STORAGE PERFORMANCE DEVELOPMENT KIT

US VIRTUAL FORUM '22

Contributions by NVIDIA in SPDK Project - NVMe Multipath and External Snapshots

Mike Gerdts, Shuhei Matsumoto

STORAGE PERFORMANCE DEVELOPMENT KIT

US VIRTUAL FORUM '22

SPDK NVMe Multipath and I/O Error Resiliency

Shuhei Matsumoto

Agenda

- Path Error Recovery
- I/O Retry
- I/O Timeout
- Multipath Mode
- Create Multipath
- I/O Channel for Multipath
- Path Selection Policy
- Asymmetric Namespace Access (ANA) Handling
- Usage
- Reference Settings

Path Error Recovery

- If NVMe driver detects an error on a qpair, it disconnects the qpair and notifies the error to the NVMe bdev module.
- Then the NVMe bdev module starts resetting the corresponding NVMe-oF controller.
  1. Disconnect and delete all I/O qpairs.
  2. Disconnect admin qpair.
  3. Connect admin qpair.
  4. Configure the NVMe-oF controller.
  5. Create and connect all I/O qpairs.
- If the step 3, 4, or 5 fails, the reset reverts to the step 3 and then it is retried after reconnect_delay_sec seconds.
- If the NVMe-oF controller is not recovered within ctrlr_loss_timeout_sec seconds, it is deleted automatically. If ctrlr_loss_timeout_sec is -1, it retries indefinitely.
- All disconnect/connect operations are done asynchronously now.
- By default, error detection on a qpair is very slow for TCP and RDMA transports. For fast error detection, a global option, transport_ack_timeout is very useful.

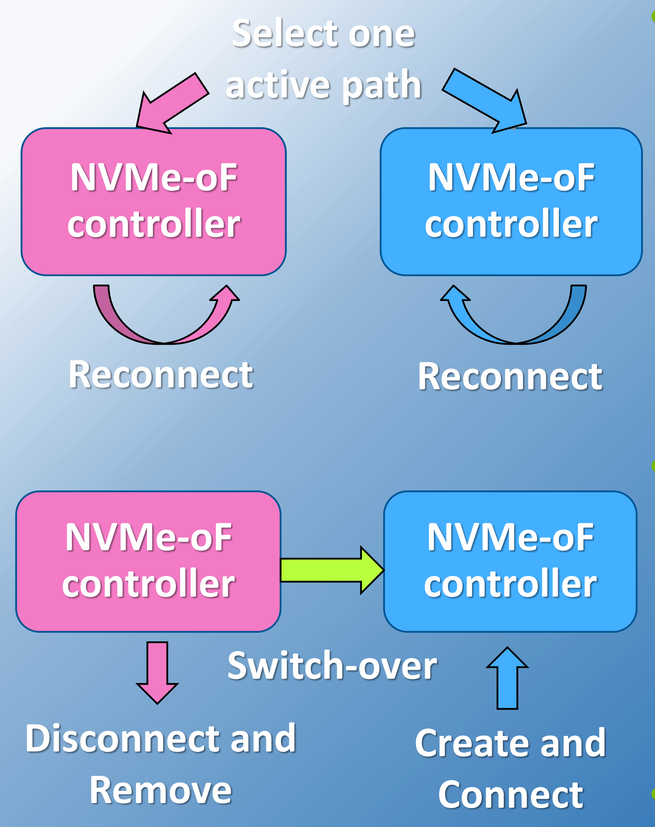
I/O Retry

- A global option, bdev_retry_count, controls the number of I/O retries for I/O errors and enables/disables the multipath failover for path errors.
- Each I/O has
  - Retry count
  - Timer to schedule a retry.
- Each I/O channel for an NVMe bdev has
  - Sorted I/O retry list
  - I/O retry poller
- If an I/O is returned with error, the following steps are executed:
  1. If the DNR bit is set or the retry count expired, complete the I/O with the returned error.
  2. If the error is a path error, insert the I/O to the I/O retry list with no delay.
  3. If the error is an I/O error, insert the I/O to the I/O retry list with the CRD value.
- When submitting an I/O, there may be no available I/O path. However, if there is any I/O path which is recovering, insert the I/O to the I/O retry list with one second delay.

I/O Timeout

- Users can choose either of the three actions, ABORT, RESET, or NONE, for I/O timeout.
- Users can set different timeout values for I/O commands and admin commands.
- If the action is ABORT, try aborting the timed out I/O. If it failed, start the NVMe-oF controller reset.
- If the action is RESET, start the NVMe-oF controller reset.
- If the action is NONE, just collect error log and do nothing.

