Proc. of The Second Intl. Conf. On Advances In Computing, Control And Networking - ACCN 2015
Copyright © Institute of Research Engineers and Doctors, USA .All rights reserved.
ISBN: 978-1-63248-073-6 doi: 10.15224/ 978-1-63248-073-6-13

Overcoming Malicious Read/Write Requests in Internet File Systems

Yosi Ben Asher Gabriel Mizrahi Gadi Haber

Abstract—Hashing of data blocks is a common approach in peer-to-peer and Internet based distributed file systems as a way to simulate a virtual hard disk over a set of servers. We consider the case where a malicious adversary attempts to overload a server assuming that this adversary “knows” how the underlying hashing scheme works. We overcome this problem by using Universal hashing a known theoretical solution. In this work we how to port such a complex hashing scheme in real implementations of Internet based file systems. In this respect our work also contributes to the well known problem of VOD (video on demand) over the Internet. The implemented system (IFS) supports cooperative caching and for concurrent read/write operations it supports Weak-Consistency. Our results suggest that Universal hashing based distributed file systems can be extremely efficient for driving Internet file systems and file systems for large clusters.

I. Introduction

We consider the problem of overcoming “dense/malicious” read/write requests of clients in multi-servers file systems (MSFSs). Such a file system should efficiently simulate a shared hard-disk created from many local disks over the Internet and support concurrent access to files created/deleted and updated by a dynamic set of web clients. IFS was specially designed to work with sudden “dense/malicious pattern” of requests namely cases where the pattern of client’s requests either deliberately or due to some need could have been mapped to a single server had we used current mapping techniques of files to servers. Basically, file systems (distributed, multi-servers and regular) can be viewed as a way to produce a map from the logical entities of files and directories to physical blocks of a hard-disk or a set of hard-disks. One of the main tasks of file systems is to maintain this mapping and keep it consistent for all the users of the system. Usually this mapping is realized by two search trees (also called the “meta-data”), one (the directory tree) to locate a specific file in the tree of directories and the other (the I-node tree) to locate physical blocks of files on the hard disk. In addition to the two search trees there is also a list of free blocks on the disk and other global information and a cache of the most recently used blocks. In MSFS using such search trees or even distributed versions of these trees (e.g., XFS [12]) may result in large communication delays as MSFS should satisfy the following requirements: A:They should allow a dynamic set of unknown clients to access files over the Internet concurrently. B: The storage space used to store files should be composed of many local disks of remote servers. These servers are often geographically dispersed over the Internet yielding significant communication delays. C: MSFSs should support concurrent read/write operations possibly to the same file. This excludes massive replication of data which prevent the users from obtaining a consistent view of the files. D: Due to the relatively long communication times, each access to a file (read/write) should involve very few servers as possible. E: It is preferable to let the client do most of the work needed to maintain the meta-data. Due to the above factors recent MSFSs tend to use hashing to map directories and physical block to servers. Basically, hashing based MSFSs view the shared disk as a large hash table of blocks partitioned between the disks of the servers. To execute read(file1, block255) a client may compute a hash value hash_1(“file1.255”) indicating the address of a physical block in the shared disk, and consequently the server address where the block can be obtained.

These potential advantages of hashing has been used in many MSFSs. Peer-to-Peer file systems where clients also play the role of the servers uses hashing in various forms. For example, in Freehaven [2] a document is split into n “shares” or blocks where any k shares can reconstruct the document. The shares are mapped to different servers according hashing values of the shares. In a read requests, all the servers holding shares of the document will send the share to the reader. Many other systems use hashing including: Peer-to=peer systems such as IVY [8] using Chord for lookup services, Pastry [11] and CAN [10] simulating a distributed hash table over the Internet; and hashed based storage/file systems such as Venti [9], CFS [1], LBFS [7] and Farsite [4] and OceanStore [6].

