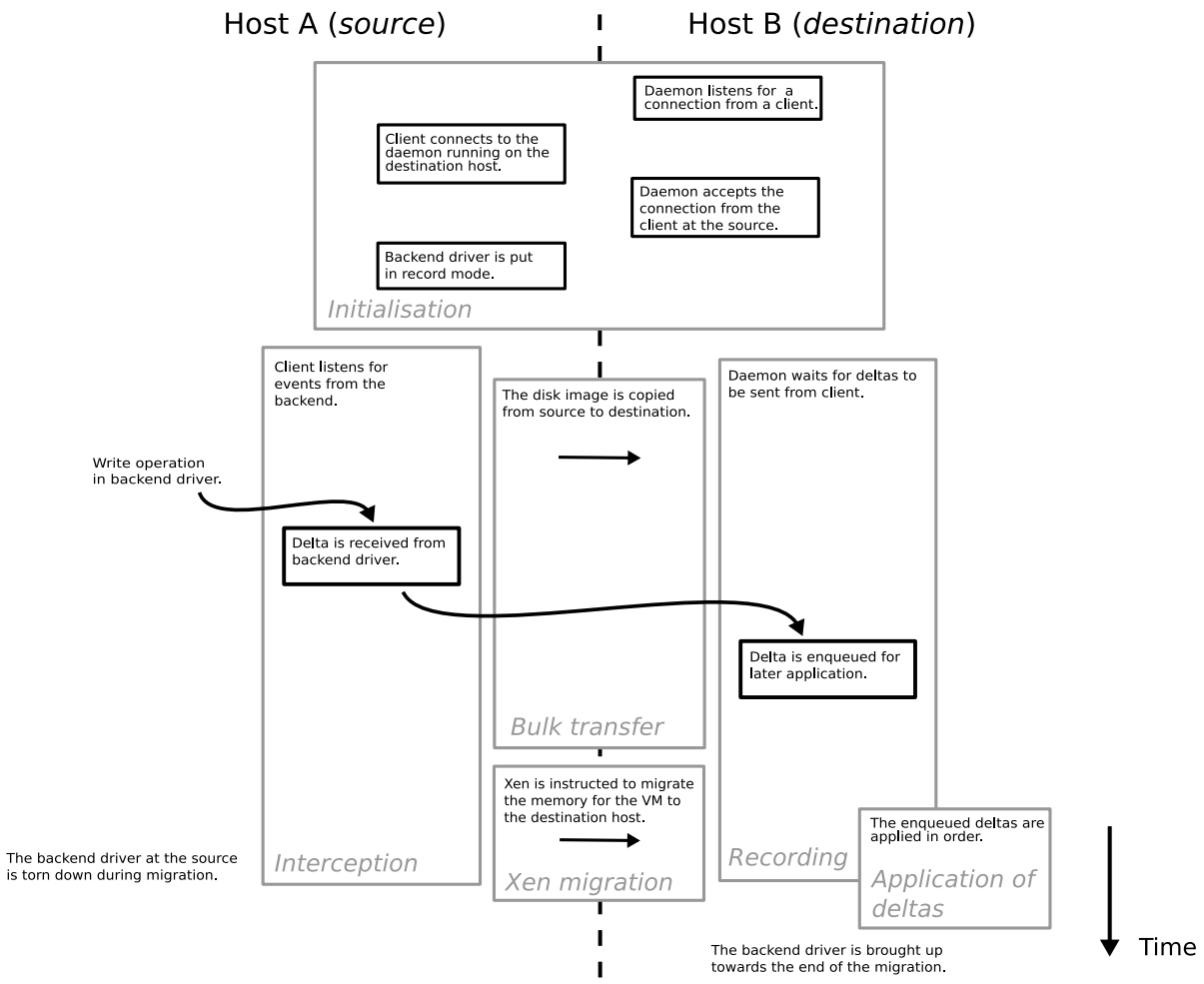

Figure 1. Overview of the migration process (without network redirection and write throttling)

source VM is paused, and its copy at the destination is started. At the same time the temporary network redirection system is started if the VM is migrated across the Internet — if the destination VM has to use a different IP address than the source VM. It keeps open connections alive and ensures that new connections are made directly to the destination VM with its IP address.

3. Implementation

In the following sections we describe how the migration of local persistent state can be realized using standard techniques. We present the concept of write throttling to tackle highly write-intensive workloads, and outline our temporary network redirection framework.

3.1 System operation

This section describes the implementation and operation of the system during the subsequent stages of the migration process. Figure 2 provides an an overview of the system's components and focuses on the handling of I/O requests during migration.

Initialisation Authentication, authorisation, and access control on the requests are handled by the XenoServer platform software [9], which interfaces with the migration daemon at the destination. When a VM migration is requested through the migration client on the source host, the client process forks, creating a separate *listener* process, and signals to the user-space block device driver to enter *record mode*. Once in this mode, the driver copies the details of any writes made through the block device to the listener process, which transfers them to the migration daemon at the destination. Meanwhile, the client process performs the bulk transfer of the disk image to the destination. At the destination the migration daemon process also forks into two for receiving the deltas and the bulk transfer in parallel.

Bulk transfer During this migration stage the system copies the VM's disk image from the source to the daemon at the destination whilst the file is in use by the running VM at the source. It is worth noting that the data transferred during migration does not need to be the entire file system used by the VM. The XenoServers platform uses Copy-on-Write and persistent caching of immutable *template disk images* on all XenoServers, allowing transferring only the differences between the template and the customised disk image

