A simple and scalable pNFS server for Linux

Christoph Hellwig

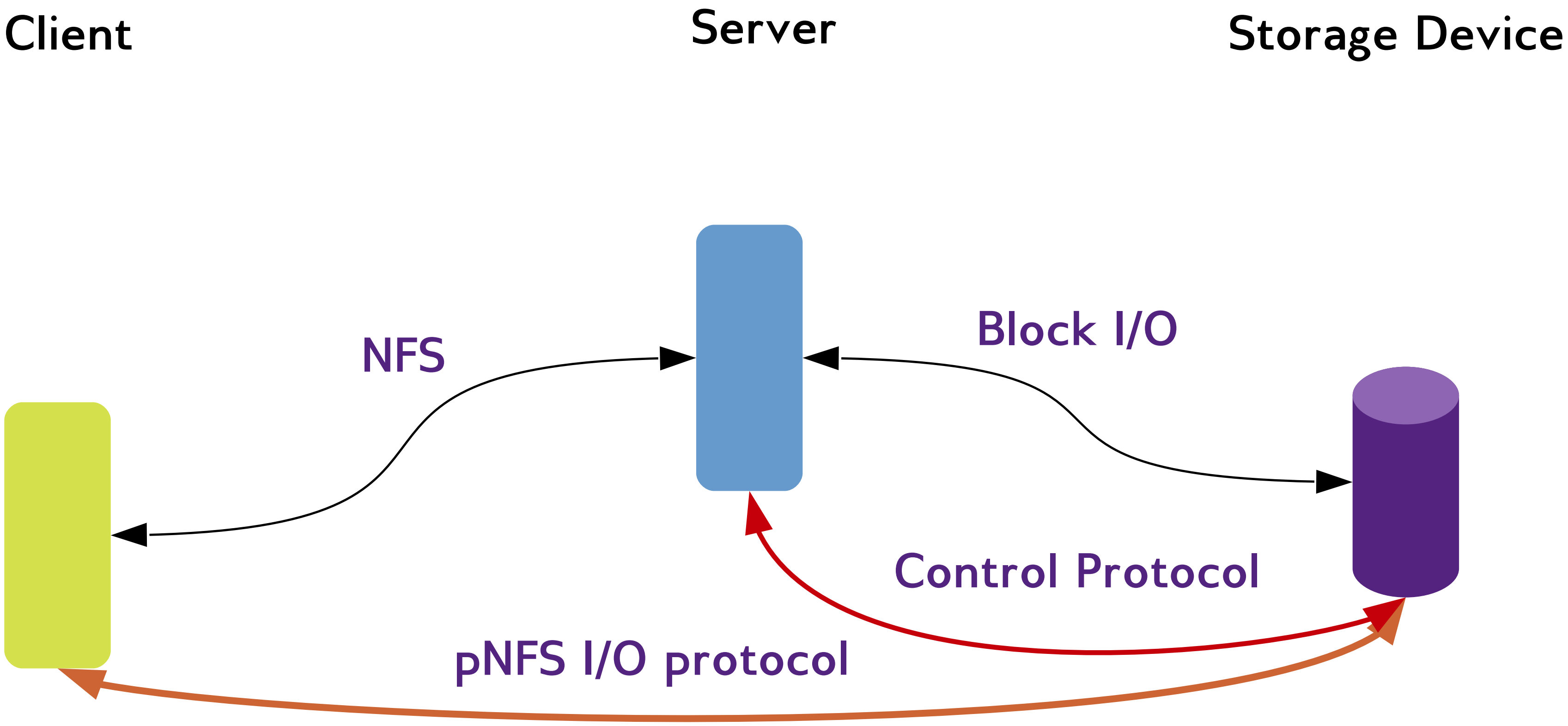
pNFS Overview

- Parallel NFS (pNFS) is a part of the NFSv4.1 spec (RFC5661)
  - Allow to bypass the Meta Data Server (MDS) for data I/O
  - Various different protocols for data I/O which are not part of the main (p)NFS standard

pNFS layout types

- **File layout** (part of *RFC5661*)
  - Uses a subset of NFSv4.1 to storage devices
- **Block layout** (*RFC5663*)
  - Performs block I/O directly to shared block storage
- **Object layout** (*RFC5664*)
  - Uses T10 OSD commands to talk to storage devices

pNFS Overview



pNFS operations

- **LAYOUTGET** (*handle, range*)

