Automated Virtual Machine Replication and Transparent Machine Switch Support for An Individual Personal Computer User

by

Beom Heyn Kim

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Abstract

Automated Virtual Machine Replication and Transparent Machine Switch Support for An Individual Personal Computer User

Beom Heyn Kim

Master of Science

Graduate Department of Computer Science

University of Toronto

2011

As the price of computing devices drops, the number of machines managed by an individual PC user has been increased. In many cases, a user with multiple computers wants to use an exact desktop environment across machines. For personal data replication, there are several tools like DropBox, Live Mesh and Tonido. However, these tools can not provide software environment replication. Without a tool to replicate software environment, users have to manage each device's desktop environment separately which is hassling and difficult task to do manually. In this work, we explore the solution to help modern day PC users to perceive the consistent view of not only personal data but also a whole desktop environment. Our approach automatically replicates modifications made on the desktop environment of one machine to other devices using different replication policies for different network properties. Moreover, users can switch machines instantly and still see the exact environment.

Acknowledgements

First and foremost, I would really appreciate to Professor David Lie for his patient mentoring, invaluable advice, financial support, and large amount of time spent on discussions for this research. Also, I am thankful to my fellow graduate students. I would like to thank Professor Ashvin Goel and the members of Computer Systems Lab (CSL) group for their valuable feedback. I am very grateful to my family for their much needed moral support. Finally, I would like to thank University of Toronto and the department of Computer Science for their financial support

Contents

Chapter 1

Introduction

As computing devices are becoming more and more ubiquitous, an individual PC user can have more numbers of machines at the cheaper prices. Hence, it becomes a common PC usage environment where users have one or two desktops at home and one or two laptops for the mobility plus one or two workstations at the workplace. However, this luxurious situation came with the cost of managing desktop environments on different physical machines separately. In many cases, users want to share a consistent desktop environment across different machines they own. For example, a user has an up-to-date Windows 7 environment installed with all of his favourite applications configured with his own settings and preferences. The user has spent much time and put much effort to craft the Windows box to make it secure and handy for his personal usage. Assuming this was installed on one of desktop at home, if the user wants to work with the same computing environment on the workstation at his workplace or on his laptop, the other machines should undergo the same procedure of installation, configuration, data transfer, and so on. Considering frequent updates and short software release cycle, it is heavy burden to users to manage each PC separately.

One way to solve this issue is to have a main desktop environment on only one machine which can be accessed from other devices using thin client products. However,

this approach might be too expensive due to the performance overhead from the slow network. It is critical for interactive application because users can perceive the lag very sensitively. Moreover, putting all important data and software environment on one place makes it a very attractive attack point. Also, this approach has weak reliability when the disk fails without a backup storage. Thus, for personal PC users, this is not an ideal option for the desktop management. Another way to solve this issue is to replicate data across devices automatically so that users can transparently work with the same data from any machine. This approach does not suffer from problems of the foregoing option.

For personal data, the task might be easy since there are several public cloud storage services like DropBox, Live Mesh, Tonido, etc. Users' personal data on one client machine is automatically replicated to other clients. So the data can be fetched from the local disk instead of the slow and expensive commercial network. There is no extra management effort needed other than installing client software and configuring user settings like user ID and whom users want to share data with. These services provide reliability and mobility by allowing data replication to the server managed by the service provider. Also, replicated data on the local disk provides high performance suitable for interactive applications and lets users continuously use data even if the network is disconnected. Yet, those services have several shortcomings. First, they do not replicate the software environment. Second, updates might not be visible instantly because the update has not made to the server yet. Third, they are lack of strict consistency mechanism. So, it can result in multiple versions of a file because concurrent writers can write different contents to the same file at the same time. Therefore, the consistency model of cloud storage services is not suitable for users who want to replicate desktop environment and keep it consistent without conflicting changes.

One might argue that application configuration or personal setting might be replicated to other devices by using application specific plug-in or add-on like Firefox's bookmark synchronization tool. However, these approaches are limited to the certain sets of ap-

plications. Furthermore, this neither reflects updates of software including applications and the operating system nor installs the same version of software on other computers.

Therefore, the tool to automatically replicate the entire desktop environment across user's PCs is needed and it is explored in this work. We are proposing automatic virtual machine replication and transparent support for the instant machine switch for an individual PC user. Although we designed the system to fulfill the purpose described above, only two replication policies have been implemented and evaluated at the moment of writing. We will keep developing the rest of the system and optimize it in the future. In the following section, related works are summarized to give some background. The detailed description of our approach is in the overview section. Then, the implementation section will talk about issues resolved during the development of two replication policies, and the evaluation section will show how these two replication policies affect application performance. Then, we present our future work and conclude.

Chapter 2

Related Work

There are several research works and commercial products which are attempting to provide consistent desktop environment using Virtualization. Numerous Distributed File Systems(DFS) have been built in order to allow remote data access and data sharing among multiple users. Many of them allow clients to cache the replica of the data onto the local disk of client for performance and scalability. Also, DFSs provide cache coherence mechanisms to manage replicas consistently. Some of DFS also provide disconnected operation mode and thus users can use the file system even if network is disconnected or server crash. There is a category of DFS for replicating data across different types of devices or devices with some limited set of hardware resources or network capacity. Cloud storage services can be used not only to replicate personal data but also to replicate virtual disk of the virtual machine(VM) - a file on host OS can be used as a disk to the VM. Mirroring the disk state over the network has been exploited in the area of high availability(HA) cluster systems. In following sections, we would discuss more about these systems.

2.1 Desktop Virtualization

Desktop Virtualization uses the virtualization layers on the desktop PC to decouple the machine state from the hardware. Thus, it can encapsulate and serialize the whole state of the running machine which can be replicated across different devices. There are a couple of previous works done by the research community and several commercial desktop virtualization products.

2.1.1 Internet Suspend/Resume

The goal of Internet Suspend/Resume(ISR) was to provide infrastructure-based mobility to users so that the exact desktop environment can be perceived any time and at any place [25, 23, 13]. Emulating suspend and resume feature of laptops, ISR pursued the goal of seamless mobile computing by allowing users to suspend and resume a VM at different sites. They envisioned there will be computing devices deployed ubiquitously in the future. Thus, users do not need to carry personal devices such as laptops. Instead, they can simply use pervasively deployed computing devices to work with their own customized desktop environments. Because VMs are checked out from a centralized server, users can access their VM location-transparently.

However, if users travel quickly or try to switch machines instantly, users might have to wait until all updated blocks are transferred from the suspend site to the resume site via the centralized server. Also, ISR is not trying to conserve the bandwidth of commercial networks which is expensive. Today, the disk size of the virtual machine is expected to be larger than few gigabytes which was the disk size ISR is tested with. This can lead to longer resume latency and users might have to wait for a longer time than ISR expected. Furthermore, compared to the approach transferring data from the suspend site to the resume site directly, ISR's approach involves additional network bandwidth consumption and longer latency because of the server. Users of ISR should check out VM

before they use it and cache blocks on the local disk. In order to keep the consistency of cached blocks, ISR uses a simple locking mechanism. That is, users are not allowed to check out the VM until the VM, previously checked out on the other machine, is checked in and releases the lock. This can be a hassling task for users if they forgot to suspend the virtual machine before they leave the previous site. Since ISR's goal was mobility, ISR provided a centralized server for users to access their VM from any device. Although this can provide location-transparent VM access, it prevents ISR from employing proactive replication to multiple targets since the resume site might be unknown.

In order to reduce resume latency and slowdown, ISR tries to reduce the amount of data to be downloaded from the server by exploiting portable storage devices like USB keys as a look-aside buffer. So, the changes made by users will be contained in the USB keys and the USB keys can be carried by users[9]. If there is a block is needed to be fetched at the resume site, ISR looks at the portable storage device first and then tries the server next. In order to conserve network bandwidth consumption and improve performance, ISR used collision resistant hash to compare content of blocks. If the hash value of the block on the portable storage device is same as the hash value of the block on the server, the client fetches the block from the portable storage device. Otherwise, it goes to the server to fetch the corresponding block. However, this approach requires users to carry the portable storage device which is contradicting to ISR's original goal which is providing seamless mobility to users without necessity of carrying any personal device.

Unlike ISR, our system's goal is not an infrastructure based mobility but manageability for an individual PC user. Since users have fixed set of PCs, the machines to manage are well-known in advance. So, we could transfer data from one peer to another peer directly without involving a central server. Also, we could do proactive replication to all devices the user wants to synchronize. Moreover, since we have a directory server which knows the location of the blocks that are not yet replicated from other devices,

our system could support transparent machine switch without requiring explicit check-in and check-out commands and users can switch machine instantly.

2.1.2 The Collective

The Collective group also independently came up with the idea of using virtualization to encapsulate state of the running machine and migrate them across different physical devices [4, 20, 22]. While ISR tries to improve mobility for the PC users, the Collective focused on improving manageability for administrators who configured and maintain desktops in the corporate environment. Individual users are having hard time to configure their desktop environment properly, and this leads to the security breaches. On the other hand, managing each desktop by an administrator is overwhelming task. Instead, the Collective eases the burden by having a central repository containing the golden image of VM for consistent software environment, and users will access this copy to use well configured desktop environment while their personal settings and configurations are provided by the Network File System(NFS) server.

