Userland TCP Transport for NVMe

John Baldwin

BSDCan

1 June 2024

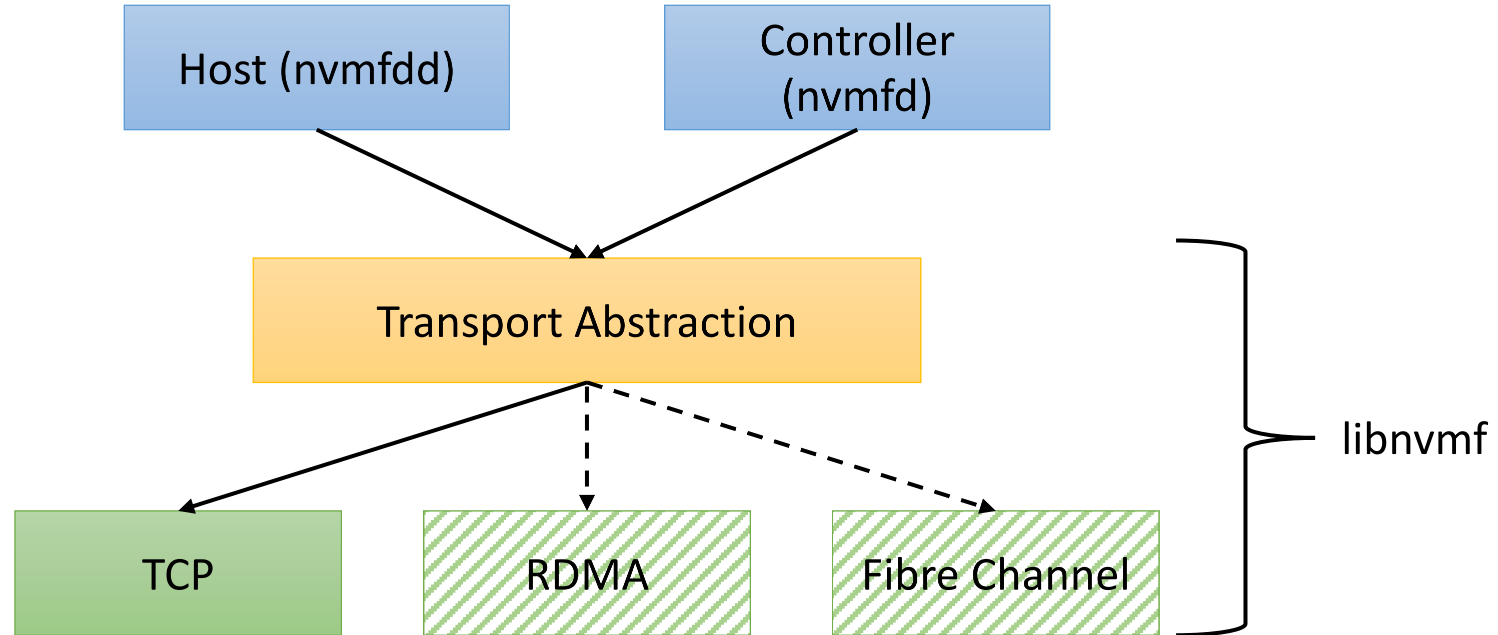
Overview

- Brief Introduction to NVMe over Fabrics
- libnvmf public API
- Host (nvmfdd)
- Controller (nvmfd)
- libnvmf internal API
- TCP transport

NVMe over Fabrics Overview

- NVMe is a protocol similar to SCSI/ATA for storage devices
- NVMe commands and completions are encapsulated in Command Capsules and Response Capsules
- Capsules are sent and received over a transport queue pair
- Commands may have associated data
- For more details, see my talk at EuroBSDCon 2023
  - https://papers.freebsd.org/2023/eurobsdcon/baldwin-implementing_nvme_over_fabrics_in_freebsd/