  →retrieve type specific layout pointing to a *device ID*

- **LAYOUTRETURN** (*handle, range*)

  → release layout

- **LAYOUTCOMMIT** (*handle, range, attributes*)

  → commit data so that it is visible to other clients, update meta data

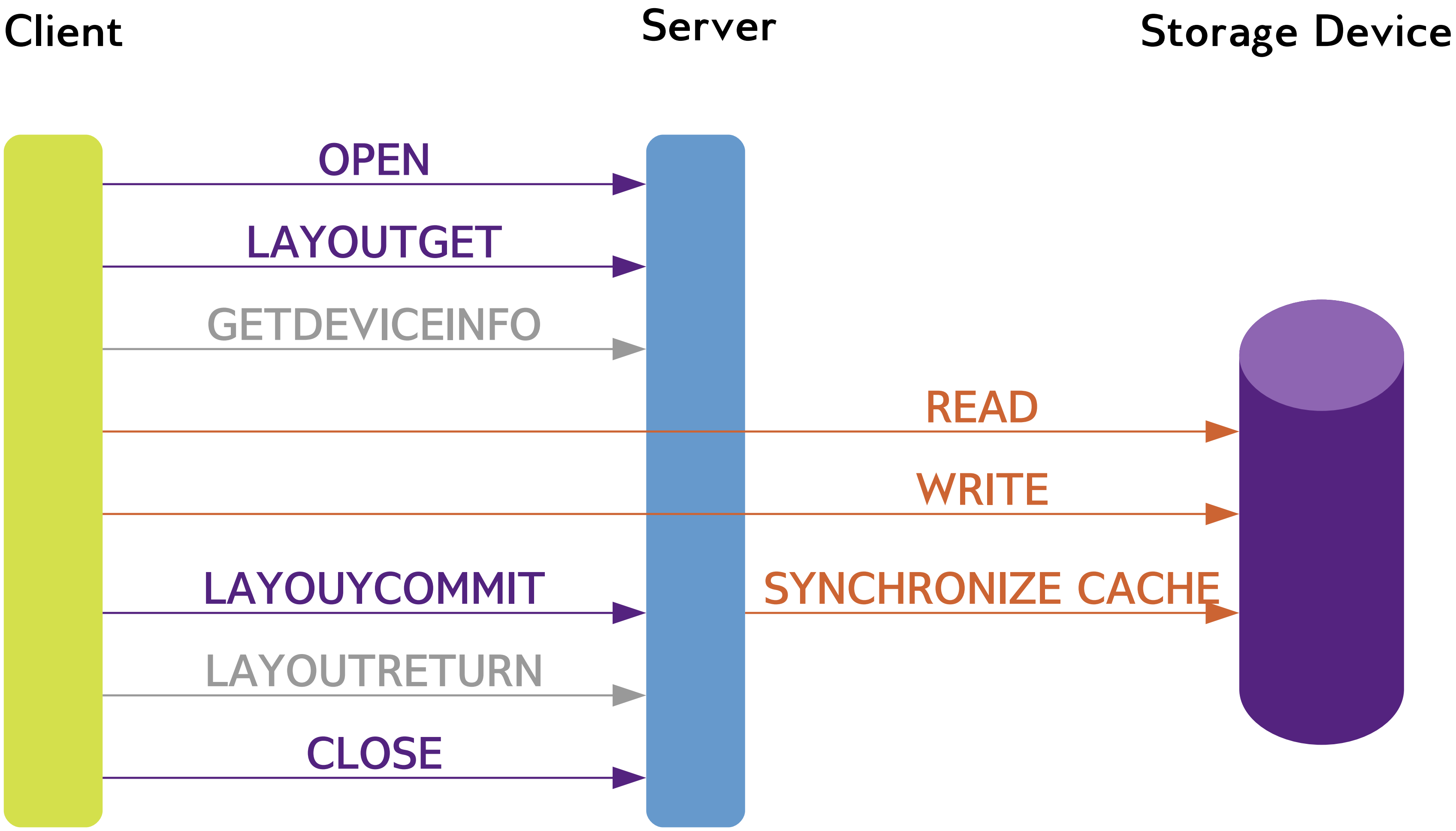
- **GETDEVICEINFO** (*device ID*)