They came up with a few interesting optimization ideas for the VM migration [21]. The copy-on-write(COW) disk technique is exploited to keep track of changes from the previous version and transfer changes only. They used ballooning userspace process to reduce the size of the memory state by shrinking the memory size taken by the VM. Also, on-demand block fetching is adapted in order to reduce the resume latency for which users should wait until they can use the VM. Finally, collision resistant hashing is used to conserve the network bandwidth consumption by avoiding sending data unnecessarily.

Although their goal is similar to ours, our system does not have a centralized repository as mentioned above. Moreover, for individual PC users, any of their machines should be able to act like a repository for the golden copy. In other words, users should be able to make changes on any machine and still be able to replicate it to other devices. Furthermore, we do not require one place to have all latest blocks for users to use the

VM. Our system can fetch the latest blocks from any devices.

2.1.3 Commercial Products

XenDesktop is provided by Citrix and built for the corporate environment. Also, VMware's counter part is VMware View. They provide remote desktop, thick client mode and disconnected mode. Other virtualization solutions can be integrated with those. For example, VMware provides ThinApp integrated with View to support virtualized application for better security and flexibility of using legacy apps independent from platforms. They both are based on the client-server architecture and provide the strength of centralized management like the goal of The Collective. The Collective ends up with start up company, MokaFive. MokaFive exploits using personal portable devices like USB to transfer personal data and user settings while the central server provides locked down system disk image. To the best of our knowledge, they do not provide peer-to-peer data transfer or automatic block device replication to multiple devices.

2.2 Distributed File Systems

Since the Distributed File System has the long history, numerously many Distributed File Systems(DFS) have been designed and implemented. We would highlight interesting ones only. We discuss DFSs of different architectures such as traditional Client-Server Architectures, Symmetric Architectures and Cluster-Based Distributed File System. At last, we present distributed storage system for portable devices designed for the environment limited with hardware resources like network capacity or battery, etc.

2.2.1 Client-Server Architectures

Client-Server architecture uses a centralized server to store and maintain shared data. Servers export these data organized in file system tree structure. Thus, clients

import and mount the exported file trees onto the local file system's mount point, and share data across different devices and with different users through the UNIX-like file system interface. Sun Microsystems's Network File System(NFS) is one of the first DFSs and probably the most widely deployed DFS. The initial version of NFS did not have mechanism allowing client side caching on the local disk. The server was considered as the primary location for all up-to-date data and clients have to contact servers very frequently. Hence, the old version of NFS suffers from several shortcomings such as the centralized server becoming a performance bottleneck and single point of failure. More than one server can be composed and collaboratively export a single file system tree in order to distribute the clients' requests to different servers. However, it is possible that one of the servers holds very popular file in the whole filesystem like root directory which will be accessed frequently by many clients in short time. Then, this harmonized group of servers still suffers from the scalability problem. Moreover, partitioning filesystem and distributing subtrees to multiple file servers for good performance and scalability is not a trivial task even for experienced administrators. Furthermore, the NFS was designed for the LAN environment where the network bandwidth is large and the latency is small. Recent version of NFS has added features allowing client side caching on the local hard disk. In addition, it provides callback mechanisms for replica management, although they implemented optimistic policy which can lead to conflicting changes made by write-sharing clients. The recent version of NFS is affected greatly by the CMU's AFS which is described in the following paragraph.

In order to solve the scalability issue, researchers at CMU came up with AFS which heavily utilizes a local persistent storage for client side caching [24]. Since AFS caches a whole file on the local disk, clients' request can be handled locally which results in much lesser accesses to the server. Hence, the scalability was greatly improved and performance could be enhanced so that AFS could support thousands of clients. Nevertheless, as the number of users and machines AFS can support increases, the network failure or

server failure became more frequent and critical issues. This leads to the development of Coda distributed file system with disconnected operation mode for weakly connected networking environments [3]. Coda also provides trickle reintegration which allows clients to commit changes to the server steadily and gradually without significantly hurting the performance of the client. Moreover, Coda supports transparent server replications to provide high availability by servicing clients from an alternative server in the event of the server crash. For systems like AFS or Coda which allows client side caching, replica management policy is an important design issue. AFS once implemented token-based pessimistic cache consistency model which enforces no sharing of a file while one writer is writing to the file. Although this can provide much stronger consistency model than the optimistic ones like Coda's, communication bandwidth for protocol related message exchange can be expensive and performance can be decreased as the number of sharer increases. However, researchers found that for DFS's target environment where desktops sharing data through the DFS the update rate is low and data sharing is usually sequential but rarely concurrent [11]. Moreover, for DFS like Coda, it is difficult to implement efficient pessimistic cache consistency model with disconnected clients. Thus, the recent version of AFS and Coda implement optimistic cache consistency model.

These DFSs are providing the repository for clients so that they can read, write and share files. However, our system does not just providing shared information but also it actively replicates to each device. In terms of cache consistency, we adapt directory based cache consistency protocol which is pessimistic protocol. Since we are dealing with disk consistency and disk corruption can be much more severe damage than a file corruption, it is better to choose more conservative approach then optimistic consistency protocols. Since our system targets the environment where there is no concurrent sharer, protocol related message exchange would be much lesser frequent than AFS's old pessimistic cache consistency protocol where multiple sharers can ping-pong the exclusive ownership of files.

2.2.2 Symmetric Architectures

xFS, is designed to improve scalability of DFS by removing the centralized server and distribute server's responsibility to clients [2]. xFS implemented pessimistic cache consistency model inspired by the directory based cache consistency protocol. This cache consistency model uses invalidation to guarantee one exclusive writer and no data sharing while writing. Also, the metadata manager directs client to the peer holding the latest data. It distinguishes its cache consistency model from AFS by managing consistency at the granularity of a block rather than a file. Thus, this consistency model becomes useful to increase scalability by striping a popular large file across different servers and allowing access to different parts of the file concurrently. Also, xFS uses cooperative cache which allows fetching cached data from peers' memory cache prior to fetching from a persistent storage, because LAN can provide faster data fetching than a hard disk. Since there is no centralized server, the scalability could be improved so that throughput increases linearly with respect to the number of clients. However, xFS is targeted only for LAN environment. Moreover, frequent message communication with expensive networking protocols limits the performance of xFS.

Many of peer-to-peer(P2P) distributed storage systems were proliferated in early 2000 [7, 14, 19, 26, 16, 1]. The main advantage of P2P system is scalability through the symmetric architecture and high availability through redundancy of data. OceanStore is built envisioning there will be a need for the global-scale storage shared across the globe. It aims global-scale storage utility envisioned to scale up to 10 billion people with 10 thousands files per person. Since OceanStore is targeting global-scale, it is important to replicate data on servers distributed across the globe. This can complicate cache management but provides availability. OceanStore tries to decouple data from physical location and let it flow through the servers depending on data access patterns of users. Since it is system shared by multiple users, the data is encrypted and allows write only to allowed users based on ACL.

CFS is a scalable and robust p2p read only file system which is utilizing Chord, a scalable lookup system, and DHash which is distributed block storage. Applications using CFS will interact through UNIX-like interfaces, while DHash managed the blocks and Chord looks up the location of the requested block. For large scale systems, network topology is never static but keeps changing. In CFS, underlying Chord manages dynamically and implicitly the mapping from the block identifier to the server holding the requested block. DHash replicates files in advance over different servers for load balancing and high availability. Only the publisher can update data while others can only read. In terms of the concept of single writer, CFS is similar to our system. However, they do not try to replicate entire file on the local disk of each peer, but rather distribute blocks over servers. Fetching blocks for cache miss is adding the performance overhead.

While CFS is read-only file system, Ivy is p2p distributed file system for both read and write accesses. It is another P2P system built on top of DHash and Chord. Ivy implements optimistic cache consistency model and uses a per-participant log to support conflict resolution after network partitioned and disconnected peers made conflicting changes. Performance was two or three times slower than the regular NFS. The main performance bottleneck was network latency and the cryptographic operation to generate digital signature. The cryptographic operation was needed, since Ivy client does not fully trust each other and DHash server might be corrupted.

Farsite is a serverless DFS that logically functions as a centralized file server but physically dispersed across networked desktop PCs. Farsite uses cryptographic techniques to secure user data, randomly replicate data for higher availability and reliability and Byzantine-fault-tolerant protocol to maintain integrity of file and directory data. It is designed for desktop I/O workloads but not for high performance I/O of scientific applications or the large scale write sharing of database applications. Administration effort is minimal which is mainly signing certificates such as Machine, user, namespace certificates. Also, Farsite was designed with the intention to emulate the traditional local

file system's behaviour, typically NTFS.

Unlike many P2P systems, we have a directory server which keeps track of locations of blocks. Also, we cache the whole VM instead of just distributing it over servers. Moreover, we prefer pessimistic consistency protocol than the optimistic ones.

