VaporDisk: A Reliable and Portable File System

Aaron Nathan
amn32@cornell.edu
CS6464 Final Project

Abstract

This paper introduces VaporDisk, a fast, **reliable** and **portable** Windows mixed kernel and user-space file system. This system is designed for use by the "casual computer user", i.e. someone with little or no technical knowledge and is therefore is designed to be entirely transparent to the end user. It is also designed to work with the simplest back end storage systems, such as Amazon's S3 while being extensible to later take advantage of more complex data models. VaporDisk simultaneously provides almost the same performance as a local disk, minimally impacting the user experience. This is accomplished with a write-through cache and smart revision collapsing, described in detail in the paper. Additionally, VaporDisk provides a friendly and secure web interface to allow remote access of files from an HTTP server, making data portable to any device with a web browser.

Keywords

Distributed File System, Backup, Windows, Driver, Storage, Web, S3

Introduction

This paper explores how to provide a reliable and portable file management solution that casual computer users will find convenient. Specifically, the system addresses the problem of a single user on a single workstation who occasionally needs to have access to his or her files on other computers. It aims at minimizing complexity presented to the user while simultaneously providing useful tools that will prevent users from employing bad practices when sharing and backing up data.

As computers become more and more integrated into people's daily activities, data accessibility and reliability is becoming even more important, even for the casual computer user. Currently, many people utilize thumb disks and other "portable" storage devices to bring their files with them, much like the physical briefcase office workers have (or backpacks students have). Thumb drives allow users to have access to their files from home, work, school, or any other location with a computer, and frees them from being bound to one location to access and modify their data. The use of these devices also facilitates sharing of data as these drives are commonly shared amongst people; for instance to quickly copy a series of photographs from one laptop to another. In addition to thumb disks, email too has started to become a common medium for distributing files either to other people or amongst one user's many computers. With complicated corporate and school networks, it is generally more of a hassle to use the storage and sharing mediums provided by these entities than it is to just use something as simple as email. However, email was never really designed to handle large files, and therefore, most systems rejects files larger than a few megabytes. Additionally, the messages with attachments that fit under this arbitrary limit still place a large burden on the underlying mail systems and generally irritate the IT support staff.

These ad-hoc types of file management outlined above bring upon a host of problems, known all too well to those of us who have employed such solutions. First, both email and thumb drives offer little or no revision control, so although the data is available on multiple devices, it is entirely unclear which device has the most recent copy, who is currently working on the latest revision, and how to merge together any conflicts that may arise when the revisions don't match. It is entirely up to the user to perform all revision control activities, although they may not even be aware any conflicts exist! Secondly, it is not uncommon for portable devices to be far less reliable than their non-portable counterparts simply due to the fact that they are portable. Thumb drives are sent through the washing machine and emails get "lost" somewhere from your mail server to the destination server, and with these failures the data is lost. So although consistent accessibility and reliability are becoming more important, the current methods used by casual computer users hardly address any of these issues.

At the same time, cloud computing has emerged from infancy and is beginning to be used in main stream data-intensive and compute-intensive applications. Large datacenter owners such as Amazon and Microsoft have begun to offer their excess computing resources for relatively small fees, promising both reliable and fast systems that are fairly simple to use and inexpensive. This provides for an offsite location, ideal for backing up files, with capacity that grows as the user demands. In the case of Amazon S3, a simple PUT/GET/ LIST/DELETE/COPY interface is provided.

The remainder of this paper first describes an in-depth review of the design and implementation of VaporDisk and how differs from prior work that attempts to solve this problem. We then test VaporDisk's performance using typical Windows office applications, and compare this to the performance using SAMBA shares on Windows Server and the experience of a few other backup solutions. We conclude with discussion on our system and the work that is left to be done.

