Vidi: A Venti To Go

Latchesar Ionkov
Los Alamos National Laboratory[*]
lionkov@lanl.gov

ABSTRACT

Vidi is a Venti proxy that allows certain clients to work when there is no connection to the Venti server. Vidi can be used on computers, such as laptops, to create archives of the file system even when disconnected, and later to transfer the archives to the Venti server. This paper describes an archival configuration used by the author as well as the design and implementation of a proxy that allows it to work in a disconnected state.

1. Introduction

Venti [7] archival server and the utilities for using it, Vac and Vacfs, allow simple and convenient way of keeping history of computer's files forever. Venti's interface doesn't allow data to be deleted or modified once it is stored. The fact that block's address depends on its content allows Venti to coalesce all blocks with the same content and keep a single copy in its storage. The archival utilities that use Venti don't need to implement complex algorithms to detect which files on the filesystem are modified. Archiving multiple filesystems with similar files leads to even better utilization of disk's space. Initially Venti was designed to replace Plan9's WORM [6] filesystem, but with Plan9 from User Space [2] the server and clients are also available for POSIX compatible operating systems.

A common Venti setup consists of a server with many disks, possibly in RAID configuration, and multiple clients in the same network archiving their filesystems daily on the server. This setup doesn't work well when the clients are mobile and can be used disconnected for long periods of time. If the clients cannot connect to Venti, gaps are introduced in history of the filesystem, data might be lost, and some of the important advantages of using Venti no longer exist.

One of the solutions for mobile computers is to run Venti locally, eventually copying the local Venti content to the central Venti server later. A major drawback for this approach is that Venti requires at least 105 percent as much disk space as the data stored. The disk space of the mobile computers is not as abundant as for the servers and desktops and the restriction is very often unacceptable.

Vidi introduces an alternative solution for disconnected archival that only uses 0.5 to 2 percent of the space required by running a local Venti server. Instead of keeping locally the content of all the blocks when making an archive, it keeps only the addresses of the blocks that were already sent to Venti, and the content of the blocks that were written while the Venti server was unavailable. Once the mobile computer is connected to its home network, Vidi copies the blocks to the central Venti server and deletes them from the local disk. A notable disadvantage of using Vidi is that it doesn't allow access to previous snapshots when disconnected.

2. Plan9 dump for Linux

2.1. Venti

Venti is a network storage system that uses a hash value of a block's content as an address for the block. Once data is stored into the Venti storage, it cannot be deleted. Venti provides simple interface for storing and retrieving data. When a client sends a data block for storage,

[*]LANL publication: LA-UR-08-05603

the Venti server responds with the SHA-1 [1] hash of the block contents called `score`. If the client needs to retrieve the data, it sends the score to the server and Venti sends back the block's content. The maximum size of a block Venti can store is 56 Kilobytes.

Using the SHA-1 hash of a block as an address allows the Venti server to detect blocks with the same content and ensure that they are stored only once on the disk. This property simplifies considerably the archival clients because they no longer need to figure out which files on the file system were changed. If the files are not modified, their subsequent archival is not going to use any more space in the archival system.

At a higher level, Venti supports storing and retrieving larger files by splitting them into blocks. The scores of the data blocks are combined into indirect blocks, their scores are combined further until a single score is produced that can later be used to retrieve the whole file. Venti files don't have names or any metadata information typically present for any modern operating system files. Venti also supports "directory" files that contain description (scores and some additional information) of Venti files. Each block in Venti has assigned a type value that indicates whether it is a data block, an indirect block (and the level of indirection), or a directory block.

Venti ships with utilities to store, copy or retrieve Venti files and directories.

2.2. Vac

Vac is a utility for storing files and directories in Venti. Venti converts the specified list of files into a list of Venti files and directories, saving the score of the top directory in a special *root* block. The score of the root block is returned to the user and can be used to retrieve the file hierarchy. Vac stores the regular files as a single Venti file. Because the Venti directories don't store files' metadata, each directory is represented with two Venti files – a Venti data file containing the metadata of the files from the directory, and a Venti directory.

Vacfs is a 9P [4] file server that given a score for a Vac root block can serve all the files stored with Vac. Vacfs can be used natively in Plan9, or using the v9fs [3] filesystem in Linux.

2.3. Using vac for archival

The Plan9 [5] dump filesystem provides a convenient view of its previous states. Each night a snapshot of the filesystem is taken and its content is available forever. The content of the file system on January 1st, 2001 can be reviewed by going to /n/dump/2001/0101 directory.

It is possible to achieve similar results on Linux by using Vac and Vacfs. Each night Vac is run to store the Linux filesystem in Venti, and the resulting score is saved in separate directory */YYYY/MMDD*. Then Vac is run again with `-m` option to expand and merge all vac scores in a single tree. The resulting score can be mounted using v9fs to provide the convenient Plan9 dump interface.

