Internet Engineering Task Force (IETF)
Request for Comments: 5664
Category: Standards Track
ISSN: 2070-1721

B. Halevy
B. Welch
J. Zelenka
Panasas

January 2010

Object-Based Parallel NFS (pNFS) Operations

Abstract

Parallel NFS (pNFS) extends Network File System version 4 (NFSv4) to allow clients to directly access file data on the storage used by the NFSv4 server. This ability to bypass the server for data access can increase both performance and parallelism, but requires additional client functionality for data access, some of which is dependent on the class of storage used, a.k.a. the Layout Type. The main pNFS operations and data types in NFSv4 Minor version 1 specify a layout-type-independent layer; layout-type-specific information is conveyed using opaque data structures whose internal structure is further defined by the particular layout type specification. This document specifies the NFSv4.1 Object-Based pNFS Layout Type as a companion to the main NFSv4 Minor version 1 specification.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc5664.

Table of Contents

1. Introduction

In pNFS, the file server returns typed layout structures that describe where file data is located. There are different layouts for different storage systems and methods of arranging data on storage devices. This document describes the layouts used with object-based storage devices (OSDs) that are accessed according to the OSD storage protocol standard (ANSI INCITS 400-2004 [1]).

An "object" is a container for data and attributes, and files are stored in one or more objects. The OSD protocol specifies several operations on objects, including READ, WRITE, FLUSH, GET ATTRIBUTES, SET ATTRIBUTES, CREATE, and DELETE. However, using the object-based layout the client only uses the READ, WRITE, GET ATTRIBUTES, and FLUSH commands. The other commands are only used by the pNFS server.

An object-based layout for pNFS includes object identifiers, capabilities that allow clients to READ or WRITE those objects, and various parameters that control how file data is striped across their component objects. The OSD protocol has a capability-based security scheme that allows the pNFS server to control what operations and what objects can be used by clients. This scheme is described in more detail in the "Security Considerations" section (Section 13).

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 [2].

2. XDR Description of the Objects-Based Layout Protocol

This document contains the external data representation (XDR [3]) description of the NFSv4.1 objects layout protocol. The XDR description is embedded 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 NFSv4.1 objects layout protocol:

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

That is, 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 > pnfs_osd_prot.x
```

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

The embedded XDR file header follows. Subsequent XDR descriptions, with the sentinel sequence are embedded throughout the document.

Note that the XDR code contained in this document depends on types from the NFSv4.1 nfs4_prot.x file ([4]). This includes both nfs types that end with a 4, such as offset4, length4, etc., as well as more generic types such as uint32_t and uint64_t.

2.1. Code Components Licensing Notice

The XDR description, marked with lines beginning with the sequence "///", as well as scripts for extracting the XDR description are Code Components as described in Section 4 of "Legal Provisions Relating to IETF Documents" [5]. These Code Components are licensed according to the terms of Section 4 of "Legal Provisions Relating to IETF Documents".

```
/// /*
///  * Copyright (c) 2010 IETF Trust and the persons identified
///  * as authors of the code.  All rights reserved.
///  *
///  * Redistribution and use in source and binary forms, with
///  * or without modification, are permitted provided that the
///  * following conditions are met:
///  *
///  * o Redistributions of source code must retain the above
///  *   copyright notice, this list of conditions and the
///  *   following disclaimer.
///  *
///  * o Redistributions in binary form must reproduce the above
///  *   copyright notice, this list of conditions and the
///  *   following disclaimer in the documentation and/or other
///  *   materials provided with the distribution.
///  *
///  * o Neither the name of Internet Society, IETF or IETF
///  *   Trust, nor the names of specific contributors, may be
///  *   used to endorse or promote products derived from this
///  *   software without specific prior written permission.
///  *
///  *   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS
///  *   AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED
///  *   WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
///  *   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
///  *   FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO
///  *   EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
///  *   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
///  *   EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
///  *   NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
///  *   SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
///  *   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
///  *   LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
///  *   OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
///  *   IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
///  *   ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
///  *
///  * This code was derived from RFC 5664.
///  * Please reproduce this note if possible.
///  */
///
/// /*
///  * pnfs_osd_prot.x
///  */
///
/// %#include <nfs4_prot.x>
///
```

3. Basic Data Type Definitions

The following sections define basic data types and constants used by the Object-Based Layout protocol.

3.1. pnfs_osd_objid4

An object is identified by a number, somewhat like an inode number. The object storage model has a two-level scheme, where the objects within an object storage device are grouped into partitions.

```
/// struct pnfs_osd_objid4 {
///     deviceid4       oid_device_id;
///     uint64_t        oid_partition_id;
///     uint64_t        oid_object_id;
/// };
///
```

The pnfs_osd_objid4 type is used to identify an object within a partition on a specified object storage device. "oid_device_id" selects the object storage device from the set of available storage devices. The device is identified with the deviceid4 type, which is an index into addressing information about that device returned by the GETDEVICELIST and GETDEVICEINFO operations. The deviceid4 data type is defined in NFSv4.1 [6]. Within an OSD, a partition is identified with a 64-bit number, "oid_partition_id". Within a partition, an object is identified with a 64-bit number, "oid_object_id". Creation and management of partitions is outside the scope of this document, and is a facility provided by the object-based storage file system.

3.2. pnfs_osd_version4

```
/// enum pnfs_osd_version4 {
///     PNFS_OSD_MISSING    = 0,
///     PNFS_OSD_VERSION_1  = 1,
///     PNFS_OSD_VERSION_2  = 2
/// };
///
```

pnfs_osd_version4 is used to indicate the OSD protocol version or whether an object is missing (i.e., unavailable). Some of the object-based layout-supported RAID algorithms encode redundant information and can compensate for missing components, but the data placement algorithm needs to know what parts are missing.

At this time, the OSD standard is at version 1.0, and we anticipate a version 2.0 of the standard (SNIA T10/1729-D [14]). The second generation OSD protocol has additional proposed features to support more robust error recovery, snapshots, and byte-range capabilities. Therefore, the OSD version is explicitly called out in the information returned in the layout. (This information can also be deduced by looking inside the capability type at the format field, which is the first byte. The format value is 0x1 for an OSD v1 capability. However, it seems most robust to call out the version explicitly.)

3.3. pnfs_osd_object_cred4

