Towards Dynamic Scripted pNFS Layouts

Matthias Grawinkel
University of Paderborn
grawinkel@upb.de

Tim Süß
Johannes-Gutenberg
University Mainz
t.suess@uni-mainz.de

Gregor Best
University of Paderborn
gbe@mail.upb.de

Ivan Popov, André Brinkmann
Johannes-Gutenberg
University Mainz
brinkman@uni-mainz.de

***Abstract*—Today's network file systems consist of a variety of complex subprotocols and backend storage classes. The data is typically spread over multiple data servers to achieve higher levels of performance and reliability. A metadata server is responsible for creating the mapping of a file to these data servers. It is hard to map application specific access patterns to storage system specific features, which can result in a degraded IO performance.**

We present an NFSv4.1 / pNFS protocol extension that integrates the client's ability to provide hints and I/O advices to metadata servers. We define multiple storage classes and allow the client to choose which type of storage fits best for its desired access pattern. Furthermore, we propose flexible, script based file layouts that describe a logical file's mapping to its storage locations by an interpreted script instead of a fixed mapping.

We discuss the possibilities of such an extension, show that it integrates well with the pNFS protocol and show that the latency and compute overhead of interpreted layouts justifies the gained features.

I. Introduction

Today's file systems for *big data* and *HPC systems* have many challenges to tackle. Besides the massive amount of data that must be stored, file systems must provide fast and concurrent access to data for an arbitrary number of users. However, users and their applications have different reliability and performance requirements when storing and accessing their data. Existing parallel file systems are based on a huge variety of protocols, semantics, and storage classes, and applications have many different access patterns ranging from parallel access to shared files to write-once-read-many, append-only, or write-once archival workloads.

To scale capacity and performance requirements, data is spread to multiple data servers (DS), which are managed by a metadata server (MDS). A file may be striped or mirrored to multiple DS and the MDS manages and stores layout descriptors to map files to their locations. To achieve the best performance, the applications' access patterns have to match the storage system's capabilities and the data distribution pattern. A mismatch can have a huge impact on throughput and latency. When the MDS is responsible for the placement of data to the data servers and for creating layouts for the clients, it is hard to match the clients access patterns. To include this clint side information, the NFSv4.1 / pNFS protocol provides layout hints and proposes advices to communicate hints on client access patterns to the storage system for future versions [1], [2]. Once the layout is created by the MDS, the client has to use it, even if it does not match its usage patterns or if the client's access behavior changes.

In the world of software development, scripting languages as additions to static programs have gained importance. These languages allow modifications of algorithms during run-time. Code can be injected from an external source to enable the program to react on a specific situation. One language that has gained a lot of traction is Lua [3], which is used throughout many areas [4] and which is also one of the fastest scripting languages [5].

To overcome the impacts of mismatching layouts and to provide an improved client flexibility, we investigate the introduction of scripted elements in the context of file layout descriptors. On file creation, a client gives some hints on its estimated access patterns or a wish-list of storage classes or locations and the MDS returns a list that matches the client's hint without specifying a layout. The client then uses the storage on its own behalf and stores a script based layout descriptor back to the MDS.

This scheme introduces a lot of flexibility for the clients while being compatible with existing protocols. A client can use a default layout or provide its own script that maps ranges of the logical file to its storage locations. This script can be generated by the client's storage driver or it can even be injected by the application that created the file. This mechanism can help to match the client's access patterns to the storage system.

In pNFS, a layout contains a list of data locations and a parameterization of a RAID scheme to use on these locations. By introducing scripts as layout descriptors, more sophisticated placement strategies can be used without updating the clients or servers. Instead of sending a parameterization for a RAID scheme, simple mappings, standard RAID schemes, or more complex data distribution strategies as analyzed by Miranda et al. [6] can be used by the clients, while reducing the metadata server's complexity.

In this paper we discuss the feasibility of such an approach, show that it fits into the pNFS protocol stack, discuss some examples and their benefits and shortcomings, and provide an evaluation setup to measure the actual overhead of an integration of the scripting engine into the storage stack.