One problem not addressed by current hashing techniques used by the above systems is overcoming sudden “Dense Pattern” of read/write requests (DPR) that are mapped to a small subset of servers:

Definition 1.1: Consider a given set of k servers, n blocks and a mapping of these blocks to the k servers (possibly using a constant number of copies of each block). Let A be an adversary trying to find a “bad” pattern of k blocks that are all mapped to a small subset of servers. In each try A submit k requests and learns the resulting load at each server. Can we find a mapping of the n blocks that would require many tries of A before a “bad” pattern can be computed.

Note that this requirement is stronger than a requirement for average load balancing over all possible k requests. Practically, the above definition requires that the chosen mapping will be good for any sequence of requests which is not too large

Yosi Ben Asher, Gabriel Mizrahi, Gadi Haber
University of Haifa
Israel

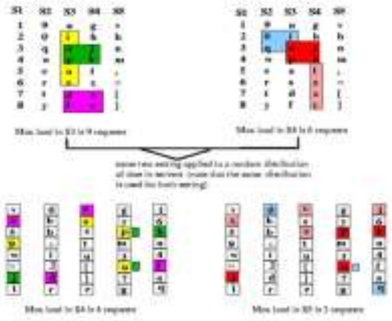

Fig. 1. Two cases of DPRs that are solved by using a different mapping of blocks to servers.

Though hashing is believed to have good load balancing properties it is clearly not capable of blocking bad cases of DPRs. This problem has been studied in a different context, namely that of simulating shared memory over a set processors each having a local module of memory. This problem, known as DMM (Distributed Memory Machines) simulation, was solved using a complex hashing schemes [3] which will be described later on. In this work we show that these hashing schemes can be also used for MSFSs. Intuitively, DMM simulation uses universal hashing to obtain an almost randomly mapping among the modules (the disks of remote servers in our case). In this way the probability that a sequence of requests will fail (one of them will load some memory module) is low. Special care was also given to implementing other aspects of MSFS such as cooperative caches namely fetching blocks from near clients instead of from remote servers. The issue of maintaining some notion of consistency had to be checked as well. This was done in order to verify that using such complex hashing scheme does not prevent the usage of cooperative caches and consistency notion.

II. General mode of operation

We describe general features of the IFS and how it works. We mainly focus on the way hashing can be used to create a file system which satisfies the requirements given in the introduction. The IFS consists of the following components: 1) A large virtual hard-disk (VHD) is created over the servers of the IFS, consisting of equal sized storage areas in each machine. The blocks of the VHD are distributed among the local disks of the machines constituting the IFS. Each machine runs an IFS server that send/receives physical blocks of the VHD to the clients. There can be several user applications which access the VHD through the same IFS client. The client holds suitable data to allow each application to access its set of opened files. 2) The mapping between logical blocks $\mathbf{f}\text{ile} \times \#\text{block}_{\text{infile}}$ and physical blocks on the VHD is made by means of hash functions. This mapping forms the first level of hashing as described in figure 2. 3) IFS clients can use local caching; when the cache becomes full, the client flashes a block from the cache to the suitable servers. The IFS allows a client to fetch a block from a client's cache rather than from a remote IFS server. This mechanism is called "cooperative caches" and is used to reduce the load of sending complete blocks from the servers. 4) There is another level of hashing

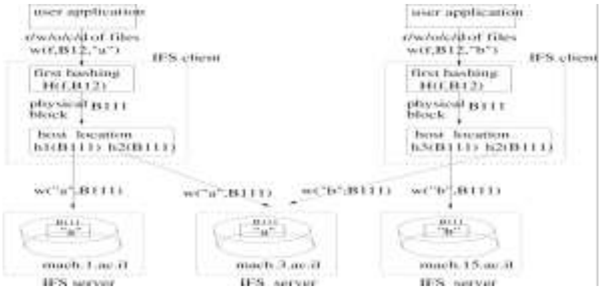

Fig. 2. Two clients updating the same block (main stages).