2.2.3 Cluster-Based Distributed File Systems

There are some DFS modified the traditional Client-Server architecture to make DFS suitable for server clusters which are often used for parallel applications [30, 8]. Google File System(GFS), Ceph, Parallel Network File System(pNFS) are allowing clients parallel accessing to a pool of persistent storage devices. In order to improve performance for their target applications, file-stripping techniques are used in those cluster-based DFSs. File-stripping techniques distribute a single file across multiple servers, and enables fetching different part of the file in parallel. For large server clusters, clients read and write files across thousands of machines, and files are usually very large, easily ranging up to multiple gigabytes. Files contain lots of smaller objects and updates to files are usually appending rather than overwriting. For large scale systems like Google's infrastructure, it is relatively frequent that machine crashes. GFS constructs clusters of servers that consist of a single master and multiple chunk servers. The master is only contacted for the location information of the required part of file which is chunked and distributed across chunk servers. Chunks are replicated across different chunk servers for high availability. Since the master node's load is only metadata management, it can scale much better than traditional client-server architectures.

We do not use striping technique. Also, our system is not for a larger cluster environment, but for a small group of PCs owned by an individual.

2.2.4 Miscellaneous Distributed Storage Systems

Data on various devices like consumer electronics or portable storage devices are manually replicated and synchronized, but this is time-consuming task and hard to keep all replicas consistent by hand. Also, they are restricted with hardware resources like network bandwidth or battery life. Thus, this leads to many researches to develop distributed storage system for replicating data across devices in the restricted environments. These works resemble to our work in the sense that it helps users to replicate data across different devices automatically with the network capacity limitation and some of them support the disconnected operation mode.

Bayou [27] is designed for applications to replicate data across servers and it allows concurrent accesses by multiple applications. When many clients concurrently read and write with weakly connected network, it is unavoidable to have confliction. Bayou is an infrastructure for confliction management introduced by concurrent data accesses of applications on mobile devices such as laptops or PDAs. Authors claim that they do not have the notion of disconnected mode, because Bayou allows devices to communicate in pairwise even if they are disconnected from other devices and propagate changes between them. Bayou store a full collection of data on several servers, and clients can access different servers with session guarantees [28] to reduce possible inconsistencies perceived by applications. Once done with write, each client does not care whether updates have been propagated to other servers or not. Bayou servers tend to move toward eventual consistency which means writes will eventually propagated to all servers through intermittently connected network connections. Bayou provides support for the application specific confliction detection and resolution.

Footloose [15] tries to let users to manage data replicated across computing devices such as cell phones, PDA, laptop, and desktop PCs. They assumed devices are connected to some devices but not all other devices. Also, mobile devices will travel along users and connect to different devices from place to place. So they are used as the data carrier.

So they replicate data from device to device whenever devices are connected towards physical eventually consistency. For conflict resolution, they let the conflicts to flow to other devices which can resolve them.

BlueFS [17] is designed to manage personal data replicas distributed across different consumer electronics devices at home. Those devices have limited battery life, and BlueFS tries to provide an energy-efficient distributed storage system for them. The system lets devices use persistent storage to cache replicas for both performance and network disconnected operation. There is a centralized server which has reliable network connection, and the server keeps the primary copy of each replicated data for the high availability. Also, the central server provides the reliability when devices lost or stolen. They allow a client to fetch data from the replica on the location where the power consumption can be minimized. BlueFS asynchronously update the changes to the server and other peers across diverse devices. BlueFS implements optimistic cache consistency model similar to Coda's and uses callback mechanisms similar to AFS's for cache coherence. They maintain callbacks per-device and queues the invalidation messages in order to support portable media which frequently hibernate for energy conservation. BlueFS keeps the version number for confliction resolution. By keeping version number which increases at every write access, the server can easily detect confliction by looking at the version number. If the version number is one more than previous version number, that means there was no confliction. Otherwise, it implies that the conflicting changes have been made by clients.

EnsemBlue [18] showed how to integrate consumer electronics with distributed file system. EnsemBlue is based on BlueFS and added extended features. The persistent queries were supported by the EnsemBlue to customize the file system behaviour for each different CED. CEDs are heterogeneity and can only work with certain file format. EnsemBlue also provides the namespace diversity to support different file organization schemes on different CEDs. Moreover, EnsemBlue allows a set of disconnected devices

to form an ensemble to share data with each other and with a pseudo-server, castellan. The castellan keeps track of the cache contents of each member of the ensemble and lets a client to fetch data from the proper peer in the event of cache miss. Also, it receives changes made by a client and propagates the changes to clients.

Our system is targeting the environment where network connection is limited but not disconnected frequently. Even if we support disconnected mode, the system should be explicitly informed by the client going into the disconnected mode. Also, we provide more pessimistic cache consistency protocol than optimistic ones. At last, we do not want to distribute latest blocks over the devices, but rather want to keep all latest blocks replicated on every device. Even if it is not possible to keep all replicas on each device up-to-date all the time, we try to make the whole VM image up-to-date on each device eventually.

2.3 Cloud Storage Service

Cloud Storage Services like DropBox, Live Mesh and Tonido are more and more widely used by users who want to replicate personal data to other machines. Cloud Storage Services can be used to synchronize the virtual disk of VM by synchronizing a file on the host OS backing the VM's virtual disk. However, because these tools provide eventual consistency of shared data, consistency of virtual disk can not be guaranteed.

Dropbox provides a folder to the user and the user can place files he wants to replicate. Files in this folder are replicated to the DropBox server and then to other devices. In order to conserve network bandwidth consumption and improve speed to replicate data, Dropbox supports Peer-to-peer mode. However, files must be synchronized with file server before replicated to other devices. Also, peer-to-peer mode is only for LAN environment. It leaves a snapshot of a file every time users save changes to allow users to rollback to a previous version. If conflicting changes are made concurrently, one of them is kept as

the new version of the existing file and other changes will create different files marked as conflicted version in its file name. Moreover, it synchronizes data at the granularity of file. Also, storing data or disk replica on the third party server might compromise one's privacy.

Live Mesh is similar to DropBox, but it has interesting features itself such as let users synchronize Microsoft's application user settings. Live Mesh also has peer-to-peer data synchronization feature as well. Unlike DropBox's LAN sync feature, Live Mesh does not need to synchronize data with the central server. Thus, users can be protected from privacy issues with this feature. If users want, they can replicate data to the server, sky drive, for reliability and availability at the cost of privacy. Unlike DropBox, they do not leave snapshots of different versions of a file. So they do not provide means to revive a file accidentally erased.

Tonido is a P2P cloud storage service. Although it also provides Tonido server for high availability and reliability, its primary goal is providing the basis for forming personal cloud storage.

These systems are mainly for file replication. So, these solutions implements optimistic consistency protocol and there is possibility of conflicting changes to occur. However, we want more strict consistency mechanisms which can be used for VM replication. Also, these systems do not automatically replicate changes unless users explicitly save the modified files. Yet, we automatically and transparently replicate changes.

2.4 Networked High Availability Cluster System

There is a previous work for replicating the whole VM including all states of running computer to create backup image for High Availability. Remus replicates the snapshot of the entire running machine as fast as 25 milliseconds. It delays releasing output until the snapshots is completely replicated to the secondary node, backup machine. Since

this delay is very short, the client would not perceive it. However, they are targeted for HA cluster consisted of two nodes and works for the LAN environment.

DRBD is a virtual block device that replicates every disk write to the backup node over the network. Although this replicates disk state, it can not replicate the CPU and memory state of the running machine. It is also designed for LAN primarily, and it only works for two nodes.

Since systems for HA clusters usually include only two nodes and they aggressively replicate each change, different properties of network is not considered. This is because their main purpose is to provide high availability not manageability. Thus, their approach is not suitable for our purpose, because it does not consider network capacity of WAN and its cost. Our system does not completely give up the high availability though. It tries to replicate aggressively for peers in LAN environment so that reliability through redundancy can be achieved.

Chapter 3

Overview

In this work, we propose an automated VM replication system to provide users exact desktop environment for manageability and mobility. Virtualization technologies have been used to encapsulate the entire desktop environment including personal data, software environments as well as hardware states of the running machines. Each device has a replica of an entire VM on the local disk for performance, reliability through redundancy and disconnected operation. Our system provides a cache consistency mechanism for connected clients by using a meta data server which collects cache state of each device and provides location information of the latest blocks. So any disk block change made on any device is visible immediately on any other device. Therefore, we can allow users instant switch of machines transparently. Although our cache consistency mechanism requires a central server to manage proper mapping information, block contents are transferred in P2P manner. This approach saves network bandwidth consumption. Also, the server is only responsible to maintain the meta data used for the consistency protocol during the system initialization. Hence, the server is lightly loaded and any PC can be used as a server machine. Our system is aware of network properties between peers and incorporates different replication policies for different connections.

3.1 Limitations

Currently, our approach has several limitations. It works with devices with similar architecture only because of the limitation of currently existing VMMs. Currently, it is not transparent to do suspend and resume between different hardware architectures, because of different instruction sets supported by different CPU families. We do not try to fix this problem here, although we expect that there will be some solution in the future. Although we provide a consistency mechanism for connected clients, our system can not prevent the confliction while clients are operating in disconnected mode. Our current implementation can handle only two nodes in WAN environment because the directory server has not been realized yet. Also, disconnected mode, network conservation and the fault tolerance are not yet implemented either. At the time of writing, only the LAN replication and WAN replication policies are implemented.