Three Layers in Userspace



libnvmf

- Built as an internal lib in FreeBSD’s base system
  - Base system components can use it, but not ports
  - LIBADD=nvmf
- Headers
  - <dev/nvme/nvme.h>: Base NVMe structures and constants
  - <dev/nvmf/nvmf_proto.h>: NVMe over Fabrics protocol definitions
  - <libnvmf.h>: libnvmf public API
- Prioritizes simplicity and correctness over performance
  - Uses blocking I/O
  - Not thread-safe

libnvmf: Associations

- An association represents a logical connection between a remote host and controller
  - Includes all queue pairs, can span multiple transport connections
- nvmf_allocate_association() creates a struct nvmf_association with a single reference
- nvmf_free_association() releases reference from creation

libnvmf: Queue Pairs

- SQ/CQ pairs (struct nvmf_qpair) are allocated from an association by nvmf_allocate_qpair()
  - Wrappers around nvmf_allocate_qpair() are used in practice
- Active queue pairs hold a reference on the parent association
- Reference from creation dropped by nvmf_free_qpair()

libnvmf: Capsules

- struct nvmf_capsule represents Command and Response capsules
- Capsules are allocated from a queue pair by nvmf_allocate_command() and nvmf_allocate_response()
- Active capsules hold a reference on the queue pair
- Freed by nvmf_free_capsule()

libnvmf: Command Data

- For a host, a data buffer can be attached to a Command Capsule by calling nvmf_capsule_append_data()
- The contents of the data buffer can be sent along with the Command, or the remote controller can write data into the data buffer while completing the command
- For a controller, nvmf_receive_controller_data() is used to copy data from a received Command Capsule’s data into a local buffer, or nvmf_send_controller_data() is used to copy data from a local buffer into a portion of the remote host’s data buffer

libnvmf: Host Overview

- Each association creates a new struct nvmf_association object
- Queue pairs are created via nvmf_connect() which also sends the CONNECT command and receives its response
- Command Capsules are sent via nvmf_host_transmit_command(), and nvmf_host_wait_for_response() waits for a matching reply
- Admin queue pair must retrieve CDATA and call nvmf_update_association() before creating I/O queue pairs
- Various higher level wrappers provided:
  - nvmf_read_property() and nvmf_write_property()
  - nvmf_host_identify_controller() and nvmf_host_identify_namespace()

nvmfdd: Simple Userspace Host

- Source in tools/tools/nvmf/nvmfdd/nvmfdd.c
- Can read one or more blocks from a single namespace and write the contents to stdout, or read one or more blocks from stdin and write the contents to a contiguous range of LBAs on a single namespace
- Exercises the host APIs in libnvmf including various wrapper routines and direct operations using capsules

nvmfdd: Connecting Admin Queue

```
connect_admin_queue(…)
      struct nvmf_qpair *qp;
      uint64_t cap;
      int error;

      qp = nvmf_connect(na, params, 0, NVMF_MIN_ADMIN_MAX_SQ_SIZE,
          hostid, cntlid, subnqn, hostnqn, 0);
      if (qp == NULL)
            return (NULL);

      error = nvmf_read_property(qp, NVMF_PROP_CAP, 8, &cap);
      if (error != 0)
            errc(1, error, "Failed to fetch CAP");
```

Creates admin queue pair,
Handles CONNECT command and response

Reads CAP “register”

nvmfdd: Connecting Admin Queue

```
connect_admin_queue(…)
      struct nvme_controller_data cdata;
      …

      /* Fetch controller data. */
      error = nvmf_host_identify_controller(qp, &cdata);
      if (error != 0)
            errc(1, error, "F

      nvmf_update_assocation(na, &cdata);
      …
      return (qp);
```