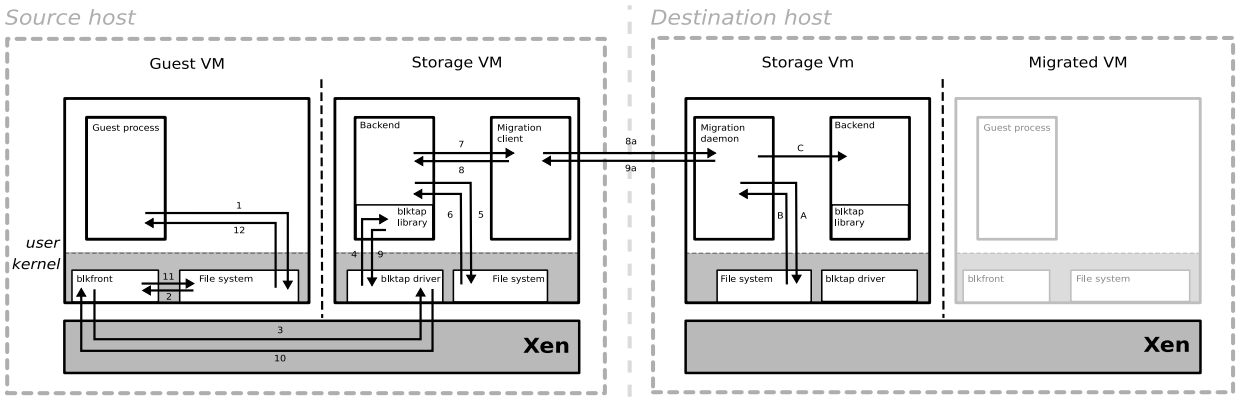

Figure 2. Implementation of our migration architecture

of the VM when migrating [14]. In order to prevent disrupting locality in the VM's buffer cache and ensure performance, the system ensures that block reads bypass the buffer cache during this stage.

Xen live migration After bulk transfer the system invokes Xen's live migration interfaces. Xen iteratively logs dirtied memory pages and copies them to the destination host without stopping the VM being migrated [8]. In its final phase it pauses the source VM, copies any remaining pages to the destination, and resumes execution there.

Intercepting and recording Xen supports a *split-driver architecture* for block devices — a frontend that provides a raw block device in the kernel of the guest VM (`blkfront`) communicating with a backend in a privileged storage VM via a ring buffer. To intercept write operations on the source VM, we use the *block tap* (`blktap`) framework [33]. This provides a small backend kernel module that allows forwarding requests to the backend to a user-space process, which enables us to implement our block device driver in user space for safety and practicality.

I/O requests are handled as shown in Figure 2. When a file system access is made by a process in the guest VM, an I/O request is passed through the file system layer, frontend driver, and `blktap` driver in the storage VM, to the user-space backend using a shared memory interface via a character device (operations 1–4). The backend writes to the local disk image and sends the details of the write operation to the migration client, which packages it as a *delta* and sends it to the daemon (operations 5–9a).

Application of deltas After bulk transfer, and in parallel to Xen live migration, the migration daemon applies deltas to the copy of the disk image at the destination (operations A and B in Figure 2). Once the queue of deltas is empty and the Xen migration process is completed, the destination disk image is in a consistent state. Therefore, our system delays any attempted I/O operations by the VM until that point. Processing of I/O requests is resumed as soon as the daemon indicates the completion of the application of deltas (operation C). In the majority of cases the migration daemon completes its work during the final stage of the Xen migration. Thus, no delay is added to the migration process.

3.2 Write throttling

In order to handle highly write-intensive workloads we have implemented a simple write throttling mechanism. This is based on a *threshold* and a *delay*: when the VM reaches the number of writes defined by the threshold, each further write it attempts is delayed by the delay parameter. Also, the threshold and delay are doubled to slow writes further if the new threshold is reached. In practice, we have found 16384 to be a suitable value for the initial threshold, and 2048 microseconds to be an effective initial delay parameter. We have found this algorithm to be simple yet effective in handling highly write-intensive workloads. The enforcement part of the mechanism is separated from the policy one for extensibility.

3.3 Wide-area network redirection

When migration takes place between servers in different networks, the migrated VM has to obtain a new IP address and thus existing network connections break. Our temporary network redirection scheme overcomes this by combining IP tunneling [21] with Dynamic DNS [34].

Just before the VM is to be paused at the source such that the live migration can complete, we start the redirection scheme. With the help of `iproute2` we set up an IP tunnel between the old IP address at the source and its new IP at the destination. Once the migration has completed and the VM can respond at its new network location we update the Dynamic DNS entry for the services the VM provides. This ensures that future connections are directed to the VM's new IP address. In addition we begin forwarding all packets that arrive at the source for the VM's old IP address to the destination, through the tunnel. Packets that arrive during the final migration step have to either be dropped or queued in order to avoid connection resets. We choose to drop them using `iptables`. After the restart of the VM at the destination the VM has two IP addresses: its old one, used by existing connections through the tunnel, and its new one, used by new connections directly. The tunnel is torn down when no connections remain that use the VM's old IP address.

We have implemented and tested the above solution, and found it to be a practical and effective way of achieving migrating VMs without interrupting their network connections. The performance degradation experienced by the VM is minimal. Our solution requires the cooperation of the source server for only a limited amount of time as most connections are short-lived — for instance those to VMs running Web servers. Furthermore, in the worst-case — if the source server does not cooperate — the approach is as good as Xen migration without redirection.

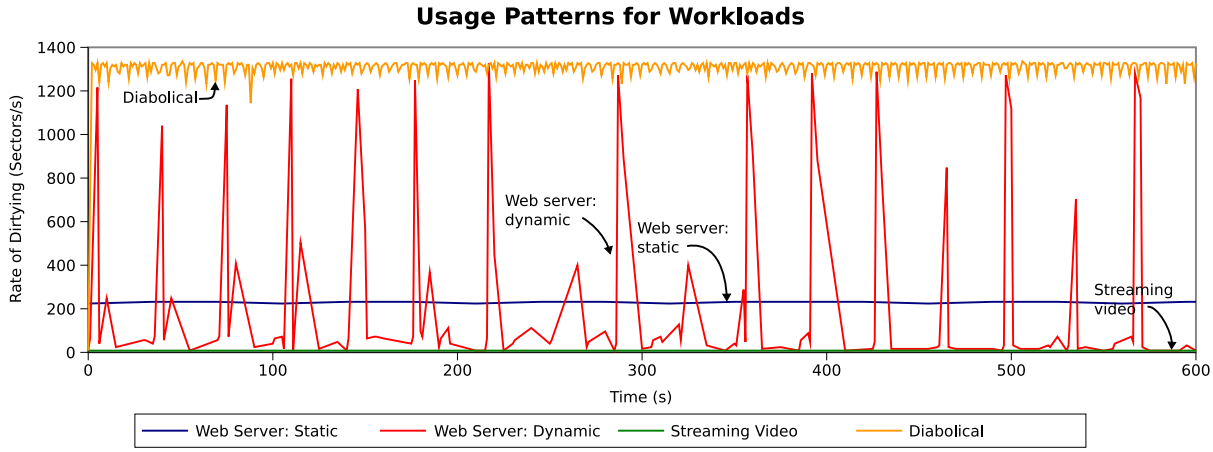

Figure 3. Comparison of the disk write behaviour of the different workloads

4. Evaluation

Our implementation has been tested in a number of realistic usage cases and shown to successfully support live migration of entire VMs including their local persistent state. It allows VMs to continue running during the migration and transparently redirects network connections to the migrated VM at the destination. Furthermore, it ensures that the destination file system is consistent and identical to the one at the source.