A. Distributed File Access

Many network and parallel file systems like PanFS [7], Lustre, Ceph [8], or PVFS [9] use dedicated access paths for

metadata operations and data access. A (clustered) MDS is responsible for namespace operations, access rights, locking, and the general management of the file system, while the actual data is stored on dedicated data servers. When a client wants to access a file, it needs to contact the MDS that returns a layout descriptor that describes the mapping of the file to the data servers. Following `READ` or `WRITE` operations are then directly sent to the data servers.

The mapping of a file to data servers imposes many challenges, including different underlying storage protocols and interconnects, different storage classes ranging from RAM over SSDs to disks and tapes, and different performance metrics. Client access patterns have to match these storage systems' capabilities, e.g., Isaila and Tichy argued that it is important to provide enough flexibility to represent these access patterns to match the different application demands [10].

The idea of different file policies for different application demands or file types has been introduced with the automatic tiering capabilities of the HP AutoRaid storage system [11]. Hildebrand et al. [12] improve the performance of small file accesses by circumventing the full pNFS I/O path and taking a fallback route via a standard NFSv4 server. Therefore, they are able to circumvent the pNFS overhead. Additionally, Panasas enables client side RAID [7] by using a dedicated layout per file. Small files can be mirrored while bigger files can be striped as an object via RAID5.

B. The pNFS protocol

pNFS is an extension of the NFS4 storage protocol that defines the separation of the file system metadata from the location of the file. All control and file management operations are handled by a MDS and the file I/O is handled by a storage specific driver. The current NFSv4.1 version [1] defines mechanisms for a client to access files on NFS4 data servers alongside block / volume [13] and object-based [14] storage backends. To access a file on any of these backends, a client needs to obtain the file's layout, which is used by the client's storage specific driver to access the data. In general this layout contains a list of storage servers, information about striping and mirroring of the file, alongside some specifics for the target storage type. The MDS is responsible for managing these layouts and for deciding about placement, mirroring, and striping strategies for the file. Some flexibility is given to the client that can supply a `layout_hint` upon a file `CREATE` operation. This hint contains information about mirroring or striping parameters, but once the MDS decided on these parameters, they are not supposed to be changed during the file's lifetime.

A layout is only given to a client when it has a file handle and the mandatory access rights and file locks. Furthermore, it has an *iomode* that defines read (`R`) or read-write (`RW`) access and it is only valid for a range. A file layout is defined precisely by a tuple `<client ID, filehandle, layout type, iomode, range>`. Multiple clients can hold multiple non-intersecting `RW` layouts on different parts of a file or hold intersecting `R` layouts on shared parts of a file.

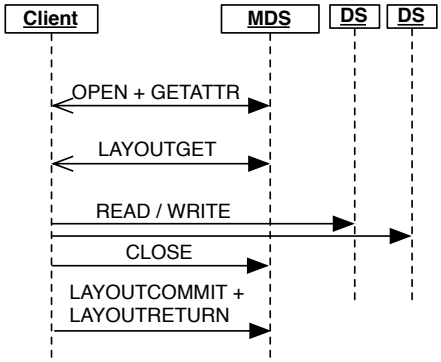

Fig. 1. Simplified pNFS protocol for data access.

If a client has a `RW` layout, it can also change the content of the layout, i.e., when a new set of blocks is written in a block layout in a copy-on-write system. Then, a `LAYOUTCOMMIT` call writes the updated layout to the MDS. After closing the file, the client can return the layout with a `LAYOUTRETURN`.

A client's layout can be called back by the MDS with a `CB_LAYOUTRECALL` when the layout becomes invalid. This happens when either a conflicting layout is requested or the state encapsulated by the layout becomes invalid. The latter can happen when an event directly or indirectly modifies the layout. Layouts can be recalled for specific ranges, clients, or file systems [1].

II. Dynamic Layouts

While the pNFS protocol and its extensions [1], [13], [14] offer mechanisms to distribute data to file servers, object stores, and block devices via configurable layouts, the protocol is still missing some possibilities to support the exact access patterns or changing requirements of the clients.

