vscsi(4) and iscsid

iSCSI initiator the OpenBSD way

by Claudio Jeker

iSCSI initiator support in OpenBSD was requested for some time. Thanks to the addition of vscsi(4) implementing an initiator got fairly simple. So simple that someone like me with no clue about SCSI was able to write a prototype in 3 days. The initiator is fully written in userland and passes the embedded SCSI datagrams unmodified to the kernel via vscsi(4). The SCSI midlayer of the kernel will issue the SCSI commands while iscsid is handling the higher level iSCSI protocol. iscsid is responsible to establish and to keep session alive. This split simplifies the design massively. The kernel code for vscsi(4) is just around 650 lines of commented C code. The userland process implementing the FSM and error handling is still far below 10k lines of code.

Overview

In the beginning large storage area networks (SAN) had to be built with Fibre Channel. Some call it the $4000 SCSI cable since the Fibre Channel switches are expensive compared to Ethernet ones.
I guess that because of this high pricetag, iSCSI was developed. Another reason could be the complexity of managing large FC SAN. For a service provider it may indeed be interesting to run everything over one unified infrastructure to reduce infrastructure, administration and management cost.
iSCSI provides blocklevel access to remote devices which is different to network filesystems like NFS or *gulp* SMB which allow direct file access. Another blocklevel network access protocol is ATA over Ethernet which is mostly dead since it was more of a quick and dirty hack. There is no real authentication and it is limited to the LAN since it is a L2 protocol.

The SCSI protocol

The Small Computer Systems Interface (SCSI) protocol provides a uniform way to talk to various I/O devices, especially storage devices. Disk, CD-ROMs, tape drives, scanners and even disk enclosures can be attached via SCSI. SCSI protocols are request/response application protocols with basic commands and special commands per device class. The SCSI protocols can be implemented on various physical interfaces. Parallel and serial attached SCSI for direct attached storage are obvious but Fibre Channel, IEEE-1394 aka Firewire and even USB mass storage are using the SCSI protocol to talk to the storage devices. In OpenBSD RAID controllers and modern SATA controllers convert internal SCSI messages into their native commands. So almost all disks in OpenBSD are now accessed as sd(4) using the SCSI midlayer.

The iSCSI protocol

iSCSI packs SCSI data units into a TCP data stream and the data is then routed via the Internet. The iSCSI protocol implements special session handling to work around the unreliable nature of the Internet.
The SCSI protocol defines two types of endpoints for a SCSI transport, the initiator and the target. Initiators issue SCSI commands to request services from components, logical units of a server, known as a target. Each Logical Unit has an address within a target called a Logical Unit Number (LUN). So a specific

device is identified by the target id and the LUN.
In the iSCSI case the naming is the same. The target is the server providing the storage and the initiator is running on the client where the exported disk is attached. A server can export multiple LUNs which are identified by unique names.

Sessions, Connections and Tasks

For each exported block device there is a iSCSI session between the initiator and the target. A session consists of one or more (TCP) connections. Having multiple connections open at the same time allows to load share the data across multiple TCP/IP streams and makes the system more resilient to failing connections (in theory, it also adds a lot of additional complexity to the protocol). At any given time, only one session can exist between a given iSCSI initiator node and an iSCSI target node for a specific device but the session may have multiple connections. It is possible to have multiple sessions between initiator and target open but each one will be for a different device.
Tasks are distributed onto the various connections but a specific task can only run over a single connection. Tasks are iSCSI requests for which a response is expected. SCSI commands are packed into tasks. If a connection fails all open task of that connection are rescheduled on other working connections. In the worst case a new connection will be opened.

State Machine

iSCSI connections and iSCSI sessions go through several well-defined states from the time they are created to the time they are cleared. For simplicity only the initiator state machines are covered.
The session state machine is very simple since there are only three states. The initial idle state (FREE), the full feature state (LOGGED_IN) and a state for session recovery and continuation (FAILED).
The connection state machine is much more complex.