```
/// enum pnfs_osd_cap_key_sec4 {
///     PNFS_OSD_CAP_KEY_SEC_NONE = 0,
///     PNFS_OSD_CAP_KEY_SEC_SSV  = 1
/// };
///
/// struct pnfs_osd_object_cred4 {
///     pnfs_osd_objid4         oc_object_id;
///     pnfs_osd_version4       oc_osd_version;
///     pnfs_osd_cap_key_sec4   oc_cap_key_sec;
///     opaque                  oc_capability_key<>;
///     opaque                  oc_capability<>;
/// };
///
```

The pnfs_osd_object_cred4 structure is used to identify each component comprising the file. The "oc_object_id" identifies the component object, the "oc_osd_version" represents the osd protocol version, or whether that component is unavailable, and the "oc_capability" and "oc_capability_key", along with the "oda_systemid" from the pnfs_osd_deviceaddr4, provide the OSD security credentials needed to access that object. The "oc_cap_key_sec" value denotes the method used to secure the oc_capability_key (see Section 13.1 for more details).

To comply with the OSD security requirements, the capability key SHOULD be transferred securely to prevent eavesdropping (see Section 13). Therefore, a client SHOULD either issue the LAYOUTGET or GETDEVICEINFO operations via RPCSEC_GSS with the privacy service or previously establish a secret state verifier (SSV) for the sessions via the NFSv4.1 SET_SSV operation. The pnfs_osd_cap_key_sec4 type is used to identify the method used by the server to secure the capability key.

- PNFS_OSD_CAP_KEY_SEC_NONE denotes that the oc_capability_key is not encrypted, in which case the client SHOULD issue the LAYOUTGET or GETDEVICEINFO operations with RPCSEC_GSS with the privacy service or the NFSv4.1 transport should be secured by using methods that are external to NFSv4.1 like the use of IPsec [15] for transporting the NFSV4.1 protocol.

- PNFS_OSD_CAP_KEY_SEC_SSV denotes that the oc_capability_key contents are encrypted using the SSV GSS context and the capability key as inputs to the GSS_Wrap() function (see GSS-API [7]) with the conf_req_flag set to TRUE. The client MUST use the secret SSV key as part of the client's GSS context to decrypt the capability key using the value of the oc_capability_key field as the input_message to the GSS_unwrap() function. Note that to prevent eavesdropping of the SSV key, the client SHOULD issue SET_SSV via RPCSEC_GSS with the privacy service.

The actual method chosen depends on whether the client established a SSV key with the server and whether it issued the operation with the RPCSEC_GSS privacy method. Naturally, if the client did not establish an SSV key via SET_SSV, the server MUST use the PNFS_OSD_CAP_KEY_SEC_NONE method. Otherwise, if the operation was not issued with the RPCSEC_GSS privacy method, the server SHOULD secure the oc_capability_key with the PNFS_OSD_CAP_KEY_SEC_SSV method. The server MAY use the PNFS_OSD_CAP_KEY_SEC_SSV method also when the operation was issued with the RPCSEC_GSS privacy method.

3.4. pnfs_osd_raid_algorithm4

```
/// enum pnfs_osd_raid_algorithm4 {
///     PNFS_OSD_RAID_0     = 1,
///     PNFS_OSD_RAID_4     = 2,
///     PNFS_OSD_RAID_5     = 3,
///     PNFS_OSD_RAID_PQ    = 4     /* Reed-Solomon P+Q */
/// };
///
```

pnfs_osd_raid_algorithm4 represents the data redundancy algorithm used to protect the file's contents. See Section 5.4 for more details.

4. Object Storage Device Addressing and Discovery

Data operations to an OSD require the client to know the "address" of each OSD's root object. The root object is synonymous with the Small Computer System Interface (SCSI) logical unit. The client specifies SCSI logical units to its SCSI protocol stack using a representation

local to the client. Because these representations are local, GETDEVICEINFO must return information that can be used by the client to select the correct local representation.

In the block world, a set offset (logical block number or track/ sector) contains a disk label. This label identifies the disk uniquely. In contrast, an OSD has a standard set of attributes on its root object. For device identification purposes, the OSD System ID (root information attribute number 3) and the OSD Name (root information attribute number 9) are used as the label. These appear in the pnfs_osd_deviceaddr4 type below under the "oda_systemid" and "oda_osdname" fields.

In some situations, SCSI target discovery may need to be driven based on information contained in the GETDEVICEINFO response. One example of this is Internet SCSI (iSCSI) targets that are not known to the client until a layout has been requested. The information provided as the "oda_targetid", "oda_targetaddr", and "oda_lun" fields in the pnfs_osd_deviceaddr4 type described below (see Section 4.2) allows the client to probe a specific device given its network address and optionally its iSCSI Name (see iSCSI [8]), or when the device network address is omitted, allows it to discover the object storage device using the provided device name or SCSI Device Identifier (see SPC-3 [9].)

The oda_systemid is implicitly used by the client, by using the object credential signing key to sign each request with the request integrity check value. This method protects the client from unintentionally accessing a device if the device address mapping was changed (or revoked). The server computes the capability key using its own view of the systemid associated with the respective deviceid present in the credential. If the client’s view of the deviceid mapping is stale, the client will use the wrong systemid (which must be system-wide unique) and the I/O request to the OSD will fail to pass the integrity check verification.

To recover from this condition the client should report the error and return the layout using LAYOUTRETURN, and invalidate all the device address mappings associated with this layout. The client can then ask for a new layout if it wishes using LAYOUTGET and resolve the referenced deviceids using GETDEVICEINFO or GETDEVICELIST.

The server MUST provide the oda_systemid and SHOULD also provide the oda_osdname. When the OSD name is present, the client SHOULD get the root information attributes whenever it establishes communication with the OSD and verify that the OSD name it got from the OSD matches the one sent by the metadata server. To do so, the client uses the root_obj_cred credentials.

4.1. pnfs_osd_targetid_type4

The following enum specifies the manner in which a SCSI target can be specified. The target can be specified as a SCSI Name, or as an SCSI Device Identifier.

```
/// enum pnfs_osd_targetid_type4 {
///     OBJ_TARGET_ANON             = 1,
///     OBJ_TARGET_SCSI_NAME        = 2,
///     OBJ_TARGET_SCSI_DEVICE_ID   = 3
/// };
///
```

4.2. pnfs_osd_deviceaddr4

The specification for an object device address is as follows:

```
/// union pnfs_osd_targetid4 switch (pnfs_osd_targetid_type4 oti_type) {
///     case OBJ_TARGET_SCSI_NAME:
///         string              oti_scsi_name<>;
///
///     case OBJ_TARGET_SCSI_DEVICE_ID:
///         opaque              oti_scsi_device_id<>;
///
///     default:
///         void;
/// };
///
/// union pnfs_osd_targetaddr4 switch (bool ota_available) {
///     case TRUE:
///         netaddr4            ota_netaddr;
///     case FALSE:
///         void;
/// };
///
/// struct pnfs_osd_deviceaddr4 {
///     pnfs_osd_targetid4      oda_targetid;
///     pnfs_osd_targetaddr4    oda_targetaddr;
///     opaque                  oda_lun[8];
///     opaque                  oda_systemid<>;
///     pnfs_osd_object_cred4   oda_root_obj_cred;
///     opaque                  oda_osdname<>;
/// };
///
```

4.2.1. SCSI Target Identifier

When "oda_targetid" is specified as an OBJ_TARGET_SCSI_NAME, the "oti_scsi_name" string MUST be formatted as an "iSCSI Name" as specified in iSCSI [8] and [10]. Note that the specification of the oti_scsi_name string format is outside the scope of this document. Parsing the string is based on the string prefix, e.g., "iqn.", "eui.", or "naa." and more formats MAY be specified in the future in accordance with iSCSI Names properties.

Currently, the iSCSI Name provides for naming the target device using a string formatted as an iSCSI Qualified Name (IQN) or as an Extended Unique Identifier (EUI) [11] string. Those are typically used to identify iSCSI or Secure Routing Protocol (SRP) [16] devices. The Network Address Authority (NAA) string format (see [10]) provides for naming the device using globally unique identifiers, as defined in Fibre Channel Framing and Signaling (FC-FS) [17]. These are typically used to identify Fibre Channel or SAS [18] (Serial Attached SCSI) devices. In particular, such devices that are dual-attached both over Fibre Channel or SAS and over iSCSI.

When "oda_targetid" is specified as an OBJ_TARGET_SCSI_DEVICE_ID, the "oti_scsi_device_id" opaque field MUST be formatted as a SCSI Device Identifier as defined in SPC-3 [9] VPD Page 83h (Section 7.6.3. "Device Identification VPD Page"). If the Device Identifier is identical to the OSD System ID, as given by oda_systemid, the server SHOULD provide a zero-length oti_scsi_device_id opaque value. Note that similarly to the "oti_scsi_name", the specification of the oti_scsi_device_id opaque contents is outside the scope of this document and more formats MAY be specified in the future in accordance with SPC-3.

The OBJ_TARGET_ANON pnfs_osd_targetid_type4 MAY be used for providing no target identification. In this case, only the OSD System ID, and optionally the provided network address, are used to locate the device.

4.2.2. Device Network Address

The optional "oda_targetaddr" field MAY be provided by the server as a hint to accelerate device discovery over, e.g., the iSCSI transport protocol. The network address is given with the netaddr4 type, which specifies a TCP/IP based endpoint (as specified in NFSv4.1 [6]). When given, the client SHOULD use it to probe for the SCSI device at the given network address. The client MAY still use other discovery mechanisms such as Internet Storage Name Service (iSNS) [12] to locate the device using the oda_targetid. In particular, such an

external name service SHOULD be used when the devices may be attached to the network using multiple connections, and/or multiple storage fabrics (e.g., Fibre-Channel and iSCSI).

The "oda_lun" field identifies the OSD 64-bit Logical Unit Number, formatted in accordance with SAM-3 [13]. The client uses the Logical Unit Number to communicate with the specific OSD Logical Unit. Its use is defined in detail by the SCSI transport protocol, e.g., iSCSI [8].

5. Object-Based Layout

The layout4 type is defined in the NFSv4.1 [6] as follows:

```
enum layouttype4 {
    LAYOUT4_NFSV4_1_FILES   = 1,
    LAYOUT4_OSD2_OBJECTS    = 2,
    LAYOUT4_BLOCK_VOLUME    = 3
};

struct layout_content4 {
    layouttype4             loc_type;
    opaque                  loc_body<>;
};

struct layout4 {
    offset4                 lo_offset;
    length4                 lo_length;
    layoutiomode4           lo_iomode;
    layout_content4         lo_content;
};
```

This document defines structure associated with the layouttype4 value, LAYOUT4_OSD2_OBJECTS. The NFSv4.1 [6] specifies the loc_body structure as an XDR type "opaque". The opaque layout is uninterpreted by the generic pNFS client layers, but obviously must be interpreted by the object storage layout driver. This section defines the structure of this opaque value, pnfs_osd_layout4.

5.1. pnfs_osd_data_map4

```
/// struct pnfs_osd_data_map4 {
///     uint32_t                    odm_num_comps;
///     length4                     odm_stripe_unit;
///     uint32_t                    odm_group_width;
///     uint32_t                    odm_group_depth;
///     uint32_t                    odm_mirror_cnt;
///     pnfs_osd_raid_algorithm4    odm_raid_algorithm;
/// };
///
```

The pnfs_osd_data_map4 structure parameterizes the algorithm that maps a file's contents over the component objects. Instead of limiting the system to simple striping scheme where loss of a single component object results in data loss, the map parameters support mirroring and more complicated schemes that protect against loss of a component object.

"odm_num_comps" is the number of component objects the file is striped over. The server MAY grow the file by adding more components to the stripe while clients hold valid layouts until the file has reached its final stripe width. The file length in this case MUST be limited to the number of bytes in a full stripe.

The "odm_stripe_unit" is the number of bytes placed on one component before advancing to the next one in the list of components. The number of bytes in a full stripe is odm_stripe_unit times the number of components. In some RAID schemes, a stripe includes redundant information (i.e., parity) that lets the system recover from loss or damage to a component object.

The "odm_group_width" and "odm_group_depth" parameters allow a nested striping pattern (see Section 5.3.2 for details). If there is no nesting, then odm_group_width and odm_group_depth MUST be zero. The size of the components array MUST be a multiple of odm_group_width.

The "odm_mirror_cnt" is used to replicate a file by replicating its component objects. If there is no mirroring, then odm_mirror_cnt MUST be 0. If odm_mirror_cnt is greater than zero, then the size of the component array MUST be a multiple of (odm_mirror_cnt+1).

See Section 5.3 for more details.

5.2. pnfs_osd_layout4

```
/// struct pnfs_osd_layout4 {
///     pnfs_osd_data_map4      olo_map;
///     uint32_t                olo_comps_index;
///     pnfs_osd_object_cred4   olo_components<>;
/// };
///
```

The pnfs_osd_layout4 structure specifies a layout over a set of component objects. The "olo_components" field is an array of object identifiers and security credentials that grant access to each object. The organization of the data is defined by the pnfs_osd_data_map4 type that specifies how the file's data is mapped onto the component objects (i.e., the striping pattern). The data placement algorithm that maps file data onto component objects assumes that each component object occurs exactly once in the array of components. Therefore, component objects MUST appear in the olo_components array only once. The components array may represent all objects comprising the file, in which case "olo_comps_index" is set to zero and the number of entries in the olo_components array is equal to olo_map.odm_num_comps. The server MAY return fewer components than odm_num_comps, provided that the returned components are sufficient to access any byte in the layout's data range (e.g., a sub-stripe of "odm_group_width" components). In this case, olo_comps_index represents the position of the returned components array within the full array of components that comprise the file.

Note that the layout depends on the file size, which the client learns from the generic return parameters of LAYOUTGET, by doing GETATTR commands to the metadata server. The client uses the file size to decide if it should fill holes with zeros or return a short read. Striping patterns can cause cases where component objects are shorter than other components because a hole happens to correspond to the last part of the component object.

5.3. Data Mapping Schemes

This section describes the different data mapping schemes in detail. The object layout always uses a "dense" layout as described in NFSv4.1 [6]. This means that the second stripe unit of the file starts at offset 0 of the second component, rather than at offset stripe_unit bytes. After a full stripe has been written, the next stripe unit is appended to the first component object in the list without any holes in the component objects.

5.3.1. Simple Striping

The mapping from the logical offset within a file (L) to the component object C and object-specific offset O is defined by the following equations:

```
L = logical offset into the file
W = total number of components
S = W * stripe_unit
N = L / S
C = (L-(N*S)) / stripe_unit
O = (N*stripe_unit)+(L%stripe_unit)
```

In these equations, S is the number of bytes in a full stripe, and N is the stripe number. C is an index into the array of components, so it selects a particular object storage device. Both N and C count from zero. O is the offset within the object that corresponds to the file offset. Note that this computation does not accommodate the same object appearing in the olo_components array multiple times.

For example, consider an object striped over four devices, <D0 D1 D2 D3>. The stripe_unit is 4096 bytes. The stripe width S is thus 4 * 4096 = 16384.

Offset 0:

```
  N = 0 / 16384 = 0
  C = 0-0/4096 = 0 (D0)
  O = 0*4096 + (0%4096) = 0
```

Offset 4096:

```
  N = 4096 / 16384 = 0
  C = (4096-(0*16384)) / 4096 = 1 (D1)
  O = (0*4096)+(4096%4096) = 0
```

Offset 9000:

```
  N = 9000 / 16384 = 0
  C = (9000-(0*16384)) / 4096 = 2 (D2)
  O = (0*4096)+(9000%4096) = 808
```

Offset 132000:

```
  N = 132000 / 16384 = 8
  C = (132000-(8*16384)) / 4096 = 0 (D0)
  O = (8*4096) + (132000%4096) = 33696
```

5.3.2. Nested Striping

The odm_group_width and odm_group_depth parameters allow a nested striping pattern. odm_group_width defines the width of a data stripe and odm_group_depth defines how many stripes are written before advancing to the next group of components in the list of component objects for the file. The math used to map from a file offset to a component object and offset within that object is shown below. The computations map from the logical offset L to the component index C and offset relative O within that component object.

```
L = logical offset into the file
W = total number of components
S = stripe_unit * group_depth * W
T = stripe_unit * group_depth * group_width
U = stripe_unit * group_width
M = L / S
G = (L - (M * S)) / T
H = (L - (M * S)) % T
N = H / U
C = (H - (N * U)) / stripe_unit + G * group_width
O = L % stripe_unit + N * stripe_unit + M * group_depth * stripe_unit
```

In these equations, S is the number of bytes striped across all component objects before the pattern repeats. T is the number of bytes striped within a group of component objects before advancing to the next group. U is the number of bytes in a stripe within a group. M is the "major" (i.e., across all components) stripe number, and N is the "minor" (i.e., across the group) stripe number. G counts the groups from the beginning of the major stripe, and H is the byte offset within the group.

For example, consider an object striped over 100 devices with a group_width of 10, a group_depth of 50, and a stripe_unit of 1 MB. In this scheme, 500 MB are written to the first 10 components, and 5000 MB are written before the pattern wraps back around to the first component in the array.

```
   Offset 0:
     W = 100
     S = 1 MB * 50 * 100 = 5000 MB
     T = 1 MB * 50 * 10 = 500 MB
     U = 1 MB * 10 = 10 MB
     M = 0 / 5000 MB = 0
     G = (0 - (0 * 5000 MB)) / 500 MB = 0
     H = (0 - (0 * 5000 MB)) % 500 MB = 0
     N = 0 / 10 MB = 0
     C = (0 - (0 * 10 MB)) / 1 MB + 0 * 10 = 0
     O = 0 % 1 MB + 0 * 1 MB + 0 * 50 * 1 MB = 0

   Offset 27 MB:
     M = 27 MB / 5000 MB = 0
     G = (27 MB - (0 * 5000 MB)) / 500 MB = 0
     H = (27 MB - (0 * 5000 MB)) % 500 MB = 27 MB
     N = 27 MB / 10 MB = 2
     C = (27 MB - (2 * 10 MB)) / 1 MB + 0 * 10 = 7
     O = 27 MB % 1 MB + 2 * 1 MB + 0 * 50 * 1 MB = 2 MB

   Offset 7232 MB:
     M = 7232 MB / 5000 MB = 1
     G = (7232 MB - (1 * 5000 MB)) / 500 MB = 4
     H = (7232 MB - (1 * 5000 MB)) % 500 MB = 232 MB
     N = 232 MB / 10 MB = 23
     C = (232 MB - (23 * 10 MB)) / 1 MB + 4 * 10 = 42
     O = 7232 MB % 1 MB + 23 * 1 MB + 1 * 50 * 1 MB = 73 MB
```

5.3.3. Mirroring

The odm_mirror_cnt is used to replicate a file by replicating its component objects. If there is no mirroring, then odm_mirror_cnt MUST be 0. If odm_mirror_cnt is greater than zero, then the size of the olo_components array MUST be a multiple of (odm_mirror_cnt+1). Thus, for a classic mirror on two objects, odm_mirror_cnt is one. Note that mirroring can be defined over any RAID algorithm and striping pattern (either simple or nested). If odm_group_width is also non-zero, then the size of the olo_components array MUST be a multiple of odm_group_width * (odm_mirror_cnt+1). Replicas are adjacent in the olo_components array, and the value C produced by the above equations is not a direct index into the olo_components array. Instead, the following equations determine the replica component index RCi, where i ranges from 0 to odm_mirror_cnt.

```
C = component index for striping or two-level striping
i ranges from 0 to odm_mirror_cnt, inclusive
RCi = C * (odm_mirror_cnt+1) + i
```

5.4. RAID Algorithms

pnfs_osd_raid_algorithm4 determines the algorithm and placement of redundant data. This section defines the different redundancy algorithms. Note: The term "RAID" (Redundant Array of Independent Disks) is used in this document to represent an array of component objects that store data for an individual file. The objects are stored on independent object-based storage devices. File data is encoded and striped across the array of component objects using algorithms developed for block-based RAID systems.

5.4.1. PNFS_OSD_RAID_0

PNFS_OSD_RAID_0 means there is no parity data, so all bytes in the component objects are data bytes located by the above equations for C and O. If a component object is marked as PNFS_OSD_MISSING, the pNFS client MUST either return an I/O error if this component is attempted to be read or, alternatively, it can retry the READ against the pNFS server.

5.4.2. PNFS_OSD_RAID_4

PNFS_OSD_RAID_4 means that the last component object, or the last in each group (if odm_group_width is greater than zero), contains parity information computed over the rest of the stripe with an XOR operation. If a component object is unavailable, the client can read the rest of the stripe units in the damaged stripe and recompute the missing stripe unit by XORing the other stripe units in the stripe. Or the client can replay the READ against the pNFS server that will presumably perform the reconstructed read on the client's behalf.

When parity is present in the file, then there is an additional computation to map from the file offset L to the offset that accounts for embedded parity, L'. First compute L', and then use L' in the above equations for C and O.

```
L = file offset, not accounting for parity
P = number of parity devices in each stripe
W = group_width, if not zero, else size of olo_components array
N = L / (W-P * stripe_unit)
L' = N * (W * stripe_unit) +
     (L % (W-P * stripe_unit))
```

5.4.3. PNFS_OSD_RAID_5

PNFS_OSD_RAID_5 means that the position of the parity data is rotated on each stripe or each group (if odm_group_width is greater than zero). In the first stripe, the last component holds the parity. In

the second stripe, the next-to-last component holds the parity, and so on. In this scheme, all stripe units are rotated so that I/O is evenly spread across objects as the file is read sequentially. The rotated parity layout is illustrated here, with numbers indicating the stripe unit.

```
0 1 2 P
4 5 P 3
8 P 6 7
P 9 a b
```

To compute the component object C, first compute the offset that accounts for parity L' and use that to compute C. Then rotate C to get C'. Finally, increase C' by one if the parity information comes at or before C' within that stripe. The following equations illustrate this by computing I, which is the index of the component that contains parity for a given stripe.