Design

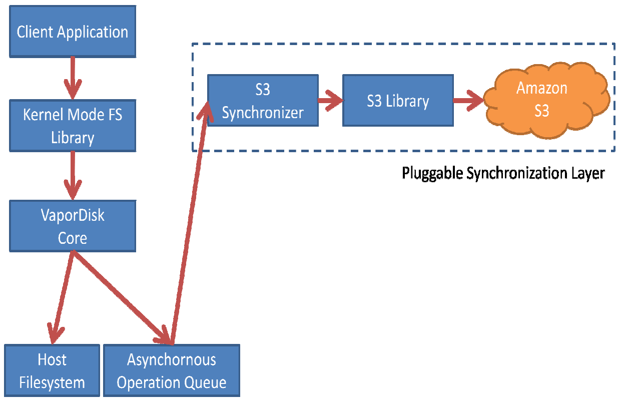


Figure 1: Architecture of VaporDisk

Here we introduce a preliminary concept of the full VaporDisk System. The eventual system will have many more features, however, the design presented here addresses the specific pieces that have been implemented and together results in a functioning system for basic remote backup and remote read only availability. Figure 1 depicts the VaporDisk architecture from the view point of a single user. Each component is described in detail below:

The Client Application

One of the major design goals of VaporDisk was to have no impact on the client application *and* how the user uses that client application. We therefore designed around the goal of making all existing Windows based software compatible with VaporDisk, and furthermore, examined how users currently save files using devices such as thumb drives. In the implementation section we will discuss how we achieved this goal further.

Kernel Mode File System Library

Much like FUSE in Linux[6], we needed to be able to handle kernel events, but wish to do so with code that runs in user space. This approach gives us several advantages, the main advantage being that Kernel mode code is notoriously difficult to debug and is generally resultant in "Blue Screen" errors when bugs are encountered. This topic and the design decisions made are covered in much more detail in the Implementation section of this paper.

VaporDisk Core

This component acts as the director between all the components of the VaporDisk System. It handles distributing the calls from Kernel Mode File System component to the underlying permanent storage on the host and the asynchronous replication queue, both of which are described later. This component is also responsible for the underlying logic for managing the VaporDisk such as mounting and unmounting.

The Host File System and Asynchronous Operation Queue

In general, users are impatient and will expect their VaporDisk drive to perform just as well as any normal hard drive. It is therefore not realistic to rely on writing data to some remote network location on every disk operation and to wait until that operation is completed. To address this, VaporDisk utilizes a local disk cache which actually contains all the files on the VaporDisk. File system events are passed to this local cache and return as soon as the cache is up to date. By directing writes and reads immediately to the host file system, we can give near local disk performance (we only add overhead in proxying the file system call through our system). At the same time, every file system call is sent to the Asynchronous Operation Queue. The design of this system is setup so that eventual consistency will be achieved with a remote storage source and the local host. We abstract all file system calls to simple operations (identified later) designed to be independent of the underlying backing store.

Due to the fact that we have an eventual consistency model, user data loss could still occur during the time window that the local cache and remote store are out of sync. However, we felt this tradeoff for overall performance was well worth the sacrifice of a few seconds of possible work loss between cache synchronizations.

Pluggable Synchronization Layer

At this stage, the VaporDisk is implemented against Amazon's S3, but we designed VaporDisk to be useable with a variety of backing stores by creating a Synchronization Layer which abstracts the specifics of the backing store away from the Asynchronous Operation Queue. For instance, a developer could write a new Synchronizer that worked with proprietary NAS systems or a different online backing store. Note that this layer provides both the ability to send the data to the backing store and also to retrieve and restore the data in case of failure on the local host.

VaporNodes

In order to store the data in a way that is general enough to be used with many different kinds of backing stores, we created the VaporNode, which is similar in construction to a UNIX inode. Each VaporNode represents either a file or folder in the file system and contains a collection of metadata and a pointer to the file data itself. The metadata consists of a 64 bit identifier, a 64bit parent identifier, the name of the entity, its creation, access and modification times, its size in bytes, a list of its children, its MD5 hash code (only for files) and a string indicating the machine name of the computer that last edited the file with VaporDisk. We assume the root always has identifier zero, and from the root we can recreate the entire file system tree. The hash code is computed on every write by the client.

Implementation

The Kernel Mode Driver

In Windows, there are several ways to present "storage" to a user. One method is to register a Folder Watch object, which essentially relays callbacks to some user-process whenever modifications occur on a folder and its children. Writes still go down to the underlying file system and cannot be intercepted or even modified. This is the method used by Dropbox[4]. Another method involves registering a Shell Handler which presents a virtual "store" in the My Computer object. This is not technically a disk drive and not all Windows programs will be able to use this type of system, particularly legacy applications. Typical backup sites use this type of implementation such as IDrive and GmailFS[5].