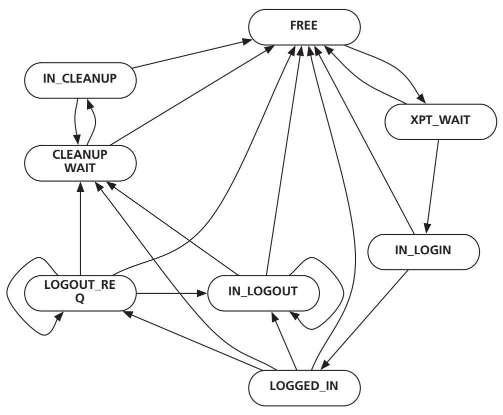

Figure 1: Connection State Machine

Table 1: State description

FREE	Initial state for an idle session
XPT_WAIT	Waiting for a response to a connection establishment request
IN_LOGIN	Waiting for the Login process to conclude
LOGGED_IN	Full Feature Phase
IN_LOGOUT	Waiting for a Logout response
LOGOUT_REQ	Waiting for an internal event signalling readiness to proceed with Logout
CLEANUP_WAIT	Waiting for cleanup processing
IN_CLEANUP	Waiting for the connection cleanup to conclude

A connection has to pass through a few phases until a full featured connection is established. Initially the connection is in the Login Phase where authentication happens. One or both sub stages, for security and operational negotiation, need to be passed through to complete the login process. When the Login Phase is finished the connection moves to the full featured phase. There is also a special discovery type that is a restricted full featured connection to get a list of all targets to which an initiator may have access, as well as the list of addresses on which these targets may be accessed.

iSCSI request/response types

All iSCSI protocol data units (PDUs) are built as a set of one or more headers and zero or one data segement. There is an optional CRC digest for both header and data segments. The main iscsi header has a fixed size of 48 bytes.
The PDU all contain various sequence numbers and allow to issue multiple tasks at the same time. During the login phase all parameters for the sessions are exchanged including the number of outstanding commands and the maximum PDU data size.

- **SCSI-Command**

  SCSI CDB (Command Descriptor Block) sent from initiator to target (I2T). It is possible that all or part of the data for write commands are included in the data segment part of the SCSI-Command PDU.

- **SCSI-Response**

  SCSI response from target to initiator (T2I). The PDU data segment may contain associated sense data in case of errors (e.g. SCSI CHECK CONDITION).

- **Task Management Function Request**

  Initiator can explicitly control the execution of one or more tasks via the Task Management Function Request.

- **Task Management Function Response**

  Response from the target for a previous Task Management Function Request which includes possible success or failure responses.

- **SCSI Data-Out and SCSI Data-In**

  SCSI Data-Out and SCSI Data-In are the main vehicles by which SCSI data payload is carried between initiator and target. Large data transfers can be segmented into multiple Data-Out or Data-In PDUs.

- **Ready To Transfer (R2T)**

  R2T is the mechanism by which the SCSI target “requests” the initiator for output data.

- **Asynchronous Message**

  Asynchronous Messages are used to carry SCSI or asynchronous events or asynchronous iSCSI messages.

- **Text Request and Text Response**

  Text requests and responses are designed as a parameter negotiation vehicle. The included data are UTF-8 encoded strings consisting of key=value pairs.

- **Login Request and Login Response**

  Special Text messages used only during the Login Phase.

- **Logout Request and Response**

  Logout Requests and Responses are used for the orderly closing of connections.

- **SNACK Request**

  With the SNACK Request, the initiator requests retransmission of data from the target.

- **Reject**

  Reject enables the target to report an iSCSI error condition.

- **NOP-Out Request and NOP-In Response**

  Empty messages to implement a 'ping' mechanism.

“Features” -- knobs and buttons

iSCSI comes with many buttons most of them are never used but result in a more complex standard and implementation. Others can result in session failures because both ends have a different opinion about how the session works.