which is responsible for distributing the physical blocks of the VHD among the IFS servers. This level (refer to as the second level of hashing in figure 2) is used to generate pseudo-random distribution of the blocks in the VHD as described in the introduction. This level of hashing partitions the load equally among the IFS servers and allows concurrent access to the VHD. 5) Each physical block is replicated three times, on three different IFS servers. Each read/write operation of a block requires fetching/updating two out of these three copies (also called the "majority rule"). Time stamps are used to determine which of these two copies is the most updated. 6) Using local caches destroys the consistency of the VHD and each client can have a different view of the VHD. IFS allow applications to use a mechanism of "tokens" in order to update caches with a correct view of the VHD.

Figure 2 depicts the main stages in the execution of two concurrent write operations $\text{w}(\mathbf{f}, \text{B}12, \text{"a"}) || \text{w}(\mathbf{f}, \text{B}12, \text{"a"})$ made by two IFS clients. This involve three IFS servers holding the three copies of B12. The majority rule of IFS implies that the result of these two writes is that B12 will contain "a".

III. Detailed operation of the IFS

Following the basic idea of DMM simulations, the hashing scheme used by the IFS to access data works as follows:

- There is a family of universal hash functions $H = \{h_{a,b}(x) = ((a \cdot x + b) mod\ k) mod\ p$ where $k = prime(/vhd/), a, b \in [1, ..., k]$ and p is the number of servers (prime) and $/vhd/$ is the size of the virtual hard disk (total number of blocks).
- Initially we choose at random three functions h^1, h^2, h^3 from H by choosing their coefficients a, b at random, and store them along with some other information at a common location. When an IFS server ($\text{IFS}_{\boldsymbol{i}}$) starts to run (our analogue to the mounting operation of regular file systems) it will access this common location and thus gets the chosen hash functions and a pointer to its local part in the VHD. Note that the same machine can host several separate IFSs. Each server($\text{IFS}_{\boldsymbol{i}}$) is responsible for storing every block B_x of $\boldsymbol{f}$ for which either $h^1(B_x) = i, h^2(B_x) = i$ or $h^3(B_x) = i$ consequently there will be three copies of each block. For fault tolerance ability, one should resolve the rare case where $h^1(B_x) = h^2(B_x) = h^3(B_x)$ so that not all three copies will be stored on the same machine.

- When a client wants to access (read/write) a block B_x which is not present in its local cache it sends requests to two servers. Fetching a block B_x to the cache of client CL_i requires fetching two copies B^1_x, B^2_x out of the three possible. This is done using the three functions h^1, h^2, h^3 in a random order to fetch two copies. Of the two copies, we select the one with the highest global time tag, and store it in the cache. The global time tag indicates the last global time that this block. has been updated. Finally, the list of clients associated with B_x in the two servers from which B_x was fetched is updated to include the fact that the initiating client has the most recent copy of B_x in its cache.
- When a client's cache becomes full the least recently used blocks are "flashed" to the servers to be stored. Assume that a client wants to flash B_x: we choose (at random) two hash functions (out of h^1, h^2, h^3) and send two copies of B_x to the suitable servers. The list of clients associated with B_x (in the two servers from which B_x was flashed to) is updated to include the fact that the initiating client does not have a copy of B_x in its cache. Note that the list of clients associated with B_x in one of the servers can point to a client that does not have a copy of B_x, in case this client has already flashed B_x to a different server from the one it was originally fetched from. Finally, the current version does not support "diffs", so when a block is flashed to a server it overwrites the old version of the block at that server. Consequently, for the current version the consistency unit is a block.
- Each server is multi-threaded and thus can handle several requests from different clients simultaneously. The load per server is thus the number of active requests that are being served in a given time unit.