In this section we describe the experiments undertaken to quantitatively evaluate the *disruption* that migration causes to the services, as perceived by their users in terms of delayed service responses. Our results demonstrate that our system does not significantly impact running services during live VM migration, outperforming freeze-and-copy approaches by several orders of magnitude. Additionally, our experiments show that the system is able to successfully handle workloads where VMs perform disk writes at an excessive rate — we term these *diabolical workloads*. Furthermore, we demonstrate that the system does not introduce a significant *disk performance overhead* compared to disk access using the unmodified block tap-based drivers.

We define the following metrics: *downtime* refers to the time between pausing the VM on the source and resuming it on the destination, during the final stage of memory migration. *Disruption time* is the time during which clients of the services running in the migrated VM observe a reduction of service responsiveness — requests by the client take a longer time to return. We term the difference between disruption time and downtime *additional disruption*; it is an indicator of the additional disruption caused by our system compared to Xen memory-only migration. *Migration time* is the total time between the point at which the migration is started by a migration request, to the point at which it is completed and the VM is running at the destination. Finally, the *number of deltas* and *delta rate* denote the amount of changes made to the source disk image during the bulk transfer stage, indicating how write-intensive a given workload is.

4.1 Workload overview

We considered three workloads: that of a web server serving *static content*, that of a web server hosting a *dynamic web application*, and that of *video streaming*. These workloads reflect realistic usage cases found in XenoServers, and at the same time neatly trifurcate the spectrum of I/O disk write load, as shown in Figure 3. The dynamic web workload generates a large number of writes in bursts, the static web workload generates a medium, roughly constant number of writes, and the video streaming workload creates relatively few writes but is very latency sensitive. To these we added a diabolical workload, generating disk writes at a very high rate, to test our write throttling facility.

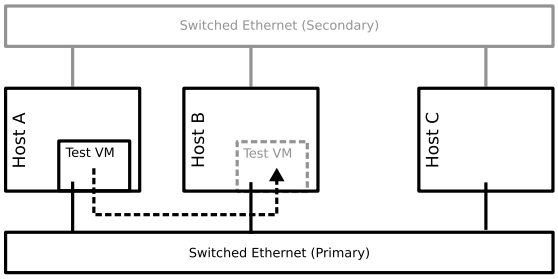

Figure 4. Experimental setup

Furthermore, the chosen workloads are unsuitable for network-attached storage based migration, due to cost — storage server too expensive to run for migrating a web server, and performance — the dynamic web server and streaming server would experience performance degradation by accessing their data over the network.

4.2 Experimental setup

We used the configuration shown in Figure 4 for all experiments described in this section. All hosts were dual Xeon 3.2GHz with 4GB DDR RAM and mirrored RAID arrays of U320 (10k RPM) SCSI disks. All hosts were connected by two separate 100Mbps switched Ethernet networks. Installed on hosts A and B was a snapshot of the Xen unstable release and XenLinux 2.16.13.

The migrated VM was provided with 512MB of RAM and was configured to use a single CPU. The VM used a 1GB `ext3` disk image with installations of Debian GNU/Linux 3.1, Apache 2.0, and all the required libraries and dependencies of the target web application such as MySQL. The privileged storage VMs (VM0) on hosts A and B were using `ext3` disk images and were hosting the backend block device driver as well as our migration client and daemon, as described in Section 3.1.

Workload	Migration time (s)	Disruption time (s)	Number of deltas	Delta rate (deltas/s)	Total transferred (MB)	Overhead
Static Web Content	680	1.04	511	0.75	1643	7%
Dynamic Web Application	196	3.09	1173	5.98	2085	36%
Streaming Video	148	3.04	5	0.03	1546	1%
Diabolical Load	*811*	*–*	*39853*	*49.14*	*1692*	*10%*

Table 1. Summary of results recorded for the different workloads. The diabolical load experiment was the only one performed with write throttling enabled

In all cases the VM was migrated from host A to host B over the *primary network*. Host C was used as the client of the services running in the VM, generating HTTP requests to its services during the migration. These requests and the corresponding replies were transmitted over the primary network in all cases except the freeze-and-copy and wide-area migration experiments — Sections 4.4 and 4.5. During the evaluation period the machines were not used for any service that was not strictly necessary for the experiments undertaken. All throughput, disruption time, and interarrival time measurements were obtained using `tcpdump`. All experiments were repeated several times.

To emulate a wide-area network for the experiments in Sections 4.4 and 4.5, we used the Linux traffic shaping interface[1] to limit bandwidth to 5Mbps and add a round-trip latency of 100ms between hosts A and B. This is representative of the connectivity observed between a host in London and a host in the east coast of the US.

4.3 Local-area migration results

We summarise these results in Table 1. The apparent discrepancy between our results and those reported in [8] is due to the lower bandwidth available in our testing environment — 100Mbps compared to 1Gbps in the experiments conducted by the Xen project team.

4.3.1 Web server: static content

This test measured the disruption caused when running a static web server in a VM on host A, while it was being migrated to host B. Client HTTP traffic was generated on host C using SIEGE[2]. The load consisted of 25 clients simultaneously and repetitively downloading a 512KB file from the web server, making the experiment network-limited. To evaluate the disruption observed by the client during migration, we measured the throughput of the HTTP traffic before, during, and after the migration. We also measured the throughput of the different protocols in use in the different migration stages.

