NFSv4 Working Group   D. Hildebrand
Internet Draft   M. Eshel
Intended status: Standards Track   IBM Almaden
Expires: June 2011   December 6, 2010

Simple and Efficient Read Support for Sparse Files
draft-hildebrand-nfsv4-read-sparse-02.txt

Status of this Memo

Copyright Notice

Abstract

This document proposes a new READPLUS operation for NFSv4.2 to support efficient reading of sparse files, which are growing in the data center due to the increasing number of virtual disk images. READPLUS has all the features and functionality of READ, but has an extensible return value that includes an easy and efficient way for administrators to copy and manage sparse 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 puts forth a proposal for the NFSv4.2 protocol to support efficient reading of sparse files.

A sparse file is a common way of representing a large file without having to reserve disk space for it. Consequently, a sparse file uses less physical space 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 a hole in a sparse file, the file system must returns all zeros 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 adds a new READPLUS operation to efficiently read from sparse files by avoiding the transfer of all zero regions from the server to the client. READPLUS supports all the features of READ but includes a minimal extension to support sparse files. In addition, the return value of READPLUS is now compatible with NFSv4.1 minor versioning rules and could support other future extensions without requiring yet another operation. READPLUS is guaranteed to perform no worse than READ, and can dramatically improve performance with sparse files. READPLUS does not depend on pNFS protocol features, but can be used by pNFS to support sparse files.

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

o Regular file: An object of file type NF4REG or NF4NAMEDATTR.

o Sparse File. A Regular file that contains one or more Holes.

o Hole. A byte range within a Sparse file that contains regions of all zeroes. For block-based file systems, this could also be an unallocated region of the file.

3. Applications and Sparse Files

Applications may cause an NFS client to read holes 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 holes 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 holes and non-holes, 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 file holes.

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 is transferred from the NFS server to the NFS client. Once a byte range of the file 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 all zero regions, whereas scp does not support sparse files and will transfer every byte of the file.

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 holes 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 READPLUS 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 Data Region Map from the server. A Data Region Map would specify all zero and non-zero regions in a file. While this option seems simple, it is less useful and can become inefficient and cumbersome for several reasons:

o Data Region Maps can be large, and transferring them can reduce overall read performance. For example, VMware's .vmdk files can have a file size of over 100 GBs and have a map well over several MBs.

o Data Region Maps can change frequently, and become invalidated on every write to the file. This can result the map being 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 Data Region Map not only for itself, but for all other clients accessing the same file system image.

o Data Region Maps do not handle all zero-filled sections of the file, reducing the effectiveness of the solution. While it may be possible to modify the maps to handle zero-filled sections (at possibly great effort to the server), it is almost impossible with pNFS. With pNFS, the owner of the Data Region Map is the metadata server, which is not in the data path and has no knowledge of the contents of a data region.

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 READPLUS

The section introduces a new read operation, named READPLUS, which allows NFS clients to avoid reading holes in a sparse file. READPLUS is guaranteed to perform no worse than READ, and can dramatically improve performance with sparse files.

READPLUS supports all the features of the existing NFSv4.1 READ operation [3] and adds a simple yet significant extension to the format of its response. The change allows the client to avoid returning all zeroes from a file hole, wasting computational and

network resources and reducing performance. READPLUS uses a new result structure that tells the client that the result is all zeroes AND the byte-range of the hole in which the request was made. Returning the hole's byte-range, and only upon request, avoids transferring large Data Region Maps that may be soon invalidated and contain information about a file that may not even be read in its entirely.

A new read operation is required due to NFSv4.1 minor versioning rules that do not allow modification of existing operation's arguments or results. READPLUS is designed in such a way to allow future extensions to the result structure. The same approach could be taken to extend the argument structure, but a good use case is first required to make such a change.