IV. Metadata and Commands in the IFS

We describe briefly how the metadata is organized in the clients and the servers and how the basic commands of the IFS are executed. As explained in the introduction, the metadata used in the IFS system should be fully distributed and accessing it should never cause communication among the servers or among the clients. Due to the direct access property of the hashing the metadata in IFS does not include any search trees as in other DFS systems but is mainly focused to manage the files generated by the clients as follows:

Filezero- is a special file in which each block describes a given file (e.g., filenumber (fnum), name, size, last update, permissions, etc.). Thus the information on existing files is kept on the VHD and is accessed when we mount a set of servers. A special property of filezero is that there is only one copy of each block (only one hash function is used to access it), and its blocks are not accessed through the cache. Consequently, only one client can access a given block of file zero at any given time, so that filezero is always consistent with all the clients.

Table of blocks- is a hashed list of all the blocks used in every server; each entry points to a physical block on the local storage of the current server. Each entry contains a bit indicating if this block is free or being used by a file. This is needed due to the fact that the hashing can cause collisions and we have to prevent overwriting blocks on the VHD due to collisions. In addition, in the current version we associate with each block a pointer to the last client that requested this block.

Table of open files- Each server has a table of all the opened files (opened by clients so far). For each such file there is a counter counting the number of clients that have opened this file so far. Each entry in this table points to the entry of this file in the table of blocks. The client has also such a table, but with a more complex structure since it has to support several processes/applications which read/write from different locations in a given file.

The set of commands of the IFS include:

create(fnum, params...)- creates a new file with file number *fnum*. A client executing this command has to access the server that holds the suitable entry in filezero and to update it if necessary. Note that since the blocks of filezero are not replicated, there is only one server that needs to be updated.

delete(fnum)- deletes a file making its blocks on VHD free so that they can be used by future write operations. The delete is never executed until all the clients that have opened this file have closed it (using the counters in the tables of open files). Since there is only one server which "owns" the entry of *fnum* in filezero, only one client can delete a file at any given time (i.e., future delete operations will be directed to that server and will be rejected). Similarly, it is not possible to open or create a file which is currently being deleted. After the server that owns the suitable block in filezero acknowledges the delete, the initiating client sends a "free" instruction to all the servers indicating that they should mark all the blocks allocated to this file as free. When all the servers acknowledge that all the blocks of the deleted file have been marked as free (searching in the table of blocks) the client can instruct the master server to modify the entry of that file in filezero. The open/close instructions work in a similar way, updating the counter of opened files in the table of opened files in the server that owns the entry of the underlying file in filezero.

write(fnum, buffer, #bytes)- writes *#bytes* from the buffer at the "end" of the file fnum. The write operation verifies that the file has been opened by the initiating client. The client has suitable data-structures (similar to those used in common file systems) allowing a set of processes on the client's machine to write at the end of the files which they have opened. The data is written to the local cache to be flashed later on to two servers when the cache becomes full. The read instruction works in a similar way. IFS also supports seek operations which are basically local operations of the client.

V. Consistency

We now discuss consistency problems in the IFS and how they have been addressed. Consistency problems are basically generated when clients can have different views of the vhd.

This can happen when the local caches of two or more clients have different values of the same block.

We consider two main types of consistency models: Strict consistency: This requires that a read operation will return the value written by the most recent write operation according to a global clock (all Unix file systems and many other obey this semantic). Implementing strict consistency in a DFS implies that before every update of a block, all the copies of this block in the caches of other clients must be invalidated. These invalidations are too costly, so most Distributed file systems do not implement Strict consistency. In AFS, for example, Strict consistency is implemented only at the granularity of complete files. Release consistency: Introduced in [5], this allows multiple copies of a block in several clients to be updated concurrently without any invalidation. It assumes an explicit synchronization instruction of exchanging a token between two machines, such that after receiving the token the local cache of one machine is made consistent with the cache of the machine from which the token was received. This is a very efficient form of consistency, but requires the programmer explicitly to use the token mechanism.