3.2 Assumptions

We assume a user owns multiple PCs such as desktop, laptop, netbook or tablet, and tries to manage them with a single consistent desktop environment. Each computer will be connected through a gigabit Ethernet for LAN, and 7 Mbps commercial network for WAN. It is common that the gigabit Ethernet is used in a LAN environment these days. On the other hand, the commercial network for a WAN environment is limited to 7 Mbps upload speed today and has longer network latency. These commercial networks are asymmetric and download speed is usually much larger. However, transferring data between PCs over WAN is limited by the upload speed. As our key insight, we assume only one active node at any given time. Since our target environment is only for a single user, it is reasonable to assume that there is only one computer that is being used by the user. That is, VM state is accessed by only one computer, primary node, at any given time. Other nodes which are not running VM are defined as secondary nodes. Also, our

system works in the environment where network connections are stable and there is no frequent network failure or machine crash.

3.3 Disk I/O Bandwidth Measurement

In order to see how much disk I/O bandwidth is consumed by the VM running a regular desktop environment, we ran some disk I/O intensive applications normal desktop users would use and measured disk I/O statistics using iostat utility program. iostat reports CPU rate and device I/O rate during the given interval as many times as a user specified. A user can adjust time window and see the average I/O rate per interval of the given duration. Since we were interested in how many disk I/O requests are received by the block device used as the VM's disk and we knew that the clock in VM is skewed, we ran iostat on the Host OS for the primary partition used by VM instead of running it inside the VM. We were typically interested in the feasibility of the aggressive replication policy, the policy where we replicate each disk write, for LAN and whether it is possible to be applied to WAN environment as well. This would depend on the disk dirty rate because it shows how fast we must replicate. To stress disk heavily, we tried to pick some disk write intensive jobs. Although we have not considered benchmarking tools and other unusually disk intensive tasks, we tried to test the normal desktop applications often used by PC users.

Usually decompression and installation of software shows the high disk dirty rate, and in order to generate swapping more frequently we tried to run multiple processes at the same time. We created a Windows XP guest VM on VirtualBox 3.2.10 client hypervisor. We picked VirtualBox because it is an open source, allows VM to use raw hardware partition directly and provide the teleporting feature. Seven Internet Explorer processes were executed and one of then was running YouTube movies. Concurrently, we installed a BitTorrent client program and downloaded the Torrent file for Eclipse IDE.

Then, we downloaded an eclipse zip file using the torrent, decompressed the zip file and installed the Eclipse. Afterwards, we measured disk I/O consumption for ten consecutive intervals each of which is 5 minute long. The result is provided in the Table 3.1. Based on our assumption, the bandwidth for the LAN is 1 Gbps which is 125 MB/s and the bandwidth for WAN is 7 Mbps which is 0.875 MB/s. The max dirty rate average was 2880.57 blocks/sec in the interval 4 which is about 1.407 MB/s, considering the block size is 512 MB in the iostat's result. Even if we considered the overhead of TCP protocol, it is feasible to aggressively replicate to multiple peers within LAN environment. However, for WAN environment, the disk dirty rate exceeds the network bandwidth sometimes. Considering we want to support multi-target replication, aggressive replication can not be used for WAN.

Table 3.1: Disk I/O Bandwidth Consumption with 5 min intervals

Interval	Blk_read/s	Blk_wrtn/s	Blk_read	Blk_wrtn	Dirty Rate(MB/s)
1	32.50	45.21	9750	13563	0.022
2	1637.75	805.79	491424	241784	0.393
3	1136.21	897.97	340864	269392	0.438
4	2728.21	2880.57	818544	864256	1.407
5	1756.97	1831.92	527160	549648	0.894
6	778.85	610.17	233672	183064	0.298
7	102.63	60.18	30792	18056	0.029
8	43.25	26.50	12976	7952	0.013
9	176.41	1405.49	52928	421688	0.686
10	96.58	321.82	28976	96552	0.157

Next, we tried to see how disk dirty rate changes depending on the length of the interval when decompressing a large file. We tested with two different intervals, 5 sec and 90 sec. The result is presented in the Table 3.2 and Table 3.3.

Table 3.2: Decompression disk I/O measurement with 5sec intervals

Interval	Blk_wrtn/s	Blk_wrtn	Dirty Rate(MB/s)
1	82.45	412	0.040
2	27.31	136	0.013
3	0.00	0	0
4	6.40	32	0.003
5	7646.22	38384	3.734
6	28988.80	144944	14.155
7	20162.87	101016	9.845
8	13422.04	66976	6.554
9	22707.20	113536	11.088
10	19402.79	97208	9.474
11	20643.20	103216	10.080
12	10179.20	50896	4.970
13	2870.68	14296	1.402

The table 3.3 shows that 3.716 MB/s was max dirtying rate for 90 sec interval, and the table 3.2 shows that 14.155 MB/s was the max dirtying rate for 5 sec. The decompression task took around 1 min and 8 sec and we did not run any other disk intensive applications. So both measurements cover the disk I/O rate of the decompression mainly. The average dirtying rate was lower as longer the period of the measurement is. During the decompression, the steep dirty rate curve was observed in some intervals, but in other intervals it showed much lower dirty rate than the peak dirty rate. It is true not only for the decompression but also for mixed workload. If we aggregate interval 3 and 4 in table 3.1, then the average dirty rate goes down to 0.923 MB/s. This implies we can replicate at lower rate than the peak dirty rate but over the longer period of time for WAN. Because WAN is slow the replication rate over WAN can not catch up the dirty

Table 3.3: Decompression disk I/O measurement with 90sec intervals

Interval	Blk_wrtn/s	Blk_wrtn	Dirty Rate(MB/s)
1	61.38	5524	0.030
2	7611.06	685376	3.716

rate. However, disk intensive work does not last long and the disk will be lightly loaded for the rest of time. Thus, we can replicate blocks over the long period of time gradually to achieve eventual consistency. One problem with this lazy replication policy is that not all of latest blocks might be replicated to the machine by the time the user tries to use it after traveling. As a fallback, we provide on-demand fetching and background synchronization based on the directory based cache consistency protocol.

In short, our measurement showed that it is feasible to aggressively replicate each disk write to multiple machines connected with 1Gbps network, and it is better to provide eventual consistency for peers connected over WAN while the disk is being idle or lightly loaded.

3.4 System Architecture

System Components Our system has three components a userspace utility, kernel module and directory server. Figure 3.1 displays these components, their interactions with each other and interactions with the block device driver and other peers. First, the userspace component is to provide user-defined configuration parameters to the kernel module so that it can be initialized properly. For example, IP address of the peer should be specified in the configuration file. Then, the userspace utility will reads configuration information and pass it to the kernel module so that the network connection can

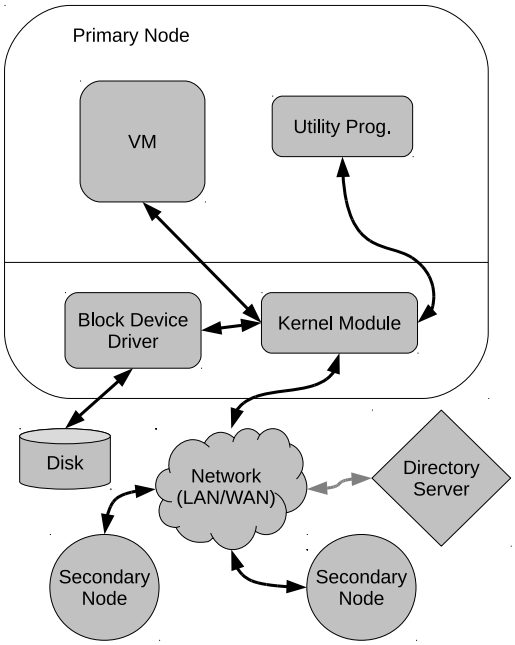

Figure 3.1: System Architecture

be established with the correct peer. Also, other parameters like a primary disk partition to be replicated and synchronization rate should be specified in the configuration file and passed to the kernel module by the userspace utility during the kernel module initialization.

Second, the kernel module is the component actually participating in the replication. It is sitting between the file system of guest OS and the backing block device driver which is the block device driver for the physical hard disk. The VM thinks the kernel module as its block device and sends disk I/O to the kernel module component. Then, the kernel module component receives disk I/O from its upper layer, VM, and sends the disk I/O to the backing block device driver. At the same time, the kernel module sends replicated data to peers over LAN connections. For WAN peers, the kernel module simply forwards disk I/O without sending replicated data. Then, dirtied blocks will be

replicated when the VM does not get any input from the user - pushing synchronization. This pushing synchronization might not be able to catch up the dirty rate as we saw in the disk I/O measurement section. Thus, the kernel module on the new node runs background synchronization - pulling synchronization - with on-demand fetching. The locations of latest blocks are known to the directory server, and the kernel module can obtain this meta data from the directory server. While the pushing synchronization is controlled by the kernel timer, pulling synchronization is continued until all latest blocks are replicated on the local disk. This is because to reduce the chance of cache misses which cause significant performance overhead.