Fetches controller data via IDENTIFY

Required before creating I/O queues

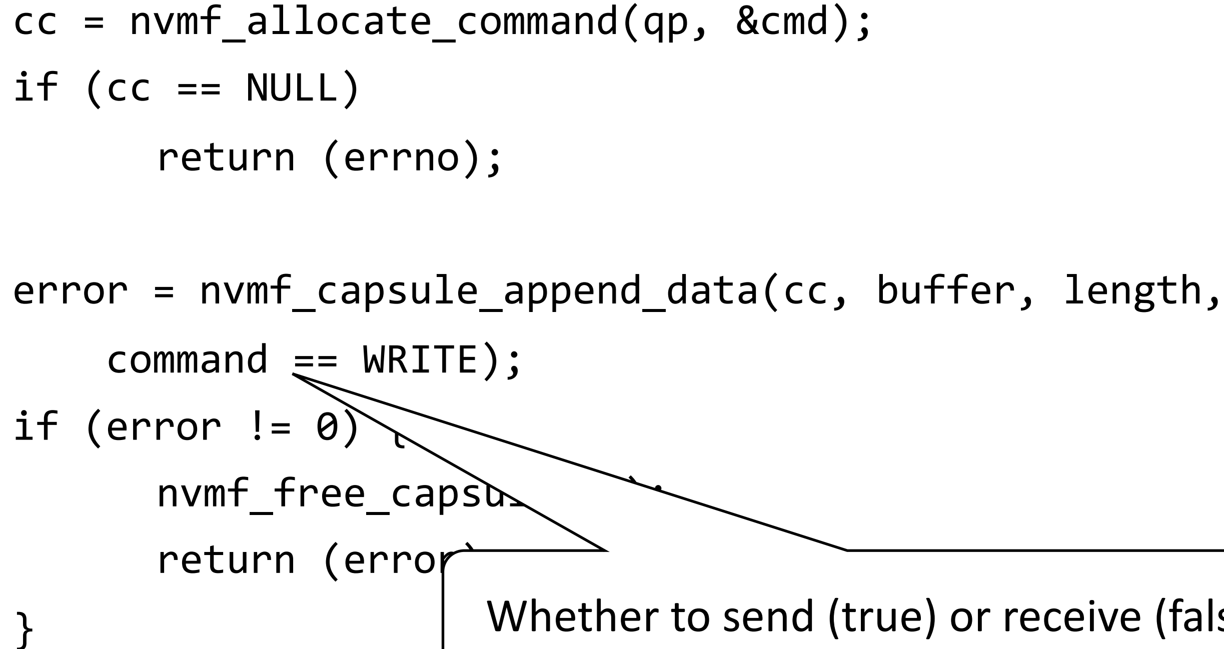
nvmfdd: Completing an I/O Command

```
static int
nvmf_io_command(struct nvmf_qpair *qp, u_int nsid, enum rw command,
    uint64_t slba, uint16_t nlb, void *buffer, size_t length)
{
	struct nvme_command cmd;
	const struct nvme_completion *cqe;
	struct nvmf_capsule *cc, *rc;
	int error;
	uint16_t status;
```

nvmfdd: Completing an I/O Command

```
memset(&cmd, 0, sizeof(cmd));
cmd.opc = command == WRITE ? NVME_OPC_WRITE : NVME_OPC_READ;
cmd.nsid = htole32(nsid);
cmd.cdw10 = htole32(slba);
cmd.cdw11 = htole32(slba >> 32);
cmd.cdw12 = htole[illegible]lb - 1);
/* Sequential Reques[illegible]dw13? */
```

nvmfdd: Completing an I/O Command



nvmfdd: Completing an I/O Command

```
error = nvmf_host_transmit_command(cc);
if (error != 0) {
      nvmf_free_capsule(cc);
      return (error);
}

error = nvmf_host_wait_for_response(cc, &rc);
nvmf_free_capsule(cc);
if (error != 0)
      return (error);
```

Data transfer has completed when this returns

`rc` allocated internally

nvmfdd: Completing an I/O Command

```
	cqe = nvmf_capsule_cqe(rc);
	status = le16toh(cqe->status);
	if (status != 0) {
		printf("NVMF: %s failed, status %#x\n", command == WRITE ?
		    "WRITE" : "READ", status);
		nvmf_free_capsule(rc);
		return (EIO);
	}

	nvmf_free_capsule(rc);
	return (0);
}
```

libnvmf: Controller Overview

- All associations for a controller share a single struct nvmf_association object
  - One object for all I/O controllers, separate object for Discovery controller
- Queue pairs are created via nvmf_accept() which receives the CONNECT command and does initial validation, but does *not* send a reply
  - Caller must send a reply after performing additional validation
  - Error reply sent via nvmf_connect_invalid_parameters()
  - Successful reply sent via nvmf_finish_accept()
- Command Capsules are received via nvmf_controller_receive_capsule()
- Response Capsules can be sent via nvmf_controller_transmit_capsule()
  - In practice, nvmf_send_response() and wrappers like nvmf_send_success() used instead
- Not required to send a reply before waiting for another command

nvmfd: Simple(-ish) Userspace Controller

- Source in usr.sbin/nvmfd/*.c
- Implements Discovery Controller in userspace always
- I/O Controller can be in kernel or userspace, but initial connection always in userspace
- Userspace I/O controller can use a file, disk device, or memory disk as backing store for each namespace

nvmfd: Accepting a Queue Pair

Each QP runs in a dedicated thread

CONNECT command capsule

CONNECT command data

```
io_socket_thread(…)
	struct nvmf_fabric_connect_data data;
	…
	qp = nvmf_accept(io_na, &qparams, &nc, &data);
	…
	if (strcmp(data.subnqn, nqn) != 0) {
		warn("I/O qpair with invalid SubNQN: %.*s",
		    (int)sizeof(data.subnqn), data.subnqn);
		nvmf_connect_invalid_parameters(nc, true,
		    offsetof(struct nvmf_fabric_connect_data, subnqn));
		goto error;
	}
```