```
L = file offset, not accounting for parity
W = odm_group_width, if not zero, else size of olo_components array
N = L / (W-1 * stripe_unit)
(Compute L' as describe above)
(Compute C based on L' as described above)
C' = (C - (N%W)) % W
I = W - (N%W) - 1
if (C' <= I) {
  C'++
}
```

5.4.4. PNFS_OSD_RAID_PQ

PNFS_OSD_RAID_PQ is a double-parity scheme that uses the Reed-Solomon P+Q encoding scheme [19]. In this layout, the last two component objects hold the P and Q data, respectively. P is parity computed with XOR, and Q is a more complex equation that is not described here. The equations given above for embedded parity can be used to map a file offset to the correct component object by setting the number of parity components to 2 instead of 1 for RAID4 or RAID5. Clients may simply choose to read data through the metadata server if two components are missing or damaged.

5.4.5. RAID Usage and Implementation Notes

RAID layouts with redundant data in their stripes require additional serialization of updates to ensure correct operation. Otherwise, if two clients simultaneously write to the same logical range of an object, the result could include different data in the same ranges of mirrored tuples, or corrupt parity information. It is the

responsibility of the metadata server to enforce serialization requirements such as this. For example, the metadata server may do so by not granting overlapping write layouts within mirrored objects.

6. Object-Based Layout Update

layoutupdate4 is used in the LAYOUTCOMMIT operation to convey updates to the layout and additional information to the metadata server. It is defined in the NFSv4.1 [6] as follows:

```
struct layoutupdate4 {
    layouttype4             lou_type;
    opaque                  lou_body<>;
};
```

The layoutupdate4 type is an opaque value at the generic pNFS client level. If the lou_type layout type is LAYOUT4_OSD2_OBJECTS, then the lou_body opaque value is defined by the pnfs_osd_layoutupdate4 type.

Object-Based pNFS clients are not allowed to modify the layout. Therefore, the information passed in pnfs_osd_layoutupdate4 is used only to update the file's attributes. In addition to the generic information the client can pass to the metadata server in LAYOUTCOMMIT such as the highest offset the client wrote to and the last time it modified the file, the client MAY use pnfs_osd_layoutupdate4 to convey the capacity consumed (or released) by writes using the layout, and to indicate that I/O errors were encountered by such writes.

6.1. pnfs_osd_deltaspaceused4

```
/// union pnfs_osd_deltaspaceused4 switch (bool dsu_valid) {
///     case TRUE:
///         int64_t     dsu_delta;
///     case FALSE:
///         void;
/// };
///
```

pnfs_osd_deltaspaceused4 is used to convey space utilization information at the time of LAYOUTCOMMIT. For the file system to properly maintain capacity-used information, it needs to track how much capacity was consumed by WRITE operations performed by the client. In this protocol, the OSD returns the capacity consumed by a write (*), which can be different than the number of bytes written because of internal overhead like block-level allocation and indirect blocks, and the client reflects this back to the pNFS server so it can accurately track quota. The pNFS server can choose to trust this

information coming from the clients and therefore avoid querying the OSDs at the time of LAYOUTCOMMIT. If the client is unable to obtain this information from the OSD, it simply returns invalid olu_delta_space_used.

6.2. pnfs_osd_layoutupdate4

```
/// struct pnfs_osd_layoutupdate4 {
///     pnfs_osd_deltaspaceused4    olu_delta_space_used;
///     bool                        olu_ioerr_flag;
/// };
///
```

"olu_delta_space_used" is used to convey capacity usage information back to the metadata server.

The "olu_ioerr_flag" is used when I/O errors were encountered while writing the file. The client MUST report the errors using the pnfs_osd_ioerr4 structure (see Section 8.1) at LAYOUTRETURN time.

If the client updated the file successfully before hitting the I/O errors, it MAY use LAYOUTCOMMIT to update the metadata server as described above. Typically, in the error-free case, the server MAY turn around and update the file's attributes on the storage devices. However, if I/O errors were encountered, the server better not attempt to write the new attributes on the storage devices until it receives the I/O error report; therefore, the client MUST set the olu_ioerr_flag to true. Note that in this case, the client SHOULD send both the LAYOUTCOMMIT and LAYOUTRETURN operations in the same COMPOUND RPC.

7. Recovering from Client I/O Errors

The pNFS client may encounter errors when directly accessing the object storage devices. However, it is the responsibility of the metadata server to handle the I/O errors. When the LAYOUT4_OSD2_OBJECTS layout type is used, the client MUST report the I/O errors to the server at LAYOUTRETURN time using the pnfs_osd_ioerr4 structure (see Section 8.1).

The metadata server analyzes the error and determines the required recovery operations such as repairing any parity inconsistencies, recovering media failures, or reconstructing missing objects.

The metadata server SHOULD recall any outstanding layouts to allow it exclusive write access to the stripes being recovered and to prevent other clients from hitting the same error condition. In these cases, the server MUST complete recovery before handing out any new layouts to the affected byte ranges.

Although it MAY be acceptable for the client to propagate a corresponding error to the application that initiated the I/O operation and drop any unwritten data, the client SHOULD attempt to retry the original I/O operation by requesting a new layout using LAYOUTGET and retry the I/O operation(s) using the new layout, or the client MAY just retry the I/O operation(s) using regular NFS READ or WRITE operations via the metadata server. The client SHOULD attempt to retrieve a new layout and retry the I/O operation using OSD commands first and only if the error persists, retry the I/O operation via the metadata server.

8. Object-Based Layout Return

layoutreturn_file4 is used in the LAYOUTRETURN operation to convey layout-type specific information to the server. It is defined in the NFSv4.1 [6] as follows:

```
struct layoutreturn_file4 {
        offset4         lrf_offset;
        length4         lrf_length;
        stateid4        lrf_stateid;
        /* layouttype4 specific data */
        opaque          lrf_body<>;
};

union layoutreturn4 switch(layoutreturn_type4 lr_returntype) {
        case LAYOUTRETURN4_FILE:
                layoutreturn_file4      lr_layout;
        default:
                void;
};

struct LAYOUTRETURN4args {
        /* CURRENT_FH: file */
        bool                    lora_reclaim;
        layoutreturn_stateid    lora_recallstateid;
        layouttype4             lora_layout_type;
        layoutiomode4           lora_iomode;
        layoutreturn4           lora_layoutreturn;
};
```

If the lora_layout_type layout type is LAYOUT4_OSD2_OBJECTS, then the lrf_body opaque value is defined by the pnfs_osd_layoutreturn4 type.

The pnfs_osd_layoutreturn4 type allows the client to report I/O error information back to the metadata server as defined below.

8.1. pnfs_osd_errno4