In designing VaporDisk, we made it a requirement that the VaporDisk itself shows up as a physical disk, not some special folder. To achieve this, VaporDisk makes use of the Windows IFS (installable file system) framework accessible with the in Window DDK (driver development kit). With this framework, there are three modes that a developer can use:

- File System- This is the lowest level access and is designed for creating actual file systems on real volumes, like NTFS or FAT32, that talk to physical hard drives. Microsoft explicitly recommends against using this model.
- File System Filter Driver – This allows the developer to customize existing file systems. The driver presents itself as a filter to intercept requests to the underlying volume. It can choose either to continue to pass the request to the underlying file system or intercept it and mask it away.
- MiniFilter Driver– These are designed for virus scanner and indexing type applications that need access to all files regardless of their use, but have no ability to write or intercept requests to the low level file system.

All of the above methods use IRP (Input/Output Request Packets) for communication between the Operating System and the Kernel Mode Driver.

VaporDisk uses the File System Filter Driver pattern, but intercepts *all* calls and does not pass them through to the underlying file system. In order to achieve this, we have used the Dokan library. Dokan[7] is an open source project which is uses the Microsoft Windows IFS File System Filter Driver model. Extensive work was required to make this driver stable especially under both 64bit and 32bit flavors of Windows.

It is important to note that the Filter Driver code executes in Kernel Mode and can therefore cause system panics, also known as “blue screens” if any bugs are encountered. In order to minimize this possibility, we chose to separate as much of the code as possible into user mode by proxying calls from the driver as soon as possible. Additionally, since the user mode components of VaporDisk are implemented in C#, a wrapper was needed to access the kernel-mode API calls through the Dokan device driver.

The User Mode Application

In the VaporDisk framework, a user mode process runs in the background and constantly intercepts the proxied file system operations. This process takes all of these operations and runs the Asynchronous Operation Queue to synchronize these operations to a backing store. For the purposes of this paper, we only implement a synchronizer for Amazon S3, but we plan to implement more in future iterations.

Since the file system is only writable by one client, the local disk cache will always be "freshest" from the local host with write access (except in the case of client failure, which is discussed later). Changes to the file system are saved locally on the client and therefore, each file system operation immediately returns to the client application. At this moment, the data written to the client's disk is not backed up or even available remotely. We assume "eventually consistency"; eventually the client and the backing store will have the same data available to all other "read only" clients. To accomplish this, an event is put on an asynchronous queue. It is important to note that during this inconsistent period, data loss *can still occur*. However, the goal of this system is to minimize the chance of this happening by keeping this window of vulnerability as small as possible.

The Synchronizer

The queue performs operations through use of the Synchronizer. The Synchronizer’s main job is to do as its name implies: synchronize the state between the local client and the backing store. This means each synchronizer is written for a particular backing store. The Synchronizer also works in both directions, specifically, during normal operation it proxies writes from the local host to the remote store, but during backup, it must process reads from the remote store to the local host.

The Synchronizer itself has several operations. These include write, create, delete, move, setAttr and setLen. Each of these operations is implemented with the assumption that the underlying VaporDisk cache has "checked" the validity of the operation, for instance, checking that a file already exists when renaming another file.

For this implementation we have chosen to create a Synchronizer that uses Amazon's S3 service as its backing store. This service provides data access through "buckets" which are essentially single level folders which contain a series of keys. Each key is a file that is up to 5GB in size. Operations that are valid on S3 are GET/PUT/LIST/DELETE and COPY[8]. Each operation is entirely atomic, and the backing store is assumed to provide adequate redundancy.

The Synchronizer for S3 makes use of an encrypted SSL connection over HTTP to make the data transfer secure. In Windows, files are stored in exactly one folder and are entirely identified by their path, unlike in many Unix systems in which files are identified by inode and can be "hard linked" to several folders. Essentially, the file system structure in Windows is a tree and not a general graph.. To accommodate this (and not to limit the system for use in the Windows environment), each element is converted to a VaporNode. We store a table of VaporNodes in memory on the local host in order to help minimize the number of hits on the remote backing store. We assume that since this is the only client writing to the backing store that we will not have consistency issues since the local host will always have the most up to date copy.