nvmfd: Accepting a Queue Pair

```
io_socket_thread(…)
        …
        /* Is this an admin or I/O queue pair? */
        cmd = nvmf_capsule_sqe(nc);
        if (cmd->qid == 0)
                connect_admin_qpair(s, qp, nc, &data);
        else
                connect_io_qpair(s, qp, nc, &data, le16toh(cmd->qid));
```

nvmfd: Accepting an I/O Queue Pair

```
connect_io_qpair(…)
        …
        error = nvmf_finish_accept(nc, io_controller->cntlid);
        if (error != 0) {
                pthread_mutex_unlock(&io_na_mutex);
                warnc(error, "Failed to send CONNECT response");
                goto error;
        }
        …
        nvmf_free_capsule(nc);

        handle_io_qpair(ioc, qp, qid);
        return;
```

nvmfd: Handling I/O Commands

```
handle_io_commands(…)
        while (!disconnect) {
                error = nvmf_controller_receive_capsule(qp, &nc);
                if (error != 0) {
                        if (error != ECONNRESET)
                                warnc(error, "Failed to
                        break;
                }

                cmd = nvmf_capsule_sqe(nc);

                switch (cmd->opc) {
                …
                }
                nvmf_free_capsule(nc);
        }
```

Read a Command Capsule

Handle Command and Send Response

nvmfd: Handling I/O Commands

Constructs a SCT_GENERIC Completion in a Response Capsule and Sends It

```
handle_io_commands(…)
                  switch (cmd->opc) {
                  case NVME_OPC_FLUSH:
                           if (cmd->nsid == htole32(0xff    fff))
                                    nvmf_send_generic_error(nc,
                                        NVME_SC_INVALID_NAMESPACE_OR_FORMAT);
                                    break;
                           }
                           handle_flush(nc, cmd);
                           break;
                  case NVME_OPC_WRITE:
                           handle_write(ioc, nc, cmd);
                           break;
                  case NVME_OPC_READ:
                           handle_read(ioc, nc, cmd);
                           break;
```

nvmfd: Handling WRITE Command

```
static void
handle_write(struct io_controller *ioc, const struct nvmf_capsule *nc,
    const struct nvme_command *cmd)
{
        size_t len;

        len = nvmf_capsule_data_len(nc);
        device_write(le32toh(cmd->nsid), cmd_lba(cmd), cmd_nlb(cmd), nc);
        hip_add(ioc->hip.host_write_commands, 1);

        len /= 512;
        len += ioc->partial_duw;
        if (len > 1000)
                hip_add(ioc->hip.data_units_written, len / 1000);
        ioc->partial_duw = len % 1000;
}
```

Performs the actual write

Statistics in Health Information log page

nvmfd: Handling WRITE Command

```
static void
device_write(uint32_t nsid, uint64_t lba, u_int nlb,
    const struct nvmf_capsule *nc)
{
        …
        if (dev->type == RAMDISK) {
                p = NULL;
                dst = (char *)dev->mem + off;
        } else {
                p = malloc(len);
                dst = p;
        }
```

nvmfd: Handling WRITE Command

```
error = nvmf_receive_controller_data(nc, 0, dst, len);
if (error != 0) {
       nvmf_send_generic_error(nc,
           NVME_SC_TRANSIENT_TRANSPOR   RROR);
       free(p);
       return;
}
```

nvmfd: Handling WRITE Command

```
        if (dev->type != RAMDISK) {
                if (!write_buffer(dev->fd, p, len, off)) {
                        free(p);
                        nvmf_send_generic_error(nc,
                            NVME_SC_INTERNAL_DEVICE_ERROR);
                        return;
                }
        }
        free(p);
        nvmf_send_success(nc);
}
```

Non-Ramdisk data written to backing store

Sends SC_SUCCESS completion

libnvmf Internals

- Capsules (struct nvmf_capsule) and queue pairs (struct nvmf_qpair) are abstract base classes
- Transport backends provide concrete implementations (struct nvmf_tcp_capsule, struct nvmf_tcp_qpair)
- libnvmf defines an internal set of virtual functions implemented by each transport (struct nvmf_transport_ops)