3. Vidi: archive when disconnected

Vidi is a server that speaks the Venti protocol. When the Venti server is available, Vidi acts as a proxy, redirecting client's requests to Venti, and Venti's responses back to the client. In addition to the redirection, Vidi builds a locally stored cache of scores for blocks that were sent to Venti. The cache is used in the disconnected state to detect blocks that Venti has and not store them for later transmission. Blocks whose scores are not present in the Vidi's score cache are saved in a block log. Both the score cache and the block log are stored on a local disk.

Figure 1 shows Vidi's operation when it is connected to Venti. Reading a block is always sent to the Venti server. The read operations don't affect Vidi's score cache. Writing a block first checks if its score is present in Vidi's score cache, and if it is present, a "success" response is sent back to the client without contacting the Venti server. Otherwise, the block is sent to the Venti server and on success, the score of the block is saved in Vidi's score cache.

When Vidi is not connected to Venti (Figure 2), read operations check if the score is present in the score cache, and if so whether the block is available from the local block log. In the unlikely case when the block is available locally, its content is sent back to the client, otherwise Vidi responds with an error. On write, if the score of the block is found in the score cache, a "success" response is sent to the client. Otherwise, Vidi appends the block content to its block log and adds the score to the score cache.

```
Get(score)
  if (data = venti.blockget(score))
    cache.putscore(score)
    respond(data)
  else
    responderror("not found")
```

```
Put(data)
  score = sha1(data)
  found = cache.putscore(score)
  if (found)
    respond(score)
    return

  if (venti.blockput(data))
    respond(score)
  else
    responderror(...)
```

Figure 1: Operation when Vidi is connected to the Venti server

```
Get(score)
  entry = cache.getscore(score)
  if (entry && blklog.valid(entry.address)
    data = blklog.read(entry.address)
    respond(data)
  else
    responderror("not found")
```

```
Put(data)
  score = sha1(data)
  found = cache.putscore(score)
  if (!found)
    found.address = blklog.append(data)

  respond(score)
```

Figure 2: Operation when Vidi is disconnected from the Venti server

Vidi keeps two pointers into the block log – of the first block that wasn't sent to Venti yet, and the position where the next block should be written to. When Vidi is reconnected to the Venti server, it starts sending the blocks from the block log to Venti. When all blocks are sent, i.e. the two pointers have the same value, the block log size is reset to the initial size. When a block is appended to the block log, its address doesn't directly reflect the offset where it is written in the log file. The block addresses always grow, even when the block log is shrunk after all blocks are submitted to Venti. This prevents updating the score cache addresses once the blocks are not in the block log anymore. When the block file is shrunk, Vidi updates a third pointer it keeps which keeps the logical address of the first block in the file. To check if a block for a score stored in the score cache is still available in the block log, Vidi checks if the "start" pointer is greater than the address of the block.

Unlike Venti's index, Vidi's score cache can drop scores of existing blocks. That can cause blocks that are already present in Venti (and even in the block log itself) to be added to Vidi's block log. The duplicates don't cause incorrect operation for Venti or Vidi. The only issue is the increased size of the local block log file. Our results show that with a reasonable size of the score cache, the number of duplicate blocks is not outrageous.

In addition to saving the score cache on a local disk, Vidi keeps some of the scores in RAM to improve the performance.

The prototype Vidi server is implemented for Unix operating system in 4000 lines of C code. It doesn't use the standard libventi libraries that are distributed with Plan9 from User Space.

3.1. Score cache disk layout

The disk layout (Figure 3) of Vidi's score cache is similar to Venti's index layout. The available disk space is divided into buckets (64K by default) and each bucket contains a map for a slice of the score space. The entries in the bucket are sorted by score. Unlike Venti, which depends on its index not overflowing, Vidi is designed to handle overflows and keep the most recently used scores in a bucket. Vidi doesn't keep a global LRU list. Instead it keeps per bucket LRU list. If a score needs to be added to a bucket, the least recently used entry in the bucket is

removed. In order to keep a LRU list, in addition to the block score, and its position in the local block log, the entry has pointers to the previous and next entry in the list.