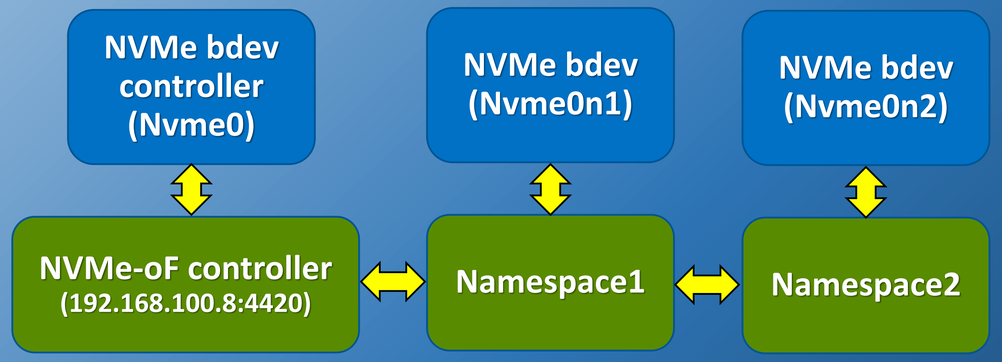
Multipath Mode



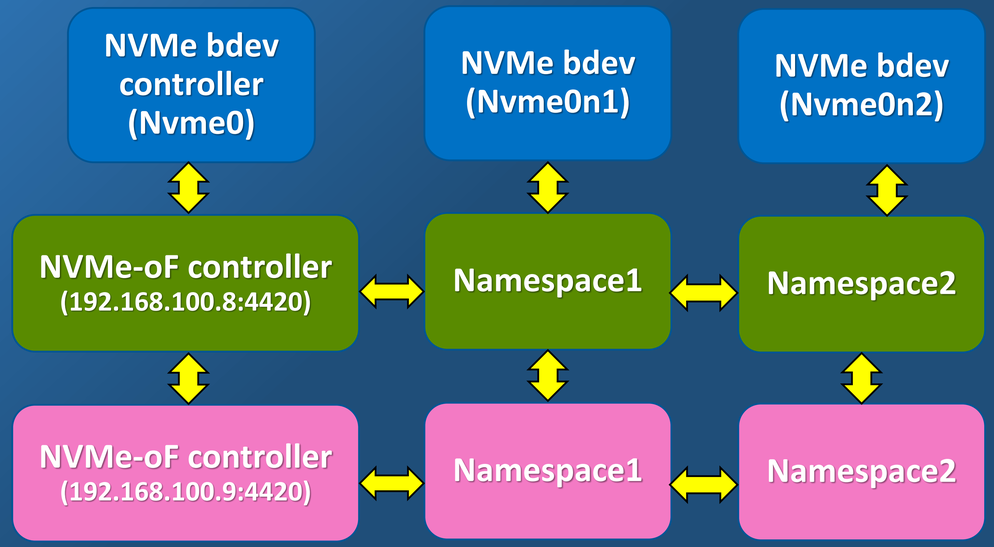
- **Multipath**
  - Multipath has been supported since SPDK v22.01. It provides high performance and the same functionalities as the Linux kernel native NVMe multipath.
  - Active connections are used based on a policy of either active-passive or active-active.
  - Asymmetric Namespace Access (ANA) is also supported and used for path selections.
- **(Cold) Failover**
  - One active connection is maintained, and an alternate path is connected during the switch-over.
  - This reduces the number of active connections at any given time, but switch-over is delayed.
- **Disable**
  - Each path is maintained independently.

Create Multipath

- To create multipath to the same NVMe-oF subsystem, call the bdev_nvme_attach_controller RPC multiple times with the same NVMe bdev controller name.
- The bdev_nvme_attach_controller RPC for each path takes the following steps:
  - Create a NVMe-oF controller.
  - Discover namespaces of the controller.
  - For each found namespace, add the namespace as an alternate path if NVMe bdev is found, or create a new NVMe bdev otherwise.



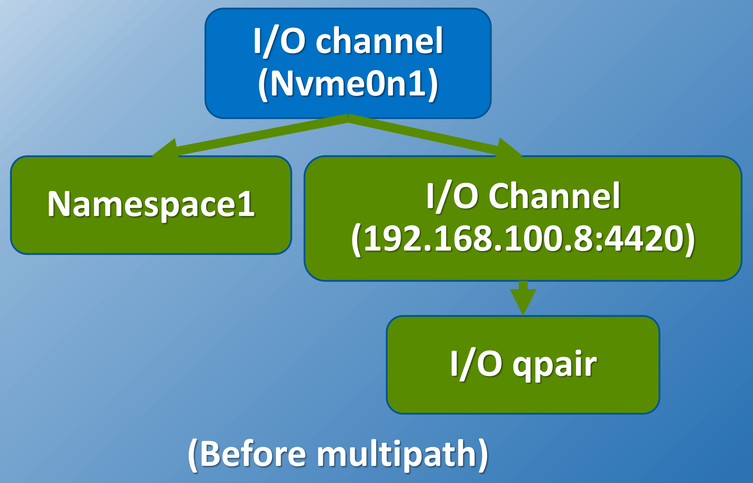
(Attaching the first path)