We currently approximate strict consistency by using the local clock for time stamps and leave it to the application

ing/storing blocks. Release consistency in IFS is supported via a token mechanism, which works as follows. Each server has a unique token which can be held by at most one client. A client that tries to acquire a token is blocked until the token is released by another client or by the server that owns the token. A list of updates is associated with every token; each update in the list describes a modification of some block made by a client to a cached copy of that block when this client acquired the token. Thus, accessing the vhd can be serialized if all the writes to a given file or block are executed under the scope of acquiring and releasing a pre-designated token. By acquiring a token, a client updates its cache using the update list. Similarly, when a token is released the updates made by the client while holding the token are added to the update list of that token. Several techniques can be used to keep the update list from becoming too big (the current version of the IFS supports one of them). Using the tokens is an expensive operation because the update list must be sent through the Internet. It thus seems more reasonable not to use the tokens frequently but instead to limit the cache size to a small number of blocks. In this way the servers are frequently updated by blocks that are being flashed by the clients, yielding relatively consistent view of the VHD.

VI. Experimental results

Two experiments where designed to test two premises of the IFS: A: For a fixed number of clients, using more servers, should increase the quantity of blocks that each client can receive in a time unit. B: The DMM simulation scheme used in the IFS successfully distributes the load among the servers for sufficiently large sequences of requests. Note that we do not test the system against a malicious adversary trying to fail the system as this property is theoretically proved in the DMM simulation problem. We only focus on showing that the system work efficiently in spite of the underlying complex hashing scheme. We used a cluster of about 16 PC machines connected by two fast Ethernet switches to approximate an IFS setting over the Internet. Each client read consecutive blocks from a small set of files that where written on the VHD before the experiments took place. A central monitor was used to measure the amount of block that each client received and each server sent in a given time interval.

To test the expected slowdown caused by serving too many clients running a video player, one must use significantly more machines than those available to us. For example, we tested a system with one server and several video players clients and still observed reasonable display rates and reasonable video images. Evidently, such a small number of machines is not sufficient to produce meaningful loads the IFS such that viewing a video movie is damaged (as frequently happens when video movie is watched over the Internet). We therefore used in some cases an artificial client which just read the blocks of a video file but did not display them and by so obtained clients which can "load" the system (each such client represents several read VOD clients watching movies over the

Figure 3 describes the average number of received blocks of one client, in a time unit, as a function of different number of clients (*C* =) and servers (*S* =)used in the IFS test configuration. For example, the results for five clients show a clear speed up when more servers where used, e.g. 500 blocks with five servers compared with 125 blocks with two servers. In general, these results verify the expectation that the hashing scheme balances the load among the servers. However, some non-linear effects are also revealed by these results, e.g., using more clients (8 . . . 13) with five servers did not decreased the average number of received blocks as expected. We assume that these are only negligible effects caused by the average operation of the number of packets received by each client.

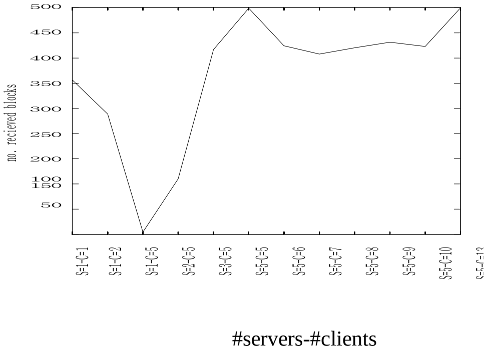

Fig. 3. Average number of received blocks in a time unit for a single client.