The results of this experiment are shown in Figure 5. Migration starts after 25 seconds, causing a drop in HTTP throughput as part of the network is used for the bulk transfer. After 450 seconds Xen migration starts. Migration terminates after approximately 700 seconds, causing a brief drop in HTTP throughput — and thus an increase of the Xen migration throughput — due to pausing the VM. The deltas generated under this workload are primarily caused by the writing of the log files for the web server and that of other generic system logging during migration. This explains the regularly spaced and small in volume peaks in delta throughput, as the log files are only flushed to disk regularly.

The measured disruption time is *1.04 seconds*, and practically unnoticeable by a human user. The HTTP throughput is reduced during the migration due to the throughput taken by the bulk transfer, as this experiment is network-limited. Note that the amount of throughput allocated to the bulk transfer can be adjusted using straightforward rate limiting.

This experiment demonstrates that our system, integrated with Xen live migration, is able to migrate a running common service (along with its local persistent state) without any noticeable disruption to its clients. Moreover, the additional disruption caused by our system is less than 1% of downtime.

This also highlights the effectiveness of our framework for system maintenance and upgrades. Given a high-availability SLA of 99.999% server uptime, our system allows up to 289 migrations in a year — or nearly six migrations a week, while still satisfying the SLA.

4.3.2 Web server: dynamic web application

For this test we run the `phpBB`[3] bulletin board software in a VM on host A, and evaluated the effect of the migration of this VM to host B. To generate HTTP requests and file system writes, we built a web robot that crawled the bulletin boards at random and, when it came across a form for submitting posts to the board, it submitted an arbitrary post — this happened in 5% of cases. Each such post led to a MySQL query, which inserted data into a database. We run 250 simultaneous threads of this web robot on host C for the duration of the experiment.

The HTTP throughput as well as that of migration protocols was measured as described in Section 4.3.1, throughout the migration. The results of this experiment are shown in Figure 6. Triggered by a migration request, the bulk transfer starts at 25 seconds, with the first large burst of deltas occurring after 62 seconds. Xen migration is invoked after 125 seconds, and the experiment finishes after 225 seconds.

Our system enables the migration of a running web server providing dynamic content backed by a database to 250 clients, with a disruption time of only *3.09 seconds*. This is higher than in the static web workload experiment — as a result of the larger number of deltas due to the bulletin board posts, but it is still low enough not to be noticed by most human users in a web context. Furthermore, given the high-availability SLA described in the previous section, our system allows 98 migrations in a year for system maintenance and upgrade.

In contrast to the static web workload, HTTP throughput is not significantly affected by migration. The reason is that the experiment is not network-limited — there is sufficient capacity for all flows. This is why the total migration time is shorter. The higher variation of the bulk and Xen migration throughput is a result of the workload's bursty nature and the varying size of the web pages downloaded.

This experiment demonstrates that our system is effective in migrating running services that write to the local disk at a higher rate, and in a bursty fashion, without disrupting them severely.

[1] `http://sourceforge.net/projects/tsiface/`

[2] `http://www.joedog.org/JoeDog/Siege`

[3] `http://www.phpbb.com/`

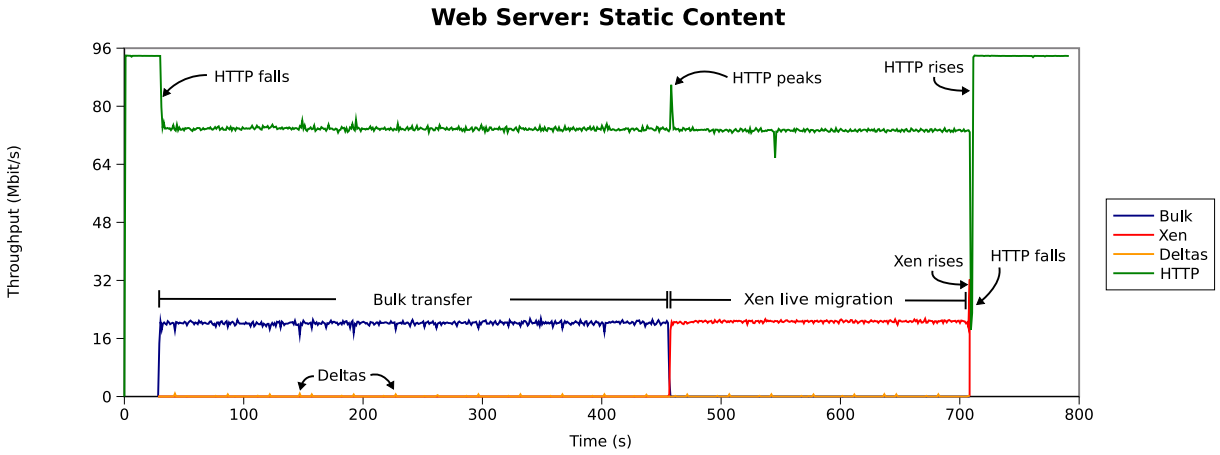

Figure 5. Throughput of migration protocols during VM migration under static web load

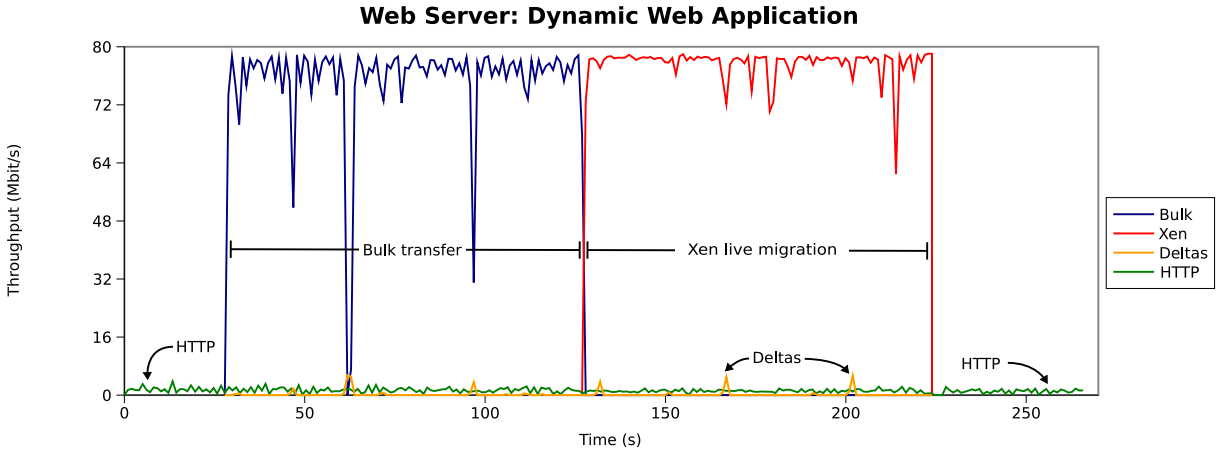

Figure 6. Throughput of migration protocols during VM migration under dynamic web load.

