GANESHA, a multi-usage with large cache NFSv4 server

Philippe Deniel Thomas Leibovici Jacques-Charles Lafoucrière
CEA/DIF
{philippe.deniel,thomas.leibovici,jc.lafoucriere}@cea.fr

Abstract

GANESHA is a user-space NFSv2, NFSv3, and NFSv4 server. It runs on Linux, BSD variants, and POSIX-compliant UNIXes. It is available under the CeCILL license, which is a French transposition of the GPL and is fully GPL-compatible. The protocol implementation is fairly complete, including GSSAPI security hooks. GANESHA is currently in production at our site, where, thanks to a large cache and a lot of threads, it delivers up to a hundred thousand NFS operations per minute. This paper describes the current implementation as well as future developments. This includes GANESHA as a NFS Proxy server and NFSv4.1 enhancements, but also the access to LDAP and SNMP information using the file system paradigm.

1 Introduction

NFS is a well known and venerable network protocol which is used widely. NFSv4 is the latest version of the protocol. It fully reconsiders its semantic and the way NFS can be used.

We manage a huge compute center at CEA. In the past three years, we had to face a strong increase in the amount of data produced by our supercomputer, up to tens of terabytes a day. Archived results and files are stored in HPSS, a third-party vendor's HSM which had a NFS interface. NFS fits our need well in terms of files meta-data management, but there were several limitations in the product that made for a difficult bridge between the HSM and NFS, and we believed it was time to step to something new. The HPSS product has a user-space API, complete enough to do all manipulation on files and directories. The decision to write a brand new daemon to handle the NFS interface we needed to HPSS was natural, but the following ideas lead the design process:

- The new product should be able to manage very large data and meta-data caches (up to millions of records), to avoid congestion on the underlying file system.
- The new product should be able to provide the NFS interface we needed to HPSS, but should also be able to access other file systems.
- The new product should support the NFSv4 protocol, and its related features in term of scalability, adaptability, and security.
- The new product should be able to scale as much as possible: software congestion and bottlenecks should be avoided, the only limits would come from the hardware.
- The new product should be a free software program.
- The new product should be running on Linux, but portable to other Unix platforms.

These considerations drove the design of GANESHA. This paper will provide you with additional information about it. The generic architecture and the way it works will be described and you'll see how GANESHA can be turned into a "very generic" NFS server (using only POSIX calls from LibC) or a NFSv4 Proxy as well. Information will also be provided on the way to write packages to extend GANESHA in order to make it manage various names-spaces.

The paper first describes NFSv4 and the technical reasons that lead to a user-space NFS daemon. The architecture of the product is then detailed including the issues that were met and how they were solved. Some actual results are shown before concluding.

2 Why a NFSv4 server in User Space?

GANESHA is not a replacement for the NFSv4 server implemented in the kernel; it is a brand new program, with its advantages and disadvantages. For some aspects, the NFSv4 server in the kernel should be more efficient, but there are several domains (for example building a NFSv4 Proxy server) in which the user-space approach will provide many interesting things.

First of all, working in user space makes it possible to allocate very large piece of memory. This memory can then be used to build internal caches. Feedback of using GANESHA in production showed that 4 Gigabytes were enough for making a million-entry cache. On a x86_64 platform, it is possible to allocate even bigger memory chunks (up to 16 or 32 GB, depending on the machine's resources). Caching about 10 million entries becomes possible.

A second point is portability. If you write kernel code, then it will be acquainted with the kernel's structure and it won't be possible to port it to a different OS. We kept Linux (i686 or x86_64) as the primary target, but we also wanted to compile and run it on different architectures, keeping them as secondary targets. Most of the Free Software Community is very close to Linux, but there are other free operating systems (FreeBSD or OpenSolaris) and we have wanted to be compatible with them since the beginning of the project. Another consideration is the code itself: something that compiles and runs on different platforms is generally safer than a "one target" product. Our experience as developers showed that this approach always pays back; it often reveals bugs that would not have been so easily detected on Linux, because resources are managed differently. Portability doesn't only mean "running on several OSes," for a NFS server it also means "managing different file systems." The NFSv4 semantics bring new ideas that need to be considered there. The NFSv2 and NFSv3 protocols have semantics very close to the way Unixes manage file systems. Because of this, it was almost impossible to have NFS support for a non UNIX-related file system. One design consideration of NFSv4 was to make the protocol able to manage as many file systems as possible. Because of this, it requires a very reduced subset of file/directory attributes to be supported by the underlying file system and can manage things as simple as a FAT16 file system (which has almost none of the attributes you expect in "regular" file systems). When designing GANESHA, we wanted to keep this idea: managing as many file systems as possible. In fact, it is possible with the NFSv4 semantics to manage every set of data whose organization is similar to a file system: trees whose nodes are directories and leaves are files or symbolic links. This structure (that will be referenced as the *name-space* structure in this paper) maps to many things: files systems of course, but also information accessible through a SNMP MIB or LDAP-organized data. We choose to integrate this functionality to GANESHA: making it a generic NFSv4 server that can manage everything that can be managed by NFSv4. Doing this is not very easy within the kernel (kernel programming is subject to lots of constraints): designing the daemon for running in user space became then natural.