For example the Task Management Function allow initiators to issue commands that may influence all other systems connected to this target. The included functions include among others `ABORT TASK`, `ABORT TASK SET`, `LOGICAL UNIT RESET`, `TARGET WARM RESET`, and `TARGET COLD RESET`. It may make sense to abort open tasks and task sets but `LOGICAL UNIT RESET`, `TARGET WARM RESET`, and `TARGET COLD RESET` will affect more then just this session. Especially the `TARGET COLD RESET` is equivalent to a power on event, thus terminating all of the target's TCP connections to all initiators. Luckily it is possible to return a “Function rejected” response.

iSCSI also allows to negotiate various parameters of a connection. While this makes sense, it often results in disabled features. For example is the number of concurrent connections by default 1 and many implementations actually don't support more then one connection. There is also various settings that change the behaviour of how to send data -- immediate, unsolicited or via initial R2T requests -- again it seems that targets implement one of the possibilities without correctly negotiating them with the initiator. It also makes the implementation more complex since multiple ways to send and receive data need to be implemented. Another such group of such settings are the various segment length parameters where non-default values are used or wrong parameters are used. So while it makes sense to allow initial parameters to be negotiated it often results in incompatibilities since implementations handle them a bit differently.

In the end the more options a protocol allows the harder is it to implement fully compliant daemons and interoperability is getting a gamble.

vscsi(4) -- a virtual scsibus

The vscsi device takes commands from the kernel SCSI midlayer and makes them available to userland for handling. Using this interface it is possible to implement virtual SCSI devices that are usable by the kernel.

ioctl commands

vscsi(4) defines 6 ioctl(2) commands to allow userland to dequeue SCSI commands and reply to them.

- **VSCSI_I2T**

  Dequeue a SCSI command. The SCSI midlayer sets the target, LUN and tag which will be used by the userland to identify the right transaction and for the kernel to map subsequent ioctl calls to this SCSI command.

- **VSCSI_DATA_READ**

  Read data from userland in response to a SCSI command identified by tag that had direction set to `VSCSI_DIR_READ`.

- **VSCSI_DATA_WRITE**

  Write data to userland in response to a SCSI command identified by tag that had direction set to `VSCSI_DIR_WRITE`. This data is available right after the `VSCSI_I2T` ioctl was received.

- **VSCSI_T2I**

  Signal completion of a SCSI command identified by tag. Depending on the status it is necessary to fill in the SCSI sense data.

- **VSCSI_REQPROBE**

  Signal the SCSI midlayer to probe a specific target and LUN. The SCSI midlayer will send an INQUERY command which will show up as a `VSCSI_I2T` ioctl.

- **VSCSI_REQDETACH**

  Signal the SCSI midlayer to detach a specific target and LUN.

I/O multiplexing

It is possible to use poll or kqueue on a vscsi(4) device file descriptor. It will signal readability as soon as a `VSCSI_I2T` request is available. This allows to write non-blocking applications without the use of threading. iscsid for example is libevent based.

Basic operation example

A process will open */dev/vscsi0* at startup afterwards no super-user privileges are needed so privileges can be dropped. The first operation on vscsi(4) is to inform the kernel that a new target is available. The kernel will then probe the device and attach it to the system. The main loop will wait for I2T commands. Depending on the command data is read or written and transactions are closed with a T2I response. The example code illustrates this basic operation. Before exiting the disk should be detached by using the `VSCSI_REQDETACH` ioctl.