  → retrieve mapping information for device

pNFS callback operations

- ❒ CB_LAYOUTRECALL
  - → recall a layout or all layouts from a file
- ❒ CB_RECALLABLE_OBJ_AVAIL
  - → tell client that a layout is now available
- ❒ CB_NOTIFY_DEVICEID
  - → tell client about changed device mappings

pNFS protocol workflow



pNFS block layout

- ❐ Any block protocol can be used for I/O
  - – Typically some form of SCSI
  - – The client just reads and writes to the device
- ❐ Essentially a traditional shared disk file system using NFS to manage meta data

pNFS block layout – device discovery

- Device identification by content:
  - The client scans for a UUID at a specific offset
  - Requires iterating over **all** block devices available to the client
- Generally a pretty bad idea, potential fix in: http://tools.ietf.org/html/draft-hellwig-nfsv4-scsi-layout-00

block layout: LAYOUTGET/LAYOUTCOMMIT

```
enum pnfs_block_extent_state {
        PNFS_BLOCK_READWRITE_DATA       = 0,
        PNFS_BLOCK_READ_DATA            = 1,
        PNFS_BLOCK_INVALID_DATA         = 2,
        PNFS_BLOCK_NONE_DATA            = 3,
};

struct pnfs_block_extent {
        struct nfsd4_deviceid           vol_id;
        u64                             foff;
        u64                             len;
        u64                             soff;
        enum pnfs_block_extent_state    es;
};
```

pNFS block layout – error handling

- The server needs to cut off a client that does not behave (fencing)
  - Clients can access the whole disk
- Not very well specified at the protocol level, server is expected to fence the target / switch
  - Requires an IP-based storage protocol
  - Requires NFS and storage to use the same network interface and address
- My scsi-layouts proposal proposes to use SCSI3 reservations to fix this issue

The first Linux pNFS server

- Linux pNFS support has been under development since 2006
  - But the server never made it out of out-of-tree prototype state
  - Structured to have very little common code, and hand off all pNFS work to the file system
  - Did have file, object and block layout drivers of various sorts, but none of them was very useful

The new Linux pNFS server

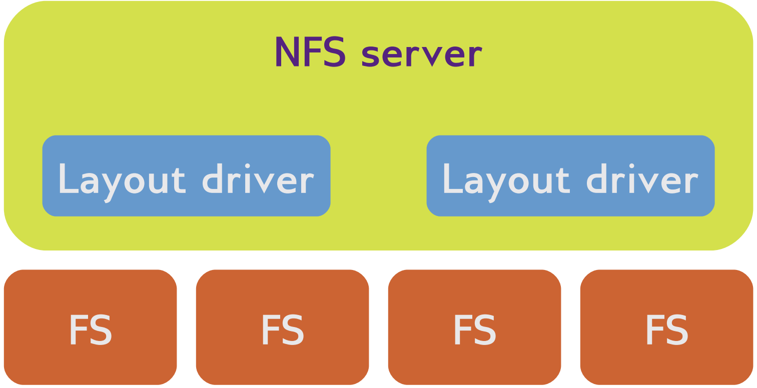
- Started out as a pNFS block layout driver to export XFS file systems
  - Turned into an entirely new server implementation
- The new server is very simple:

```
38 files changed, 1361 insertions(+), 90 deletions(-)
```

Linux pNFS server architecture

❒ Structured to:

- Keep as much common code as possible
- keep protocol specific code in the NFS server
- Do as little as possible work in the file system



Layout driver design

- In general the Linux NFS server is split into three phases:
  1. XDR decoding
  2. Processing
  3. XDR result encoding
- The pNFS server has separate methods for these phases

Layout driver: methods

```
struct nfsd4_layout_ops {
        u32             notify_types;

         __be32 (*proc_getdeviceinfo)(struct super_block *sb,
                        struct nfsd4_getdeviceinfo *gdevp);
        __be32 (*encode_getdeviceinfo)(struct xdr_stream *xdr,
                        struct nfsd4_getdeviceinfo *gdevp);

        __be32 (*proc_layoutget)(struct inode *,
                        const struct svc_fh *fhp,
                        struct nfsd4_layoutget *lgp);
        __be32 (*encode_layoutget)(struct xdr_stream *,
                        struct nfsd4_layoutget *lgp);

        __be32 (*proc_layoutcommit)(struct inode *inode,
                        struct nfsd4_layoutcommit *lcp);
};
```