A last point is also to be considered: accessing services located in user space is very easy when you already are in user space. NFSv4 support in the kernel introduced the *rpc_pipefs* mechanism which is a bridge used by kernel services to address user-space services. It is very useful for managing security with kerberos5 or when the *idmapd* daemon is asked for a user-name conversion. This is not required with GANESHA: it uses the regular API for the related service.

These reasons naturally lead the project to a user-space daemon. We also wanted to write something new and open. There was already an efficient support of NFSv4 support within kernel code. Rewriting something else would have had no sense. This is why GANESHA is a user-space daemon.

3 A few words about NFSv4

NFS in general, and more specifically NFSv4, is a central aspect to this paper. People are often familiar with NFS, but less are aware of the features of NFSv4.

NFSv2 was developed by Sun Microsystems in 1984. It showed limits and this lead to the birth of NFSv3, which was designed in a more public forum by several companies. Things were a bit different with NFSv4. The protocol has been fully developed by an IETF working group (IETF is responsible for standardization of protocol like IPv4, IPv6, UDP, TCP, or "higher-level" things like FTP, DNS, and HTTP). The design began with a birds-of-a-feather meeting at IETF meetings. One of the results was the formation of the *NFS version 4* working group in July, 1997.

Goals of the working group when designing the protocol were:

- improve access and performance on the Internet;
- strong security with negotiation built into the protocol;
- easier cross-platform interoperability;
- the protocol should be ready for protocol extensions.

NFSv4 integrates features that allow it to work correctly on a WAN, which is a network with low bandwidth and high latency. This is done through using experience obtained with protocols like WebNFS. NFSv4 will then use compound requests to limit messages and send as much information as possible in each of them. To reduce traffic, the caching capability were truly extended, making the protocol ready for implementation of very aggressive caching and an NFSv4 proxy server.

Scalability and availability were improved, too; a strong stateful mechanism is integrated in the protocol. This is a major evolution compared to NFSv2 and NFSv3, which were stateless protocols. A complex negotiation process occurs between clients and server. Due to this, NFSv4 can allow a server with a strong load to relocate some of its clients to a less busy server. This mechanism is also used when a client or server crash occurs to reduce the time to full recovery on both sides.

Security is enhanced by making the protocol a connection-oriented protocol. The use of RPCSEC_GSS is mandatory (this protocol is an evolution of ONC/RPC that supports extended security management—for example the krb5 or SPKM-3 paradigm—by use of the GSSAPI framework) and provides "RPC-based" security. The protocol is connection-oriented, and will require TCP (and not UDP like NFSv2 and NFSv3), which makes it easier to have connection-based security.

The structure and semantics of NFSv3 were very close to those of UNIX. For other platforms, it was difficult to "fit" in this model. NFSv4 manages attributes as bitmaps, with absolutely no link to previously defined structures. Users and groups are identified as strings which allow platforms that do not manage uid/gid like UNIX to interoperate via NFSv4.

The protocol can be extended by support of "minor versions." NFSv4 is released and defined by RFC3530, but evolutions are to be integrated in it, providing new features. For example, the support of RDMA, the support of the PNFS paradigm, and the new mechanism for "directory delegation" are to be integrated in NFSv4. They will be part of NFSv4.1, whose definition is in process.

4 Overview

This section describes the design consideration for GANESHA. The next sections will show you how these goals were achieved.

4.1 The CeCILL License

GANESHA is available as a Free Software product under the terms of the CeCILL license. This license is a French transposition of GPL made by several French research organizations, including CEA, CNRS, and INRIA. It is fully GPL-compatible.

The use of the GNU General Public License raised some legal issues. These issues lead to uncertainties that may prevent contributions to Free Software. To provide better legal safety while keeping the spirit of these licenses, three French public research organizations, the CEA, the CNRS, and INRIA, have launched a project to write Free Software licenses conforming to French law. CEA, CNRS, and INRIA released CeCILL in July, 2004. CeCILL is the first license defining the principles of use and dissemination of Free Software in conformance with French law, following the principles of the GNU GPL. This license is meant to be used by companies, research institutions, and all organizations willing to release software under a GPL-like license while ensuring a standard level of legal safety. CeCILL is also perfectly suited to international projects.

4.2 A project on Open-Source products

GANESHA was fully written and developed using Free Software. The resources available for system programming are huge and comprehensive, and this made the task much easier on Linux than on other Unixes.

The tools used were:

- *gcc* (of course...)