5.1. ARGUMENTS

```
   struct READPLUS4args {
           /* CURRENT_FH: file */
           stateid4        stateid;
           offset4         offset;
           count4          count;
   };
```

5.2. RESULTS

```
     union nfs_readplusreshole switch  (holeres4 resop) {
         CASE HOLE_NOINFO:
             void;
         CASE HOLE_INFO:
             offset4         hole_offset;
             length4         hole_length;
      };
      union nfs_readplusresok4 switch  (readplusrestype4 resop) {
         CASE READ_OK:
              opaque          data<>;
         CASE READ_HOLE:
              nfs_readplusreshole   reshole4;
      };

      union READPLUS4res switch (nfsstat4 status) {
       case NFS4_OK:
               bool            eof;
               nfs_readresok4  resok4;
       default:
```

```
               void;
      };
```

5.3. DESCRIPTION

The READPLUS operation is based upon the NFSv4.1 READ operation [3], and similarly reads data from the regular file identified by the current filehandle.

The client provides an offset of where the READPLUS is to start and a count of how many bytes are to be read. An offset of zero means to read data starting at the beginning of the file. If offset is greater than or equal to the size of the file, the status NFS4_OK is returned with nfs_readplusrestype4 set to READ_OK, data length set to zero, and eof set to TRUE. The READPLUS is subject to access permissions checking.

If the client specifies a count value of zero, the READPLUS succeeds and returns zero bytes of data, again subject to access permissions checking. In all situations, the server may choose to return fewer bytes than specified by the client. The client needs to check for this condition and handle the condition appropriately.

If the client specifies an offset and count value that is entirely contained within a hole of the file, the status NFS4_OK is returned with nfs_readplusresok4 set to READ_HOLE, and if information is available regarding the hole, a nfs_readplusreshole structure containing the offset and range of the entire hole. The nfs_readplusreshole structure is considered valid until the file is changed (detected via the change attribute). The server MUST provide the same semantics for nfs_readplusreshole as if the client read the region and received zeroes; the implied holes contents lifetime MUST be exactly the same as any other read data.

If the client specifies an offset and count value that begins in a non-hole of the file but extends into hole the server should return a short read with status NFS4_OK, nfs_readplusresok4 set to READ_OK, and data length set to the number of bytes returned. The client will then issue another READPLUS for the remaining bytes, which the server will respond with information about the hole in the file.

If the server knows that the requested byte range is into a hole of the file, but has no further information regarding the hole, it returns a nfs_readplusreshole structure with holeres4 set to HOLE_NOINFO.

If hole information is available on the server and can be returned to the client, the server returns a nfs_readplusreshole structure with the value of holeres4 to HOLE_INFO. The values of hole_offset and hole_length define the byte-range for the current hole in the file. These values represent the information known to the server and may describe a byte-range smaller than the true size of the hole.

Except when special stateids are used, the stateid value for a READPLUS request represents a value returned from a previous byte-range lock or share reservation request or the stateid associated with a delegation. The stateid identifies the associated owners if any and is used by the server to verify that the associated locks are still valid (e.g., have not been revoked).

If the read ended at the end-of-file (formally, in a correctly formed READPLUS operation, if offset + count is equal to the size of the file), or the READPLUS operation extends beyond the size of the file (if offset + count is greater than the size of the file), eof is returned as TRUE; otherwise, it is FALSE. A successful READPLUS of an empty file will always return eof as TRUE.

If the current filehandle is not an ordinary file, an error will be returned to the client. In the case that the current filehandle represents an object of type NF4DIR, NFS4ERR_ISDIR is returned. If the current filehandle designates a symbolic link, NFS4ERR_SYMLINK is returned. In all other cases, NFS4ERR_WRONG_TYPE is returned.

For a READPLUS with a stateid value of all bits equal to zero, the server MAY allow the READPLUS to be serviced subject to mandatory byte-range locks or the current share deny modes for the file. For a READPLUS with a stateid value of all bits equal to one, the server MAY allow READPLUS operations to bypass locking checks at the server.