```
/// enum pnfs_osd_errno4 {
///     PNFS_OSD_ERR_EIO            = 1,
///     PNFS_OSD_ERR_NOT_FOUND      = 2,
///     PNFS_OSD_ERR_NO_SPACE       = 3,
///     PNFS_OSD_ERR_BAD_CRED       = 4,
///     PNFS_OSD_ERR_NO_ACCESS      = 5,
///     PNFS_OSD_ERR_UNREACHABLE    = 6,
///     PNFS_OSD_ERR_RESOURCE       = 7
/// };
///
```

pnfs_osd_errno4 is used to represent error types when read/write errors are reported to the metadata server. The error codes serve as hints to the metadata server that may help it in diagnosing the exact reason for the error and in repairing it.

- PNFS_OSD_ERR_EIO indicates the operation failed because the object storage device experienced a failure trying to access the object. The most common source of these errors is media errors, but other internal errors might cause this as well. In this case, the metadata server should go examine the broken object more closely; hence, it should be used as the default error code.

- PNFS_OSD_ERR_NOT_FOUND indicates the object ID specifies an object that does not exist on the object storage device.

- PNFS_OSD_ERR_NO_SPACE indicates the operation failed because the object storage device ran out of free capacity during the operation.

- PNFS_OSD_ERR_BAD_CRED indicates the security parameters are not valid. The primary cause of this is that the capability has expired, or the access policy tag (a.k.a., capability version number) has been changed to revoke capabilities. The client will need to return the layout and get a new one with fresh capabilities.

- PNFS_OSD_ERR_NO_ACCESS indicates the capability does not allow the requested operation. This should not occur in normal operation because the metadata server should give out correct capabilities, or none at all.

- PNFS_OSD_ERR_UNREACHABLE indicates the client did not complete the I/O operation at the object storage device due to a communication failure. Whether or not the I/O operation was executed by the OSD is undetermined.

- PNFS_OSD_ERR_RESOURCE indicates the client did not issue the I/O operation due to a local problem on the initiator (i.e., client) side, e.g., when running out of memory. The client MUST guarantee that the OSD command was never dispatched to the OSD.

8.2. pnfs_osd_ioerr4

```
/// struct pnfs_osd_ioerr4 {
///     pnfs_osd_objid4     oer_component;
///     length4             oer_comp_offset;
///     length4             oer_comp_length;
///     bool                oer_iswrite;
///     pnfs_osd_errno4     oer_errno;
/// };
///
```

The pnfs_osd_ioerr4 structure is used to return error indications for objects that generated errors during data transfers. These are hints to the metadata server that there are problems with that object. For each error, "oer_component", "oer_comp_offset", and "oer_comp_length" represent the object and byte range within the component object in which the error occurred; "oer_iswrite" is set to "true" if the failed OSD operation was data modifying, and "oer_errno" represents the type of error.

Component byte ranges in the optional pnfs_osd_ioerr4 structure are used for recovering the object and MUST be set by the client to cover all failed I/O operations to the component.

8.3. pnfs_osd_layoutreturn4

```
/// struct pnfs_osd_layoutreturn4 {
///     pnfs_osd_ioerr4             olr_ioerr_report<>;
/// };
///
```

When OSD I/O operations failed, "olr_ioerr_report<>" is used to report these errors to the metadata server as an array of elements of type pnfs_osd_ioerr4. Each element in the array represents an error that occurred on the object specified by oer_component. If no errors are to be reported, the size of the olr_ioerr_report<> array is set to zero.

9. Object-Based Creation Layout Hint

The layouthint4 type is defined in the NFSv4.1 [6] as follows:

```
struct layouthint4 {
    layouttype4           loh_type;
    opaque                loh_body<>;
};
```

The layouthint4 structure is used by the client to pass a hint about the type of layout it would like created for a particular file. If the loh_type layout type is LAYOUT4_OSD2_OBJECTS, then the loh_body opaque value is defined by the pnfs_osd_layouthint4 type.

9.1. pnfs_osd_layouthint4

```
/// union pnfs_osd_max_comps_hint4 switch (bool omx_valid) {
///     case TRUE:
///         uint32_t            omx_max_comps;
///     case FALSE:
///         void;
/// };
///
/// union pnfs_osd_stripe_unit_hint4 switch (bool osu_valid) {
///     case TRUE:
///         length4             osu_stripe_unit;
///     case FALSE:
///         void;
/// };
///
/// union pnfs_osd_group_width_hint4 switch (bool ogw_valid) {
///     case TRUE:
///         uint32_t            ogw_group_width;
///     case FALSE:
///         void;
/// };
///
/// union pnfs_osd_group_depth_hint4 switch (bool ogd_valid) {
///     case TRUE:
///         uint32_t            ogd_group_depth;
///     case FALSE:
```

```
///         void;
/// };
///
/// union pnfs_osd_mirror_cnt_hint4 switch (bool omc_valid) {
///     case TRUE:
///         uint32_t            omc_mirror_cnt;
///     case FALSE:
///         void;
/// };
///
/// union pnfs_osd_raid_algorithm_hint4 switch (bool ora_valid) {
///     case TRUE:
///         pnfs_osd_raid_algorithm4    ora_raid_algorithm;
///     case FALSE:
///         void;
/// };
///
/// struct pnfs_osd_layouthint4 {
///     pnfs_osd_max_comps_hint4        olh_max_comps_hint;
///     pnfs_osd_stripe_unit_hint4      olh_stripe_unit_hint;
///     pnfs_osd_group_width_hint4      olh_group_width_hint;
///     pnfs_osd_group_depth_hint4      olh_group_depth_hint;
///     pnfs_osd_mirror_cnt_hint4       olh_mirror_cnt_hint;
///     pnfs_osd_raid_algorithm_hint4   olh_raid_algorithm_hint;
/// };
///
```

This type conveys hints for the desired data map. All parameters are optional so the client can give values for only the parameters it cares about, e.g. it can provide a hint for the desired number of mirrored components, regardless of the RAID algorithm selected for the file. The server should make an attempt to honor the hints, but it can ignore any or all of them at its own discretion and without failing the respective CREATE operation.

The "olh_max_comps_hint" can be used to limit the total number of component objects comprising the file. All other hints correspond directly to the different fields of pnfs_osd_data_map4.

10. Layout Segments

The pnfs layout operations operate on logical byte ranges. There is no requirement in the protocol for any relationship between byte ranges used in LAYOUTGET to acquire layouts and byte ranges used in CB_LAYOUTRECALL, LAYOUTCOMMIT, or LAYOUTRETURN. However, using OSD byte-range capabilities poses limitations on these operations since

the capabilities associated with layout segments cannot be merged or split. The following guidelines should be followed for proper operation of object-based layouts.

10.1. CB_LAYOUTRECALL and LAYOUTRETURN

In general, the object-based layout driver should keep track of each layout segment it got, keeping record of the segment's iomode, offset, and length. The server should allow the client to get multiple overlapping layout segments but is free to recall the layout to prevent overlap.

In response to CB_LAYOUTRECALL, the client should return all layout segments matching the given iomode and overlapping with the recalled range. When returning the layouts for this byte range with LAYOUTRETURN, the client MUST NOT return a sub-range of a layout segment it has; each LAYOUTRETURN sent MUST completely cover at least one outstanding layout segment.

The server, in turn, should release any segment that exactly matches the clientid, iomode, and byte range given in LAYOUTRETURN. If no exact match is found, then the server should release all layout segments matching the clientid and iomode and that are fully contained in the returned byte range. If none are found and the byte range is a subset of an outstanding layout segment with for the same clientid and iomode, then the client can be considered malfunctioning and the server SHOULD recall all layouts from this client to reset its state. If this behavior repeats, the server SHOULD deny all LAYOUTGETs from this client.

10.2. LAYOUTCOMMIT

LAYOUTCOMMIT is only used by object-based pNFS to convey modified attributes hints and/or to report the presence of I/O errors to the metadata server (MDS). Therefore, the offset and length in LAYOUTCOMMIT4args are reserved for future use and should be set to 0.

11. Recalling Layouts

The object-based metadata server should recall outstanding layouts in the following cases:

- When the file's security policy changes, i.e., Access Control Lists (ACLs) or permission mode bits are set.
- When the file's aggregation map changes, rendering outstanding layouts invalid.

- When there are sharing conflicts. For example, the server will issue stripe-aligned layout segments for RAID-5 objects. To prevent corruption of the file's parity, multiple clients must not hold valid write layouts for the same stripes. An outstanding READ/WRITE (RW) layout should be recalled when a conflicting LAYOUTGET is received from a different client for LAYOUTIOMODE4_RW and for a byte range overlapping with the outstanding layout segment.

11.1. CB_RECALL_ANY

The metadata server can use the CB_RECALL_ANY callback operation to notify the client to return some or all of its layouts. The NFSv4.1 [6] defines the following types:

```
const RCA4_TYPE_MASK_OBJ_LAYOUT_MIN     = 8;
const RCA4_TYPE_MASK_OBJ_LAYOUT_MAX     = 9;

struct  CB_RECALL_ANY4args      {
    uint32_t        craa_objects_to_keep;
    bitmap4         craa_type_mask;
};
```

Typically, CB_RECALL_ANY will be used to recall client state when the server needs to reclaim resources. The craa_type_mask bitmap specifies the type of resources that are recalled and the craa_objects_to_keep value specifies how many of the recalled objects the client is allowed to keep. The object-based layout type mask flags are defined as follows. They represent the iomode of the recalled layouts. In response, the client SHOULD return layouts of the recalled iomode that it needs the least, keeping at most craa_objects_to_keep object-based layouts.

```
/// enum pnfs_osd_cb_recall_any_mask {
///     PNFS_OSD_RCA4_TYPE_MASK_READ = 8,
///     PNFS_OSD_RCA4_TYPE_MASK_RW   = 9
/// };
///
```

The PNFS_OSD_RCA4_TYPE_MASK_READ flag notifies the client to return layouts of iomode LAYOUTIOMODE4_READ. Similarly, the PNFS_OSD_RCA4_TYPE_MASK_RW flag notifies the client to return layouts of iomode LAYOUTIOMODE4_RW. When both mask flags are set, the client is notified to return layouts of either iomode.

12. Client Fencing

In cases where clients are uncommunicative and their lease has expired or when clients fail to return recalled layouts within a lease period at the least (see "Recalling a Layout"[6]), the server MAY revoke client layouts and/or device address mappings and reassign these resources to other clients. To avoid data corruption, the metadata server MUST fence off the revoked clients from the respective objects as described in Section 13.4.

13. Security Considerations

The pNFS extension partitions the NFSv4 file system protocol into two parts, the control path and the data path (storage protocol). The control path contains all the new operations described by this extension; all existing NFSv4 security mechanisms and features apply to the control path. The combination of components in a pNFS system is required to preserve the security properties of NFSv4 with respect to an entity accessing data via a client, including security countermeasures to defend against threats that NFSv4 provides defenses for in environments where these threats are considered significant.

The metadata server enforces the file access-control policy at LAYOUTGET time. The client should use suitable authorization credentials for getting the layout for the requested iomode (READ or RW) and the server verifies the permissions and ACL for these credentials, possibly returning NFS4ERR_ACCESS if the client is not allowed the requested iomode. If the LAYOUTGET operation succeeds the client receives, as part of the layout, a set of object capabilities allowing it I/O access to the specified objects corresponding to the requested iomode. When the client acts on I/O operations on behalf of its local users, it MUST authenticate and authorize the user by issuing respective OPEN and ACCESS calls to the metadata server, similar to having NFSv4 data delegations. If access is allowed, the client uses the corresponding (READ or RW) capabilities to perform the I/O operations at the object storage devices. When the metadata server receives a request to change a file’s permissions or ACL, it SHOULD recall all layouts for that file and it MUST change the capability version attribute on all objects comprising the file to implicitly invalidate any outstanding capabilities before committing to the new permissions and ACL. Doing this will ensure that clients re-authorize their layouts according to the modified permissions and ACL by requesting new layouts. Recalling the layouts in this case is courtesy of the server intended to prevent clients from getting an error on I/Os done after the capability version changed.

The object storage protocol MUST implement the security aspects described in version 1 of the T10 OSD protocol definition [1]. The standard defines four security methods: NOSEC, CAPKEY, CMDRSP, and ALLDATA. To provide minimum level of security allowing verification and enforcement of the server access control policy using the layout security credentials, the NOSEC security method MUST NOT be used for any I/O operation. The remainder of this section gives an overview of the security mechanism described in that standard. The goal is to give the reader a basic understanding of the object security model. Any discrepancies between this text and the actual standard are obviously to be resolved in favor of the OSD standard.

13.1. OSD Security Data Types

There are three main data types associated with object security: a capability, a credential, and security parameters. The capability is a set of fields that specifies an object and what operations can be performed on it. A credential is a signed capability. Only a security manager that knows the secret device keys can correctly sign a capability to form a valid credential. In pNFS, the file server acts as the security manager and returns signed capabilities (i.e., credentials) to the pNFS client. The security parameters are values computed by the issuer of OSD commands (i.e., the client) that prove they hold valid credentials. The client uses the credential as a signing key to sign the requests it makes to OSD, and puts the resulting signatures into the security_parameters field of the OSD command. The object storage device uses the secret keys it shares with the security manager to validate the signature values in the security parameters.

The security types are opaque to the generic layers of the pNFS client. The credential contents are defined as opaque within the pnfs_osd_object_cred4 type. Instead of repeating the definitions here, the reader is referred to Section 4.9.2.2 of the OSD standard.

13.2. The OSD Security Protocol

The object storage protocol relies on a cryptographically secure capability to control accesses at the object storage devices. Capabilities are generated by the metadata server, returned to the client, and used by the client as described below to authenticate their requests to the object-based storage device. Capabilities therefore achieve the required access and open mode checking. They allow the file server to define and check a policy (e.g., open mode) and the OSD to enforce that policy without knowing the details (e.g., user IDs and ACLs).

Since capabilities are tied to layouts, and since they are used to enforce access control, when the file ACL or mode changes the outstanding capabilities MUST be revoked to enforce the new access permissions. The server SHOULD recall layouts to allow clients to gracefully return their capabilities before the access permissions change.

Each capability is specific to a particular object, an operation on that object, a byte range within the object (in OSDv2), and has an explicit expiration time. The capabilities are signed with a secret key that is shared by the object storage devices and the metadata managers. Clients do not have device keys so they are unable to forge the signatures in the security parameters. The combination of a capability, the OSD System ID, and a signature is called a "credential" in the OSD specification.

The details of the security and privacy model for object storage are defined in the T10 OSD standard. The following sketch of the algorithm should help the reader understand the basic model.

LAYOUTGET returns a CapKey and a Cap, which, together with the OSD System ID, are also called a credential. It is a capability and a signature over that capability and the SystemID. The OSD Standard refers to the CapKey as the "Credential integrity check value" and to the ReqMAC as the "Request integrity check value".