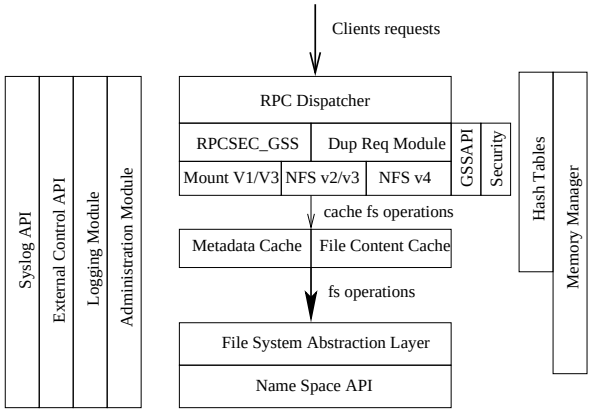

Figure 1: GANESHA's layered architecture

- *gdb* for debugging, often used jointly with Electric Fence or the Dmalloc library for memory debugging.
- *valgrind* for caring about memory leaks.
- *doxygen* for generating the various documents about the APIs' calls and structures.
- *GIT* as source code repository manager.
- *PERL* and *SWIG* to wrap API calls in order to write non-regression scripts.
- *Connectathon test suite* which is a test suite designed for the validation of NFS client-server behavior.
- *PyNFS* a non-regression test tool written in Python by the CITI folks.[1]

4.3 A layered product

GANESHA is designed as a layered product. Each layer is a module dedicated to a specific task. Data and meta-data caching, RPCSEC_GSS and protocol management, accessibility to the file system... All these functionalities are handled by specific modules. Each module has a well defined interface that was designed before starting to write a single line of code. Such a modular design is good for future code maintenance. Furthermore, one can write new algorithms within a layer without changing the rest of the code. A better description is that cache management could change the cache layers, or a different name-space could be managed, but these changes should not impact the other modules. Efforts were made to reduce adherences between layers. This was costly at the beginning of the project, but on a mid-range time scale, it appeared that this simplified a lot in the rest of the project. Each layer could be developed independently, by different developers, with their own validation and non-regression tests. A "global make" step can then re-assemble all the pieces. It should be reduced if all of them complete their validation tests.

A few modules are the very core of GANESHA:

- The Buddy Malloc module manages the memory used by GANESHA.
- The RPCSEC_GSS module handles the data transport via the RPCSEC_GSS protocol. It manages security by accessing the security service (usually krb5, SPKM-3, or LIPKEY).
- The NFS protocol modules perform the management of the structures used for the NFS messages.
- The Cache Inode Layer manages a very large cache for meta-data.
- The File Content Layer manages data caching. It is closely acquainted with the Cache Inode Layer.
- The File System Abstraction Layer is a very important module: it wraps, via a well defined interface, the calls to access a name-space. The objects it addresses are then cached by the Cache Inode and File Content layers.
- The Hash Table Module provides Red-Black-Trees-based hash tables. This generic module is widely used within GANESHA to provide associative addressing.

These modules will be discussed in more details in the next sections.

4.4 Managing memory

The main issue is memory management. Almost all modules within GANESHA's architecture will have to perform dynamic memory allocation. For example, a thread managing a NFS request may need to allocate a buffer for storing the requested result. If the regular

[1] CITI's site contains bunches of interesting stuff for people interested in NFSv4.

LibC malloc/free calls are used, there are risks of fragmenting memory because some modules will allocate large buffers when others will use much smaller ones. This could lead to a situation where part of the memory used by the program is swapped to disk, and performance would quickly drop.

For this reason, GANESHA implements its own memory manager. This module, which is used by all the other parts of GANESHA, allows each thread to allocate its own piece of memory at startup. When a thread needs a buffer, it will look into this space to find an available chunk of the correct size. This allocation is managed by the Buddy Malloc algorithm, the same that is used by the kernel. Use of the syscall *madvise* is also made to tell the Linux memory manager not to move the related pages. The behavior of the daemon towards memory will then be to allocate a single large piece of memory. If there is no other "resource consuming" daemon running on the same machine, the probability for this piece of memory not to be swapped is high. This will maintain performance at a good level.

4.5 Managing the CPU resource

The second resource is the CPU. This is much easier to manage than memory. GANESHA is massively multi-threaded, and will have dozens of threads at the same time (most of them are "worker threads," as we'll see later). POSIX calls for managing threads help us a lot here, we can use them to tell the Linux scheduler not to manage the pack of threads as a whole, but to consider each of them separately.[2] With a multi-processor machine, such an approach will allow the workload to "spread across" all of the CPUs. What is also to be considered is potential deadlocks. In a multi-threaded environment, it is logical to have mutexes to protect some resources from concurrent accesses. But having bunches of threads is not useful if most of them are stuck on a bottleneck. Design considerations were taken into account to avoid this situation.

First, reader/writer locks were preferred to simple mutexes. Because the behavior of reader/writer locks may differ from one system to another, a small library was written to provide this service (which was a required enhancement in terms of portability).

