NFSv4 Working Group
Internet Draft
Intended status: Standards Track
Expires: March 2011

D. Hildebrand
M. Eshel
IBM Almaden
September 29, 2010

Simple and Efficient Read Support for Sparse Files
draft-hildebrand-nfsv4-read-sparse-01.txt

Status of this Memo

Abstract

This document extends the NFSv4.1 protocol to support efficient reading of sparse files. The number of sparse files is growing in the data center, most notably due to the increasing number of virtual disk images. This simple extension provides an easy and efficient way for administrators to copy and manage these files without wasting disk space or transferring data unnecessarily.

Table of Contents

1. Introduction

NFS is now used in many data centers as the sole or primary method of data access. Consequently, more types of applications are using NFS than ever before, each with their own requirements and generated workloads. As part of this, sparse files are increasing in number while NFS continues to lack any specific knowledge of a sparse file's layout. This document extends the NFSv4.1 protocol to support efficient reading of sparse files.

A sparse file is a common way of representing a large file without having to pre-allocate data for it. Consequently, a sparse file uses fewer blocks than its size indicates. This means the file contains 'holes', byte ranges within the file that contain no data. Most modern file systems support sparse files, including most UNIX file systems and NTFS, but notably not Apple's HFS+. Common examples of sparse files include VM OS/disk images, database files, log files, and even checkpoint recovery files most commonly used by the HPC community.

If an application reads 'holes' in a sparse file, the file system converts empty blocks into "real" blocks filled with zeros, and returns them to the application. For local data access there is little penalty, but with NFS these zeroes must be transferred back to the client. If an application uses the NFS client to read data into memory, this wastes time and bandwidth as the application waits for the zeroes to be transferred. Once the zeroes arrive, they then steal memory or cache space from real data. To make matters worse, if an application then proceeds to write data to another file system, the zeros are written into the file, expanding the sparse file into a full sized regular file. Beyond wasting disk space, this can actually prevent large sparse files from ever being copied to another storage location due to space limitations.

This document simply adds a new return value to the READ RPC to avoid reading holes in sparse files and to tell the client the location of the next valid data block. This solution is intentionally very simple and does not build on complicated and optional features such

as pNFS. This hopefully ensures that sparse files become supported by the widest number of client implementations.

The XDR description is provided in this document in a way that makes it simple for the reader to extract into a ready to compile form. The reader can feed this document into the following shell script to produce the machine readable XDR description of the metadata layout:

```
#!/bin/sh
grep "^  *///" | sed 's?^  *///  ??' | sed 's?^.*///??'
```

I.e. if the above script is stored in a file called "extract.sh", and this document is in a file called "spec.txt", then the reader can do:

```
 sh extract.sh < spec.txt > md.x
```

The effect of the script is to remove leading white space from each line of the specification, plus a sentinel sequence of "///".

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119 [1].

2. Terminology

- Regular file: An object of file type NF4REG or NF4NAMEDATTR.
- Sparse File. A Regular file that has a size greater than the number of blocks allocated for the file.
- Hole. A byte range within a Sparse file that contains no data or simply zeroes.

3. Applications and Sparse Files

Applications may cause an NFS client to read empty blocks in a file for several reasons. This section describes three different application workloads that cause the NFS client to transfer data unnecessarily. These workloads are simply examples, and there are probably many more workloads that are negatively impacted by sparse files.

The first workload that can cause empty blocks to be read is sequential reads within a sparse file. When this happens, the NFS client may perform read requests ("readahead") into sections of the

file not explicitly requested by the application. Since the NFS client cannot differentiate between allocated and unallocated sections, the NFS client may prefetch empty sections of the file.

This workload is exemplified by Virtual Machines and their associated file system images, e.g., VMware .vmdk files, which are large sparse files encapsulating an entire operating system. If a VM reads files within the file system image, this will translate to sequential NFS read requests into the much larger file system image file. Since NFS does not understand the internals of the file system image, it ends up performing readahead into unallocated sections. Note that it is also common for several VMs on different NFS clients to share a single file system image file, which exacerbates the problem by resending empty blocks to multiple clients.