Third, the directory server is used mainly for directory based cache consistency protocol. It is expected to be lightly loaded because it manages meta data only and the message exchange for consistency protocol in our system is not as frequent as the shared memory multiprocessor architecture due to our single primary assumption. Essentially, it takes care of the location information for each block and provides the knowledge of where the latest version of the block replica is stored. The directory based cache consistency protocol is a pessimistic consistency protocol because it does not allow both read and write sharing of data while the data is being written by one entity. Therefore, the directory server contributes to minimizing the disk corruption which is much more severe issue than file inconsistency in distributed file systems. The directory server can also keep track of which machine is being used by the user by storing the machine's ID when the user starts running VM.

When user runs the start-up command of our system, the kernel module will be configured and initialized by communicating with the userspace utility which provides knowledge about the user's configuration file. Then, VM configured to use the kernel module as a disk is started running. If the VM was running on another peer, VirtualBox triggers teleporting which is essentially similar to the live migration[5] to switch the machine transparently on which the VM is running. Otherwise, VirtualBox simply powers

up the VM. The directory server will interact with the kernel module during the system initialization and execution in order to keep the consistency of the disk.

Mobility Protocol DRBD which our design and implementation are based on defines two roles for each computer or node. The primary node is the node that allows applications to write to its disk. Yet, the secondary node does not allow applications to write to its disk while its primary peer can write to its disk. The secondary node can be promoted to be the primary node in the case of the primary node crash or network failure. Also, the primary node can be demoted to be the secondary node. We adapted the same terminology in terms of the role, but we promote and demote each node for different purpose.

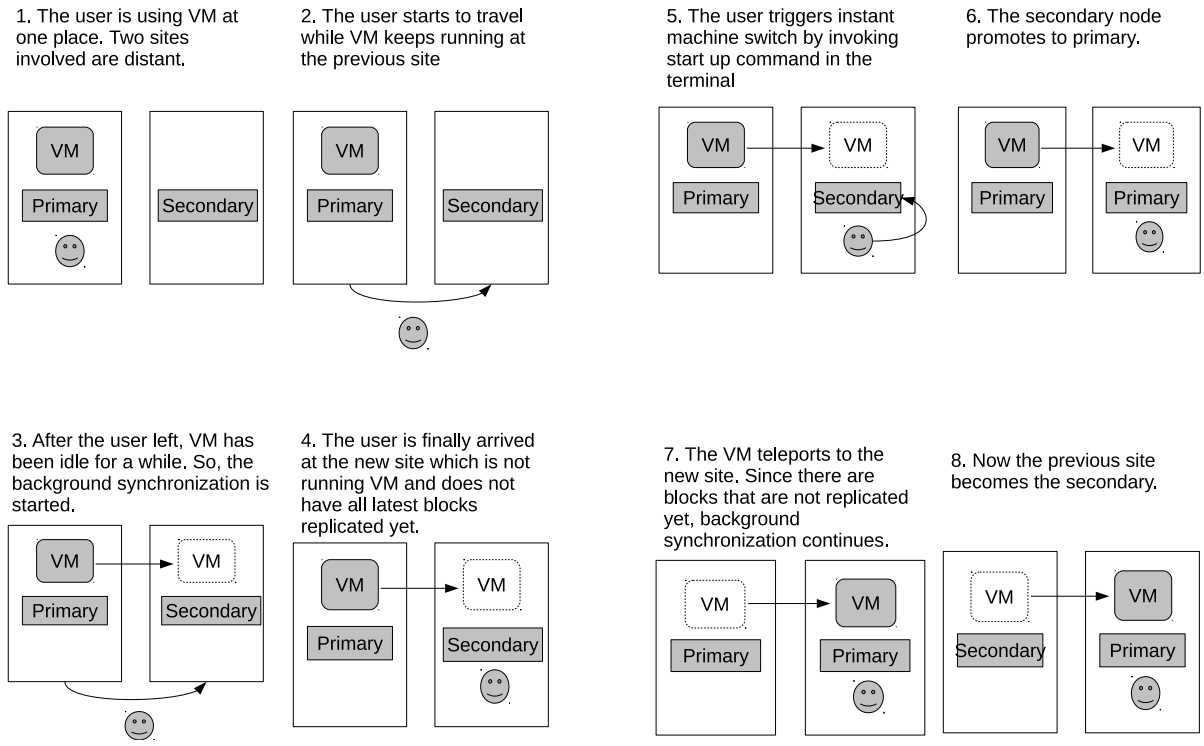

Figure 3.2: Mobility Protocol

The mobility protocol is described in the Figure 3.2. The figure assumes that two

nodes are connected over WAN. While the user is traveling, the VM at the previous site becomes idle for a while and the background synchronization is started. Then, the user invokes some command on the terminal to trigger the machine switch after arriving at the new site. The script will inform the directory server that the user is trying to use the different device to run VM. Since the directory server can figure out the device currently running VM and let the new device knows about it. Then, the new device promotes itself to become the primary node and sends a machine switch request to the other primary node to trigger teleporting. Note that if the user was remained at the previous site, the background synchronization will be stopped as soon as the user starts giving inputs to the VM again. However, since users moved to the new site in this example and the new site still does not have all of latest blocks yet, the background synchronization is continued in order to bring all the modified blocks as soon as possible to reduce chance of cache miss. After the teleporting is finished, the old primary node becomes the secondary node to prevent others modify the replica on that device. Even though there are two primary nodes temporarily during the teleporting, the teleporting actually never runs VM on both sides simultaneously. It suspends VM on the previous site before resumes the VM on the new site. Therefore, this still does not violate the essence of our single primary node assumption because there is only one writer and no concurrent sharer at any given time. If the VM was running on no other machine by the time the user invokes the start-up command, the situation is much simpler. The new device will just promote to the primary and start up the VM as soon as the directory node informs it that there is no other device currently running the VM.

If two sites are connected through the LAN, all latest blocks are supposed to be already replicated since we are using aggressive replication policy. However, if they are connected through the WAN, there is a chance that not all of latest blocks are replicated on the new device. This is where we need the directory based cache consistency protocol as the fallback mechanism of the background synchronization. Every cache miss will

trigger on-demand fetching based on the directory based cache consistency protocol. On-demand fetching is slow especially over the WAN. Therefore, our system uses background synchronization to more aggressively replicate modified blocks with on-demand fetching. This can help to reduce slowdown that users will perceive at the new site because there will be lesser cache miss.

3.5 Overall System Design

In this section, we describe more details about how our system actually does disk replication and memory replication for a VM. Also, three additional features which are necessary for making our system more practical are introduced. Not all of these features are implemented yet. However, we introduce them here to give overall impression of our system's goal.

Disk Replication We considered two replication policies which are aggressive replication policy for LAN and lazy replication policy for WAN. Although we could consolidate disks of desktops and came up with distributed block storage like DHash [16], we decided to store the whole VM image on the local disk and try to keep all disks consistent for following reasons.

- Better performance for interactive application
- Disconnected Operation
- Reducing network bandwidth consumption
- Hard disks are cheap and large
- Reliability through redundancy

Virtual Machines used for a desktop environment is configured with a large disk size which makes difficult to replicate the VM on different machines over the network. Both

ISR and Collective attempted to exploit the blocks already existing on the destination assuming the user ran the VM on the devices previously. This assumption allows an efficient optimization, copying only the blocks of different contents based on hash values. However, previous systems are lack of support for instant switch of devices without any portable storage devices. This is due to the lack of mechanism to fetch latest blocks directly from other peers and still maintaining consistency of VM's disk. We provide a logically consistent disk even if the disk is physically inconsistent through directory based cache consistency protocol.

For LAN environment, the bandwidth is large and the latency is small so that the disk I/O is a bottleneck. Therefore, we try to keep each disk in the same subnet as closely replicated as possible by replicating disk writes aggressively to other peers. Note that since we are targeting a single PC user we are assuming only limited number of machines connected within same LAN environment. If the number of devices increases and starts affecting the performance, it is possible to divide the LAN group into multiple other subnet groups and compose them as if those subnet groups are connected through WAN.

For WAN, although the measurement showed that we could achieve eventual consistency, we should consider the fallback mechanism for the worst case scenario. For example, in ISR like working environment, if the user forgot to suspend and check-in the VM, then the user has to wait until suspend and check-in are finished before checkout and resume. Moreover, if users travel very short distance like going to the coffee shop nearby and try to work with the laptop. Data transfer will be made over the slow network and it is possible that some blocks are not yet replicated on the laptop. We provide concurrent on-demand fetching and background synchronization to handle this scenario. The directory server will maintain the meta data for the latest blocks' locations and give it to the users when they start up the system on the laptop.

The block level synchronization is more efficient than the aggressive replication. For example, files opened for write access is frequently overwritten. Aggressively replicating each write is more wasteful than just transferring the final version of the series of the changes. This will save hardware resources including network bandwidth. In addition, the collision resistant hash can be used to compare the block contents on potential sender and receiver. If the hash values are matched, it means block contents on both side are not needed to be synchronized. Therefore, network bandwidth consumption can be even more conserved.