Second, if threads share resources, this common pool could turn to a bottleneck when many threads exist together. This was avoided by allocating resources per thread. This consideration has a strong impact on the threads' behavior, because there can't be a dedicated garbage collector. Each thread has to perform its own garbage collection and has to reassemble its resources regularly. To avoid congestion, some mechanism (located on the "dispatcher thread" described below) will prevent too many threads from performing this operation at the same time (a period during which they are not available for doing their "regular" job). Cache layers that require this kind of garbage collection to be done have been designed so that this process could be divided in several steps, each undertaken by a separate agent. Experience "in real life" shows that this solution was suitable when the number of threads is large compared to the number of threads allowed to start garbage collecting (60 threads running concurrently when 3 could stop working at the same time). This experience shows that the required memory chunk was much less that what is needed for a single request (about 20 times the size). In this situation, the impact of memory management is almost invisible: an incoming request finds a non-busy thread most of the time. Side effects will only become visible under a very large load (hundreds to thousands of requests per second).

4.6 The Hash Tables: a core module for associative addressing

Associative addressing is a service that is required by many modules in GANESHA—for example, finding an inode knowing its parent and name, or finding the structure related to a NFSv4 client, knowing its client ID. The API for this kind of service is to be called very often: it has to be very efficient to enhance the daemon's global performance. The choice was made to use an array of Red-Black Trees.[3] RBTs have an interesting feature: they re-balance themselves automatically after add/update operations and so stay well balanced. RBTs use a computed value, defined as the *RBT value* in this document, to identify a specify contiguous region of the tree. Several entries stored in the RBT can produce the same RBT value, they'll reside the same area, but this will decrease the performance. Having a function to compute "well diversified" RBT values is then critical.

[2] This is the `PTHREAD_SCOPE_SYSTEM` behavior which is used here, as opposed to the `PTHREAD_SCOPE_PROCESS` policy that would not lead to the expected result.

[3] We'll use the abbreviation RBT for Red-Black Tree in the rest of this paper.

This supposes an actual knowledge of the data on which the value is computed. Because of this it is hard to have a "generic RBT value function," a new one is to be developed for each use.

Bottlenecks could occur if a single RBT is used: several threads could perform add/update operations at the same time, causing a conflicting re-balance simultaneously. It then appears that RBTs are to be protected by read/writer locks and this could quickly become a bottleneck. Working around this issue is not difficult: using several RBTs (stored in an array) will solve it. If the number of RBTs used is large (more than 15 times bigger) that the number of concurrent threads that can access them, the the probability of having two of them working on the same tree becomes pretty small. This will not use more memory: each of the 15 (or more) trees will be 15 times smaller than the single one would have been. There is an inconvenience: an additional function is required to compute the index for the RBT to be used. Implementing two functions is then needed for a single hash table: one for computing the index, the other to compute the RBT value. They must be different enough to split data across all the trees. If not, some RBTs would be very small, and others very large. Experience shows that specific non-regression tests were necessary to check for the "independence" of these two functions.

4.7 A massively multi-threaded daemon

GANESHA is running lots of threads internally. As shown in the previous sections, most of its design consideration were oriented to this massively multi-threaded architecture. The threads are of different types:

- GANESHA supports NFSv2, NFSv3, NFSv4, and the ancillary protocol MOUNT PROTOCOL v1 and v3. The *dispatcher* thread will listen for incoming NFS/MOUNT requests, but won't decode them. It will choose the least busy worker and add the request to its lists of requests to process. Duplicate request management is done here: this thread keeps track of the previously managed requests by keeping the replies sent within the last 10 minutes (they are stored in a hash table and addressed with the RPC Xid[4] value). Before associating a worker with a request, it looks at this DRC.[5] If a matching RPC Xid is found, then the former reply is sent back again to the client. This thread will use the RPCSEC_GSS layer, mostly.

- The *worker* threads do most of the job. Many instances (several dozen) of this kind of thread exist concurrently. They wait for the dispatcher thread to provide them with a request to manage. They will decode it and use Cache Inode API and File Content API calls to perform the operation required for this request. These threads are the very core of the NFS processing in GANESHA.

- The *statistics manager* collects stats from every layer for every thread. It periodically writes down the data in CSV format[6] for further treatment. A dedicated PERL script, `ganestat.pl`, is available with the GANESHA rpm as a "pretty printer" for this CSV file.

- The *admin gateway* manages a dedicated protocol. This allows administrative operations to be done remotely on the daemon. These operations include flushing caches, syncing data to FSAL storage, or performing a slow and clean shutdown. The `ganeshadmin` program, provided with the distribution, is used to interact with this thread.

4.8 Dealing with huge caches

As stated above, GANESHA uses a large piece of memory to build large caches. Data and meta-data caches will be the largest caches in GANESHA.

Let's focus first on the meta-data cache, located in the Cache Inode Layer. Each of its entries is associated with an entry in the name-space (a file, a symbolic link, or a directory[7]). This entry is itself associated with a related object in the File System Abstraction Layer (see next section) identified by a unique FSAL handle. The meta-data cache layer will map in memory the structure it reads from the FSAL calls, and it tries to keep in memory as many entries as possible, with their parent-children dependencies. Meta-data cache use hash tables

[4] See the definition of ONC/RPC protocol for details on this.

