Guaranteeing Consistency between Large Replicated Writable Disk Images

Sean Rooney Luis Garcés-Erice

IBM Research, Zurich Laboratory
8803 Rüschlikon, Switzerland
E-mail: {sro,lga}@zurich.ibm.com

***Abstract*—We describe a distributed protocol that guarantees the consistency of multiple replicas of a given disk image represented by a large file. The image replicas are used as read/write caches for the primary disk image located at a server. The protocol allows stale caches to be *incrementally* invalidated minimizing the amount of data that must be copied and recopied from the server. We prove that the protocol ensures the consistency of the disk from the point of view of any client using it, and we describe an implementation within an existing streaming framework.**

***Index Terms*—Migration, Protocol, Cache, Disk Image**

I. Introduction

Storage devices have been growing in size, and filesystems have followed supporting ever larger files. It is time consuming to work with these files in a distributed environment where the location of the file is required to change. For example, users may store these large files on a private cloud but may also require access to them when their portable computer is not attached to the network. Simply copying these large files back and forth is a very time consuming task, which may be further hindered by low bandwidth available to mobile workers. Virtualization is a primary example of applications that benefit from support for large files. A virtual disk image may be represented as one or more such files in some format, e.g. VMDK.

Streaming solutions have been developed in which operating system images can be executed on machines that are distinct from that which they are stored [1].allow a remote disk to be read and written across the network. Such a networked disk abstraction allows operating systems to only copy across the network those parts of the disk they actually access rather than the entire disk, reducing considerably the time to start the operating system. Streaming is useful when operating systems are often migrated between machines [2], or used with thin client desktops [3].

Caching the contents of the remote image disk on the local storage of the machine on which the OS executes offers benefits in terms of performance and scalability [4]. The cache is persistent, allowing the OS to be restarted with a minimal need for interaction with the storage cluster. In particular, this reduces the synchronized start-up problem [3]. Moreover, if the disk image is fully deployed to a client machine, that machine can be temporarily disconnected from the storage server allowing for example virtual machines to be executed in circumstances where no network is available.

Over time, a given OS image may be migrated among many different client machines. In particular, the OS may be migrated back to a machine on which was already executed. In such circumstances, the operating system should be able to benefit from any blocks that have been cached on the local storage and which are still valid. We describe a new distributed cache invalidation protocol that allows this to be achieved. The fundamental requirement of such a protocol is that the disk image should always appear consistent to any operating system using that image although the current valid state of the image is distributed across the network. A disk I/O protocol is consistent if the value returned from reading a disk sector via that system is always the same as the last value written to that sector via that I/O system. When reading/writing directly to a disk the system can only be inconsistent due to hardware failure. When reading/writing through a caching mechanism we must ensure that writes cannot be lost within the cache hierarchy. This is particularly problematic when the cache itself is distributed especially when the virtual machines using the cache may fail. The main contribution of this paper is the proof that the presented protocol guarantees the consistency of the disk from the point of view of distributed clients in a system with many stale read/write caches.

II. Protocol

Figure 1 shows an overview of the distributed disk cache system. Many disks may be available at the storage server and caches may exist for each of these at one or more clients. All caches are considered stale expect for that of the client currently using the disk. A sector is the basic unit in which a disk is read or written, a sector is typically 512 bytes in length. A disk may be considered as an array of sectors whose length is the size of the image divided by the sector size.

In a streaming system, there is a reference disk located on some storage infrastructure from which sectors are replicated and stored locally. The local cache is a replica of the disk with some additional information specifying whether the content of a given sector has already been read from the reference disk and is now available locally. It is also necessary to store

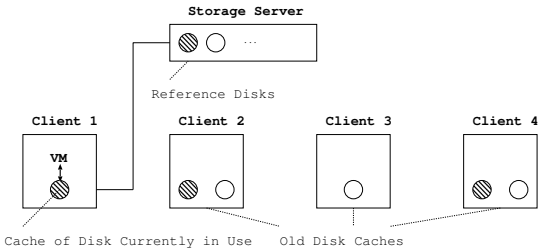

Fig. 1. Distributed Disk Caches

whether the sector content is "dirty", i.e. whether it has been written to locally and has not yet been flushed back to the reference disk. Clearly only one client may execute a given operating system instance at a given moment. We say that the client has started a session with the operating system and thereby *opened* the disk for use. A session identifier is a monotonically increasing integer. At a given time the state of the reference disk held at the server is given by:

```
struct {
    Sector [] disk;
    int session;
    int client;
    int [] sessionWritten;
} SState;
```