vscsi(4) examle code

```
struct vscsi_ioc_devevent devev;
struct vscsi_ioc_i2t i2t;
struct vscsi_ioc_t2i t2i;
struct vscsi_ioc_data data;
struct poll pfds[1];
int done = 0;
int nfds;

/* first we need to tell the kernel to probe the disk */
devev.target = target;
devev.lun = 0;/* we only use one LUN per target */
if (ioctl(fd, VSCSI_REQPROBE, &devev) == -1)
    err(1, "ioctl VSCSI_REQPROBE");

pfds[0].fd = fd;
pfds.[0].event = POLLIN;


/*
 * main loop, get I2T, work on data and finish
 * transaction with a T2I.
 */
while (!done) {
    poll(pfds, 1, INFTIM);
    if (nfds == -1 && errno == EINTR)
        continue;
    if (nfds == -1 || !(pfds[0].revents & POLLIN))
        err(1, “poll”);

    /* the kernel issued a VSCSI_I2T command */
    if (ioctl(fd, VSCSI_I2T, &i2t) == -1)
        err(1, "ioctl VSCSI_I2T");

    /*
     * i2t contains the tag, target and lun plus the
     * direction of the operation. With this info the
     * process can issue new ioctl to get data
     * VSCSI_DATA_WRITE or handle the i2t command to
     * produce a response which ends with a VSCSI_T2I
     * ioctl.
     */
    if (i2t.direction == VSCSI_DIR_WRITE) {
        data.tag = i2t.tag;
        if ((data.buf = malloc(i2t.datalen)) == NULL)
            err(1, "malloc");
        data.datalen = i2t.datalen;

        if (ioctl(fd, VSCSI_DATA_WRITE, &data) == -1)
            err(1, "ioctl VSCSI_DATA_WRITE");

        /* work with provided data */
        data_write(i2t, data.buf, data.datalen);
        free(data.buf);
    }
    if (i2t.direction == VSCSI_DIR_READ) {
        void *buf;
        size_t len;

        /* get requested data */
        buf = data_read(i2t, &len);

        data.tag = i2t.tag;
        data.buf = buf;
        data.datalen = len;

        if (ioctl(fd, VSCSI_DATA_WRITE, &data) == -1)
            err(1, "ioctl VSCSI_DATA_WRITE");

        free(buf);/* allocated by data_read */
    }

    /* finish command with a t2i ioctl */
    bzero(&t2i, sizeof(t2i));
    t2i.tag = i2t.tag;
    t2i.status = data_status(&t2i.sense, &t2i.senselen);
    if (ioctl(v.fd, VSCSI_T2I, &t2i) == -1)
        err(1, "ioctl VSCSI_T2I");
}

/* detach disk */
if (ioctl(fd, VSCSI_REQDETACH, &devev) == -1)
    err(1, "ioctl VSCSI_REQDETACH");
```

iscsid and iscsictl

OpenBSD's iscsid is not the first open-source iSCSI initiator. Both NetBSD and FreeBSD have independent initiators. In FreeBSD the initiator is a kernel module with a userland utility to configure the initiator. The NetBSD initiator runs in userland and uses the FUSE / ReFUSE userland filesystem API to attach the disks to the kernel. As usual there are multiple implementation of iSCSI initiators in Linux with various degrees of in kernel code. The Linux-iscsi project is an in kernel driver whereas the open-iscsi project uses a mix of userland daemon and kernel driver. This is also the direction we would like to move to in the future.
We started our own daemon because no project satisfied our needs and fitted into our system. OpenBSD does not have userland filesystems -- apart from nnpfs

which is only used by afs. Writing the initiator as part of the kernel did not fit the “do not bloat the kernel” mantra and running the code in userland reduces the impact of security holes. Last but not least we suffer from NIH (not invented here) syndrome, maybe not as much as others but still quite a bit.

Design

I guess some people will expect to see the usual 3 process design that is used in many daemons I worked on. But this one is different. It doesn't even use the imsg framework since it is not needed. iscsid is able to drop all privileges on startup and still be fully functional. There is no need to use syscalls that need special privileges after startup. Super-user privilege is only needed to open */dev/vscsi0*. Since no other file needs to be opened during runtime the process can run in a chroot(2) jail. The configuration is loaded via the iscsictl program.
iscsictl parses the config file and passes the configuration over to iscsid which in turn is merging the data into the running configuration
iscsid is using libevent to do non-blocking I/O multiplexing. Similar to daemons like ospfd it does not use the libevent provided buffer API. The reason for this is probably personal preference but it seems to be easier to use a buffer system that fits better then the generic bufferevent API.

Event Loop and Finite State Machine

Currently the code is split into these parts:

Table 2: Source code files

iscsid.c	main() and additional helper functions
initiator.c	Initiator specific session handling
connection.c	Connection handling, including connection FSM
pdu.c	PDU (Protocol Data Unit) and task API
buf.c	buffer API
vscsi.c	vscsi(4) related code
control.c	control socket related functions
log.c	logging and debug functions

Event Loop and Finite State Machine

As mentioned iscsid uses libevent to handle multiple connections at the same time. There are event handlers for vscsi, the control socket and for each open TCP connection. Additionally some timeouts run in the background to handle the state machinery when needed.