On success, the current filehandle retains its value.

5.4. IMPLEMENTATION

If the server returns a "short read" (i.e., fewer data than requested and eof is set to FALSE), the client should send another READPLUS to get the remaining data. A server may return less data than requested under several circumstances. The file may have been truncated by another client or perhaps on the server itself, changing the file size from what the requesting client believes to be the case. This would reduce the actual amount of data available to the client. It is possible that the server reduce the transfer size and so return a short read result. Server resource exhaustion may also occur in a short read.

If mandatory byte-range locking is in effect for the file, and if the byte-range corresponding to the data to be read from the file is WRITE_LT locked by an owner not associated with the stateid, the server will return the NFS4ERR_LOCKED error. The client should try to get the appropriate READ_LT via the LOCK operation before re-attempting the READPLUS. When the READPLUS completes, the client should release the byte-range lock via LOCKU.

If another client has an OPEN_DELEGATE_WRITE delegation for the file being read, the delegation must be recalled, and the operation cannot proceed until that delegation is returned or revoked. Except where this happens very quickly, one or more NFS4ERR_DELAY errors will be returned to requests made while the delegation remains outstanding. Normally, delegations will not be recalled as a result of a READPLUS operation since the recall will occur as a result of an earlier OPEN. However, since it is possible for a READPLUS to be done with a special stateid, the server needs to check for this case even though the client should have done an OPEN previously.

5.4.1. Additional pNFS Implementation Information

With pNFS, the semantics of using READPLUS remains the same. Any data server MAY return a READ_HOLE result for a READPLUS request that it receives.

When a data server chooses to return a READ_HOLE result, it has a certain level of flexibility in how it fills out the nfs_readplusreshole structure.

1. For a data server that cannot determine any hole information, the data server SHOULD return HOLE_NOINFO.

2. For a data server that can only obtain hole information for the parts of the file stored on that data server, the data server SHOULD return HOLE_INFO and the byte range of the hole stored on that data server.

3. For a data server that can obtain hole information for the entire file without severe performance impact, it MAY return HOLE_INFO and the byte range of the entire file hole.

In general, a data server should do its best to return as much information about a hole as is feasible. In general, pNFS server implementers should try ensure that data servers do not overload the metadata server with requests for information. Therefore, if supplying global sparse information for a file to data servers can

overwhelm a metadata server, then data servers should use option 1 or 2 above.

When a pNFS client receives a READ_HOLE result and a non-empty nfs_readplusreshole structure, it MAY use this information in conjunction with a valid layout for the file to determine the next data server for the next region of data that is not in a hole.

5.5. READPLUS with Sparse Files Example

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

Byte-Range	Contents
0-31999	Non-Zero
32K-255999	Hole
256K-287999	Non-Zero
288K-353999	Hole
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 READPLUS requests.

1. READPLUS(s, 0, 64K) --> NFS_OK, readplusrestype4 = READ_OK, eof = false, data<>[32K]. Return a short read, as the last half of the request was all zeroes.

2. READPLUS(s, 32K, 64K) --> NFS_OK, readplusrestype4 = READ_HOLE, nfs_readplusreshole(HOLE_INFO)(32K, 224K). The requested range was all zeros, and the current hole begins at offset 32K and is 224K in length.

3. READPLUS(s, 256K, 64K) --> NFS_OK, readplusrestype4 = READ_OK, eof = false, data<>[32K]. Return a short read, as the last half of the request was all zeroes.

4. READPLUS(s, 288K, 64K) --> NFS_OK, readplusrestype4 = READ_HOLE, nfs_readplusreshole(HOLE_INFO)(288K, 66K).

5. READPLUS(s, 354K, 64K) --> NFS_OK, readplusrestype4 = READ_OK, eof = true, data<>[64K].

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 Data Region Map for a file. Clients can then use this information to avoid reading holes in a file.

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, Bruce Fields, 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