Where $disk$ is the representation of the disk, $session$ is the latest session, $client$ is the identifier of the client that currently is using the session or $None$ if there is no client using that disk, and $sessionWritten$ contains the session in which a given sector was last written to. At a given time the cache state at a client is defined by:

```
struct {
    int id;
    int session;
    Sector [] diskCache;
    Bit [] dirty;
    int [] sessionCached;
} CState;
```

Where id is the unique client identifier and $session$ is the current session of the client. $diskCache$ is the local cache of the disk. $dirty$ is an array of bits indicating whether a sector has been written to but not yet synchronized with the server, while $sessionCached$ is the session in which a sector was stored in the cache. For example, when a client reads sector k for the first time from the server, $diskCache[k]$ stores the value read, while $sessionCached[k]$ contains the current session identifier $session$. If the clients writes to sector k then $dirty[k]$ is set to 1, until such time as the write is flushed from the cache back to the server.

Having defined the state retained at the client and server we now formalize what it means to open and close a disk in Procedure 1 $OpenRemoteDisk$ and Procedure 2 $CloseRemoteDisk$.

From the pre-condition and post-conditions of $OpenRemoteDisk$ it can be seen that opening a disk essentially locks the disk for use by a given client. From the pre-condition and post-condition of $CloseRemoteDisk$ it can be seen that closing the disk releases the lock. The post-condition of $CloseRemoteDisk$ also reveals that when the disk is closed all writes *must* have been written to the server.

Procedure 1 OpenRemoteDisk

procedure OPENREMOTEDISK(SState,CState)
Require: $SState.client = None$
Ensure: $SState.client = CState.id \wedge$
$SState.session' = SState.session + 1$
end procedure

Procedure 2 CloseRemoteDisk

procedure CLOSEREMOTEDISK(SState,CState)
Require: $SState.client = CState.id$
Ensure: $SState.client = None \wedge$
$\forall i\ CState.diskCache[i] = SState.disk[i] \wedge$
$\forall i\ CState.dirty[i] = 0$
end procedure

Note that closing a disk does not change the state of the client's cache. This is retained on the local storage and may be reused at some time in the future. Parts of this cache may become invalid if another client performs a write on a cached sector. If the old sector is retained with the cache then the client's disk is no longer consistent. It is necessary to remove all elements in the cache that have been written in a session more recent than the client's last session. We describe the invalidation process by Process 3 $CacheInvalidation$.

Process 3 Cache Invalidation

for i *in* $CState.sessionCached$ **do**
  if $SState.sessionWritten[i] > CState.sessionCached[i]$ **then**
    $CState.sessionCached[i] \leftarrow None$
  end if
end for

In order to invalidate the read cache each client must know whether and in which session a sector has been written by any other client. This in turn requires that the server retains this information. Hence synchronizing the client disk to the server requires the procedure shown in Procedure 4 $WriteRemote$.

Procedure 4 WriteRemote

1: **procedure** WRITEREMOTE(SState,i,Data)
2: $Write(CState.disk, i, Data)$
3: $CState.dirty[i] \leftarrow 1$
4: $SState.sessionWritten[i] \leftarrow SState.session$
5: $Write(SState.disk, i, Data)$
6: $CState.dirty[i] \leftarrow 0$
7: **end procedure**

Procedure 4 $WriteRemote$ requires that lines 2, 3 are performed in a single transaction and that lines 4, 5, 6 are also performed transactionally. As lines 2, 3 are purely local operations this is straightforward to ensure, however lines 4, 5, 6 require coordination between local and remote state.

Moreover for performance reasons the actual writing to the server is performed asynchronously with respect to writing to the local disk.

The server updates the $SState.sessionWritten$ data structure each time a write on a given sector is performed. The client needs to have access to this data structure in order to perform the invalidation procedure. It would be possible each time a session is opened to pass the entire *SState.sessionWritten* structure to the client and allow it to perform *Invalidate* before the disk is used. The required data structure would need one integer for every sector meaning that for a terabyte disk we require 2×10^9 integers, or 8 Gigabytes to be transmitted. Increasing the scope of the $SState.sessionWritten$ structure from a single sector to a run of multiple sectors, e.g. 256, would decrease the size of the structure at the cost of unnecessarily invalidating more sectors. For example, our terabyte disk would now require a $SState.sessionWritten$ of size 32 Megabytes and each time a write was performed on a given sector a run of 256 sectors would be invalidated Our expectation is that the $SState.sessionWritten$ data structure is highly compressible, as many sectors share the same session number, for example after installing updates during a given session all modified sector will have the same session number. Standards compression methods, for example *gzip*, that use sliding windows to identify frequently occurring patterns and Huffman encoding to represent them, allow such structures to be compressed/decompressed efficiently. However, in the worst case every sector has a random session identifier and the structure is incompressible. This depends on usage patterns, but the expectation is that in normal usage over time $SState.sessionWritten$ will become less compressible. Assuming that to be the case then the time to open a disk image would increase over its lifetime.