Recreating Lost Data

Since each file and directory stored by the S3 Synchronizer is a VaporNode, all information necessary to recreate the file system is available if the local machine were to fail. This includes the usual items such as creation, modify and access times, and the file's name and size. It also contains the MD5 hash of the file (calculated on write) and the file and its parents ID assigned by the synchronizer. Directories contain a list of children identified by their VaporNode ID. Another important field is the "last modified by" field, which contains a human identifiable computer name of the last machine to send a "write" to the file. This last field will eventually be used for conflict resolution, although this is not used except for display purposes in the current implementation of VaporDisk.

Each time the user mode application is started, it performs an exhaustive check of the local cache of the file system and the remote backup. It checks the MD5 hash of each file element, and restores any missing files from the local host that are on the remote store. It also updates the remote store with the latest copy of a file if the hashes differ for a given file. The general case of complete local host failure is also handled by this method since no files will exist, the entire remote store we be downloaded and recreated.

In order to maintain consistency, there is a transaction id that is stored on the client's cache within the synchronizer. This is simply a number that indicates the point at which the synchronizer queue for the local client is at. For instance if there are 100 items waiting to be sent to backing store and the client machine goes down while sending item 45, the cached item ID would contain the value 44, indicating to the client once they resume to start sending at item 45.

Asynchronous Queue Optimizations

The fact that the synchronization queue is running asynchronously brings up several complications and possibilities for optimization. For instance, imagine the following scenario: A user creates file A and write some bytes to it. Immediately, that user renames the file to file B. Since the synchronizer can be arbitrarily delayed, when it tries to get the writes to file A, it will realize that file A is missing and not know what to write. However, since file A no longer exists on the local client, we don't care what was in file A, so long as we know that this is now file B. Therefore, the synchronizer must "follow through" operations such as *move* which implicitly move data and actually recopy the data to S3.

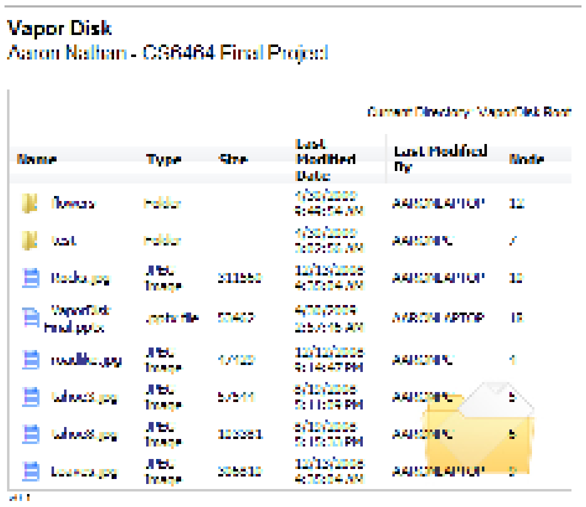
S3 does not permit partial writes to files (an entire entry associated with a key must be replaced; it cannot be appended or modified). We therefore use no differencing optimizations (i.e. write only part of the file that changed), although this is certainly something to be implemented later by implementing fixed size block-type storage to Amazon S3. Although this is a significant limitation, the synchronizer minimizes this problem by being intelligent about writing files to the backing store. The first method of this is that before each write is added to the synchronization queue, the synchronizer checks if the operation is redundant. This occurs fairly often since we impose an arbitrary delay on the queue of five seconds. This effectively allows programs to "settle" before having the operation be written to the remote store. For instance, if a file is told to be written to the disk 10 times within a five second window,, there is no need to send that same data to S3 10 times. We refer to this as "smart revision collapsing".

The second method of minimizing needless writes is during each write, the MD5 hash of the local cache file is calculated and compared against the value on the backing store. Only differing hashes trigger writes, significantly speeding up the system's performance with multi-write type applications. For instance, when copying a file using Windows Explorer, the shell will commonly fire several threads to copy blocks of the file in parallel. This results in the same number of write calls, which all may finish at a later time. If the first revision collapsing method fails to eliminate these writes, checking the hash before writing will prevent the duplicate data from being transmitted.

Read Only Web Service

The framework provides a simple website application that allows a user to access and share their files from a given URL. This accomplished with a separate application running on some server (such as EC2) that hits the backing store, decodes the file structure using the synchronizer, and sends a listing of files to the http client. This listing also shows the state of client replication to indicate if the version displayed is the most recent copy of a particular file.

A snapshot of the interface is shown in the Figure 2.