The second workload is generated by copying a file from a directory in NFS to either the same NFS server, to another file system, e.g., another NFS or Samba server, to a local ext3 file system, or even a network socket. In this case, bandwidth and server resources are wasted as the entire file, including both allocated and unallocated blocks, are transferred from the NFS server to the NFS client. Once a data block has been transferred to the client, it is up to the client application, e.g., rsync, cp, scp, on how it writes the data to the target location. For example, cp supports sparse files and will not write zero filled blocks, whereas scp does not support sparse files and will transfer every data block.

The third workload is generated by applications that do not utilize the NFS client cache, but instead use direct I/O and manage cached data independently, e.g., databases. These applications may perform whole file caching with sparse files, which would mean that even the unallocated sections will be transferred to the clients and cached.

4. Overview of Sparse Files and NFSv4

This proposal seeks to provide sparse file support to the largest number of NFS client and server implementations, and as such proposes to add a new return code to the mandatory NFSv4.1 READ operation instead of proposing additions or extensions of new or existing optional features (such as pNFS).

As well, this document seeks to ensure that the proposed extensions are simple and do not transfer data between the client and server unnecessarily. For example, one possible way to implement sparse file read support would be to have the client, on the first hole encountered or at OPEN time, request a block layout map from the server. While this option seems simple, it can become inefficient

and cumbersome. First, large block layout maps can be returned from the server, which can reduce overall READ performance. For example, VMware's .vmdk files use 64KB blocks and can have a file size of over 100 GBs. This means that the possible number of allocated (or unallocated) blocks in the file can grow very large in the worse case scenario. In addition, this large block layout map may need to be transferred multiple times with each update to the file. For example, a VM that updates a config file in its file system image would invalidate the block layout map not only for itself, but for all other clients accessing the same file system image.

Another way to handle holes is compression, but this not ideal since it requires all implementations to agree on a single compression algorithm and requires a fair amount of computational overhead.

Note that supporting writing to a sparse file does not require changes to the protocol. Applications and/or NFS implementations can choose to ignore WRITE requests of all zeroes to the NFS server without consequence.

5. Definition of Sparse Reads with NFS

The following sections details changes to the READ operation in the NFSv4.1 specification [3] to allow NFS clients to avoid reading holes in a file.

Our proposal is very simple, if a client READ request would return all zeroes from a file hole, the server does not waste computational and network overhead by sending the zeroes back to the client. Instead, the server returns a new return value and result structure that tells the client that the READ result is all zeroes AND the offset of the next non-zero segment of data. Sending the location of the next valid data block, and only upon request, avoids transferring large block layout maps that may be soon invalidated and avoids sending large amount of information about a file that may not even be read in its entirely.

5.1. Definition of the READ4res

```
///  union READ4res switch (nfsstat4 status) {
///   case NFS4_OK:
///           READ4resok     resok4;
///   case NFS4ERR_HOLE:
///           READ4reshole   reshole4;
///   default:
///           void;
///  };
```

If status is NFS4ERR_HOLE, then the entire byte range of the read request is in a hole, and can be assumed to be zero. Information regarding the location of the next non-hole, or allocated block, in the file is contained in reshole4.

5.2. Definition of the READ4reshole

```
///  struct READ4reshole {
///          offset4         data_offset;
///          length4         data_length;
///   };
```

If a READ request is into a hole, a READ4reshole structure is returned. The READ4reshole structure is considered valid until the file is changed (detected via the change attribute). If the first part of the READ request is into a section of the file that has non-zero data, and the rest of the request is all zeros, the server should return a short read.

The values of the fields are as follows,

- data_offset, which is the offset of the next region of allocated data in the file.

- data_length, which is the length of the non-zero data segment at data_offset. If data_length is not zero, then the data in the file from data_offset until data_length is allocated and does not contain a hole. If data_length is zero, then either the server has no further information regarding holes in the remainder of the file or it can be assumed that all remaining bytes in the file are allocated and contain no holes. Either way, the client can ignore the information in READ4reshole.