Similarly to the previous experiment, the measured disruption time is nearly identical to the downtime, which is entirely due to the Xen live migration process. This shows that our system provides important additional functionality (migration of local persistent state) without introducing additional disruption.

4.3.3 Streaming video

In the previous scenario, variation in response time due to either the downtime from the final stage of the Xen live migration or from the replay time did not affect the user in a considerable way. Video streaming provides a different workload with few writes, but lower tolerance towards latency. In this experiment we streamed a large video file (245MB) from the VM to a single client in host C, using standard HTTP streaming. The video was viewed on host C by a human user using a standard video player. While the video was being streamed, we measured the throughput of the video traffic and the migration protocols, as described in Section 4.3.1. We also measured the interarrival times of video packets on host C.

The throughput results of this experiment are shown in Figure 7. The bulk transfer starts at 25 seconds and switches to Xen migration after 118 seconds. The relatively constant HTTP throughput is temporarily disrupted after 168 seconds, when the VM is stopped at the source and started at the destination. The disruption time was again minimal (3.04 seconds), and it was alleviated by the buffer of the video player software. The human viewer did not notice any jitter or distortion of the video at any point during the migration of the running streaming server.

This is supported by the recorded packet interarrival times measured, shown in Figure 8, which demonstrate that there is no increase in the packet interarrival times except during the final stage of the Xen migration, when the VM is paused.

In this experiment no packets are lost as we use HTTP over TCP streaming, although we observe the effects of retransmission. Three packets have significantly increased delays followed by some packets with smaller interarrival times. Overall the disruption during that stage is short enough to be absorbed by the client buffer. As in previous experiments, the contribution of the persistent state migration process to the measured disruption time is negligible.

4.4 Comparison with freeze-and-copy

In a scenario similar to that described in Section 4.3.2, with all HTTP traffic to and from host C carried over the secondary net-

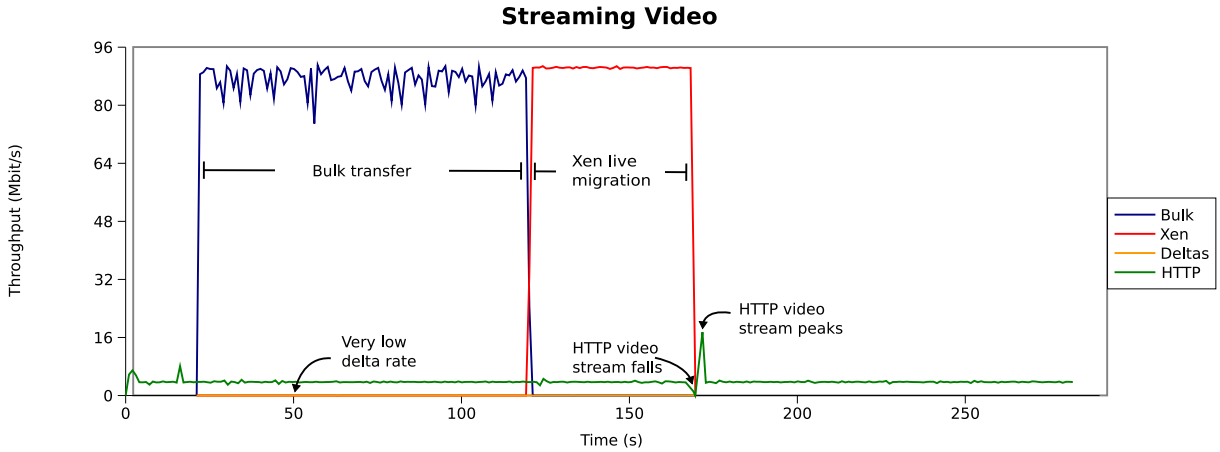

Figure 7. Throughput of migration protocols during VM migration under video streaming load

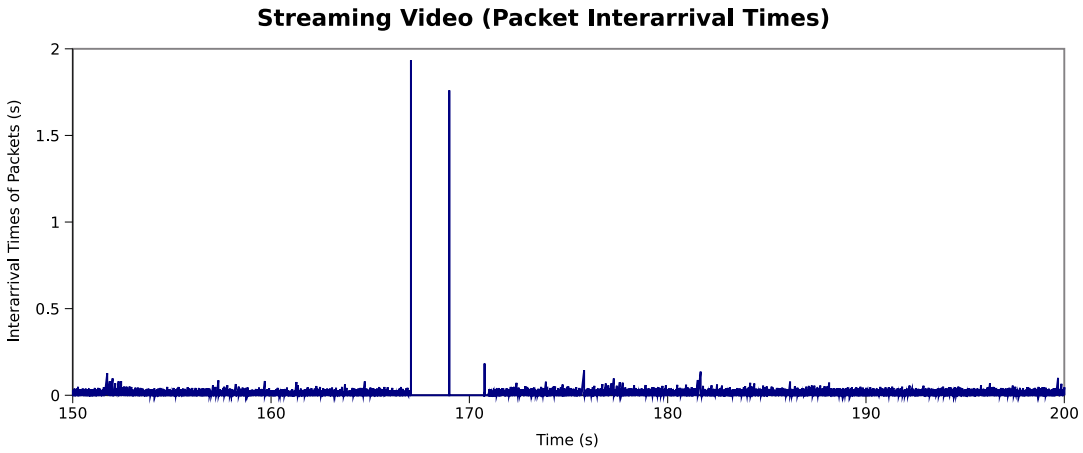

Figure 8. Packet interarrival times during VM migration under video streaming load.