We adopt a different approach; instead of transferring the entire data structure when the disk is opened, we can apply the same principle described for the disk itself and *stream* the $SState.sessionWritten$ as and when needed at the client, i.e. we represent the $SState.sessionWritten$ data structure as a streamable disk whose contents are meta-data. This means that only what is actually used is transferred and the cost of the transfer is amortized over the usage of the disk.

Procedure 5 ReadRemote

procedure READREMOTE(SState,CState,i)
  $x \leftarrow ReadSessionWritten(SState, i)$
  if $x > CState.sessionCached[i]$ **then**
    $CState.sessionCached[i] \leftarrow None$
  end if
  if $CState.sessionCached[i] \neq None$ **then**
    $result \leftarrow Read(CState.diskCache[i])$
  else
    $result \leftarrow Read(SState.disk[i])$
    $CState.diskCache[i] \leftarrow result$
    $CState.sessionCached[i] \leftarrow x$
  end if
  return $result$
end procedure

Given this, the procedure to read from a disk is presented in Procedure 5 $ReadRemote$. Note that $ReadRemote$ makes the invalidation Process 3 implicit, as the invalidation takes place dynamically as sectors are read from the reference image. Also note that the results of the operation $ReadSessionWritten$ in $ReadRemote$ can be cached themselves in a *meta-data* cache. This means that once a part of the cache is checked for validity from the server, all future accesses are local. This *meta-data cache* itself is only valid during a given session, i.e. it is completely emptied when the disk is closed.

III. Proof of Consistency

We define consistency of the disk in use by a client as a read operation on any sector of the disk always returning the last value ever written to that sector by *any* client.

We now prove that the operations described in Section II ensure consistency. The proof proceeds by contradiction; first let us assume the contrary i.e. that a write was performed on the disk by a $client_i$ but that write is not reflected later in the state of the disk at $client_j$ when that sector is read for the first time.

From Procedure 1 $OpenRemoteDisk$ and Procedure 2 $CloseRemoteDisk$ we know that only one client has access to the disk at a time and from the definition of $CloseRemoteDisk$ $client_i$ must have written all data to the server before $client_j$ opened the disk. Saying that the write is not available to $client_j$ is the same as stating that for some k which has not been written to by $client_j$ in the current session:

Assumption 6 Consequence of Disk Inconsistency

$ReadRemote(SState, CState, k) \neq Read(SState.disk[k])$

Which from the definition of Procedure 5 $ReadRemote$ is equivalent to stating that:

Assumption 7 Consequence of Disk Inconsistency

$CState.sessionCached[k] \neq None \wedge$
$Read(CState.diskCache[k]) \neq Read(SState.disk[k])$

If $CState.sessionCached[k] \neq None$ then it must have been read in a session previous to the last session of $client_j$, but as it has been written in a later session from the definition of $ReadRemote$ the value returned must be $Read(SState.disk[k])$ which contradicts Assumption 6. Therefore the protocol ensures the property of consistency.

IV. Implementation

We assume that the streaming infrastructure is using iSCSI [5], and the images are in raw format, that is they are of fixed size. Use of other protocols or image formats are possible, although implementation details would obviously change.

In Figure 2 a streaming system is depicted with a client using a local cache and the consistency protocol. Images are stored on the server in the image repository. An Image Manager entity exports the disk image and its metadata as iSCSI

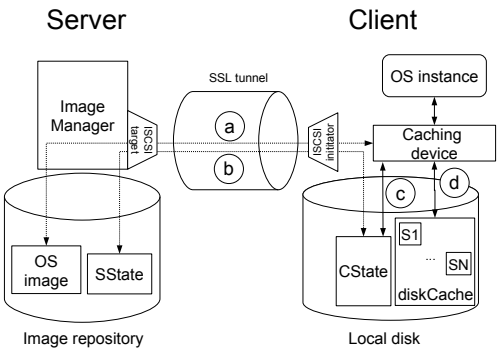

Fig. 2. A streaming system with local caching using the consistency protocol.

targets. On the client, the image is used through a caching device that keeps a local cache of the sectors $S_1, \ldots, S_N$ of the disk image. Sectors from the disk image are retrieved through the iSCSI connection ⓐ. The client state $CState$ is updated with the session information from the server state $SState$ through iSCSI connection ⓑ. When the caching device tries to read a sector not present in $diskCache$, $ReadRemote$ is used to update ⓒ the $CState$; the sector is brought to the client through ⓐ and written ⓓ to the $diskCache$. The caching device also writes ⓓ locally modified sectors to the $diskCache$ in the local disk. An example implementation by the authors of a caching device is described in [4].