```
CapKey = MAC<SecretKey>(Cap, SystemID)
Credential = {Cap, SystemID, CapKey}
```

The client uses CapKey to sign all the requests it issues for that object using the respective Cap. In other words, the Cap appears in the request to the storage device, and that request is signed with the CapKey as follows:

```
ReqMAC = MAC<CapKey>(Req, ReqNonce)
Request = {Cap, Req, ReqNonce, ReqMAC}
```

The following is sent to the OSD: {Cap, Req, ReqNonce, ReqMAC}. The OSD uses the SecretKey it shares with the metadata server to compare the ReqMAC the client sent with a locally computed value:

```
LocalCapKey = MAC<SecretKey>(Cap, SystemID)
LocalReqMAC = MAC<LocalCapKey>(Req, ReqNonce)
```

and if they match the OSD assumes that the capabilities came from an authentic metadata server and allows access to the object, as allowed by the Cap.

13.3. Protocol Privacy Requirements

Note that if the server LAYOUTGET reply, holding CapKey and Cap, is snooped by another client, it can be used to generate valid OSD requests (within the Cap access restrictions).

To provide the required privacy requirements for the capability key returned by LAYOUTGET, the GSS-API [7] framework can be used, e.g., by using the RPCSEC_GSS privacy method to send the LAYOUTGET operation or by using the SSV key to encrypt the oc_capability_key using the GSS_Wrap() function. Two general ways to provide privacy in the absence of GSS-API that are independent of NFSv4 are either an isolated network such as a VLAN or a secure channel provided by IPsec [15].

13.4. Revoking Capabilities

At any time, the metadata server may invalidate all outstanding capabilities on an object by changing its POLICY ACCESS TAG attribute. The value of the POLICY ACCESS TAG is part of a capability, and it must match the state of the object attribute. If they do not match, the OSD rejects accesses to the object with the sense key set to ILLEGAL REQUEST and an additional sense code set to INVALID FIELD IN CDB. When a client attempts to use a capability and is rejected this way, it should issue a LAYOUTCOMMIT for the object and specify PNFS_OSD_BAD_CRED in the olr_ioerr_report parameter. The client may elect to issue a compound LAYOUTRETURN/LAYOUTGET (or LAYOUTCOMMIT/LAYOUTRETURN/LAYOUTGET) to attempt to fetch a refreshed set of capabilities.

The metadata server may elect to change the access policy tag on an object at any time, for any reason (with the understanding that there is likely an associated performance penalty, especially if there are outstanding layouts for this object). The metadata server MUST revoke outstanding capabilities when any one of the following occurs:

- the permissions on the object change,
- a conflicting mandatory byte-range lock is granted, or
- a layout is revoked and reassigned to another client.

A pNFS client will typically hold one layout for each byte range for either READ or READ/WRITE. The client’s credentials are checked by the metadata server at LAYOUTGET time and it is the client’s responsibility to enforce access control among multiple users accessing the same file. It is neither required nor expected that the pNFS client will obtain a separate layout for each user accessing

a shared object. The client SHOULD use OPEN and ACCESS calls to check user permissions when performing I/O so that the server's access control policies are correctly enforced. The result of the ACCESS operation may be cached while the client holds a valid layout as the server is expected to recall layouts when the file's access permissions or ACL change.

14. IANA Considerations

As described in NFSv4.1 [6], new layout type numbers have been assigned by IANA. This document defines the protocol associated with the existing layout type number, LAYOUT4_OSD2_OBJECTS, and it requires no further actions for IANA.

15. References

15.1. Normative References

[1] Weber, R., "Information Technology - SCSI Object-Based Storage Device Commands (OSD)", ANSI INCITS 400-2004, December 2004.

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

[3] Eisler, M., "XDR: External Data Representation Standard", STD 67, RFC 4506, May 2006.

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

[5] IETF Trust, "Legal Provisions Relating to IETF Documents", November 2008, <http://trustee.ietf.org/docs/IETF-Trust-License-Policy.pdf>.

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

[7] Linn, J., "Generic Security Service Application Program Interface Version 2, Update 1", RFC 2743, January 2000.

[8] Satran, J., Meth, K., Sapuntzakis, C., Chadalapaka, M., and E. Zeidner, "Internet Small Computer Systems Interface (iSCSI)", RFC 3720, April 2004.

[9] Weber, R., "SCSI Primary Commands - 3 (SPC-3)", ANSI INCITS 408-2005, October 2005.

[10] Krueger, M., Chadalapaka, M., and R. Elliott, "T11 Network Address Authority (NAA) Naming Format for iSCSI Node Names", RFC 3980, February 2005.

[11] IEEE, "Guidelines for 64-bit Global Identifier (EUI-64) Registration Authority", <http://standards.ieee.org/regauth/oui/tutorials/EUI64.html>.

[12] Tseng, J., Gibbons, K., Travostino, F., Du Laney, C., and J. Souza, "Internet Storage Name Service (iSNS)", RFC 4171, September 2005.

[13] Weber, R., "SCSI Architecture Model - 3 (SAM-3)", ANSI INCITS 402-2005, February 2005.

15.2. Informative References

[14] Weber, R., "SCSI Object-Based Storage Device Commands -2 (OSD-2)", January 2009, <http://www.t10.org/cgi-bin/ac.pl?t=f&f=osd2r05a.pdf>.

[15] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.

[16] T10 1415-D, "SCSI RDMA Protocol (SRP)", ANSI INCITS 365-2002, December 2002.

[17] T11 1619-D, "Fibre Channel Framing and Signaling - 2 (FC-FS-2)", ANSI INCITS 424-2007, February 2007.

[18] T10 1601-D, "Serial Attached SCSI - 1.1 (SAS-1.1)", ANSI INCITS 417-2006, June 2006.

[19] MacWilliams, F. and N. Sloane, "The Theory of Error-Correcting Codes, Part I", 1977.

Appendix A. Acknowledgments

Todd Pisek was a co-editor of the initial versions of this document. Daniel E. Messinger, Pete Wyckoff, Mike Eisler, Sean P. Turner, Brian E. Carpenter, Jari Arkko, David Black, and Jason Glasgow reviewed and commented on this document.

Authors' Addresses

Benny Halevy
Panasas, Inc.
1501 Reedsdale St. Suite 400
Pittsburgh, PA 15233
USA

Phone: +1-412-323-3500
EMail: bhalevy@panasas.com
URI: http://www.panasas.com/

Brent Welch
Panasas, Inc.
6520 Kaiser Drive
Fremont, CA 95444
USA

Phone: +1-510-608-7770
EMail: welch@panasas.com
URI: http://www.panasas.com/

Jim Zelenka
Panasas, Inc.
1501 Reedsdale St. Suite 400
Pittsburgh, PA 15233
USA

Phone: +1-412-323-3500
EMail: jimz@panasas.com
URI: http://www.panasas.com/