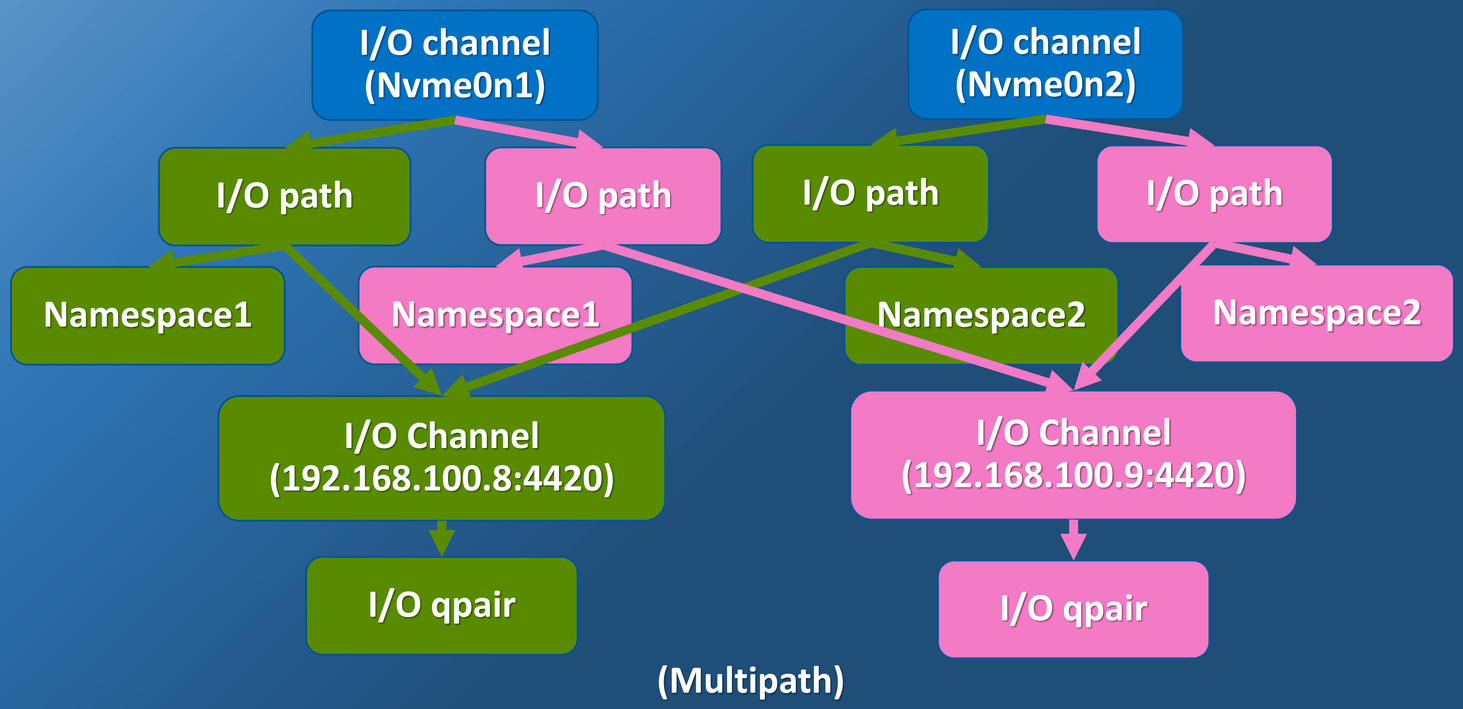


(Attaching the second path)

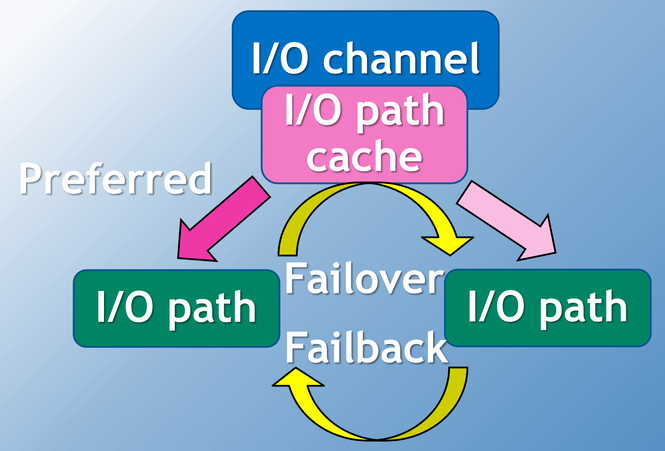
I/O Channel for Multipath

- I/O channel for an NVMe bdev is provided to the upper layer. It has an I/O path for each namespace.
- I/O path is a new additional abstraction to submit I/O to an I/O path among multiple I/O paths. I/O path consists of pointers to a namespace and an I/O channel for an NVMe-oF controller.
- I/O channel for an NVMe-oF controller has an I/O qpair as its dynamic context.

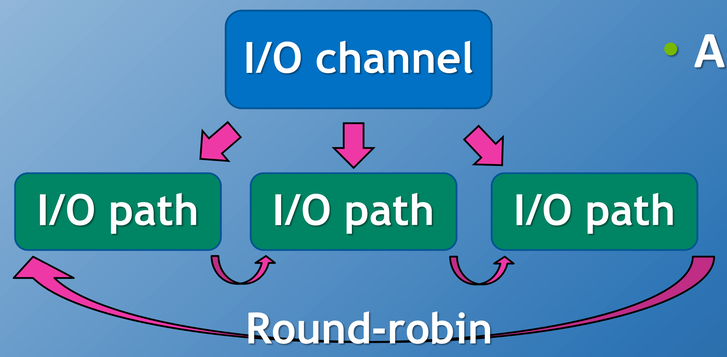




Path Selection Policy



- **Active-Passive**
  - I/O channel for an NVMe bdev has a list of I/O paths and a cache to store the optimal I/O path.
  - User can specify the preferred I/O path manually.
  - By default, if the preferred path is restored, failback to it is done automatically.
  - The automatic failback is disabled by a global option disable_auto_failback.



- **Active-Active**
  - Use the round-robin algorithm to submit an I/O to each I/O path in circular order.

Asymmetric Namespace Access (ANA) Handling

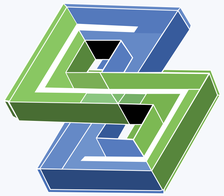
- ANA is the NVMe standard to inform an initiator of the most optimal way to access a given namespace. (ANA is synonymous with SCSI ALUA.)
- If an I/O is returned with an ANA error or an ANA change notice event is received, the ANA state may be changed. In this case, read the ANA log page to check the ANA state changes.
- Use only ANA optimal I/O paths unless no ANA optimal path is available.
- If an I/O path is in ANA transition, queue I/Os to wait until the namespace becomes accessible again. The ANA transition should end within the ANA Transition Time (ANATT) seconds. If the namespace does not become accessible within the ANATT seconds, return queued I/Os with error.