The current pNFS version provides a `layout_hint` that can give some hints for desired parameter values for the requested layout, but it still depends on the metadata server's logic. Furthermore, proposed pNFS features [2] will allow applications to provide hints on their access patterns, like random or sequential access and caching behaviors.

We propose to extend these hints to allow the client to specify partitions of storage classes and move the responsibility of how to use them to the client. For simplification we classify different storage classes into 3 categories:

- **Gold:** Low latency, fast sequential/random I/O, i.e., flash based
- **Silver:** Medium latency, fast sequential I/O, i.e., disks
- **Bronze:** High latency, archive, i.e., tapes or spin-down disks

A complex application specific file format like HDF5[1] or NetCDF[2], for example, could benefit from partitioning its data

[1] http://www.hdfgroup.org/HDF5/

[2] http://www.unidata.ucar.edu/software/netcdf/

to store frequently used metadata on a *Gold* class partition, while the bulk data is striped over a *Silver* class RAID-based partition.

As shown in Section I-B, the pNFS protocol allows a client to provide a `layout_hint` on file creation. The MDS then returns layouts that fit the requirements and may contain multiple storage addresses, like a file on a data server or an object specifier on an OSD. We propose to extend the `layout_hint` to contain a list of desired partitions, their sizes, storage classes, and distribution patterns. A hint for the aforementioned HDF5 file can, for example, contain two flash based partitions on two different servers of some megabytes and n disk based partitions on n different servers to build a RAID 4/5/6 on top of them.

Next to these additions to the `layout_hints`, we propose to use script based layout descriptors. They do not only contain some parameters that are used by a layout driver to access the storage, but they also contain a script that maps the logical file's offsets to the storage locations. Within a layout, logical ranges can be mapped in different ways to subsets of the aforementioned storage partitions. This allows simple mappings, like storing the first sectors of a file to a different partition than the remaining file, or more complex patterns as described in the Clusterfile approach [10].

By using advanced hints and providing scripted instead of fixed layout interpreters, we achieve multiple benefits. Once the script enabled client and MDS specific code is deployed, the access semantics and functionality can be changed and extended without updating the core components. Furthermore, the scripted layouts provide a huge gain of flexibility on the client side.

When a client creates a file, it requests a file size, some parameters, and a list of partitions consisting of the number of different target files/objects/volumes, their storage classes and sizes. The metadata server then creates a layout with these storage targets and lets the client decide how to use it. When the client finishes its work, the layout is committed back to the MDS. If the file is accessed by another client or during a recovery or maintenance task on the MDS, its layout is read, the script is interpreted, and the file's data can be accessed.

A. Script Engine Integration

There are multiple ways to integrate a scripting engine into the storage stack, but its general task is to transfer a script with some parameters into a kernel object that can be used by a layout driver to access the actual data. Given the Linux kernel NFS implementation, the appropriate calls to create and interpret the layout objects have to be intercepted and redirected to the scripting engine as depicted in Figure 2. For every access on a file's range, the scripted layout driver is called, which redirects read and write requests to the target files/volumes/objects according to the layout's script and parameters.

We have investigated the Lua scripting language [3] as it is very fast [5], has native C/C++ bindings, and can either be integrated in the Linux kernel via lunatik-ng [15]

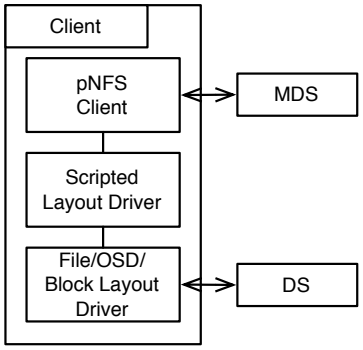

Fig. 2. Script engine integration.

or as a userspace daemon within a microdriver framework. These microdriver frameworks provide mechanisms to forward kernel functions to a userspace daemon that would integrate the scripting engine. It has been shown that this forwarding is possible within an average of $10\,\mu s$ [16]. Due to the easy integration of the Lua engine into the kernel and the direct integration into the pNFS stack, we have chosen to evaluate the kernel version.