[5] Duplicate Request Cache.

[6] Comma Separated Value, an ASCII based format for storing spreadsheets.

[7] For the current version, objects of type socket, character, or device are not managed by GANESHA.

intensively to address the entries, using the FSAL handle to address the entry associatively. With the current version of GANESHA, a simple write-through cache policy is implemented. The attributes kept for each object (the file attributes and the content of the directories) will expire after a configurable grace period. If expired, they'll be renewed if they are accessed before being erased from the cache. Garbage collection is more sophisticated. Because there is no common resources pool, each thread has to perform garbage collection itself. Each thread will keep a LRU list of the entries on which it works. A cached entry can exist only within one and only one of these lists, so if a thread accesses an entry which was previously accessed by another, it acquires this entry, forcing the other thread to release it. When garbage collection starts, the thread will go through this list, starting from the oldest entry. It then use a specific garbage policy to decide whether each entry should be kept or purged. This policy is somewhat specific. The meta-data cache is supposed to be very large (up to millions of entries) and no garbage collection will occur before at least 90% of this space is used. We choose to keep as much as possible the "tree topology" of the name-space viewed by the FSAL in the cache. In this topology, nodes are directories, and leaves are files and symbolic links. Leaves are garbage collected before nodes. Nodes are garbage only when they contain no more leaves (typically an empty directory or a directory where all entries were previously garbaged). This approach explicitly considers that directories are more important than files or symbolic links, but this should not be an issue. Usually, a name-space will contain a lot more files than directories, so it makes sense to garbage files first: they occupy most of the available space. Because the cache is very large, parts of it tend to be "sleeping areas" that are no longer accessed. The garbage collection routine within each worker thread, which manages the oldest entries first, will quickly locate these and clean them. With our workload and file system usage, this policy revealed no problem. When the garbage collection's high water mark is reached, the number of entries cached begins to oscillate regularly between low water mark and high water mark. The period of the oscillation is strongly dependent on the average load on the server.

The data cache is not managed separately: if the content of a file is stored in data cache, this will become a characteristic of the meta-data cached entry. The data cache is then a 'child cache' to the meta-data cache: if a file is data-cached, then it is also meta-data cached. This avoid incoherencies between this two caches since they are two sides of the same coin. Contents of the files which are cached are stored in dedicated directories in a local file system. A data-cache entry will correspond to two files in this directory: the index file and the data file. The index files contain the basic meta-data information about the file; the most important one is its FSAL handle. The data file is the actual data corresponding to the cached file. The index file is used to rebuild the data-cache, in the event that the server crashes without cleanly flushing it: the FSAL Handle will be read from this file and then the corresponding meta-data cache entry will be re-inserted as well, making it point to the data file for reconstructing the data cached entry. Garbage collection is performed at the same time as meta-data cache garbage collection. Before garbaging files, the meta-data cache asks the data cache if it knows this entry or not. If not, regular meta-data garbage collection is performed. If yes, the meta-data cache asks the data cache to apply its garbage policy on it, and eventually flush or purge it. If the file is cleaned from the data cache, it can be garbaged from meta-data cache. A consequence of this is that a file which has an active entry in the data cache will never be cleaned from the meta-data cache. This way of working fits well with the architecture of GANESHA: the worker threads can manage the data cache and meta-data cache at the same time, in a single pass. As stated above, the two caches are in fact the same, so no incoherence can occur between them. The data cache has no scalability issue (the paths to the related files are always known by the caches) and does not impact the performance of the meta-data cache. The policy used for data cache is "write-back" policy, and only "small" files (smaller than 10 MB) will be managed; others would be accessed directly, ignoring the data cache. Smarter or more sophisticated algorithms can be implemented—for example, the capability, for very large files, to cache a region of the file but not the whole file. This implementation could be linked to NFSv4 improvements like NFSv4 named attributes or the use of the PNFS paradigm (which is part of the NFSv4.1 draft protocol).

5 File System Abstraction Layer

FSALs (or File System Abstraction Layers) are a very important module in GANESHA. They exist in different incarnations: HPSS FSAL, POSIX FSAL, NFSv4

Proxy FSAL, SNMP FSAL, and LDAP FSAL. They provide access to the underlying file name-space. They wrap all the calls used for accessing it into a well defined API. This API is then used by the Cache Inode and File Content module. FSAL can use dedicated APIs to access the name-space (for example, the SNMP API in the case of SNMP FSAL), but this API will completely hidden from the other modules. FSAL semantics are very close to the NFSv4 semantics, an approach that is repeated in the Cache Layers. This uniformity of semantics, close to native NFSv4, makes the implementation of this protocol much easier. Objects within FSAL are addressed by an FSAL Handle. This handle is supposed to be persistent-associated with a single FSAL object by an injective relationship: two different objects will always have different handles. If an object is destroyed, its handle will never be re-used for another FSAL object. Building a new FSAL is the way to make GANESHA support a new name-space. If the produced FSAL fits correctly with the provided non-regression and validation tests, then the GANESHA daemon need only be recompiled with this new FSAL to provide export over NFS for it. Some implementation documents are available in the GANESHA distribution. External contributors may actively participate to GANESHA by writing additional FSALs. Templates for FSAL source code are available in the GANESHA package.