Every system managing replica should implement a consistency protocol. We are planning to implement the directory based cache consistency protocol. Since we do not allow concurrent active nodes, we can reduce communications with the directory server. Currently, we are thinking having the directory server to provide meta data to the primary node during the initialization period. Then, for shared cache blocks, the primary can fetch blocks from other peers without involving the directory server. For other types of cache state changes, we might need the intervention of the directory server. We would decide details of how to optimize the directory based cache consistency protocol as the implementation becomes more mature.

Memory State Replication For the scope of this work, we have decided to use built-in teleporting feature of client hypervisors for migrating memory state of running VM. Teleporting, or Live Migration, is designed to transfer VMs for a server environment with minimal service downtime perceived by clients. This feature is originally designed for the server environments where servers are using a shared storage. Thus, it considers memory state transfers through one gigabit LAN. It iteratively copies the dirtied memory pages to the resuming site until there remains a small set of memory pages which are dirtied faster than the iterative copy. This small set of memory pages are named as writable working set by Live Migration of Virtual Machine paper [5], and it is wasteful to

copy these pages iteratively because they are highly likely to be dirtied in the following iteration. Instead of the unlimited iterative copy, VM will be suspended at some point and the rest of dirtied pages will be copied to the other side. Then, the VM is resumed at the resume site after final iteration of copying is done.

Disconnected Operation Mode Our system will support mobile devices such as laptops and netbooks. Users should wait until all latest blocks are collected on the local disk before the disconnection. While a device is in the disconnected mode, other devices are recommended not to run the VM. When reconnected, all the changes made by the previously disconnected node will be replicated to peers. If other devices have run VM over the disconnected period, then conflicts can occur and possibly detected by the directory server during the reconnection stage. As soon as conflicts are detected, users should decide whether they want to try to resolve the conflicts, overwrite one version with the other version or just keep two separate desktop environments. To try to resolve conflicts, the system will go to the failure recovery mode as explained in Fault Tolerance section below.

Network Conservation

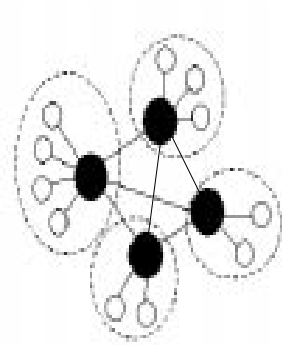

Figure 3.3: Typical Overlay Network Topology for Our System

Our system considers different properties of each network connection between the

primary node and the secondary node. We try to conserve network bandwidth by forming an overlay network so that replicated data transfers only once over WAN between different LAN groups. The typical overlay network topology in our mind is in figure 3.3. A postmaster node, the solid black nodes in the figure, is selected from each LAN group and each pair of them establishes the communication channel. Replicated data is exchanged only through this connection between postmaster nodes. When the postmaster node receives data, the node will replicate data to other peers, hollow nodes in the figure, using aggressive replication policy. This approach saves more bandwidth than broadcasting to all peers over WAN.

On the other hand, LAN is almost free compared to WAN. Thus, we prefer achieving performance, availability and reliability through aggressive replication policy. Since a gigabit LAN can transfer replicated data almost instantly, it is fair to assume all LAN group members are synchronized considering our goal. Therefore, when the postmaster node crashes, another peer can be selected as a new postmaster node with minimal chance of being inconsistent.

Fault Tolerance Only when we detect the unavailability of required blocks, we consider it as disk inconsistency issue. Because it is possible that machine can be temporarily disconnected from others, the system can operate until either a client crash or a network failure prevents the primary node from fetching the latest blocks. It is possible to tolerate crashed node or failed network by trying to choose alternative peer. When there is no peer which can handle the request, our system falls into the failure recovery mode. We have an immaterial idea for disk inconsistency issue resolution which is using fsck. First, all latest blocks from peers are gathered to one node and run fsck over the disk partition. However, we might be able to do better if we use features like taking snapshots frequently. More options will be explored as a future work.

Chapter 4

Implementation

Because it replicates data directly to its peer over the network, can fetch blocks from a peer on demand and already has the background synchronization feature, we chose to implement our system based on DRBD. However, DRBD has some shortcomings that we have overcome. First, DRBD can replicates to only one peer while we want to replicate data to multiple peers because many users have more than two PCs to manage. Second, DRBD does not guarantee the consistent view of the disk if latest blocks are dispersed over different devices. Although DRBD allows on-demand fetching, DRBD uses it only for diskless node. So DRBD works only on the local disk which is either consistent or completely inconsistent, not on the local disk which has some of latest blocks and some of outdated blocks. However, since we want to support instant machine switch, we have to provide a VM the logically consistent view to the disk even if the local disk is inconsistent physically. Therefore, we had to modify DRBD to realize aggressive replication and on-demand fetching with background synchronization. In this chapter, we discuss the background information of DRBD and how we modified DRBD to enable two replication policies. Note that on-demand fetching with background synchronization only works for two nodes currently because we have not finished implementing the directory server yet.

4.1 DRBD Background

DRBD is implemented as a virtual block device which can be inserted between the file system and the block layer which is the layer actually interact with the physical block device. DRBD is used for High Availability(HA) cluster that consists of two nodes. Usually, one node is primary and the other node is secondary while it is possible to allow dual primaries assuming the cluster file system is used. DRBD on the primary node will intercept the disk I/O and send the replicated data to the DRBD on the secondary node to mirror both disks. Thus, even if one node becomes unavailable, the other node can keep servicing applications.

DRBD provides 3 protocols to replicate data such as Protocl A, B and C. A disk write request is completed when data is written to the local disk no matter whether the data is received by the peer or not, when replicated data is arrived at the peer node and when replicated data is written to both the local disk and the peer's disk, respectively. Although Protocol C provides the strongest guarantee for consistent, reliable and highly available replicated disk, it has the biggest performance overheads since it is synchronous protocol. With Protocol A and B, there are some possibility to end up with inconsistent disk in the event of node crash or network failure. However, we decided to use Protocol A because that is the most scalable and the quickest protocol and our goal is not to provide high availability but to replicate to multiple peers even over WAN. We would explore better options for the fault tolerance mechanism as a future work.

DRBD resynchronizes two disks after the event of node crash or network failure using quick-sync bitmap and activity logging. The quick-sync bitmap on the primary node keeps track of which blocks are modified and has not been replicated to the peer at the granularity of 4KB. However, the activity logging maintains the information about which blocks are accessed recently at the granularity of 4 MB. The activity log(AL) keeps track of limited number of 4 MB regions, called extents. When the extent, not tracked by the

AL, gets newly accessed, the old extent in AL is replaced with the new one based on the least recently used cache replacement policy. With the AL, DRBD to access disk less frequently for quick-sync bitmap management. DRBD keeps small portion of the bitmap according to the extent in the AL. So, only when the old extent is replaced, the quick-sync bitmap on the disk will be updated accordingly based on the portion of bitmap in memory. When the node crashes, the quick-sync bitmap might fail to tracking some dirtied blocks because the portion stored in the memory is lost. However, using activity log kept in the disk, DRBD can revive the extents tracking the hot blocks. Then, DRBD considers all hot blocks as dirtied so that no block that might be out-of-sync is missed out during the resynchronization.

4.2 Multi-target Replication for Aggressive Replication Policy

DRBD has three main workhorse threads such that receiver, asender and worker threads. They are initialized when the kernel module component of DRBD is initialized and the TCP connection is established by the initialization code for the receiver thread by invoking drbd_connect(). DRBD keeps all required data for its operation in its main data structure drbd_conf and almost all important DRBD functions takes it as an argument. drbd_connect() also takes struct drbd_conf as an argument and obtained IP address from its member field containing drbd_socket data structure. The drbd_socket contains both IP address and socket descriptor member fields and used for both connection establishment and message communication as well as data transfer. However, drbd_socket only stores one IP address and one socket descriptor. After drbd_connect() obtain the opened socket file descriptor, it stores the descriptor in the drbd_socket data structure.

In order to extend drbd_socket, we defined our data structure vmsync_socket and

created a list of vmsync_socket, named vmsync_socket_list. We put vmsync_socket_list in the drbd_socket. The vmsync_socket contains information per connection like peer's IP address and opened file descriptor for the connection and possibly more. So, there is one vmsync_socket per peer, and when the DRBD module is initialized, we initialize vmsync_socket struct per peer in order to contain peers' IP addresses. Then, we construct the vmsync_socket_list with vmsync_socket data structures and store the list in the struct drbd_socket which is again contained in the main DRBD data structure drbd_conf. We did it this way to modify DRBD's architecture minimal because DRBD is not implemented in a modular way.

We enclosed the drbd_connect() invoking part of the receiver initialization procedure with the loop iterating the vmsync_socket_list. Since drbd_connect() obtains the IP address from and save the opened socket file descriptor in drbd_socket, we properly set the IP address of a peer in drbd_socket before invoking the drbd_connect(). After returning from drbd_connect(), we retrieve the opened socket descriptor in drbd_socket() and save it to the corresponding vmsync_socket.