Once the Lua engine is started, it can hold a state until it is stopped. This means that scripts, objects, and tables can be stored in the engine. For example, a client application can load global functions or parameters into the engine, which are then used for the layout interpretation. Next to this, the Lua engine is also capable of calling external functions. This is of special interest for some data placement strategies that use a variety of hash functions [6]. The in-kernel Lua engine, for example, can call the Linux crypto API from within the scripts. In Section III we show that this works for sha1() and analyze the latency and compute overhead induced by the scripting engine.

B. pNFS Protocol Integration

The proposed additions can be integrated into the current pNFS protocol stack. If both, client and MDS, agree to use a scripted layout, the proposed `layout_hints` can be used to define a *devicemap* by the MDS that holds the actual storage targets and a *base layout* that is valid for the full file's range.

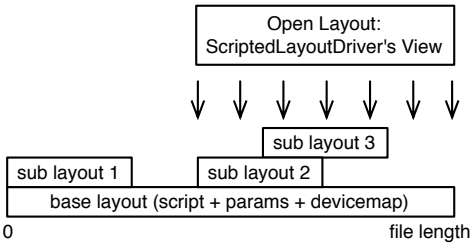

Fig. 3. Sublayouts that overwrite and extend the base layout.

As long as a client holds a `RW` layout, it has exclusive access to it and can use the layout's storage on its own behalf. Furthermore, it can update the devicemap, the layout's script, and the parameters for the layout's range and write it back to the MDS within a `LAYOUTCOMMIT` command. If the layout covers the full file, the *base layout* would be overwritten, but if the layout covers only a part of the file, a *sub layout* is created, as presented in Figure 3. When a client requests a layout, it will get a merged layout that covers all active sub layouts and the MDS is required to manage, merge, remove, and clean these layouts.

This setup allows every client to update the layout that it holds for the cost of more complex layout drivers and metadata management. A simpler version is to restrict the layout to a single base layout without any sub layouts. Here, a client needs to have an exclusive access `RW` layout for the whole file to update its script and devicemap with a `LAYOUTCOMMIT`.

C. Dynamic Layout examples

As a first example, we reinvestigate the HDF5 use case, where the MDS returns a file based layout that contains two partitions of 2 and 10 files. The layout's script provides a function that maps the logical file's offsets to the partitions and a function for each partition that defines how the data is spread to the target files. Here, the metadata can be striped with a RAID 1 to the two *Gold* devices, while the bulk data is stored on a RAID 6 over ten *Silver* partitions. For the bulk data, even different block and stripe sizes can be configured to match the application's usage patterns. For every `read` or `write` call from the application, the scripted layout driver, as shown in Figure 2, will then calculate the targets for the logical file's offsets and forward the operations to the appropriately configured layout drivers.

There are further hypothetical usage scenarios that are possible with scripted layout descriptors.

- A distributed application opens the file from multiple processors. The layout descriptor contains the partitions and a script, while the application's processors use different parameterizations (by setting parameters to the scripted layout engine) to access the actual content. This setup can be used to script concepts like Clusterfile [10] and is compatible with the pNFS protocol as long as the clients do not change the layout itself but only specify its parameterizations to their local layout driver.
- To integrate more complex, pseudo-randomized data distribution strategies, as discussed in [17] [18] [19] [19] [6]. These pseudo-randomized strategies are able to dynamically adapt the layout to a changing number of available storage systems with a minimum reconfiguration overhead. The list of available hosts (or in an advanced version, a link to a global list) could be encoded as target locations, and the layout's script would be used to determine the target hosts/volumes for the file access.
- Another branch of new optimization possibilities opens up by exploring the capability of clients to exclusively decide on how to use allocated storage within a layout after it has received a layout via a `LAYOUTGET`. Seizing the idea of pNFS directory delegations, a client would be able to create files and transfer storage partitions from a previously acquired layout to new files, without contacting the MDS the first time, when the files are closed and their layouts are returned. In general, this would allow the client to transfer preallocated storage between layouts that it holds, which again provides more flexibility without MDS interaction.