5.3. Sparse Files and pNFS

With pNFS, the semantics of NFS4ERR_HOLE remain the same. Any data server can return a NFS4ERR_HOLE result for a READ request that it receives. In addition, when a data server is returning a READ4reshole structure, it should still contain the offset and length of the next allocated block in the file, even if that block is not located on that particular data server.

When a pNFS client receives a NFS4ERR_HOLE result and a READ4reshole structure with a non-zero data_length, it uses this information in

conjunction with a valid layout for the file to determine the next data server for the next allocated block of data.

5.4. Example

To see how NFS4ERR_HOLE will work, the following table describes a sparse file. For each byte range, the file contains either non-zero data or all zero data.

Byte-Range	Contents
0-31999	Non-Zero
32K-255999	Zero
256K-287999	Non-Zero
288K-353999	Zero
354K-417999	Non-Zero

Under the given circumstances, if a client was to read the file from beginning to end with a max read size of 64K, the following will be the result. This assumes the client has already opened the file and acquired a valid stateid and just needs to issue READ requests.

1. READ(s, 0, 64K) --> NFS_OK, eof = false, data<>[32K]. Return a short read, as the last half of the request was all zeroes.

2. READ(s, 32K, 64K) --> NFS4ERR_HOLE, READ4reshole(256K, 32K). The requested range was all zeros, and the next chunk is located at offset 256K and is 32K in length.

3. READ(s, 256K, 32K) --> NFS_OK.

4. READ(s, 288K, 64K) --> NFS4ERR_HOLE, READ4reshole(354K, 64K). The client has no information regarding this range, so it issues the read request to find the next hole in the file.

5. READ(s, 354K, 64K) --> NFS_OK, eof = true.

6. Related Work

Solaris and ZFS support an extension to lseek(2) that allows applications to discover holes in a file. The values, SEEK_HOLE and

SEEK_DATA, allow clients to seek to the next hole or beginning of data, respectively.

XFS supports the XFS_IOC_GETBMAP extended attribute, which returns the allocation information for a file. Clients can then use this information to only read allocated data blocks.

NTFS and CIFS support the FSCTL_SET_SPARSE attribute, which allows applications to control whether empty regions of the file are preallocated and filled in with zeros or simply left unallocated.

7. Security Considerations

The additions to the NFS protocol for supporting sparse file reads does not alter the security considerations of the NFSv4.1 protocol [3].

8. IANA Considerations

There are no IANA considerations in this document. All NFSv4.1 IANA considerations are covered in [3].

9. References

9.1. Normative References

[1] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[2] Shepler, S., Callaghan, B., Robinson, D., Thurlow, R., Beame, C., Eisler, M., and D. Noveck, "Network File System (NFS) version 4 Protocol", RFC 3530, April 2003.

[3] Shepler, S., Eisler, M., and D. Noveck, "Network File System (NFS) Version 4 Minor Version 1 Protocol", RFC 5661, January 2010.

9.2. Informative References

[4] Shepler, S., Eisler, M., and D. Noveck, "Network File System (NFS) Version 4 Minor Version 1 External Data Representation Standard (XDR) Description", RFC 5662, January 2010.

[5] Nowicki, B., "NFS: Network File System Protocol specification", RFC 1094, March 1989.

[6] Callaghan, B., Pawlowski, B., and P. Staubach, "NFS Version 3 Protocol Specification", RFC 1813, June 1995.

10. Acknowledgments

This document was prepared using 2-Word-v2.0.template.dot. Valuable input and advice was received from Sorin Faibish, Benny Halevy, Trond Myklebust, and Richard Scheffenegger.

Authors' Addresses

Dean Hildebrand
IBM Almaden
650 Harry Rd
San Jose, CA 95120

Phone: +1 408-927-2013
Email: dhildeb@us.ibm.com

Marc Eshel
IBM Almaden
650 Harry Rd
San Jose, CA 95120

Phone: +1 408-927-1894
Email: eshel@almaden.ibm.com