		`scp`	`rsync`	Our system
LAN	Migration time (s)	148.32	228	206.79
	Disruption time (s)	148.32	72	3.12
WAN	Migration time (s)	4585	2160	3777
	Disruption time (s)	4585	279	67.96

Table 2. Comparison with freeze-and-copy approaches, with HTTP traffic carried over the secondary network.

work, we tried two approaches for copying the VM state: in the first one, we stopped the VM on the source host, copied its memory and disk state using the Secure Copy Protocol's `scp` command, and then resumed it at the destination. In the second one, we used `rsync` in the following fashion. First, we synchronised the local disk image with the remote disk image by running `rsync` three times, while the VM was still running — we experimentally found that this number of runs produced the most favourable results for the `rsync` approach. Then, we suspended the VM on the source, storing its memory state into an image file. Subsequently, we used `rsync` to transfer the VM's memory image file and disk image, and resumed the VM on the destination.

The results for freeze-and-copy migration are shown in Table 2. In the local area, `scp` results to a disruption of time of 148 seconds, while `rsync` achieves 72 seconds. Our system significantly outperforms both approaches by disrupting the service for only 3.12 seconds. The slight difference between this result and the one presented in Section 4.3.2 is due to the use of the secondary network for carrying HTTP traffic. In the wide area, the disruption that freeze-and-copy systems cause increases significantly, to 2160 and 279 seconds for `scp` and `rsync` respectively. To demonstrate the ability

of our system to reduce these severe disruption time in wide-area scenarios, we undertook the following experiment.

4.5 Wide-area migration results

This experiment evaluates the disruption caused by our system during a migration in the emulated wide-area environment described in Section 4.2. We again run a web server providing a bulletin board, see Section 4.3.2. However, in this case the HTTP traffic (requests and responses) was carried over the secondary network, in order to isolate the client-side measurements from the effect of rate limiting. This represents the case where the client is at an equal network distance from both the source and destination hosts.

The results are shown in Figure 9. The bulk transfer is started after 260 seconds, and the Xen migration after 2250 seconds. Migration finishes at 3600 seconds with only 68 seconds of disruption time. The much larger number of deltas applied to the destination file system (16787) is due to the lower bandwidth availability: the bulk transfer and Xen migration take longer, thus giving the VM the opportunity to generate more disk writes at the source. Furthermore, using our temporary network redirection mechanism the VM managed to maintain its connectivity to its clients.

Our system reduces disruption time very significantly — see Table 2, by a factor of 67.5 and 4.1 over `scp` and `rsync`, respectively, in WAN migrations. Total migration time is still relatively long (one hour and sixteen minutes) due to the limited bandwidth availability — 1/20th of that in the LAN, but that is significantly less important than reducing disruption time in the applications we consider.

Yet this demonstrates that our system can be used to perform planned wide-area migration of running services and their local persistent state. It reduces disruption by several orders of magnitude and the additional disruption introduced by including local persistent state and network traffic redirection is again negligible — less than 1% of downtime.

4.6 Diabolical workload results

If the VM to be migrated writes to the disk at a very fast rate, it generates a large number of deltas, which consume most of the CPU cycles and network bandwidth. This prevents the migration from progressing. To demonstrate the effectiveness of our write throttling mechanism, described in Section 3.2, we run Bonnie[4] as a diabolical workload generator in the VM.

Figure 10 shows the results of measuring the throughput of the different flows. Initially, the bulk transfer is starved for resources. It only transmits at 32kbps, as most of the available CPU cycles are devoted to processing the writes and deltas. After approximately 325 seconds the VM reaches the first write threshold (16384 writes) and our first-level throttle is invoked. This delays each write by 2048 microseconds and allows the bulk transfer's throughput to increase to more than 20Mbps. At around 670 seconds the second-level throttle is invoked at 32768 writes. Now the write rate is reduced even further, allowing the bulk transfer (and Xen live migration) to continue at a higher pace.

The total migration time using throttling is 811 seconds. Underlining the effectiveness of our mechanism, if the I/O load from the VM was not throttled the total time to transfer the bulk of the data would be just over 3 days.

4.7 I/O performance overhead results

We again run the Bonnie benchmark in a VM on host A, and measured the throughput achieved in four different VM I/O driver configurations: a) the *unmodified* case, where we used the off-the-shelf block tap and backend drivers found in Xen without any changes, b) the *modified* case, where we used our new backend without intercepting writes (i.e. before and after the VM migration), c) the *modified in record mode* case, where our new backend was intercepting writes, and d) the *modified during migration* case, where I/O performance was measured during a live migration of a VM running Bonnie, including its local persistent state.

The results of this are shown in Table 3. Our modified backend introduces an overhead of 0.91% when the backend is not in record mode. This does not observably degrade the performance of the VM and its services. When the record mode is turned on, I/O performance is reduced by 9.66%, which increases to 11.96% during a migration. However, as discussed in Section 4.1, this does not result in a significant disruption of the services from the clients' point of view.

This experiment demonstrates that, even under exceptionally heavy load, the I/O overhead incurred by using our system is minimal, and is only potentially observable during a migration. Combined with the observations of the previous experiments, this demonstrates that we are able to introduce additional functionality (migration of local persistent state) with only minimal reduction of performance.

5. Related work

In this section we position our solution among related systems, which we classify under the following four categories: *freeze-and-copy, remote storage only, remote block devices and on-demand fetching,* and *content mirroring*.

Freeze-and-copy The simplest way to allow the migration of a virtual machine that uses local storage is to freeze the VM — to avoid file system consistency hazards, copy its memory and persistent state, and then start the VM at the destination. For example, the Internet Suspend/Resume project [36] allows suspending a personal computing environment and transporting it to another physical machine, where it is later resumed. For file systems of a realistic size, this results in severe service interruption, as demonstrated in Section 4.4.

Remote storage only Several migration systems [10, 19, 20, 3] can only migrate VMs that a) do not use any local storage, and b) only use storage on a file server that is accessible by both the source and the destination host. For accessing the remote file server NFS [28] can be used in the local area and distributed file systems [6, 12, 37, 30] in the wide area. Peer to peer file systems [1, 4] allow VMs to use a set of unknown remote peers for storing their persistent state.