Vapor Disk

Aaron Nathan - CS6464 Final Project

Current Directory: VaporDisk Root

Name	Type	Size	Last Modified Date	Last Modified By	Node
[illegible]	Folder		[illegible]	AARONLAPTOP	12
test	Folder		[illegible]	AARONPC	7
[illegible].jpg	JPEG Image	[illegible]	[illegible]	AARONLAPTOP	12
VaporDisk Final.pptx	.pptx File	[illegible]	[illegible]	AARONLAPTOP	15
[illegible].jpg	JPEG Image	[illegible]	[illegible] PM	AARONPC	1
[illegible].jpg	JPEG Image	[illegible]	[illegible] PM	AARONPC	5
[illegible].jpg	JPEG Image	[illegible]	[illegible] PM	AARONPC	5
[illegible].jpg	JPEG Image	[illegible]	[illegible]	AARONLAPTOP	[illegible]

Figure 2: VaporDisk Web Interface

Related Work

Backup of user data is not a new idea by any stretch. Several existing tools exist for performing backups, all with varying degrees of user input. For example, Microsoft Windows XP and Vista both ship with the "Backup and Restore Center"[1] a full system backup utility complete with a graphical user interface and scheduler. Additionally, Microsoft's Live OneCare has a distributed file backup system that mirrors certain folders across all of a user's computers to be restored in the case of disaster. Although these systems are built into the operating system, they are still rarely used by casual computer users. The tools generally require either an external hard drive or full access to multiple computers, which may not be the case for all computer users. Additionally, they provide no method for a user to access the backed up files from new locations, so they really only address the problem of reliability and not portability. This oversight makes user's see no benefit in any added complexity from their normal use patterns, and therefore causes them to not use the system. Only when they've suffered from a complete data loss will they value a backup-only solution.

In a corporate environment, data redundancy and availability is also very important. For this reason, several well designed tools exist in this domain that allow even casual users to enjoy the benefits of backups and portability. For instance, Microsoft Server 2003 features DFS (Distributed File System), which is a multi-honed collection of SAMBA share servers, presented as one entity to the end user. This way although a file appears to be written to one server, it is in fact replicated to many. It is common for DFS to be employed under the user profile layer, so a Windows user's My Documents folder (for instance) is stored on the DFS system. Also, since most corporate networks use Active Directory, sharing files is generally easy for the user as it is setup by an IT administrator as a folder on the DFS where multiple users have read and/or write permissions. Although this design is great for the corporate environment, its simplicity quickly breaks down once the casual user leaves the office or office computers. In general, most companies employ a VPN in order to access company servers from remote sites. Once this connection is established, the user must generally mount their network disks manually, which requires knowledge of both the path and login credentials for the share. This is a daunting task to the casual user, and for someone not already setup on the network, near impossible.

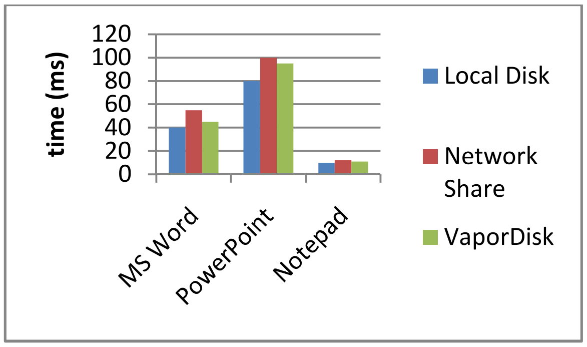
Recently, several cloud based solutions have been presented. Most of these are based in Linux and use the FUSE library to present themselves a user mode file system. Cumulus[2] is such a system for Linux. It uses the same type of backing store, but remains efficient by employing a "snapshot" system, allowing data from a single file to span across multiple snapshots if it has not changed. This design was motivated by their underlying goal of making efficient *complete* backups, which is not the goal of VaporDisk. It also adds a fair amount of complexity to the design, making parallel data retrieval difficult. Another key difference is that Cumulus *requires* access via the file system, while VaporDisk provides a simple to use HTTP interface. However, the main problem with Cumulus and other related systems is that it is tightly coupled with Linux, which is not the mainstream operating system for most casual computer users. So although it presents useful concepts for the physical process of backing up the data, it does not provide a complete solution to the problem. VaporDisk, on the other hand, is geared towards Windows users and provides a simple web interface for easy file access.