III. Evaluation

To be able to argue about scripted layouts, we first evaluated the kernel space scripting engine to check the feasibility of a script engine integration and to measure its performance and latency impact. Therefore, we built the 3.6 pnfs-all-latest branch of the current pNFS development Kernel[3] and integrated an improved version of the lunatik-ng scripting engine [15]. In general, we did not yet implement a fully working layout driver but did preliminary evaluation and benchmarking. All tests have been conducted on an Ubuntu 12.04 64bit Linux running our tailored kernel. The server has been an Intel(R) Xeon(R) CPU E3-1230 V2 @ 3.30GHz with 16 GB Ram.

For the tests, we added hooks to the scripting engine to map parts of the kernel crypto API into the engine and to make the pNFS implementation specific layout structs available for the scripts. To invoke the Lua engine, the script is embedded into a char array and the engine is called alongside the script's parameterization. When the Lua engine finishes, the result is retrieved from the engine's stack.

We conducted several tests to analyze different sources of overhead and latency. The **lua_calc_sui** test takes an offset and a layout as a parameter and returns a calculated stripe unit. The **lua_crypto** test measures the overhead to call the kernel crypto API's sha1() function. Some random data is generated, the bindings for the kernel API are called and the sha1 is returned. The **lua_create_filelayout** test takes a buffer as a parameter and returns an initialized file_layout kernel object.

```
function lua_create_filelayout (buf)
  rv = pnfs.new_filelayout()
  rv.stripe_type = "sparse"
  rv.stripe_unit = buf[1] + buf[3]
  rv.pattern_offset = buf[2] + buf[4]
  rv.first_stripe_index = buf[5] + buf[6]
  return rv
end
```

This test shows the capabilities of working with kernel objects, which can be used to build initialized structs from within the scripting engine that can be further used in the kernel context. The **lua_create_filelayout_crypto** test combines the *create_filelayout* with the *lua_crypto* test to evaluate the performance of complex scripted layouts as discussed in Section II-C.

[3] git://linux-nfs.org/~bhalevy/linux-pnfs.git

For the evaluation, the single commands have been triggered by a syscall and repeated many times within a loop inside the kernel with varying input parameters. We ran each test at least fifty times with varying numbers of rounds from 10,000 to 1,000,000. Table I presents the results in and the corresponding 95% confidence intervals.

Test	μs / op
lua_calc_sui	0.87 ($\pm$ 0.03)
lua_crypto	1.25 ($\pm$ 0.02)
lua_create_filelayout	2.18 ($\pm$ 0.05)
lua_create_filelayout_crypto	3.37 ($\pm$ 0.02)

TABLE I
TEST RESULTS LUA COMMAND EXECUTION.

The evaluation shows that for in kernel operations, the simple script based calculations can be evaluated in less than a microsecond for the *lua_add* and with less than $5\,\mu s$ for calling external functions or creating new kernel objects.

IV. DISCUSSION

We argued about the feasibility and the possibilities of moving decisive power and scripting capabilities to pNFS clients. In Section II we showed how a scripting engine can fit into the Linux storage stack and that the proposed protocol features are compatible with the existing NFSv4.1 / pNFS specification. The prototypical implementation and evaluation of Section III showed that a scripting engine can be well integrated into the kernel and that usage of script based layouts adds an average of $8\,\mu s$ when all features are used and less then $1\,\mu s$ for a simple script evaluation.

The usage scenarios of Section II-C show that scripted layouts yield a lot of flexibility in dynamic storage protocol (re)design and some optimization potential for both performance and reliability, as the clients can match their access patterns to the actual storage backends. Furthermore, the ability to address the target storage classes can be exploited for hierarchical storage management.

Nevertheless, some important points have to be considered when implementing scripted layouts and improved hints to a storage system. In contrast to the current pNFS system, clients will get more decisive power on how to address storage, and for each file access, the scripted layout has to be at least evaluated once. By allowing clients to shift acquired storage partitions, the metadata management on the MDS can become less complex.

