

# spe: A first implementation

version 1.1

## Overview

This document presents the details necessary to use the first pass at the in-kernel spe.

## **/etc/policies.spe**

This flat file serves as the stable storage for the policies. It is a list of policies where a line can either be a comment (first char is a '#') or a policy. The policies are read in order and sorted by id.

Each policy is of the form:

id, stripe count, unit size, npools, attribute expression

Where the number of datasets in the npools must be at least the stripe count.

And the attribute expression is a boolean rule composed of the operators &&, ||, and !. These join expressions of the form:

ATTRIBUTE comparison-op VALUE

Where the comparison-op is one of '==' and '!='. The allowed attributes are:

- base -- base part of the file name, without the extension
- day -- day of the month in numeric form
- domain -- DNS subdomain
- ext -- file extension
- file -- base and extension
- fqdn -- fully qualified domain name
- gid -- Group ID
- hour -- Hour of the day
- host -- Short hostname
- ip -- Machine address
- path -- Path including parent directory
- subnet -- Network Address, with a netmask (/24)

- uid -- User ID
- weekday -- day of the week in short form: “mon”

And the values are strings appropriate for the attributes.

An example */etc/policies.spe* would be:

```
[root@pnfs-4-04 /etc]> more policies.spe
10, 8, 16k, swimming:diving:wading:default, path == /pnfs1/nfs41
20, 4, 1k, swimming:diving, path == /pnfs1/pnfs
30, 4, 2k, diving:swimming, path == /pnfs1/default
40, 3, 8k, wading:diving, path == /pnfs2/nfs41
50, 4, 4k, swimming:wading, path == /pnfs2/pnfs
```

## ***/etc/npools.spe***

This flat file serves as the stable storage for the npools. It is a list of npools where a line can either be a comment (first char is a '#') or a npool.

Each npool is of the form:

npool dataset name 1 .. n

Where each dataset name is in the form:

hostname:zpool/zfs filesystem

Note, while not enforced in this implementation, each dataset can only belong to one npool.

An example */etc/npool.spe* is:

```
[root@pnfs-4-04 /etc]> more npools.spe
default pnfs-4-05:pnfs1/ds1 pnfs-4-06:pnfs1/ds1 pnfs-4-05:pnfs2/ds2 pnfs-4-06:pnfs2/
ds2
swimming pnfs-4-07:pnfs1/ds1 pnfs-4-08:pnfs1/ds1
diving pnfs-4-07:pnfs2/ds2 pnfs-4-08:pnfs2/ds2
wading pnfs-4-09:pnfs2/ds2 pnfs-4-09:pnfs1/ds1
```

## ***/usr/lib/nfs/sped command***

Policies are not loaded into the kernel until the sped command is run. If it is not run, then a default policy is selected. At present, this uber-default policy is to build up a layout comprised of the set of all datasets presented to the MDS with a unit\_size of 32k.

## nfssys() Interface

NFS\_SPE is a new nfssys\_op for the nfssys() syscall. It uses a struct nfsspe\_args

```
struct nfsspe_args {
    nfsspe_op_t    nsa_opcode;    /* operation discriminator */
    uint_t        nsa_did;       /* Door id to upcall */
    char          *nsa_xdr;       /* XDR data */
    size_t        nsa_xdr_len;   /* Size of XDR data */
};
```

Where the opcodes are:

```
typedef enum nfsspe_op {
    SPE_OP_SET_DOOR,           /* Not in Use */
    SPE_OP_POLICY_POPULATE,   /* Unload policies and use new ones */
    SPE_OP_POLICY_NUKE,       /* Unload all policies */
    SPE_OP_POLICY_ADD,        /* Add a single policy */
    SPE_OP_POLICY_DELETE,     /* Delete a single policy */
    SPE_OP_SCHEDULE           /* Not in Use */
} nfsspe_op_t;
```

When sped loads the policies from */etc/policies.spe*, it converts them into XDR format for transfer to the kernel.

## Kernel Interface

When a file is opened, the corresponding layout is generated based either on the on disk layout (odl) if the file already exists or the kspe is called at creation time to determine which policy, if any, applies. If no policy applies (or if none exist), then the uber-default one applies. (Note that this fail safe will be removed once the policies and npools are stored in SMF.)

The function `mds_createfile()` is responsible for the accounting necessary to create a file. Before it returns, it is responsible for building up a `layout_core_t` structure which has the unit size, stripe count, and array of `mds_sids`. To do that, it calls:

```
static nfsstat4
mds_createfile_get_layout(struct svc_req *req, vnode_t *vp,
    struct compound_state *cs, caller_context_t *ct, mds_layout_t **plo)
```

This function is basically a wrapper function to prep and cleanup after a call to:

```
int
nfs41_spe_allocate(nfs_server_instance_t *instp, vattr_t *vap,
    struct netbuf *addr, char *dir_path,
    count4 *stripe_count, uint32_t *unit_size,
    mds_sid **mds_sids, int bServer)
```

- `instp` tells the spe code which instance of nfsd to use

- vap is a vattr and supplies the UID and GID of the creator of the file
- addr is the machine address of the client
- dir\_path is the full path to the file
- stripe\_count is a pointer to the actual stripe\_count
- unit\_size is a pointer to the actual unit size
- mds\_sids is a pointer to an array of mds\_sids
- bServer lets the spe know if it is working on behalf of the client or server code

The kspe evaluates policies in order. If it finds a matching policy, it determines the stripe\_count and unit\_size from the policy. If it is a client request, it then returns.

If it is a server request, it builds up an array of mds\_sids to return to the caller.

That array is then used in mds\_createfile\_get\_layout() to find the corresponding layout. If one is found, we bump the refcnt. If one is not found, then we create it and also at this time create a new mpd, which in turn generates a new device id to send over the wire.

Finally, the mds\_sids will be used to generate the ds\_filehandles sent to the client.

## Getting mds\_sids

The mapping from dataset name to mds\_sid is provided via the DS\_REPORTAVAIL procedure - when a DS presents a dataset with a mds\_sid, it also provides as a named attribute the key value of “dataset” and an utf8string name.

When the spe determines that a dataset is going to be added to the layout selected, it does this mapping via mds\_ds\_path\_to\_mds\_sid():

```
int
mds_ds_path_to_mds_sid(utf8string *dataset_name, mds_sid *sid)
```

This routine searches the ds\_guid\_info table for a matching entry on the dataset\_name. This search is done per policy evaluation, i.e., there is no caching.