OpenRemoteDisk: The target is created when the client calls $OpenRemoteDisk$. If the client has the right to access the disk, $OpenRemoteDisk$ creates a target.

CloseRemoteDisk: ensures that any dirty blocks are synchronized back to the server and then removes the target. Each individual update to the reference image is performed transactionally.

WriteRemote: uses the standard iSCSI write operation except that the iSCSI server also updates the $SState.sessionWritten$ structure, conveniently implemented using another iSCSI disk. At the server the iSCSI write operation is mapped to writing the data on the primary disk but also updating the meta-data.

Writes to the local disk are not synchronous with that of the remote disk as this would be prohibitively slow. Instead during the write the corresponding sectors are simply marked as dirty within the client's cache. Dirty sectors are flushed back to the server by a separate process that runs concurrently with the virtual machines. The process uses a two phase commit, in which the data to be written is first copied from the local disk to a journal and the sector is marked as clean. Only when the write is confirmed on the server is the journal removed. The journal is implemented using a simple file accessed in *direct mode* i.e. without using any filesystem cache. Write operations on the iSCSI disk are also synchronous, ensuring that the write actually takes place in the server before updating the journal. At any moment a given sector may be in one of three states: *dirty* (has not been synchronized with the server), *clean but in the journal* (is in the process of being synchronized with the server), *clean*.

ReadRemote: is the operation used by the caching device to read any sector of the disk. The $SState.sessionWritten$ structure is read using iSCSI to check in $SClient.sessionCached$ whether the local cache is valid for that sector. $SState.sessionWritten$ is typically in the cache and the request is satisfied immediately from RAM. The sector is returned directly from the local disk if the cache is valid, otherwise it is read through iSCSI and the local $SClient$ cache is updated.

V. Analysis

The cost at the server is the combined cost of the sectors that are read and written. Assume furthermore that R_s blocks of 4 KB are required to be read during the execution of the operating system and W_s blocks are written. Then for a pure streaming (ps) solution we have the following cost function:

$Cost_{ps} = Read_{cost}(R_s) + Write_{cost}(W_s)$

Both a pure streaming system and streaming with disk cache, benefit equally from the filesystem cache at the client so we do not need to consider this in the analysis. The cached streaming solution will reuse cached blocks that are still valid, but needs to read the metadata associated with these blocks in order to determine if they are valid. Assume an aggregation of 8 sectors per entry in $SState.sessionWritten$, giving one 4 byte integer for every 4 KB of data. Written blocks are stored on the local disk and only synchronized with the server at some period in the future. Here for simplicity we assume only a synchronization at the end meaning that only unique blocks are written to the server within a given session. The cost for the cached streaming (cs) system is then:

$$Cost_{cs} = Read_{cost}(R_s - R_c) + Read_{cost}(R_s/1024) + Write_{cost}(Unique(W_s))$$

Where R_c are the reads satisfied by the local cache directly. If a disk is reopened at the same client without any intervening execution on another client then the read cost is $Read_{cost}(R_s/1024)$ for the cached streaming solution compared to the $Read_{cost}(R_s)$ for the pure streaming solution. More generally it can be seen that for reads, $Cost_{cs} < Cost_{ps}$ if $R_c > R_s/1024$. For this *not* to hold the cache would need to be completely ineffective, i.e., consecutive runs of the operating system would need to use absolutely distinct disk parts, even for booting, which is not realistic.

Increasing the aggregation in $SState.sessionWritten$ would decrease the metadata cost, but might also decrease R_c, as more sectors get incorrectly invalidated.

VI. Conclusion

Disk images are an efficient way of archiving data and applications together with a supporting operating system. We have presented a simple protocol that allows cached copies of a disk image on different machines to be reused, reducing both the load on the server and the time required to synchronize the changes back.

REFERENCES

[1] B. Madden, "A better way to manage Citrix servers: centralized block-level disk image streaming," *Citrix White paper*, March 2006.

[2] C. C. Keir, C. Clark, K. Fraser, S. H, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *In Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI*, 2005, pp. 273–286.

[3] C. Spencer and C. Black, "Streaming and virtual hosted desktop study, benchmark results," *Intel White paper*, January 2008.

[4] D. Clerc, L. Garcés-Erice, and S. Rooney, "OS Streaming Deployment," in *Proceeding of IPCCC'10*. Albuquerque, NM, USA: IEEE Computer Society, December 2010, pp. 169–179.

[5] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner, *Internet Small Computer Systems Interface*, Network Working Group RFC, April 2004.