A new layout driver has to be implemented that includes the scripting engine and exports a set of secure and well defined kernel functions and objects into the Lua engine and is able to either address the existing layout drivers to access data or implement its own client to the file / object / block protocols.

A new syscall has to be implemented that allows userspace applications to interact with the Lua engine, e.g, to set parameters or load global functions. Here, the processes have to be separated in some way so that they cannot overwrite each others data in the Lua engine. Especially for multi-user systems, this has to be considered.

In conclusion, we think it is worth to have a closer look at scripted layouts and the proposed layout hint extension. They will ease experimentation with new placement and distribution strategies and can be tailored to exactly match applications' access patterns to the used storage system. We will investigate the implementation and evaluation of a full prototype in future works.

ACKNOWLEDGEMENTS

We would like to thank Sebastian Moors and Dominic Eschweiler for their knowledge during discussions.

REFERENCES

[1] S. Shepler, M. Eisler, and D. Noveck, "Network File System (NFS) Version 4 Minor Version 1 Protocol," RFC 5661, 2010.
[2] D. Hildebrand, M. Eisler, T. Myklebust, and S. Falkner, "Support for Application IO Hints (draft-hildebrand-nfsv4-ioadvise-02.txt)," 2012.
[3] "The Programming language Lua. http://www.lua.org."
[4] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes, "The evolution of Lua," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, ser. HOPL III, 2007.
[5] "The computer language shootout benchmarks. http://shootout.alioth.debian.org."
[6] A. Miranda, S. Effert, Y. Kang, E. L. Miller, A. Brinkmann, and T. Cortes, "Reliable and randomized data distribution strategies for large scale storage systems," in *Proc. of the* 15^{th} *International Conference on High Performance Computing (HiPC)*, 2011.
[7] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable Performance of the Panasas Parallel File System," in *Proc. of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.
[8] S. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," in *Proc. of the 7th Conference on Operating Systems Design and Implementation (OSDI)*, Nov. 2006.
[9] P. H. Carns, W. B. L. III, R. B. Ross, and R. Thakur, "PVFS: A Parallel File System for Linux Clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.
[10] F. Isaila and W. F. Tichy, "Clusterfile: A Flexible Physical Layout Parallel File System," in *Proceedings of IEEE International Conference on Cluster Computing (Cluster)*, 2001.
[11] J. Wilkes, R. A. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID Hierarchical Storage System," in *Proc. of the* 15^{th} *ACM Symposium on Operating Systems Principles (SOSP)*, 1995, pp. 96–108.
[12] D. Hildebrand, P. Honeyman, and L. Ward, "Large Files, Small Writes, and pNFS," in *Proc. of the 20th Annual international conference on Supercomputing (ICS)*, 2006.
[13] D. Black, S. Fridella, and J. Glasgow, "Parallel NFS (pNFS) Block/Volume Layout," RFC 5663, 2010.
[14] B. Halevy, B. Welch, and J. Zelenka, "Object-Based Parallel NFS (pNFS) Operations," RFC 5664, 2010.
[15] "lunatik-ng – kernel scripting with Lua. http://github.com/lunatik-ng/lunatik-ng."
[16] A. Brinkmann and D. Eschweiler, "A Microdriver Architecture for Error Correcting Codes inside the Linux Kernel," in *Proc. of the ACM/IEEE Conference on Supercomputing (SC)*, 2009.
[17] A. Brinkmann, K. Salzwedel, and C. Scheideler, "Compact, adaptive placement schemes for non-uniform distribution requirements," in *Proc. of the* 14^{th} *ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Winnipeg, Manitoba, Canada, 2002, pp. 53–62.
[18] A. Brinkmann, S. Effert, F. Meyer auf der Heide, and C. Scheideler, "Dynamic and Redundant Data Placement," in *Proc. of the* 27^{th} *IEEE International Conference on Distributed Computing Systems (ICDCS)*, Toronto, Canada, Jun. 2007.
[19] R. J. Honicky and E. L. Miller, "A fast algorithm for online placement and reorganization of replicated data," in *Proc. of the* 17^{th} *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Nice, France, Apr. 2003.