When a block input/output(bio) request is made by VM, drbd_make_request_common() interposes it. If the bio request is for write access then it needs to replicate the write to other peers in the same LAN group. So DRBD schedules the replication work into the queue which the worker thread gets jobs from and sends the bio request to the backing block device driver to write to the local disk. When the worker thread starts working on the replication job, the worker invokes drbd_send_dblock() function to send the data to the peer over the network. We again enclose drbd_send_dblock() with the loop iterating vmsync_socket_list and send the packets to multiple targets by changing the socket descriptor retrieved from vmsync_socket_list at the each iteration.

4.2.1 Issue with Write Ordering

Some applications like journaling filesystems or relational databases require to keep write-ordering correctly. Since Ext filesystems are journaling filesystems and we would like to support these filsystems in the VM, we had to consider this write-ordering issue during modification. DRBD uses barrier packets to keep write-ordering for replicated data for Protocol A which we adapt. The reason why DRBD needed to use the barrier packets is that disk I/O scheduler on the peer can mix up the order of disk writes even if packets are sent in the correct order by the primary. For example, if applications want to write A and then B to the same block and they have causal dependency, applications can force the write-ordering by using method like flushing (e.g. fsync). This can keep the write-ordering for the local disk. However, if write A and B are sent over the network, there is no mechanism such as flushing over the network. Thus, DRBD implemented the network version of flushing which is basically sending the barrier packet between write A and B. Then, the peer receives A and the barrier packet prior to B. When a barrier packet is received by the secondary node, the secondary node will flush write A to its local disk before receiving B. DRBD puts every write request that should be written before following write requests into an epoch set. Since the Protocol A is asynchronous, DRBD collects replicated data which do not have causal dependency in the same epoch set. Different epoch sets are separated by barrier packets. Until acknowledgement for the barrier packet arrives, DRBD does not clear the epoch set. If we replicate data to the multi-targets, each target will reply with the barrier Ack packet. This should be received so that the TCP receive buffer does not get blocked. The barrier Ack packet is sent through the meta data socket which is different TCP socket from the data socket which is mainly used for transferring replicated data. While the data socket is used by the receiver thread, the meta data socket is a responsibility of the asender thread. So we extended DRBD's asender thread listening on the meta data socket so that when the barrier Ack packet arrived, we let the asender to iterate through the vmsync_socket_list and collect the barrier Ack packet sitting in the TCP receive buffer. Then, we clear the

corresponding epoch set when we received the barrier Ack packets of the last peer.

Since we are waiting for each peer to send the barrier Ack packet, this will degrade the performance depending on how fast every peer replies. However, this write-ordering issue is only applicable to the aggressive replication policy - we replicate using background synchronization for lazy replication policy - and other peers are not expected to experience intensive disk I/O. So, we do not expect to see particularly long delay. Thus, we just implemented as it is described here to ease the implementation. We could implement multiple asender threads per connection and handle each connection differently without waiting for lagging peer during the iteration.

4.3 Concurrent On-demand Fetching and The Background Synchronization

We provide concurrent on-demand fetching and the background synchronization for the situation where a new primary has inconsistent disk due to slow network bandwidth as mentioned in the section 3.5 under the disk replication heading. In order to provide a logically consistent disk, the new primary node should know the information of locations for the latest blocks. Based on our design, this information is managed by the directory server. However, because we did not implement the directory server yet, this feature is limited to two nodes. To enable this feature for two nodes, we depend on the bitmap sent from the old primary. The bitmap on the old primary node provides information which blocks are dirtied and not transferred to the new primary yet. This bitmap is called sync bitmap, and the old primary, the sync source, sends this sync bitmap to the new primary, the sync target. Towards the eventual consistency, we enabled synchronization in the background which also uses the sync bitmap. Here, we present how we enabled this feature for two peers. Once we implement the directory server, it can be extended

to the multiple peers easily.

4.3.1 Dirty Bitmap and Sync Bitmap

DRBD sets the bit in the bitmap whenever it is finished with handling disk write request. Since background synchronization is using the bitmap as well, we needed another bitmap to mark dirtied blocks while the sync bitmap will keep track of blocks to be synchronized during background synchronization. Otherwise, there is no means to determine why the bit is set - it can be set because the corresponding block is dirtied locally or dirtied remotely. Thus, we added the new bitmap, so called dirty bitmap, and set a bit in the dirty bitmap instead of the sync bitmap when a disk write is done. Therefore, we have the sync bitmap for the background synchronization and the dirty bitmap to mark newly dirtied blocks on the new primary node. When the synchronization is finished and all of latest blocks are on the local disk, the dirty bitmap will be merged to the sync bitmap. Afterwards, disk write request will set the bit in the sync bitmap so that those blocks to be synchronized can be remembered for the next time users switch the device.

On-demand fetching is another synchronization job which we use the sync bitmap for. It allows us to synchronize the requested out-of-sync block right away so that VM does not have to wait for the background synchronization to fetch the block. When a disk I/O request comes in, we look at the sync bitmap and determine whether we need to fetch from the peer or not. If we need to fetch the block from the peer, then we put VM into sleep and request directly to the peer. When the block is received and written to the local disk, we clear the bit in the sync bitmap and wake up the VM which will then proceeds to make disk I/O requests to the backing block device driver.

4.3.2 Race between On-demand Fetching and The Background Synchronization

While on-demand fetching is triggered by the disk I/O request from VM, the background Synchronization is triggered by a timer interface in the Linux kernel. Every time the timer is expired, the synchronization job is scheduled for the worker thread. Once the worker thread starts working on the synchronization job, it first looks at the sync bitmap and picks the next bit that is set to 1. Once next bit to sync is picked, the corresponding blocks are marked as being synchronized, and the block request is sent to the peer. The request message contains the hash value of the block content so that the sync source can compare the received hash value with the hash value of its own corresponding block content. If two hash values are equivalent, the requested block will be transferred to the new primary. Otherwise, just acknowledgement message, notifying the block is already in sync, will be sent back. When requested blocks are received by sync target, the blocks are written to the local disk and the bit is cleared in the sync bitmap.

Since both on-demand fetching and the background synchronization access the sync bitmap, there is a race between background synchronization and on-demand fetching. The race is avoided by marking atomically the block is in the middle of either on-demand fetching or the background synchronization. DRBD uses lru_cache to keep track of hot blocks that are accessed recently. Also, DRBD has the concept of extent that represents a specific region of blocks in the disk. The extent can be represented as bm_extent or al_ext. If a bm_extent exists in the cache managed by lru_cache data structure, the background synchronization is on-going for blocks in that extent. Also, DRBD keeps al_ext for recently accessed blocks in separate lru_cache, and al_ext contains reference counter. Thus, if the reference counter is greater than 0 and al_ext is in lru_cache, then blocks in the al_ext are being accessed by an application. When the background synchronization

procedure tries to synchronize blocks, it first checks whether the blocks are being accessed by VM by looking at this reference counter. On the other hand, when the disk I/O request procedure tries to do disk I/O on blocks, it checks whether those blocks are being synchronized or not by looking for the existence of corresponding bm_extent. DRBD uses al_lock spinlock data structure to atomically modify both lru_cache data structures, so only one of both procedures can modify extent and lru_cache at the same time. Thus, while one is trying to work on the disk blocks, other can not work on the overlapping set of blocks.

We extended this mechanism by merging the on-demand fetching as a part of the disk I/O request handling. When disk I/O request comes in, the disk I/O handling procedure tries to start disk I/O request by checking the existence of bm_extent. If it exists, wait until synchronization is finished for that extent. Otherwise, the procedure looks at the sync bitmap. If the corresponding bit is set, on-demand fetching is triggered. While waiting for the block, the disk I/O handling procedure is put into sleep together with VM. When the response is received and the fetched block is written to the local disk, the disk I/O handler wakes up. Then the disk I/O request is passed to the backing block device driver. Similarly, while on-demand fetching is on-going - if the corresponding al_ext's reference counter is greater than 0 - the synchronization procedure waits until the on-demand fetching is finished for disk I/O as it would wait for disk I/O handling in original DRBD.

Chapter 5

Evaluation

Our testing environment consists of two desktop PCs equipped with a gigabit capable network interface card. They are connected through gigabit Ethernet cables and a gigabit switch. One machine has Intel Pentium 4 CPU 3.00 GHz with 512 MB cache size and 1 GB of main memory. The other machine has Intel Pentium 4 CPU 3.00 GHz with 512 MB cache size and 2 GB of main memory. We chose to use VirtualBox 3.2.10 for our client hypervisor and a VM was configured with 256 MB of main memory and 10 GB of hard disk. We installed Linux 10.04 Lucid on the VM.

In this paper, we have discussed two features we have built so far. One is the aggressive replication policy and the other is concurrent on-demand fetching and background synchronization. Since the aggressive replication policy can mirror disks with the very small time window during which disks are inconsistent, we do not expect interesting performance difference between the aggressive replication policy and the base case where we run a VM with the standalone consistent up-to-date local disk. So our main interest of the evaluation was how much performance degradation is introduced by the concurrent on-demand fetching and background synchronization. Therefore, we evaluated the base case first and then tried to see the overheads of using the concurrent on-demand fetching

with the background synchronization.