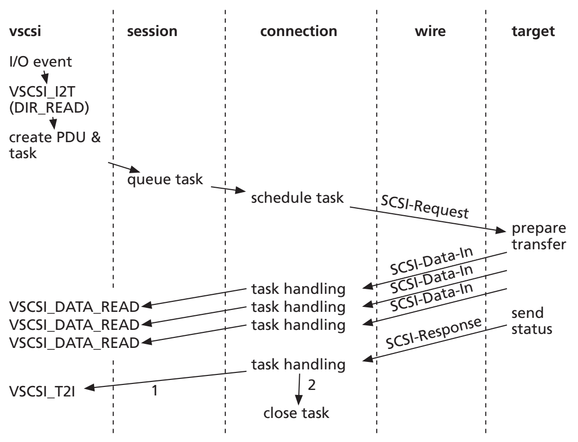

Figure 2: Read Operation Flow

During basic operation iscsid waits for a command coming from the kernel. `vscsi_dispatch()` issues the `VSCSI_I2T` ioctl and maps the target to a session. A new task is created and a PDU is assembled from the SCSI command and added to the task. This task is then issued to the session where it is scheduled onto an open connection. The connection sends the PDU to the target and handles the responses. The task is completed when a SCSI-Response message is received. The SCSI-Response is converted into a `VSCSI_T2I` ioctl message. vscsi will then complete the command and free the CDB (Command Descriptor Block) with the help of the midlayer.

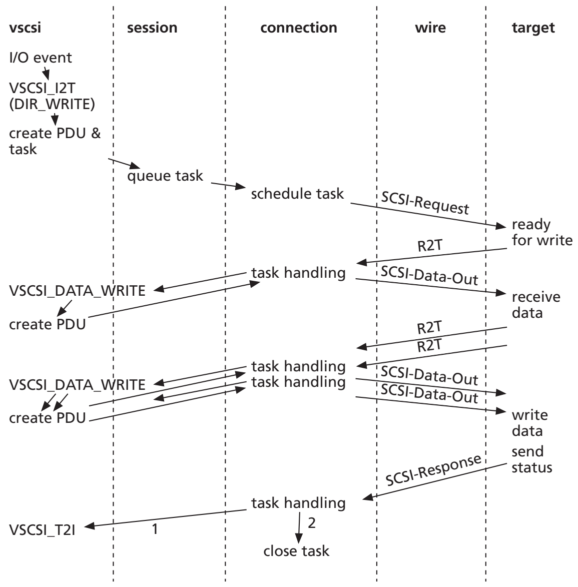

Figure 3: Write Operation Flow

Session and connection establishment

New sessions and connections need some initial configuration. This config is provided by iscsictl. A new session will try to open a connection to the target. If that succeeds the login process is run. Authentication and parameter negotiation is done and when all succeeds the session is logged in and in full feature mode. It is possible to create special discovery sessions which will just list the available devices available on a target. Discovery sessions are not full featured and only allow the exchange of text and logout messages.

Error handling

Correct error handling is very important and a must when using the daemon in production. Even local connections fail or get closed from time to time. In this moment iscsid needs to try to reopen the connection and issue all outstanding commands. In the worst case unfinished tasks need to be rescheduled on different connections. At the time of this writing almost no error recovery was implemented in iscsid.

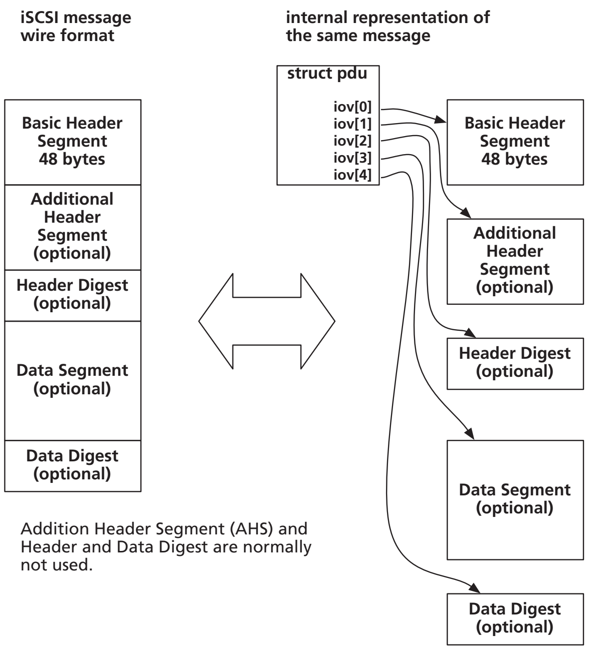
iSCSI message wire format
internal representation of the same message
struct pdu
iov[0]
iov[1]
iov[2]
iov[3]
iov[4]
Basic Header Segment 48 bytes
Additional Header Segment (optional)
Header Digest (optional)
Data Segment (optional)
Data Digest (optional)
Addition Header Segment (AHS) and Header and Data Digest are normally not used.