Transport Virtual Functions

```
struct nvmf_transport_ops {
	/* Association management. */
	struct nvmf_association *(*allocate_association)(bool controller,
	    const struct nvmf_association_params *params);
	void (*update_association)(struct nvmf_association *na,
	    const struct nvme_controller_data *cdata);
	void (*free_association)(struct nvmf_association *na);

	/* Queue pair management. */
	struct nvmf_qpair *(*allocate_qpair)(struct nvmf_association *na,
	    const struct nvmf_qpair_params *params);
	void (*free_qpair)(struct nvmf_qpair *qp);
```

Transport Virtual Functions

```
/* Create params for kernel handoff. */
int (*kernel_handoff_params)(struct nvmf_qpair *qp,
    struct nvmf_handoff_qpair_params *qparams);

/* Capsule operations. */
struct nvmf_capsule *(*allocate_capsule)(struct nvmf_qpair *qp);
void (*free_capsule)(struct nvmf_capsule *nc);
int (*transmit_capsule)(struct nvmf_capsule *nc);
int (*receive_capsule)(struct nvmf_qpair *qp,
    struct nvmf_capsule **ncp);
uint8_t (*validate_command_capsule)(const struct nvmf_capsule *nc);
```

Transport Virtual Functions

```
        /* Transferring controller data. */
        size_t (*capsule_data_len)(const struct nvmf_capsule *nc);
        int (*receive_controller_data)(const struct nvmf_capsule *nc,
            uint32_t data_offset, void *buf, size_t len);
        int (*send_controller_data)(const struct nvmf_capsule *nc,
            const void *buf, size_t len);
};
```

Transport Virtual Functions

```
struct nvmf_capsule *
nvmf_allocate_command(struct nvmf_qpair *qp, const void *sqe)
{
          struct nvmf_capsule *nc;

          nc = qp->nq_association->na_ops->allocate_capsule(qp);
          if (nc == NULL)
                    return (NULL);

          nc->nc_qpair = qp;
          nc->nc_qe_len = sizeof(struct nvme_command);
          memcpy(&nc->nc_sqe, sqe, nc->nc_qe_len);

          /* 4.2 of NVMe base spec: Fabrics always uses SGL. */
          nc->nc_sqe.fuse &= ~NVMEM(NVME_CMD_PSDT);
          nc->nc_sqe.fuse |= NVMEF(NVME_CMD_PSDT, NVME_PSDT_SGL);
          return (nc);
}
```

ops pointer stored in nvmf_association

Capsules embed SQE/CQE

TCP transport

- Uses base struct nvmf_association as-is
- Extends struct nvmf_capsule and struct nvmf_qpair
- Internal struct nvmf_tcp_command_buffer handles the data buffer associated with a command when not using In-Capsule Data
- Data transfers handled “internally”
  - For example, tcp_receive_controller_data() sends R2Ts to remote host, waits for H2C_DATA PDUs to arrive

TCP: Allocating a Queue Pair

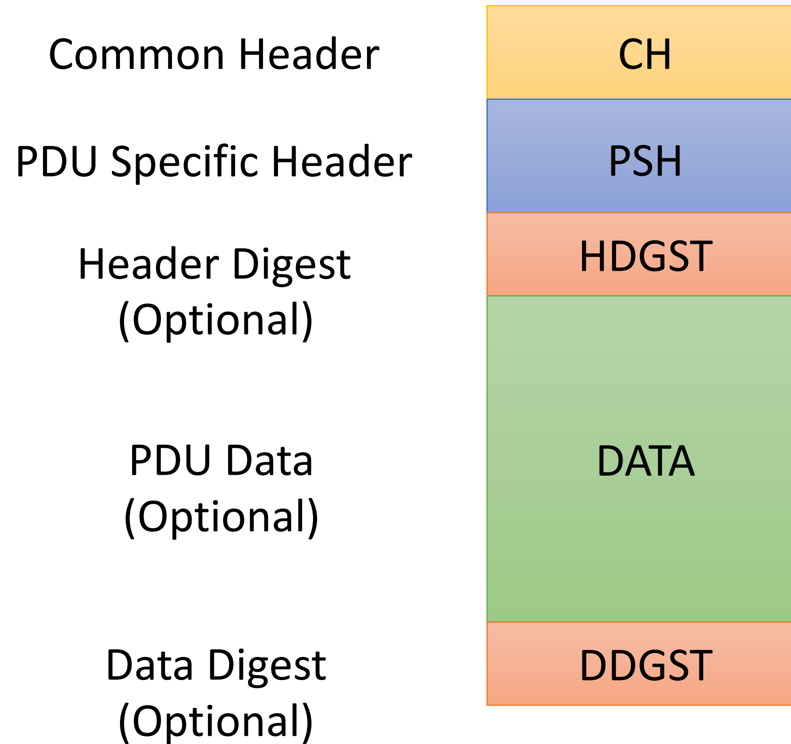
```
tcp_allocate_qpair(…)
{
        struct nvmf_tcp_qpair *qp;
        …

        qp = calloc(1, sizeof(*qp));
        qp->s = qparams->tcp.fd;
        …
        if (na->na_controller)
                error = tcp_accept(qp, na);
        else
                error = tcp_connect(qp, na, qparams->admin);
        …

        return (&qp->qp);
}
```