However, these have far lower I/O performance than local disks, thus are unsuitable for a large number of applications and types of usage — for instance, for databases or storing swap files or caches. At the same time, the storage of VM data on untrusted peers may be incompatible with the trust and privacy requirements of commercial services using the VMs. Additionally, requiring that custom file system software is installed and used by the migrated VM introduces administration overhead. Also, using the above systems for booting VMs is technically non-trivial. Furthermore in platforms such as XenoServers [9], Grids [35], and PlanetLab [23], migration is often used to bring services closer to their clients, in order to reduce round-trip latency. Wide-area communication with a remote storage server significantly degrades service performance, and negates much of the benefit of migration [13].

Remote block devices and on-demand fetching Such techniques could be used to remove the need for migrating local persistent state in advance of completing the migration. The migrated VM at the destination would access its file system exported by the source host over the network [15, 18, 24, 26]. However, using this results in a longer period of perceived performance degradation by the VM, as

[4] `http://www.acnc.com/benchmarks.html`

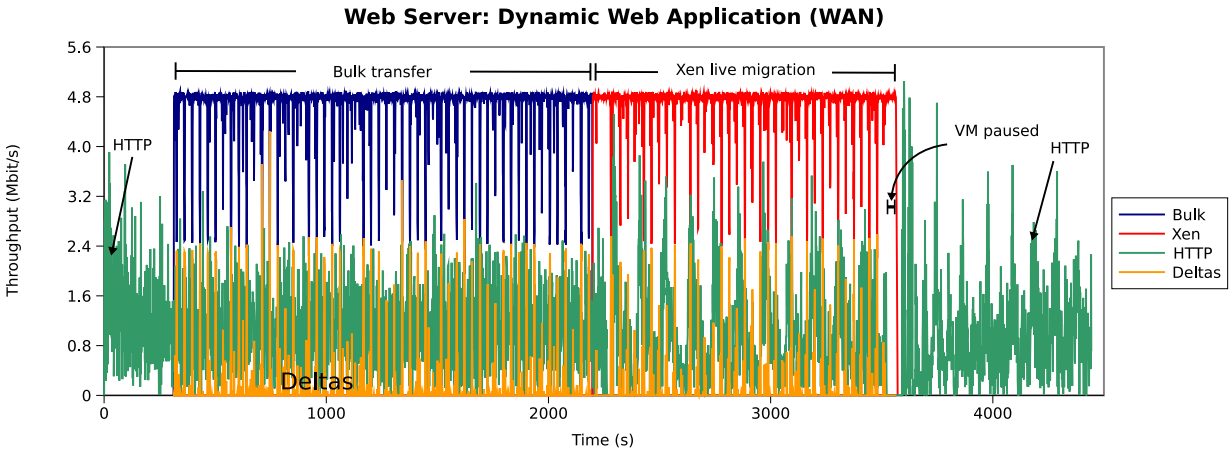

Figure 9. Throughput of migration protocols during wide-area VM migration under dynamic web load

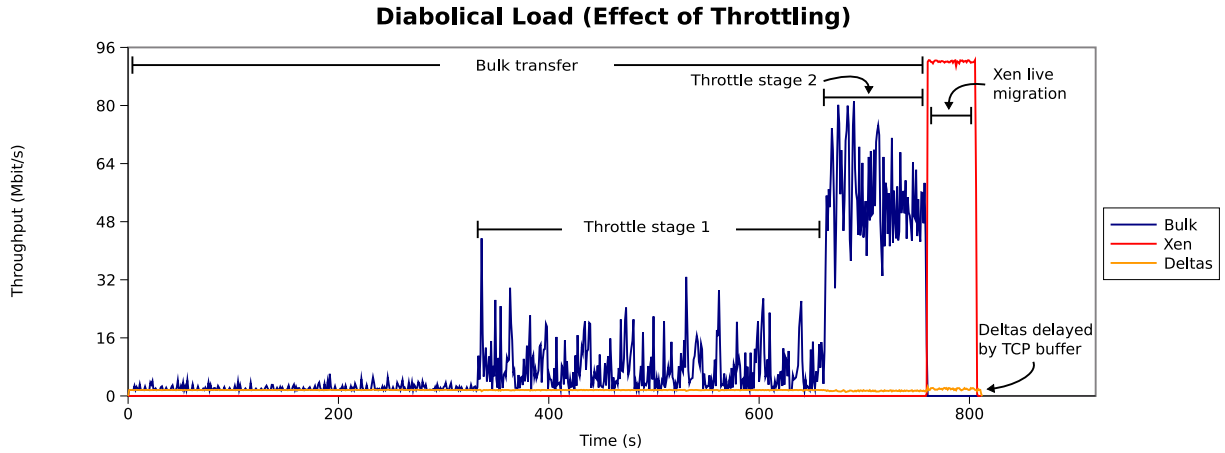

Figure 10. Throughput of migration protocols during VM migration under diabolical I/O load.

Environment	Output throughput (KB/s)	% decrease from unmodified
Unmodified	41462	–
Modified	41081	0.91%
Modified in record mode	37455	9.66%
Modified during migration	36505	11.96%

Table 3. Disk I/O overhead

blocks need to be continuously fetched over the network well after the migrated VM has started at the destination.

Furthermore, it causes *residual dependency* problems: such systems are vulnerable to unpredictable source host unavailability, for instance in the case of a network disconnection or power failure. In effect this makes the VM dependent on the availability of two hosts instead of one, and thus halves the expected time until a host the VM depends on fails.

At the same time, requiring the long-term uninterrupted cooperation of source hosts conflicts with the federated, disconnected nature of platforms such as XenoServers, Grids, and PlanetLab. In such systems servers are under the decentralised administrative control of different organisations and may be unpredictably shut down. Furthermore, the unconditional cooperation of servers cannot be readily assumed due to the lack of direct trust relationships between servers.

Additionally, many solutions that are based on on-demand fetching [5, 25] exhibit I/O performance limitations in the wide-area due to tight coupling of I/O operations to network transmission — thus magnifying the effect of higher network latency. Using such systems in the wide area would incur performance degradation of the VM and its services, which our system avoids.