Finally, there are several other automated backup solutions that exist that do provide good support for both Windows and Linux and environments. For example, Mozy[3] is a commercial system that automatically performs backups in the background, and provides an online system for retrieving files that have been backed up. However, it is not presented to the user as a File System and concentrates more on backing up the user's documents on a nightly basis. This is useful for restoring in the case of a disaster. However, since the system is designed to be "transparent", it is difficult to tell which files are synchronized. Dropbox[4] is another commercial system that very recently became available. It answers many of the problems outlined in the above systems. It works in Windows, Mac and Unix, it integrates well into the shell of each of these operating systems, it provides intelligent feedback on which files are consistent, and it has an easy to use web interface. VaporDisk is very similar to DropBox except that it is not tied to any particular backend system (which we identify as "Synchronization Layers" later) and that it is implemented as a file system rather than a mirrored folder, making it more similar to a thumb drive to the user.

VaporDisk brings together several previously disjoint ideas while keeping in mind how to make a system that is easy to use and immediately beneficial to novice computer users.

Experiments

VaporDisk itself is a system that is designed to work transparent to the user, meaning effective application performance is very important. We have benchmarked the save times for several common user applications and compared them with other common methods.



The chart shows the experiment performed where we create a new document in Microsoft Word, Power Point, and Notepad. We then pasted a large amount of text into each document, and timed the time it took for the save to finish using a file system hook.

In the graph, lower numbers are better, and as suspected the local disk performs best. In all tests, VaporDisk performed as well or better than the network share, however, it is important to remember that VaporDisk returns to the host operating system as soon as its local cache is written. The delay for writing to the backing store is actually in fact much longer, and this is not shown in the graph. It is important to note that these benchmarks are approximate since it is nearly impossible to get precise timing of the events of closed source software such as Microsoft Word.

Additionally, the web interface is currently running for a single user and is available at http://www.vapordisk.com/6464.

Conclusions and Future Work

In this paper, we have presented VaporDisk, a simple, easy to use file system that provides automated backup and instant read-only accessibility via a clean web interface. The implemented system works very well and has been tested on real hardware using Amazon S3 and EC2.

However, VaporDisk has several extensions yet to be implemented. A file lock system is needed to allow multiple clients to access the same VaporDrive simultaneously, perhaps similar to Chubby[9]. This lock would allow users to aquire exclusive access to a file and release it at their choosing. Conflicts would be handled by creating a separate copy of the file for each conflicted instance (by VaporDisk machine name), allowing a human user to later resolve the conflict manually.

Additionally, we would like to write a synchronizer for our own backing store, which we would also provide for free, making this system available to users who already have reliable storage for no additional cost. This way we could distribute a truly "no added cost" system which encourages good backup practices with minimal complexity.

Finally, we would like to enable the web interface to add new files, share files among other VaporDisk users, and add simple permissions to share files publically on the internet.

VaporDisk has many possibilities for extension beyond the few mentioned here, and we plan to address many of these in the near future.

References

[1] Features of Windows Backup and Restore Center: http://www.microsoft.com/windows/windows-vista/features/backup.aspx, Accessed May 08, 2009

[2] Cumulus: Filesystem Backup to the Cloud. Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. Appears in Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST), February 2009

[3] Mozy By DECHO: *http://support.mozy.com/docs/en-user-home-win/faq/concepts/platforms_faq.html,* Accessed May 08, 2009

[4] Dropbox - http://wiki.getdropbox.com/FrontPage, Accessed May 08, 2009

[5] Creating Shell Extension Handlers - http://msdn.microsoft.com/en-us/library/cc144067(VS.85).aspx, MSDN, Accessed May 08, 2009

[6] A Toolkit for User-Level File Systems. David Mazières. Appears in Proceedings of the USENIX Annual Technical Conference, June 2001

[7] Dokan - http://dokan-dev.net/en/about/, Accessed May 08, 2009

[8] Amaon S3 - Dynamo: Amazon's Highly Available Key-value Store, Giuseppe DeCandia, Deniz Hastorun, et al. Appears in Proceedings of the 21st ACM Symposium on Operating Principles (SOSP), October, 2007

[9] Chubby Lock Service for Loosely Coupled Distributed Systems, Mike Burrows, Google Inc. 2006