Usage

- Attach two NVMe-oF controllers and aggregate these into a single NVMe bdev controller.
  - ./scripts/rpc.py bdev_nvme_attach_controller –b Nvme0 –t rdma –a 192.168.100.8 –s 4420 –f ipv4 –n nqn.2016-06.io.spdk:cnode1 –l -1 –o 20
  - ./scripts/rpc.py bdev_nvme_attach_controller –b Nvme0 –t rdma –a 192.168.100.9 –s 4420 –f ipv4 –n nqn.2016-06.io.spdk:cnode1 –l -1 –o 20 –x multipath
- Confirm if multipath is configured correctly.
  - ./scripts/rpc.py bdev_get_bdevs –b Nvme0n1
  - ./scripts/rpc.py bdev_nvme_get_controllers –n Nvme0
- Monitor the current multipath state.
  - ./scripts/rpc.py bdev_nvme_get_io_paths –n Nvme0n1

Reference Settings for I/O Error Resiliency

#	Options	Value	Description	Note
1	bdev_retry_count	5	Controls the number of attempts when an I/O fails with the DNR (Do Not Retry) bit cleared.	bdev_retry_count is not directly used for multipath but is required to be non-zero if multipath is used.
2	transport_retry_count	7	transport_retry_count is used during connection negotiation and applied to qpairs later.	For RDMA transport, transport_retry_count=4 and transport_ack_timeout=4 is a good combination found by tuning.
3	transport_ack_timeout	0 (Driver's specific default value.)	Transport specific and to adjust the delay until the initiator detects any QP error after the connection is disconnected. 0-31. 0 does not change driver's specific default value.	For RDMA transport, transport_retry_count=4 and transport_ack_timeout=4 is a good combination found by tuning.
4	io_timeout_us	30000000 (= 30sec)	Timeout value in microseconds for the I/O commands outstanding in the NVMe driver.	
5	timeout_admin_us	60000000 (= 60sec)	Timeout value in microseconds for the admin commands outstanding in the NVMe driver.	
6	action_on_timeout	RESET	Specifies what to do for a timeout-ed I/O or admin command. none, reset, or abort.	
7	reconnect_delay_sec	10 (sec)	Time in seconds to delay a reconnect retry.	
8	ctrlr_loss_timeout_sec	-1 (infinite retries)	Time in seconds to wait until a controller is reconnected before deleting the controller.	
9	fast_io_fail_timeout_sec	0 (disabled)	Time in seconds to wait until controller is reconnected before failing I/Os to the controller.	



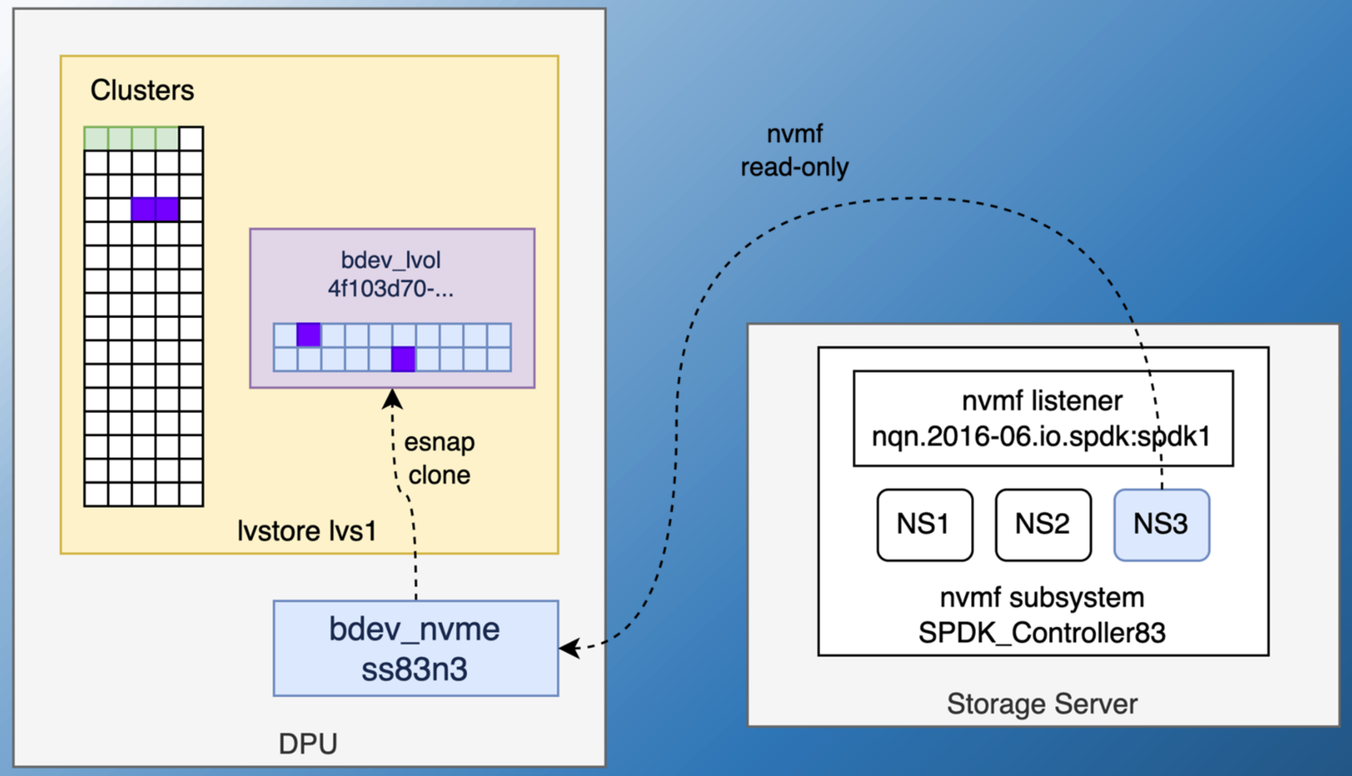
STORAGE
PERFORMANCE
DEVELOPMENT
KIT

US VIRTUAL FORUM '22

SPDK External Snapshots

Mike Gerdts

What is an External Snapshot?