5.1 The HPSS FSAL

This FSAL is not related to Free Software, but a few words must be said for historical reasons, because it strongly contributed to the origin of the project. We are using the HSM named HPSS,[8] a third-party vendor product sold by the IBM company. This HSM manages a name-space, accessible in user space via dedicated API, which fully complies with the FSAL pre-requisites. The name-space is relatively slow, and this led us to improve the caching features in GANESHA. This module is available, but not within the regular distribution of GANESHA (you need to have HPSS installed to compile it with the HPSS API libraries).

5.2 The POSIX-based FSAL

This flavor of FSAL uses the regular POSIX calls (*open*, *close*, *unlink*, *stat*) from LibC to manage file system objects. All the file systems managed by the machine on which the daemon is running (depending on its kernel) will be accessible via these functions; using them in GANESHA provides generic NFS access to all of them. The inconvenience is that POSIX calls often use the pathnames to the objects to identify them. This is no persistent information about the object (a rename could be performed on it, changing its name). This does not fit with the pre-requisite to build FSAL, as described in the previous subsection. Another "more persistent" identifier is to be found. The choice was made to use an ancillary database (basically a PostgreSQL base) to build and keep the identifier we need. The tuple (inode number, file system ID, ctime attributes) is enough to fully identify an object, but the name should be used to call the POSIX functions. The database will keep parenthood relationship between objects, making it possible to rebuild the full path to it, by making a kind of "reverse lookup" when needed. SQL optimization and pathname caching were used a lot in the module. A complete description of the process would require a full paper. Why develop such a module when it could be much easier to use the NFS interface in the kernel? The answer is linked with the resource we use at our compute center.

GANESHA can access more file systems than most available kernels at our site. We had the need to access the LUSTRE file system, but some machines were not LUSTRE clients. In most cases, they are not Linux machines. We strongly needed them to be able to access the LUSTRE name-space. This could not be done via NFS kernel support: this NFS implementation uses the VFS layer a lot, a part of the kernel that is often bypassed by the LUSTRE implementation for optimization. This approach, using the simple POSIX calls to access LUSTRE from GANESHA, was quick to write and not very costly.

This module is available.

5.3 The NFSv4 Proxy FSAL

When designing GANESHA, we had one thought: having a NFSv4 proxy would be great. NFSv4 has lots of features that are designed for implementing aggressive cache policy (file delegation is a good example of this feature). GANESHA is designed to manage huge caches. The "wedding" seems very natural here. The NFsv4 Proxy FSAL wraps NFSv4 client calls to FSAL calls. It turns the back-end part of GANESHA into a

[8] HPSS stands for *High Performance Storage System*.

NFSv4 client, turning the whole daemon into a NFSv4 proxy server. The mechanism of file delegation is a feature in NFSv4 that is quite interesting here. It allows a file to be "fully acquired" by a client for a given period of time. Operations on files, such as IO operations and modification of its attributes, will be done on the client directly, without disturbing the server; that guarantees that no other clients will access it. Depending on the kind of delegation used, the server may use transient callbacks to update information about the file. When the delegation ends, the server recovers the file, getting the new state for the file from the client. Delegation, used jointly with GANESHA meta-data and data caches, is very efficient: accessing a file's content will be done though data cache, once a delegation on the file has been acquired. The policy for the NFSv4 Proxy FSAL will be to acquire as many delegations as possible, populating the GANESHA's caches. With a well populated cache, GANESHA will become able to answer by proxy many requests. In NFSv4.1, a new feature is added: the directory delegation. This will allow the content of directories to be delegated and acquired by clients in the same way that file contents are. Used with GANESHA's meta-data cache, this feature will be very interesting.

This module is still under development.

5.4 The "Ghost FS" FSAL

This FSAL is a very simple one and is not designed for production use. It just emulates the behavior of a file system in memory, with no persistent storage. The calls to this FSAL are very quick to return because all the work is done in memory, no other resources are used. Other FSALs are always much slower than the cache layer.[9] It is hard to evaluate meta-data and cache modules performances. With the "Ghost FS" FSAL, calls to these layers can be easily qualified, and it is possible to identify the most costly calls, and thus to optimize GANESHA.

This module is available.

5.5 The LUSTRE FSAL

As mentioned above, LUSTRE is a file system we use a lot, and we would like to access it from machines that are not LUSTRE clients. We already developed the POSIX FSAL for this, but having something more acquainted with LUSTRE would be nicer. Having a user-space LUSTRE API able to perform operations in a handle-based way would be something very interesting: it would allow us to wrap the API to a LUSTRE FSAL, making the access to this file system via the GANESHA NFSv4 interface much more efficient than the one we have with the POSIX FSAL. We also hope to use the NFSv4 named attributes[10] to provide clients for LUSTRE-specific information about the file (the resident OST[11] of the file is a good example).