First, we tried to get the time taken to finish the decompression of the linux-2.6.38 source file on the consistent standalone hard disk. Then, we did the same decompression job on the inconsistent hard disk to see the overhead of the on-demand fetching(OF) with the background synchronization (BS). To evaluate the overhead of OF with BS, we invalidated all blocks on the local disk and minimize the background synchronization rate to 300 KB/s. However, we could not emulate the WAN environment because we had a problem when we used Linux Traffic Control (TC) utility for bandwidth control together with the DRBD. The overhead of WAN emulation is expected to decrease performance more. For more accurate evaluation in the future, we will have an intermediate desktop bridging two peers and make the intermediate desktop to control the network bandwidth and latency so that we do not use TC with DRBD on the same machine.

Table 5.1: Multiple Trials of Decompressing the Linux Source

Trial	Base Case	OF with BS	Trial	Base Case	OF with BS
1	182	275	6	238	354
2	258	223	7	154	285
3	255	523	8	157	353
4	242	388	9	213	382
5	244	361	10	225	231

Table 5.2: Average(Standard Variation)

Base Case	OF with BS
216.8 (32.24)	339.2 (68.56)

The average time taken to decompress the Linux source file was 216.8 sec with the standard variation 32.24 for the standalone case, and was 339.2 sec with standard variation 68.56 for OF with BS. The overhead of the OF with BS is definitely perceivable

by the user since time increases around 50% compared to the base case. Moreover, this is numbers obtained with a gigabit LAN environment. This implies two things. First, the aggressive replication is better to be employed for LAN environment due to the performance. Second, we have to synchronize two disks as much as possible before users switch devices. OF with BS should be used as the fallback mechanism rather than the primary method to replicate data even for WAN environment. Nonetheless, OF with BS can provide a new feature that no other previous desktop virtualization supported which is instant machine switch support for an individual PC user.

Chapter 6

Future Work

Since our implementation is not completed yet, most of our future work is to finish implementation. At the moment, we have built two different replication schemes for LAN and WAN. For LAN, multi-target replication is enabled, but for WAN, it works for only two nodes. Once we build the directory server, we would be able to support multiple nodes for WAN environment. Other features like disconnected operation mode, network bandwidth conservation feature and fault tolerance should be implemented as well in order to provide reasonable guarantees to users.

Also, we want to find a trade-off between minimizing the total migration time and the service downtime so that users do not suffer too much either from slowdown or from resume latency. In terms of the total migration time, the live migration will take more time then the simple copy-in and copy-out scheme due to the iterative copying stage. Thus, the live migration can hurt the performance and users will perceive performance degradation during the total migration time. On the other hand, the copy-in and copy-out approach will go through the longer service downtime which means users will suffer from the long resume latency. We will invest some time to this issue in the future.

Although performance is poor when on-demand fetching and background synchronization is used, there are many optimizations we can utilize. We will fetch more blocks

than requested ones to fetch blocks nearby the requested ones in advance. Also, we will add hash function to avoid unnecessary data transfer. Instead of synchronizing in the increasing order of the block ID, synchronizing hot blocks with higher priority may provide better performance by reducing the chance of cache misses. Furthermore, we will work to improve efficiency of the aggressive replication by batching replicated data before sending out so that avoiding sending small packets many times.

In terms of communication channel, we use TCP connection currently. Thus, we are currently sending out packets containing the same data multiple times. Thus, it is inefficient compared to broadcasting a packet to multiple peers. However, we might be able to implement reliable multicasting protocol to improve scalability.

We would also evaluate our system more rigorously with WAN emulation by having a bridging desktop machine between two peers and restricting incoming and outgoing bandwidth as well as imposing some network latency. In addition, we would want to evaluate our system in a real deployment. Then, we would be able to see how much performance degradation is observed for OF with BS in the real deployment.

Our system can automatically replicate the whole software environment considering network properties and it can provide logically consistent view of the disk even if the actual physical disk's state is inconsistent. This can provide flexibility of deploying replica of VM across different platforms and keep replicas consistent which is a general and useful property in today's cloud computing environment. Thus, we expect that it can be used for different purpose other than PC management for users and we would try to find the different application scenario of our mechanism.

Chapter 7

Conclusion

In this paper, we described the design of a system to automatically replicate whole desktop environment and the implementation of two replication policies one of which is to support instant device switch for users. The tool is designed to help an individual PC user to manage their desktop environment one time only and to provide the user an exact computing environment on any device he owns. We have done the disk I/O measurement to get some sense how intensively the desktop environment uses the disk. Based on numbers we got, the aggressive replication is feasible for LAN which can keep VM state almost identical all the time across different devices. Also, we realize that a lazy replication policy should be used for WAN due to the limited network capacity and the cost. Since we design to lazily replicate state changes for WAN, we need a fallback mechanism to support transparent and instant machine switch. Thus, we implemented on-demand fetching with background synchronization and evaluated its performance. Currently our system is only applicable to two nodes for WAN while it is applicable to multi-targets for LAN. We would implement directory cache consistency protocol to extend our system to work with multiple nodes in the WAN environment. Although we could not get the accurate performance overheads of on-demand fetching with background synchronization

due to the issue of using Linux TC with DRBD, the evaluation without TC showed about 50% performance degradation. This gives some hint that it can get much worse than 50% overhead and we need to implement optimization techniques. However, our evaluation was for the unrealistic setting where all disk blocks trigger on-demand fetching. It would be interesting to see how much we can reduce the overhead with performance tuning. At last, to make our system more useful, we would support disconnected operation mode, network bandwidth conservation feature and fault tolerance feature in the future.

Bibliography

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, pages 1–14, New York, NY, USA, 2002. ACM.

[2] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. *SIGOPS Oper. Syst. Rev.*, 29:109–126, December 1995.

[3] Peter J. Braam. The coda distributed file system. *Linux J.*, 1998, June 1998.

[4] Ramesh Chandra, Nickolai Zeldovich, Constantine Sapuntzakis, and Monica S. Lam. The collective: a cache-based system management architecture. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 259–272, Berkeley, CA, USA, 2005. USENIX Association.

[5] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[6] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *In Proc. NSDI*, 2008.

[7] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with cfs. In *In SOSP*, 2001.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[9] Benjamin Gilbert, Adam Goode, and Mahadev Satyanarayanan. Pocket isr: Virtual machines anywhere. Technical report, Carnegie Mellon University School of Computer Science, 2010.

[10] Ragib Hasan, Zahid Anwar, William Yurcik, Larry Brumbaugh, and Roy Campbell. A survey of peer-to-peer storage techniques for distributed file systems. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II - Volume 02*, ITCC '05, pages 205–213, Washington, DC, USA, 2005. IEEE Computer Society.

[11] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *SIGOPS Oper. Syst. Rev.*, 25:213–225, September 1991.

[12] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst.*, 10:3–25, February 1992.

[13] Michael Kozuch and M. Satyanarayanan. Internet suspend/resume. pages 40–46. IEEE Computer Society, 2002.

[14] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley

Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. pages 190–201, 2000.

[15] Justin Mazzola, Paluska David, Saff Tom, and Yeh Kathryn Chen. Footloose: A case for physical eventual consistency and selective conflict resolution. In *In IEE WMCSA*, 2003.

[16] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: a read/write peer-to-peer file system. *SIGOPS Oper. Syst. Rev.*, 36:31–44, December 2002.

[17] Edmund B. Nightingale and Jason Flinn. Energy-efficiency and storage flexibility in the blue file system. In *In Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 363–378, 2004.

[18] Daniel Peek and Jason Flinn. Ensemblue: Integrating distributed storage and consumer electronics. In *In Proceedings of the 7th Symposium on Operating Systems Design and Implementation. ACM SIGOPS*, pages 219–232, 2006.

[19] Antony Rowstron and Peter Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. *SIGOPS Oper. Syst. Rev.*, 35:188–201, October 2001.

[20] Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Virtual appliances for deploying and maintaining software. In *Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA 2003)*, October 2003.

[21] Constantine Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pages 377–390, December 2002.

[22] Constantine Sapuntzakis and Monica S. Lam. Virtual appliances in the Collective: A road to hassle-free computing. In *Proceedings of the Ninth Workshop on Hot Topics in Operating System*, May 2003.

[23] M. Satyanarayanan, Michael A. Kozuch, Casey J. Helfrich, and David R. O'Hallaron. Towards seamless mobility on pervasive hardware. *Pervasive Mob. Comput.*, 1:157–189, July 2005.

[24] Mahadev Satyanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23:9–18, 20–21, May 1990.

[25] Mahadev Satyanarayanan, Benjamin Gilbert, Matt Toups, Niraj Tolia, Ajay Surie, David R. O'Hallaron, Adam Wolbach, Jan Harkes, Adrian Perrig, David J. Farber, Michael A. Kozuch, Casey J. Helfrich, Partho Nath, and H. Andres Lagar-Cavilla. Pervasive personal computing in an internet suspend/resume system. *IEEE Internet Computing*, 11:16–25, March 2007.

[26] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. pages 149–160, 2001.

[27] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29:172–182, December 1995.

[28] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent W. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, PDIS '94, pages 140–149, Washington, DC, USA, 1994. IEEE Computer Society.

[29] Werner Vogels. File system usage in windows nt 4.0. *SIGOPS Oper. Syst. Rev.*, 33:93–109, December 1999.

[30] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI*, pages 307–320, 2006.