Like found in Linux device mapper

Serves as the read-only base device for lvol and blob clones.

Can be any blobstore device that supports reads and will not be written to by others.

Initial lvstore implementation creates blobstore devices from any bdev.

Example use case: write splitting for virtual machine images where only differences are stored on the VM's host.

Esnap clone is the quick way to say *clone of an external snapshot*.

How does this work?

```
static void
blob_esnap_load_done(void *arg, struct spdk_bs_dev
{
        struct spdk_blob_load_ctx *ctx = arg;
        struct spdk_blob        *blob = ctx->blob;

        assert(blob->back_bs_dev == NULL);

        if (bserrno == 0) {
                assert(dev != NULL);
+----  6 lines: if (blob->bs->io_unit_size < dev->
                } else {
                        blob->back_bs_dev = dev;
                        blob->parent_id = SPDK_BLO
                }
        } else if (dev != NULL) {
                dev->destroy(dev);
        }
        blob_load_final(ctx, bserrno);
}
```

- As always, each lvol uses a blob to store its data.
- Most of the changes are in the blobstore.
- Blobs already support thin provisioning and clones
  - Unallocated clusters read from blob->back_bs_dev
- Blobstore devices (struct spdk_bs_dev)
  - bs->dev: blob_bdev wraps a bdev (rw).
  - Thin provisioning: zeroes fills buffer with 0 (ro).
  - Clone: blob_bs_dev reads from a blob (ro).
  - **esnap clone: blob_bdev can wrap bdev (ro).**
- Flexible interface
  - lvol external snapshots wrap bdevs with blob_bdev.
  - Other consumers (like unit tests) can invent their on spdk_bs_dev implementations.

IO Channels

```
struct spdk_bs_request_set {
+---  4 lines: struct spdk_bs_cpl      cpl;-------------------------------
        /*
         * The blobstore's channel, obtained by blobstore consumers
         * via spdk_bs_alloc_io_channel(). Used for IO to the
         * blobstore.
         */
        struct spdk_bs_channel          *channel;
        /*
         * The channel used by the blobstore to perform IO on
         * back_bs_dev. Unless the blob is an external clone,
         * back_channel == spdk_io_channel_get_ctx(set->channel).
         */
        struct spdk_io_channel          *back_channel;
```

```
void
bs_sequence_readv_bs_dev(spdk_bs_sequence_t *seq, struct spdk_bs_dev
                         struct iovec *iov, int iovcnt, uint64_t lba,
                         spdk_bs_sequence_cpl cb_fn, void *cb_arg)
{
        struct spdk_bs_request_set      *set = (struct spdk_bs_reques
        struct spdk_io_channel          *channel = set->back_channel;
```

IO channel spdk_bs_alloc_io_channel() returns an spdk_bs_channel with bs_ch->dev_channel for IO on the blobstore's device. bs_ch->dev_channel is not useful with external snapshots.

External snapshot channels now stored in esnap_channels tree in each spdk_bs_channel.

Per-blob external snapshot channel allocated on first read from that blob.

The bs_batch and bs_sequence read functions now use the set's back_channel to perform IO.

How to recognize an esnap clone

An **esnap clone** has:

- **Internal XATTR** BLOB_ESNAP_ID ("EXTSNAP")
  - Can be set to anything useful to the blobstore consumer
  - lvstore sets it to the UUID provided during clone creation
- **Invalid flag** SPDK_BLOB_ESNAP
  - Ensures earlier SPDK builds leave this blob alone
- **parent_id** SPDK_BLOBID_ESNAP
  - Indicates special handling during snapshot and inflate operations

Creating an esnap clone, blob edition

```
struct spdk_blob_opts {
+--- 16 lines: uint64_t  num_clusters;----------------------------------------------------
        /**
         * If set, create a clone of an external snapshot - an "esnap clone". The memory
         * referenced by esnap_id will be copied into the blob's metadata and can be
         * retrieved with spdk_blob_get_esnap_id(), typically from an esnap_bs_dev_create()
         * callback. See struct_bs_opts.
         *
         * When esnap_id is specified, num_clusters should be specified. If it is not, the
         * blob will have no capacity until spdk_blob_resize() is called.
         */
        const void *esnap_id;

        /**
         * The size of esnap_id, in bytes.
         */
        uint32_t esnap_id_len;
};
```

```
spdk_blob_opts_init(&opts, sizeof(opts));
opts.esnap_id = esnap_uuid;
opts.esnap_id_len = strlen(esnap_uuid) + 1;
opts.thin_provision = true;
opts.num_clusters = spdk_divide_round_up(sz, spdk_bs_get_cluster_size
opts.clear_method = lvol->clear_method;
opts.xattrs.count = SPDK_COUNTOF(xattr_names);
opts.xattrs.names = xattr_names;
opts.xattrs.ctx = lvol;
opts.xattrs.get_value = lvol_get_xattr_value;
```

The esnap clone can only be opened if the blobstore was opened with external snapshot support