This module is under definition. It will be finalized as soon as a handle-based LUSTRE API is available.

5.6 The SNMP FSAL

The SNMP protocol organizes sets of data as trees. The overall structure of the trees is defined by files named MIB.[12] Knowing the MIB yields the ability to compute the OID[13] to access a given management value. This OID is basically a list of numbers: each of them identifies a node at the given level in the tree, and the last one identifies the leaf where the data resides. For example, `.1.3.6.1.4.1.9362.1.1.0` identifies the Uptime value in the SNMPv2 MIB. This OID is used to query a SNMP agent about the time since the last reboot of the machine. OIDs can also be printed in a "symbolic" way, making them more human readable. In the previous example, `.1.3.6.1.4.1.9362.1.1.0` is printed as *SNMPv2-MIB::system.sysUpTime*. This tree structure is in fact a name space: each SNMP-accessible variable can be seen as a "file object" whose content is the value of the variable. There are "directories" which are the nodes in the MIB structure. OIDs are very good candidates for being handles to SNMP objects, and are to be mapped to names (the symbolic version of the OID). This clearly shows that SNMP has enough features to build an FSAL on top of it. Using it with GANESHA will map the SNMP information into an NFS export, able to be browsed like a file system. It is then possible to browse SNMP in a similar way to the `/proc` file system. In our example, Handle `.1.3.6.1.4.1.9362.1.1.0` would

[9] Otherwise there would have been no need for caches...

[10] which are basically the way NFSv4 manages extended attributes.

[11] Object Storage Target: the way LUSTRE views a storage resource.

[12] Management Information Base.

[13] Object ID.

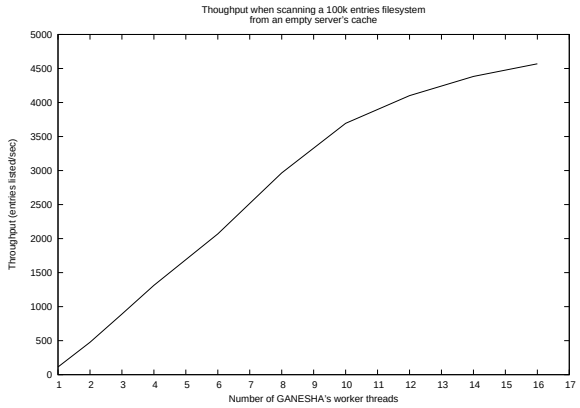

Figure 2: Performance with an empty metadata-cache

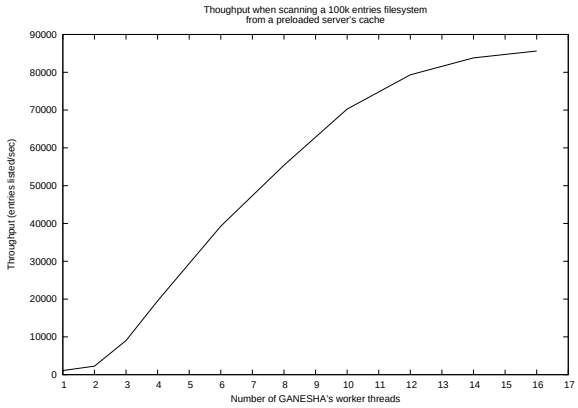

Figure 3: Performance with a preloaded metadata-cache

map to `(mounted NFS PATH)/SNMPv2-MIB/system/sysUpTime`. A read operation on `SNMPv2-MIB/system/sysUpTime` would yield the corresponding value.

Some SNMP values are settable: in this approach, they could be changed by writing to the file corresponding to them.

This module is under development.

5.7 The LDAP FSAL

The idea for this FSAL is the same as for the SNMP FSAL. LDAP has a name-space structure and is accessible via a user-space API. This FSAL simply wraps this API to provide FSAL support, then NFS support via GANESHA for LDAP. LDAP information will then be browsed like `/proc`, via NFS.

This module is under development.

6 Performances and results

In this section, we will show GANESHA's scalability feature by an actual test. The test is as follows: a specific tool was written to perform, in a multi-threaded way (the number of threads is configurable) what `find . -ls` does, which is scanning a whole large tree in a name-space. This tree contained 2220 directories on 3 levels; each of them contained 50 files (which means more that 110,000 files were in the whole tree). The test utility ran on several client nodes (up to 4 machines) using the same server. The multi-threaded test utility was run of each of these 4 clients with 64 threads each. This was equivalent to 256 cache-less clients operating concurrently. The server machine was a IBM x366 server with four Intel Xeon 3 GHz processors and 4 GB of RAM, running GANESHA built with the POSIX FSAL. Two groups of measurements were made. The first one is done with a server whose meta-data cache is empty (Figure 2), and the second (Figure 3) with the same server with a preloaded cache. In this second step, the read entries exist in the memory of the server, and the performance of the meta-data cache can be compared to the raw FSAL performances.