Content mirroring When services are stateless, an alternative to VM migration is maintaining a static network of servers and using one of the existing content distribution architectures to mirror content [32]. While this avoids complex VM migration, it comes at a high cost: that of setting up, managing, and maintaining a static global infrastructure of mirrors over a long period of time. Furthermore, such platforms are underutilised [17]. By incorporating our migration system into the XenoServer platform we allow the dynamic creation of mirrors when needed, reducing deployment costs and improving server utilisation.

6. Conclusion

In this paper we presented the design, implementation, and evaluation of a system for supporting the transparent, live wide-area migration of virtual machines that use local storage for their persistent state. Our system minimises service disruption — an unnoticeable three seconds for a running web server providing dynamic content to 250 simultaneous clients. It does so by allowing the VM to continue running on the source host during the migration. Additionally, our approach is transparent to the migrated VM, and does not interrupt open network connections to and from the VM during wide area migration. It guarantees consistency of the VM's local persistent state at the source and the destination after migration, and is able to handle highly write-intensive workloads. Furthermore, it does not introduce additional service disruption compared to memory-only migration, nor does it incur a high I/O performance overhead. In the wide-area our system reduces service disruption by several orders of magnitude compared to freeze-and-copy approaches.

In the future, we plan to queue deltas on the source host and send them to the destination host in batches. This will allow optimisations such as duplicate elimination, and improve network performance. At the same time, we plan to integrate data compression facilities in the communication between the migration client and the migration daemon, as well as add better support for sparse files.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *OSDI*, 2002.

[2] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the Art of Virtualization. In *SOSP* (2003).

[3] Douglis, F., and Ousterhout, J. K. Transparent Process Migration: Design Alternatives and The Sprite Implementation. *Software - Practice and Experience* (1991).

[4] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless Network File Systems. In *SOSP*, 1995.

[5] B. Arantsson and B. Vinter. The Grid Block Device: Performance in LAN and WAN Environments. In *EGC*, 2005.

[6] A. Barnhart. The Common Internet File System. *Softw. Dev.*, pages 75–77, 1997.

[7] A. Bestavros, M. Crovella, J. Liu, and D. Martin. Distributed Packet Rewriting and its Application to Scalable Server Architectures. In *ICNP*, 1998.

[8] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *NSDI*, 2005.

[9] S. Hand, T. L. Harris, E. Kotsovinos, and I. Pratt. Controlling the XenoServer Open Platform. In *OPENARCH*, 2003.

[10] J. G. Hansen and E. Jul. Self-Migration Of Operating Systems. In *EW11: ACM SIGOPS European workshop*, 2004.

[11] K. Kim, J. Kim, and S. Jung. GNBD/VIA: A Network Block Device Over Virtual Interface Architecture on Linux. In *IPDPS*, 2002.

[12] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *SOSP*, 1991.

[13] E. Kotsovinos. Global Public Computing. Technical report, University of Cambridge, 2005.

[14] E. Kotsovinos, T. Moreton, I. Pratt, R. Ross, K. Fraser, S. Hand, and T. Harris. Global-Scale Service Deployment in the XenoServer Platform. In *WORLDS*, 2004.

[15] H. A. Lagar-Cavilla, N. Tolia, R. Balan, E. de Lara, M. Satyanarayanan, and D. O'Hallaron. Dimorphic Computing. Technical report, Carnegie Mellon University, 2006.

[16] J. Lundberg and C. Candolin. Mobility in the Host Identity Protocol (HIP). In *IST*, Isfahan, Iran, 2003.

[17] McKinsey & Company, Inc. Two New Tools that CIOs Want, May 2006. In the McKinsey Quarterly.

[18] K. Z. Meth and J. Satran. Design of the iSCSI Protocol. In *MSS '03*, 2003.

[19] M. Nelson, B. Lim, and G. Hutchins. Fast Transparent Migration for Virtual Machines. In *USENIX 2005*, 2005.

[20] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: a System for Migrating Computing Environments. *SIGOPS Oper. Syst. Rev.*, 2002.

[21] C. Perkins. IP encapsulation within IP, 1996. RFC 2003.

[22] C. E. Perkins and A. Myles. Mobile IP. *Proceedings of International Telecommunications Symposium*, 1994.

[23] L. Peterson, D. Culler, and T. Anderson. Planetlab: a Testbed for Developing and Deploying Network Services, 2002.

[24] A. G. A. P.T. A. M. Lopez. The Network Block Device. *Linux J.*, 2000, 2000.

[25] P. Reisner. Distributed Replicated Block Device. In *Int. Linux System Tech. Conf.*, 2002.

[26] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the Migration of Virtual Computers. In *OSDI*, 2002.

[27] H. Schulzrinne and E. Wedlund. Application-Layer Mobility using SIP. *SIGMOBILE Mob. Comput. Commun. Rev.*, 2000.

[28] S. Shepler. NFS Version 4 Design Considerations. Rfc2624.

[29] A. Snoeren and H. Balakrishnan. An End-to-End Approach to Host Mobility. In *MOBICOM*, 2000.

[30] S. R. Soltis, T. M. Ruwart, and M. T. O'Keefe. The Global File System. In *NASA Goddard Conference on Mass Storage Systems*, 1996.

[31] F. Travostino, P. Daspit, L. Gommans, C. Jog, C. de Laat, J. Mambretti, I. Monga, B. van Oudenaarde, S. Raghunath, and P. Wang. Seamless Live Migration of Virtual Machines over the MAN/WAN. *iGrid*, 2006.

[32] L. Wang, K. Park, R. Pang, V. S. Pai, and L. Peterson. Reliability and Security in the CoDeeN Content Distribution network. In *USENIX*, 2004.

[33] A. Warfield, S. Hand, K. Fraser, and T. Deegan. Facilitating the Development of Soft Devices. In *USENIX*, 2005.

[34] B. Wellington. Secure DNS Dynamic Update. RFC 3007.

[35] Foster, I., Kesselman, C., and Tuecke, S. The Anatomy of the Grid: Enabling Scalable Virtual Organization. *Int'l Journal of High Performance Computing Applications* (2001).

[36] Kozuch, M., and Satyanarayanan, M. Internet Suspend/Resume. In *WMCSA* (2002).

[37] Spasojevic, M., and Satyanarayanan, M. An Empirical Study of a Wide-Area Distributed File System. *ACM ToCS* (1996).