Enabling Blobstore Support

```
struct spdk_bs_opts {
+-- 38 lines: * Size of cluster in bytes. Must be multiple of 4KiB page size. --
        /**
         * External snapshot creation callback to register with the blobstore.
         */
        spdk_bs_esnap_dev_create esnap_bs_dev_create;

        /**
         * Blobstore context to pass with esnap_bs_dev_create.
         */
        void *esnap_ctx;
} __attribute__((packed));
```

```
static void
blob_load_backing_dev(spdk_bs_sequence_t *seq, void *cb_arg)
{
+--  6 lines: struct spdk_blob_load_ctx *ctx = cb_arg;---------------------------------
        if (blob_is_esnap_clone(blob)) {
+---  7 lines: if (blob->bs->esnap_bs_dev_create == NULL) {----------------------------

                SPDK_INFOLOG(blob, "Creating external snapshot device\n");
                blob->bs->esnap_bs_dev_create(blob->bs->esnap_ctx,
                                              seq->cpl.u.blob_handle.esnap_ctx, blob,
                                              blob_esnap_load_done, ctx);
                return;
        }
```

```
static void
lvs_bs_opts_init(struct spdk_bs_opts *opts)
{
        spdk_bs_opts_init(opts, sizeof(*opts));
        opts->max_channel_ops = SPDK_LVOL_BLOB_OPTS_CHANNEL_OPS;
        opts->external_bs_dev_create = lvs_esnap_dev_create;
}
```

```
static void
lvs_esnap_dev_create(void *bs_ctx, void *blob_ctx, struct spdk_blob *blob,
                     spdk_blob_op_with_bs_dev cb, void *cb_arg)
{
        struct spdk_bs_dev      *bs_dev = NULL;
        const void              *esnap_id = NULL;
        const char              *name;
        size_t                  esnap_id_len = 0;
        int                     rc;

        rc = spdk_blob_get_esnap_id(blob, &esnap_id, &esnap_id_len);
+--  5 lines: if (rc != 0) {-----------------------------------------------------------
        name = esnap_id;
+--  6 lines: if (strnlen(name, esnap_id_len) + 1 != esnap_id_len) {-------------------

        rc = spdk_bdev_create_bs_dev_ro(name, lvs_esnap_bdev_event_cb, NULL, &bs_dev);
+--  6 lines: if (rc != 0) {-----------------------------------------------------------
        cb(cb_arg, bs_dev, rc);
}
```

Logical Volume esnap Clones

```
$ ./rpc.py bdev_malloc_create 10240 512
Malloc0

$ uuid=4b936d3a-7b0e-4f96-aa46-d2828dffb3a2

$ ./rpc.py bdev_malloc_create -u $uuid 1024 512
Malloc1

$ ./rpc.py bdev_lvol_clone_bdev lvs1 $uuid lvol1
360a67dd-8a02-4860-9879-65d69c3b96a5

$ ./rpc.py bdev_get_bdevs -b lvs1/lvol1 | jq '.[-1].driver_specific'
{
  "lvol": {
    "lvol_store_uuid": "b07423c9-7d15-4471-89e6-b2657707e57b",
    "base_bdev": "Malloc0"
    "thin_provision": true,
    "snapshot": false,
    "clone": true,
    "esnap_clone": true,
    "esnap_clone_id": "4b936d3a-7b0e-4f96-aa46-d2828dffb3a2"
  }
}
```

- This is a very thin veneer on top of the work done in the blobstore.
- Added
  - spdk_lvol_create_bdev_clone()
  - rpc_bdev_lvol_clone_bdev()
- Updated
  - vbdev_lvol_dump_info_json()
  - rpc_bdev_lvol_clone()

Shared Claims

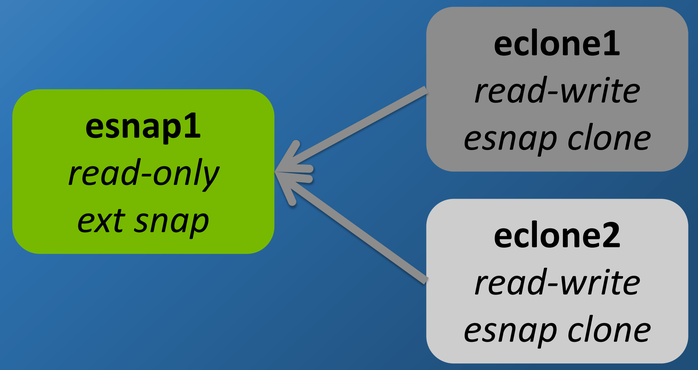
```
/** Claim types */
enum spdk_bdev_module_claim_type {
        /* Not claimed. Must not be used to request a claim. */
        SPDK_BDEV_MOD_CLAIM_NONE = 0,

        /**
         * Exclusive writer, with allowances for legacy behavior. This matches the behavior of
         * `spdk_bdev_module_claim_bdev()` as of SPDK 22.09. This claim type is deprecated.
         */
        SPDK_BDEV_MOD_CLAIM_EXCL_WRITE,

        /**
         * The descriptor passed with this claim request is the only writer. Other claimless
         * readers are allowed.
         */
        SPDK_BDEV_MOD_CLAIM_READ_WRITE_ONCE,

        /**
         * Any number of readers, no writers. Readers without a claim are allowed.
         */
        SPDK_BDEV_MOD_CLAIM_READ_ONLY_MANY,

        /**
         * Any number of writers with matching shared_claim_key. After the first writer
         * establishes a claim, future aspiring writers should open read-only and pass the
         * read-only descriptor.  If the shared claim is granted to the aspiring writer, the
         * descriptor will be upgraded to read-write.
         */
        SPDK_BDEV_MOD_CLAIM_READ_WRITE_MANY
};
```



For an external snapshot to act like a snapshot, it needs to be read-only.

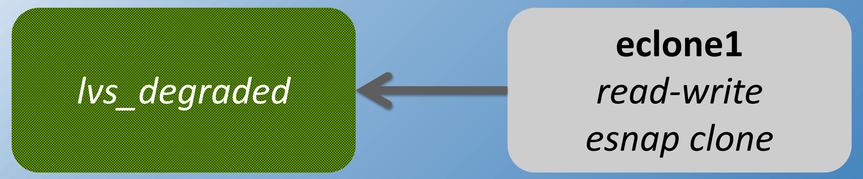
There may be many esnap clones of the same external snapshot.