Performs TCP handshake (ICReq / ICResp PDUs)

NVME/TCP PDUs



TCP: Constructing a PDU

```
/*
 * Construct and send a PDU that contains an optional data payload.
 * This includes dealing with digests and the length fields in the
 * common header.
 */
static int
nvmf_tcp_construct_pdu(struct nvmf_tcp_qpair *qp, void *hdr, size_t hlen,
    void *data, uint32_t data_len)
{
	struct nvme_tcp_common_pdu_hdr *ch;
	struct iovec iov[5];
	u_int iovcnt;
	uint32_t header_digest, data_digest, pad, pdo, plen;
```

TCP: Constructing a PDU

```
plen = hlen;
if (qp->header_digests)
        plen += sizeof(header_digest);
if (data_len != 0) {
        pdo = roundup2(plen, qp->txpda);
        pad = pdo - plen;
        plen = pdo + data_len;
        if (qp->data_digests)
                plen += sizeof(data_digest);
} else {
        assert(data == NULL);
        pdo = 0;
        pad = 0;
}
```

TCP: Constructing a PDU

```
ch = hdr;
ch->hlen = hlen;
if (qp->header_digests)
	ch->flags |= NVME_TCP_CH_FLAGS_HDGSTF;
if (qp->data_digests && data_len != 0)
	ch->flags |= NVME_TCP_CH_FLAGS_DDGSTF;
ch->pdo = pdo;
ch->plen = htole32(plen);

/* CH + PSH */
iov[0].iov_base = hdr;
iov[0].iov_len = hlen;
iovcnt = 1;
```

TCP: Constructing a PDU

```
/* HDGST */
if (qp->header_digests) {
        header_digest = compute_digest(hdr, hlen);
        iov[iovcnt].iov_base = &header_digest;
        iov[iovcnt].iov_len = sizeof(header_digest);
        iovcnt++;
}

if (pad != 0) {
        /* PAD */
        iov[iovcnt].iov_base = __DECONST(char *, zero_padding);
        iov[iovcnt].iov_len = pad;
        iovcnt++;
}
```

TCP: Constructing a PDU

```
if (data_len != 0) {
        /* DATA */
        iov[iovcnt].iov_base = data;
        iov[iovcnt].iov_len = data_len;
        iovcnt++;

        /* DDGST */
        if (qp->data_digests) {
                data_digest = compute_digest(data, data_len);
                iov[iovcnt].iov_base = &data_digest;
                iov[iovcnt].iov_len = sizeof(data_digest);
                iovcnt++;
        }
}
```

TCP: Constructing a PDU

```
        return (nvmf_tcp_write_pdu_iov(qp, iov, iovcnt, plen));
}
```

TCP: Sending a Capsule

```
static int
tcp_transmit_capsule(struct nvmf_capsule *nc)
{
        if (nc->nc_qe_len == sizeof(struct nvme_command))
                return (tcp_transmit_command(nc));
        else
                return (tcp_transmit_response(nc));
}

struct nvmf_transport_ops tcp_ops = {
        …
        .transmit_capsule = tcp_transmit_capsule,
        …
}
```

TCP: Sending a Response Capsule

```
static int
tcp_transmit_response(struct nvmf_capsule *nc)
{
	struct nvmf_tcp_qpair *qp = TQP(nc->nc_qpair);
	struct nvme_tcp_rsp rsp;

	memset(&rsp, 0, sizeof(rsp));
	rsp.common.pdu_type = NVME_TCP_PDU_TYPE_CAPSULE_RESP;
	rsp.rccqe = nc->nc_cqe;

	return (nvmf_tcp_construct_pdu(qp, &rsp, sizeof(rsp), NULL, 0));
}
```

Conclusion

- NVMe over Fabrics was merged to FreeBSD's `main` branch in early May
  - Will ship in 15.0, will not be in 14.1
- Thanks to Chelsio Communications for sponsoring this work
- Questions?