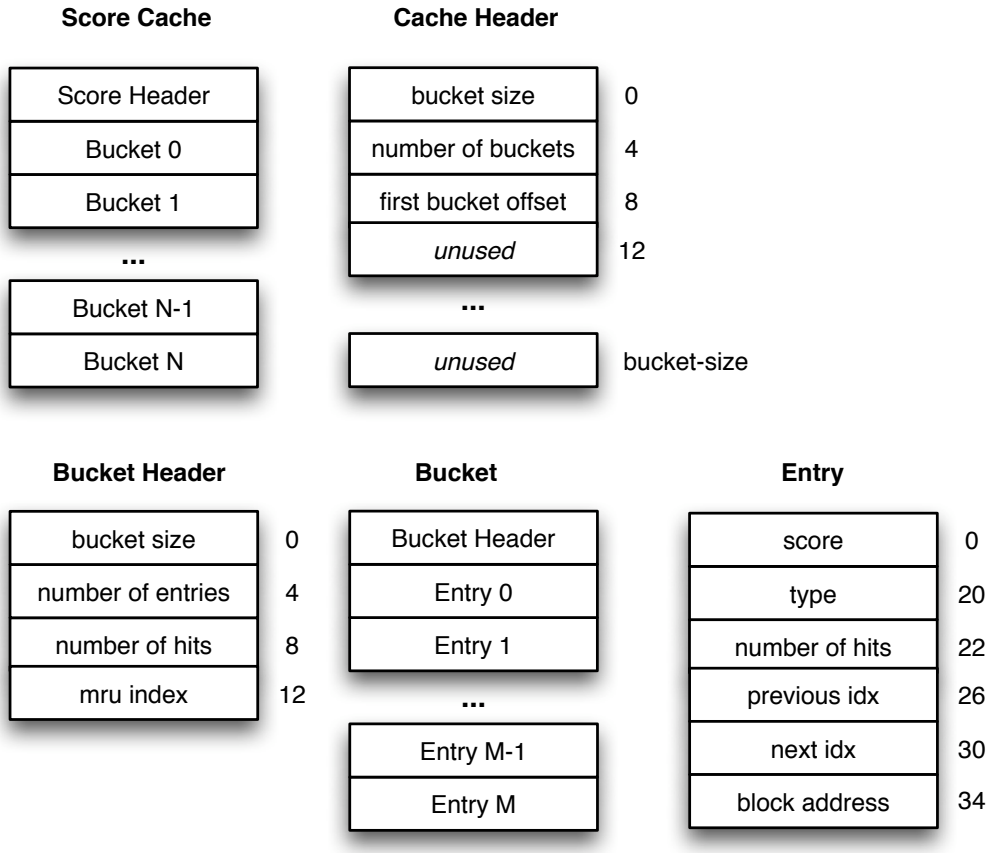


Figure 3: Score cache disk layout

3.2. Block log disk layout

Unlike Venti, Vidi's block log (Figure 4) is stored in a local file that is allowed to grow and shrink. The block log is not divided into arenas. The block log file consists of a header, list of data blocks and a trailer. The header contains a magic number and the "start" pointer. Each block contains a magic number, block's type, size and content. The log's trailer contains the "read" and "write" pointers. Vidi doesn't compress the block contents.

3.3. Using Vidi with Vac

When Vac is not used in an incremental mode, it converts the file system into a stream of "write" operations. Because Vac doesn't try to retrieve data from Venti, it would work well when connecting to Vidi even when disconnected. As Vidi doesn't always contact the Venti server even when connected, Vac's performance is improved even in non-incremental mode.

4. Performance results

The performance of the prototype is evaluated with different score cache and RAM cache sizes. The Venti server is running on a Linux server with 16 CPUs, 32GB RAM and 2.7TB arena space. The Vidi server is running on another Linux server with 2 CPUs and 2GB of RAM. Both servers are connected to the network with a Gigabit Ethernet card, but not to the same Ethernet switch and are 3 hops apart. The tests were performed using the vac program from Plan9 from User Space on a directory containing 117347 files with total size 11.38 GBytes. Before the tests were run, the directory was stored to the Venti server.