Layout driver: data structures

```
struct nfs4_layout_stateid {
        struct nfs4_stid                ls_stid;
        struct list_head                ls_perclnt;
        struct list_head                ls_perfile;
        spinlock_t                      ls_lock;
        struct list_head                ls_layouts;
        u32                             ls_layout_type;
        struct file                     *ls_file;
        struct nfsd4_callback           ls_recall;
        stateid_t                       ls_recall_sid;
        bool                            ls_recalled;
};
struct nfs4_layout {
        struct list_head                lo_perstate;
        struct nfs4_layout_stateid      *lo_state;
        struct nfsd4_layout_seg         lo_seg;
};
```

pNFS server: I/O path design

1. Common code handles all requests by doing any sort of common validation and state ID processing
2. Then calls out to the layout driver where needed, passing along the whole operation state
3. Core code handles all manipulation of the in-memory layout and layout state ID data structures

pNFS server: GETDEVICEINFO

- Device IDs are a nightmare
  - Globally valid (not per fsid)
  - 128bit identifier (less than typical fsids)
  - Must never be reused

pNFS server: GETDEVICEINFO

```
struct nfsd4_deviceid {
        u64                     fsid_idx;
        u32                     generation;
        u32                     pad;
};
```

- The first time **LAYOUTGET** is called on a device we allocate a nfsd4_deviceid_map structure and an index, and hash it
  - **GETDEVICEINFO** looks it up in the hash by the index to retrieve the export pointer
  - The structure is **never** freed

pNFS server: layout recalls

- A server may recall outstanding layouts from a client:
  - Truncate
  - Conflicting access
- We only support whole file recalls
  - Allows embedding recall information in the layout state structure
- All new **LAYOUTGET**s are blocked during outstanding recalls

pNFS server: block layout driver

- The block layout driver has two parts:
  - XDR encoding / decoding
  - A small wrapper to bridge between pNFS and three new export_operations methods:

```
int (*get_uuid)(struct super_block *sb, u8 *buf, u32 *len,
                     u64 *offset);
int (*map_blocks)(struct inode *inode, loff_t offset,
                     u64 len, struct iomap *iomap,
                     bool write, u32 *device_generation);
int (*commit_blocks)(struct inode *inode, struct iomap *iomaps,
                     int nr_iomaps, struct iattr *iattr);
```

pNFS server: XFS support code

- There is very little file system support code for pNFS block layouts:
  - Implementations of the export_operations
  - Code to recall layouts that clients might have outstanding on truncate-like operations or write()
    - Calls into the NFS server by abusing file locks

block layout: LAYOUTGET/LAYOUTCOMMIT

```
enum pnfs_block_extent_state {
        PNFS_BLOCK_READWRITE_DATA       = 0,
        PNFS_BLOCK_READ_DATA            = 1,
        PNFS_BLOCK_INVALID_DATA         = 2,
        PNFS_BLOCK_NONE_DATA            = 3,
};

struct pnfs_block_extent {
        struct nfsd4_deviceid           vol_id;
        u64                             foff;
        u64                             len;
        u64                             soff;
        enum pnfs_block_extent_state    es;
};
```

XFS extent structure

```
typedef enum {
        XFS_EXT_NORM, XFS_EXT_UNWRITTEN,
        XFS_EXT_DMAPI_OFFLINE, XFS_EXT_INVALID
} xfs_exntst_t;

typedef struct xfs_bmbt_irec {
        xfs_fileoff_t   br_startoff;
        xfs_fsblock_t   br_startblock;
        xfs_filblks_t   br_blockcount;
        xfs_exntst_t    br_state;
} xfs_bmbt_irec_t;
```

XFS pNFS I/O path

- ❒ Basically an extended version of the direct I/O path.
- ❒ Steps for **LAYOUTGET**:
  1. Invalidate the page cache before handing out extents
  2. Call into the block allocator to look for the blocks
  3. If block allocation is requires allocate blocks as unwritten extents
- ❒ **LAYOUTCOMMIT** converts unwritten extents and updates the file size and time stamps

Benchmarks?

Questions?