All v2 claims associated with a bdev descriptor: claim released on close.

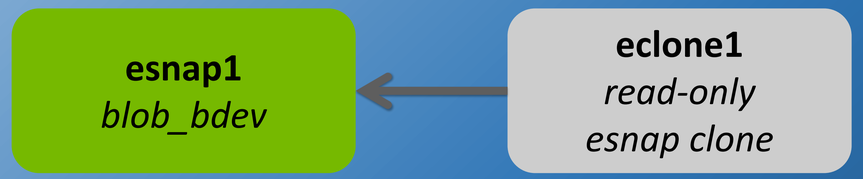
Exclusive write (v1) claim will likely be deprecated.

Missing External Snapshots

Lvstore loaded with missing esnap



After examine_config() called with missing bdev



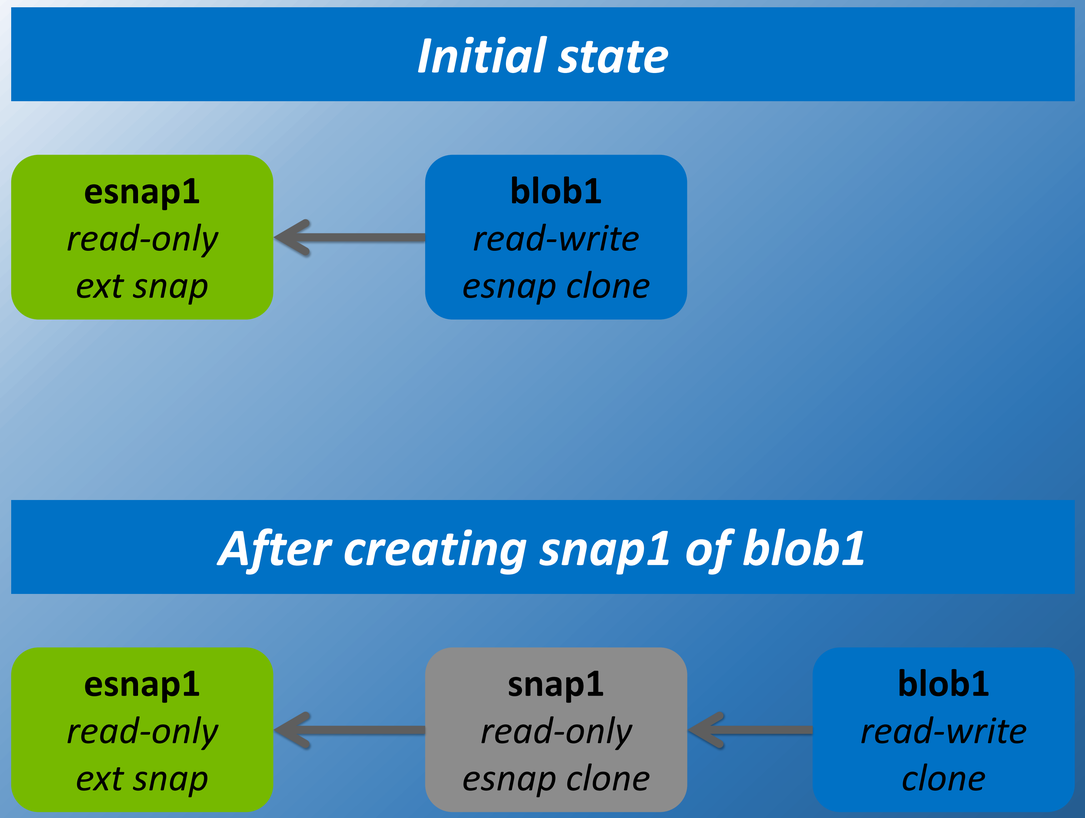
The bdev used as an external snapshot may be registered after the lvstore.

The lvol is still opened, but can only service IO involving blocks backed by allocated blobstore clusters

Missing lvols stored in per-lvstore missing bdevs tree, consulted by vbdev_lvol's examine_config() callback.

Blobstore gains the ability to hotplug blob->back_bs_dev to switch from a back_bs_dev that returns EIO for all reads to a proper blob_bdev.

Snapshotting an esnap clone



Newly created snapshot becomes the parent of the blob being snapshotted

Snapshotting an esnap clone:

- Turns the new snapshot into the (read-only) clone of the external snapshot
- The blob that was an esnap clone becomes a clone of the new snapshot.

This means invalid_flag, internal XATTR, and parent_id move to new snapshot.