We were thus ready to check more closely the distribution of non-balanced situations. We set a monitor which synchronized all the servers and the clients in the system to work synchronously by waiting for acknowledgment from the monitor after each message sent to a client or to a server. The monitor measured the number of servers that were not serving any client (called "empty" servers) in a time unit of two steps of the monitor. The results in figure 4 show that in most of the units (samples in the figure) there were no empty servers. This, verify the premise of using DMM simulations to distribute the load among the servers. Observe that using more clients than servers improves the distribution. Thus the fluctuations of the previous chart can also be attributed to the fact that in these experiments the number of clients was less than the number of servers. Other similar experiments also verified the premise of the DMM simulation even for the relatively small number of servers and clients used in our experiments. It is of course reasonable to assume that since we are dealing with a probabilistic process, much better results would have been obtained for large numbers of clients and servers. Finally, we also measured the difference in the number of requests (called max difference) between the most loaded server and the least loaded server. The max difference was measured against various number of samples, i.e., the max difference in k samples is the maximal number of requests a server received in k samples minus the minimal number of requests a server received in those k samples. Clearly, the probability of finding larger differences the is related to the size of the sample. There were up to 13 clients in this experiment so that the maximal difference is always $\leq 13 \cdot k$ Where k is the size of the sample. The results in figure 5 show that in most of the samples the maximal difference was relatively small, $2 - 6$.

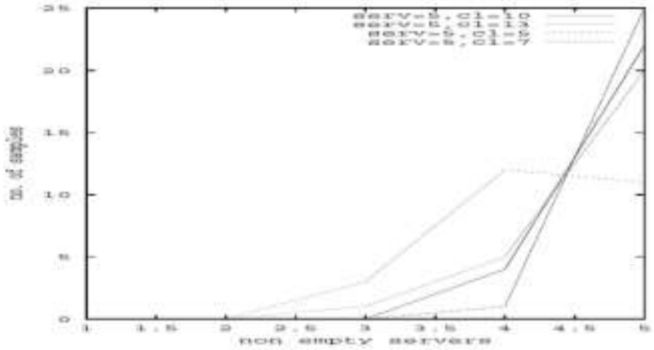

Fig. 4. Histogram of idle servers per step.

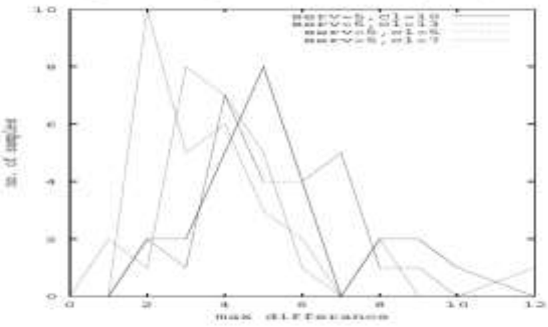

Fig. 5. Histogram of max difference (5 servers).

REFERENCES

[1] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01), Chateau Lake Louise, Banff, Canada, October 2001.

[2] R. Dingledine, M. J. Freedman, and D. Molnar. The free haven project: Distributed anonymous storage service. In Workshop on Design Issues in Anonymity and Unobservability, number 2000 in LNCS, pages 67–95, 2000.

[3] M. Ditzfelbinger and F. Meyer auf der Heide. Efficient shared memory simulations. In In Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures, pages 110–119, 1993.

[4] J. Douceur, A. Adya, W. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In International Conference on. Distributed Computing Systems ICDCS, 2002.

[5] K. Gharachorloo, D. E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In Proc. of the 17th Annual Int'l Symp. on Computer Architecture, pages 15–26, May 1990.

[6] John K., D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In Proceedings of ACM ASPLOS. ACM, November 2000.

[7] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In Symposium on Operating Systems Principles, pages 174–187, 2001.

[8] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In Proceedings of 5th Symposium on Operating Systems Design and Implementation, 2002.

[9] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In First USENIX conference on File and Storage Technologies, Monterey,CA, 2002.

[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In Proceedings of ACM SIGCOMM 2001, 2001.

[11] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), pages 329–350, November 2001.

[12] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In Proceedings of the USENIX 1996 Technical Conference, pages 1–14, San Diego, CA, USA, 22–26 1996.