Figure 2 shows that saturation of the FSAL occurs quickly. Increasing the number of worker threads increases the performance, but no larger throughput than 5,000 entries read per second can be reached. Observations made on the server showed that no CPU or memory contention led to this saturation effect. The reason was that the POSIX FSAL on top of the underlying POSIX calls did not scale to these values.

Figure 3 shows different results. Due to the meta-data cache, most of the operations are done directly in memory, reducing greatly the calls to POSIX FSAL. The throughput raises up to 90,000 entries read per second. The dependence between this throughput and the number of worker threads is linear, which shows the scalability of the process. After 11 worker threads, we can't see such linearity. The reason for this was due to CPU congestion. The OS could not allocate enough CPU time to all the workers, and they start waiting to be scheduled. This test should be performed on a larger platform.

This test shows that the multi-thread architecture in

GANESHA provides good scalability.

7 Conclusion and perspectives

GANESHA has been in production at our site for more than one full year. It fits the needs we had when the decision was taken to start the project. Its large cache management capability allowed an increase of the incoming NFS requests on the related machines, a need that was critical for several other projects.

When the product started in full production, in January, 2006, this provided us with very useful feedback that helped in fixing bugs and improved the whole daemon. Thanks to this, GANESHA is a very stable product in our production context at our site. Making GANESHA Free Software is an experience that will certainly be very positive; we expect the same kind of feedback from the Open Software community. GANESHA can also be of some interest for this community; we actually believe that is could serve well as a NFSv4 Proxy or as an SNMP or LDAP gateway.

NFSv4 is also a very exciting protocol, with plenty of interesting features. It can be used in various domains and will probably be even more widely used that the former version of NFS. Lots of work is done around this protocol, like discussion about implementing its features or extending it with new features (see NFSv4.1 drafts). GANESHA will evolve as NFSv4 will. We hope that you will find this as exciting as we did, and we are happy to share GANESHA with the community. We are eagerly awaiting contributions from external developers.

References

[1] S. Shepler, B. Callaghan, D. Robinson, Sun Microsystems Inc., C. Beame, Hummingbird Ltd., M. Eisler, D. Noveck, Network Aplliance Inc. *"Network File System (NFS) version 4 Protocol," RFC 3530*, The Internet Society, 2003.

[2] Callaghan, B., Pawlowski, B. and P.Staubach, *"NFS Version 3 Protocol Specification," RFC 1813*, The Internet Society, June, 1995.

[3] Sun Microsystems, Inc., *"NFS: Network File System Protocol Specification," RFC 1094*, The Internet Society, March, 1989.

[4] Shepler, S., *"NFS Version 4 Design Considerations," RFC 2624*, The Internet Society, June, 1999.

[5] Adams, C., *"The Simple Public-Key GSS-API Mechanism (SPKM)," RFC 2025*, The Internet Society, October, 1996.

[6] Eisler, M., Chiu, A. and L. Ling, *"RPCSEC_GSS Protocol Specification," RFC 2203*, The Internet Society, September, 1997.

[7] Eisler, M., *"NFS Version 2 and Version 3 Security Issues and the NFS Protocol's Use of RPCSEC_GSS and Kerberos V5," RFC 2623*, The Internet Society, June, 1999.

[8] Linn, J., *"Generic Security Service Application Program Interface, Version 2, Update 1," RFC 2743*, The Internet Society, January, 2000.

[9] Eisler, M., *"LIPKEY—A Low Infrastructure Public Key Mechanism Using SPKM," RFC 2847*, The Internet Society, June, 2000.

[10] Shepler, S., Callaghan, B., Robinson, D.,Thurlow, R., Beame, C., Eisler, M. and D. Noveck, *"NFS version 4 Protocol," RFC 3010*, The Internet Society, December, 2000.

[11] B. Callaghan, *"NFS Illustrated,"* Addison-Wesley Longman Ltd., Essex, UK, 2000.

[12] CITI. *Projects: NFS Version 4 Open Source Reference Implementation*, `http://www.citi.umich.edu/projects/nfsv4/linux`, June, 2006.

[13] Connectathon. *Connectathon web site*, `http://www.connectathon.org`.

[14] S. Khan. *"NFSv4.1: Directory Delegations and Notifications," Internet draft*, `http://tools.ietf.org/html/draft-ietf-nfsv4-directory-delegation-01`, Mar 2005.

[15] Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow, *"The NFS Version 4 Protocol,"* In Proceedings of Second International System Administration and Networking (SANE) Conference, May, 2000.

[16] A. Charbon, B. Harrington, B. Fields, T. Myklebust, S. Jayaraman, J. Needle, *"NFSv4 Test Project,"* In Proceedings to the Linux Symposium 2006, July, 2006.

[17] P. Åstrand, *"Design and Implementation of a Test Suite for NFSv4 Servers,"* September, 2002.

[18] CEA, CNRS, INRIA. *"CeCILL and Free Software,"* `http://www.cecill.info/index.en.html`.