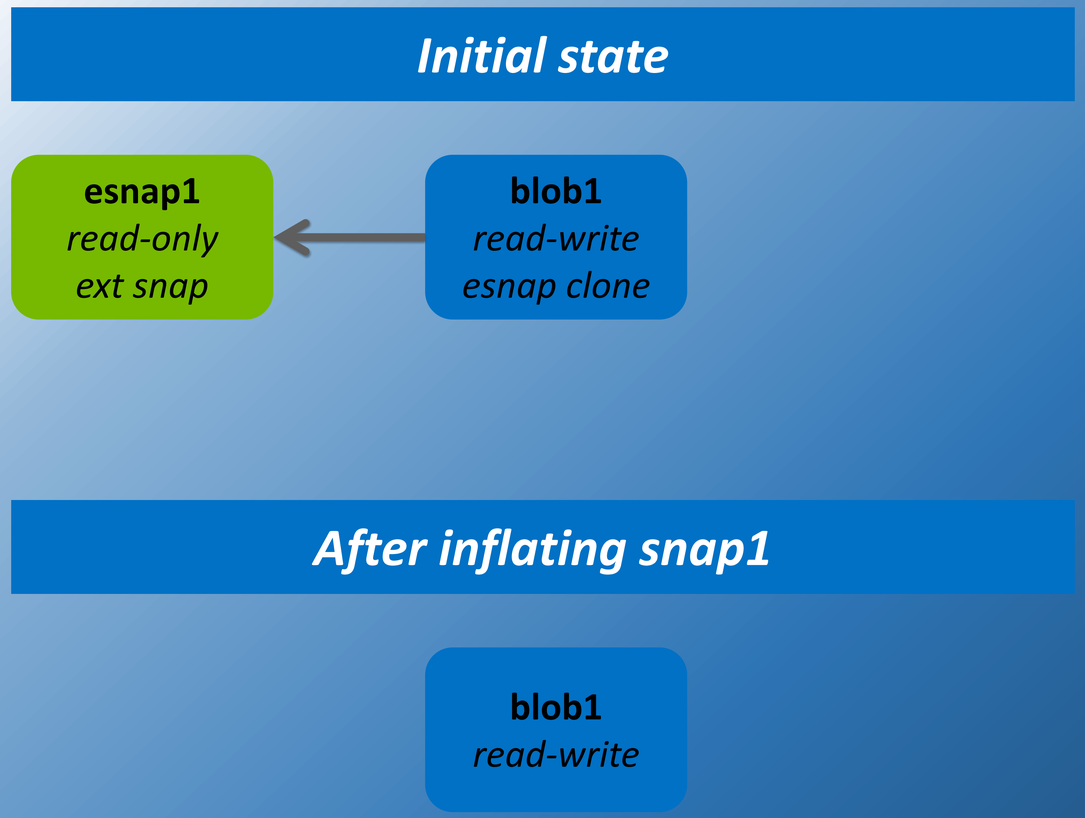
Inflating or decoupling an esnap clone

	Inflate	Decouple
Clean clusters copied from parent snapshot	Yes	Yes
Allocate space for "zero" clusters	Yes	No*
Retains dependency on back_bs_dev snapshot	No	No

* Depends on behavior of back_bs_dev->is_zeroes()

- **Inflate:** remove dependency on parent snapshot and zeroes dev.
- **Decouple:** remove dependency on parent snapshot.
- Other similar systems refer to one or both of these as hydration.

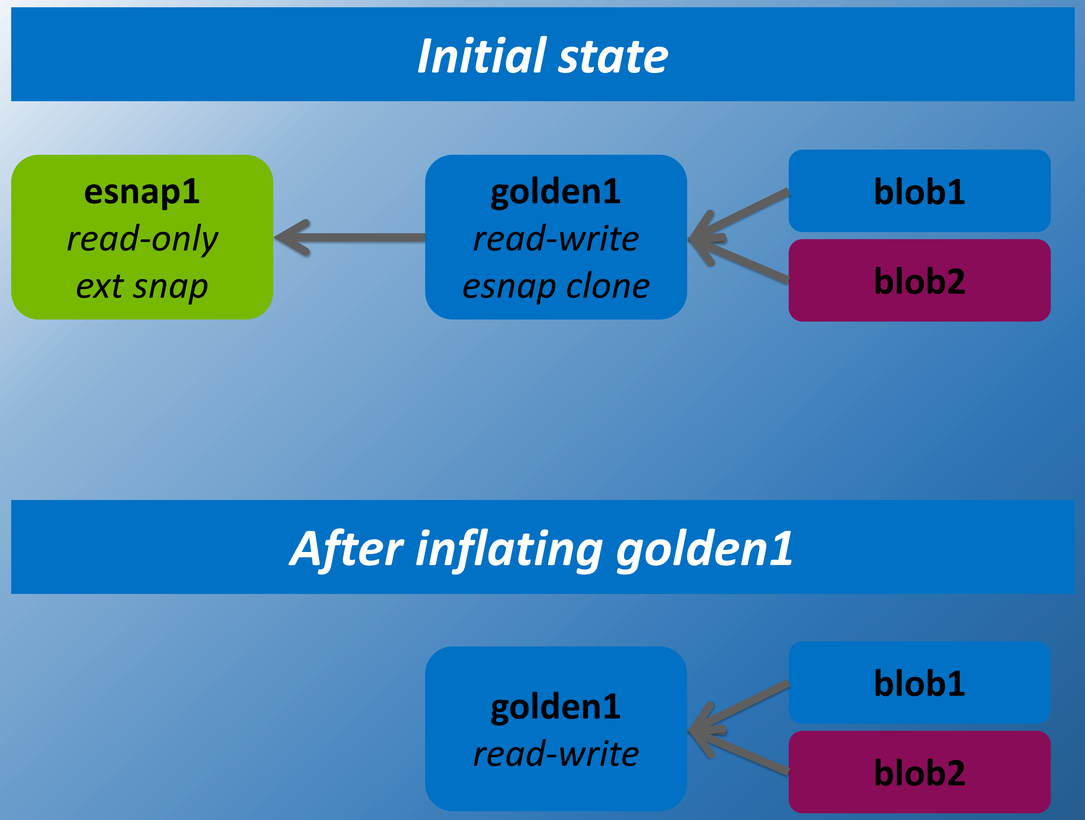
Inflating esnap Clones



An esnap clone can be inflated or decoupled to remove its dependence on its external snapshot.

This can be done while the clone is being used with no more impact than a write to each unallocated cluster.

Inflating esnap Clones



A snapshot that is itself an esnap clone can be inflated.

The snapshot, all of its current clones, and all of its future clones are now fully independent of the external snapshot.

This allows you to get a local copy of all the data needed by blob1 and blob2 without unnecessary duplication.

Thank you!