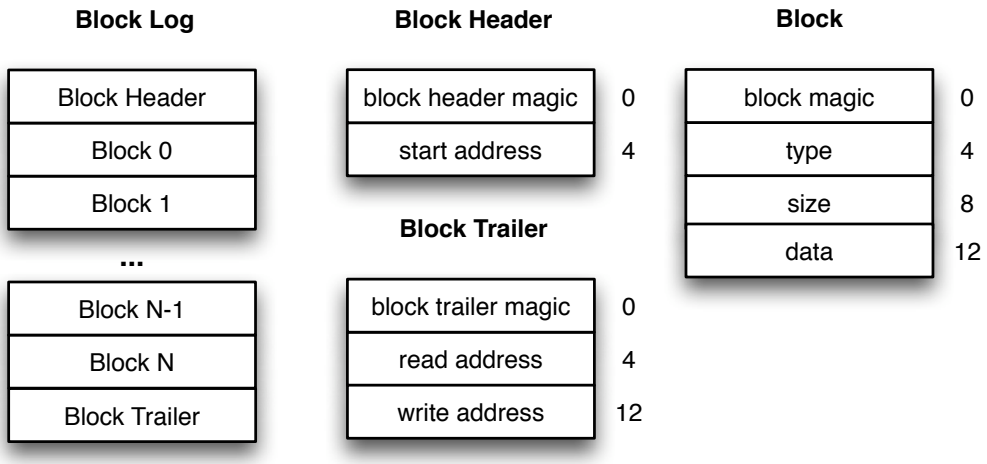
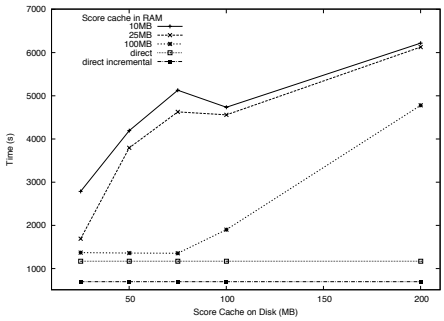
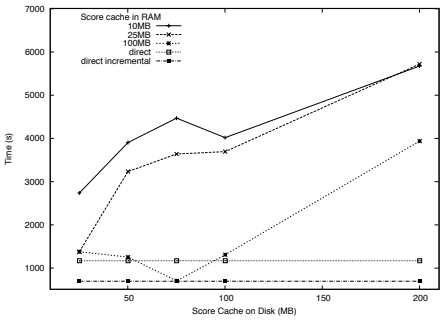


Figure 4: Block log disk layout

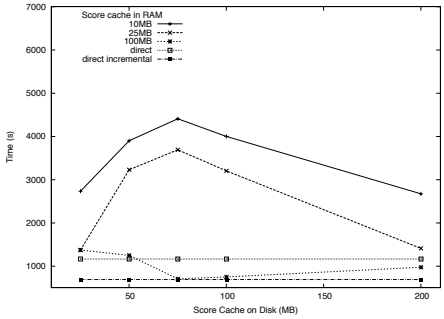
Figure 5 shows results running Vac with 64 Kilobyte buckets. Keeping information for the score recency leads to changes in the score cache even when the score is already present in the cache. This leads to higher number of operations to the local disk compared with the standard Venti which doesn't keep recency information per score. Having too small score cache leads to increased number of missed scores and even though the completion time is lower, Vidi stores an unacceptably high number of blocks (50 percent) even though they already are present in the central Venti server. Using too large score cache increases the I/O operations to the score cache too much decreasing the performance. The best results are achieved when the score



(a) Time to archive with empty score cache



(b) Time to archive with populated score cache



(c) Time to archive when disconnected

Score Cache Size (MB)	Cache Utilization (percent)	Block log size (MB)
25	100	6339
50	100	876
75	73	12.37
100	53	12.37
200	26	12.37

(d) Score cache utilization and block log size

Figure 5: Results using Vidi with 64 Kilobyte buckets.

cache is about 75 percent full. In that case, Vidi uses 0.74 percent of the storage a local Venti would use to archive the file system with performance comparable with the one achieved when using Vac in incremental archive mode.

Tests performed with smaller bucket size show improved performance at the expense of using more space used by the block log. Using smaller buckets reduces the I/O bandwidth, but the smaller number of scores in a LRU list increases the chance of score miss.

5. Conclusion and Future Work

Vidi allows standard Venti tools to be used for archiving when the central Venti server is not available. It caches locally the scores of the most recently written blocks. Vidi provides reasonable performance using a fraction of the disk space that other alternatives would use.

An interesting future work would be to extend Vidi to cache not only scores, but also the content of the blocks, allowing partial access to the archived file system. Experimentation with caching techniques other than LRU (ARC, MQ, etc.) could improve the hit ratio on both score and block cache further improving the performance and the user experience as a whole.

The implementation could be further improved by compressing the blocks in the local block log, and improving the I/O operations to the score cache.

References

[1] *Secure Hash Standard*. National Institute of Standards and Technology, Washington, 2002. Federal Information Processing Standard 180-2.

[2] Russ Cox. Plan9 from user space. http://swtch.com/plan9port/.

[3] Eric Van Hensbergen and Latchesar Ionkov. The v9fs project. http://v9fs.sourceforge.net.

[4] AT&T Bell Laboratories. Introduction to the 9p protocol. *Plan 9 Programmer's Manual*, 3, 2000.

[5] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[6] Sean Quinlan. A cached WORM file system. *Software — Practice and Experience*, 21(12):1289–1299, 1991.

[7] Sean Quinlan and Sean Dorward. Venti: A new approach to archival data storage. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 7, Berkeley, CA, USA, 2002. USENIX Association.