Figure 4: PDU buffers

Task and PDU handling

To complete a task multiple messages or PDUs need to be exchanged. Depending on the task different actions need to be taken. Because of this a task will call a callback whenever a full PDU was received. The callback will either close the task or issue another PDU request. PDU are special buffers, on read the various data parts are split into a sub buffers similar to a iovec. When creating a PDU the same approach is used. Therefore the header and data are always in independent buffers. A task can reside on either the session queue or a connection queue.

At the moment only one task is scheduled per connection. The protocol allows to have multiple tasks open at the same time but this is not yet implemented.

iscsictl -- monitoring and configuration

iscsid and iscsictl communicate via a UNIX local socket. Unlike the imsg daemons a datagram socket is used. So there is no need to convert a stream of bytes into messages. The datagrams start with a common header -- mainly a message type -- the rest of the structure is type specific. Receiving of datagrams is atomic so there is no need to include a message length in the header but there is a size limit of 4KB. This size limit should be large enough, even for complex data structures.
iscsictl is used to load the config into iscsid and to monitor the sessions of iscsid. The configuration file is parsed by iscsictl, converted into a binary representation and then passed to iscsid where the configuration is merged into the current configuration. Iscsid does not have a way to load the configuration. It starts up with an empty configuration and will do nothing until iscsictl loads the config into the daemon.
iscsictl is also used to monitor iscsid. It can be used to view session and connection states. Returning statistics about connections and sessions. This part of the code is not yet written.

Future ideas

First of all the current implementation needs to be finished. iscsictl is at the moment still more of a dream then actually a working config and monitoring tool. More parts of the basic iSCSI protocol need to be covered in iscsid and a lot of testing is needed. The next step would be some performance testing against high-performance targets. For development software based iscsi targets like the netbsd iscsi-target are very nice but to get performance numbers a more performing target is needed.

One of the plans is to move part of the protocol handling back into the kernel. vscsi and the TCP socket would handle the basic message passing directly in kernel while the userland would wait for iSCSI specific messages and do the login, logout procedures. One of the big question marks is the additional complexity added to the kernel to implement this. The basic protocol parsing is fairly simple but retransmission logic and protocol extension like the CRC digest could result in complex code. So cost and benefit need to be evaluated carefully.

A less intrusive optimization would be to use a method similar to sendfile() to remove the copy overhead from kernel to userland and back.

Adding RDMA support has not been considered for now. In the end RDMA only makes sense when the bulk data handling can be offloaded to the kernel.

References

[1] Internet Small Computer Systems Interface (iSCSI), RFC 3720, April 2004

[2] Internet Small Computer Systems Interface (iSCSI) Naming and Discovery, April 2004

[3] A Portable iSCSI Initiator, Alistair Crooks (The NetBSD Foundation), http://2008.asiabsdcon.org/papers/P6A-paper.pdf

[4] NetBSD initiator source code, http://cvsweb.netbsd.org/bsdweb.cgi/src/external/bsd/iscsi/initiator/

[5] FreeBSD initiator source code, http://www.freebsd.org/cgi/cvsweb.cgi/~checkout~/src/sys/dev/iscsi/initiator/

[6] OpenBSD Project, http://www.openbsd.org/

[7] FreeBSD Project, http://www.freebsd.org/

[8] NetBSD Project, http://www.netbsd.org/

[9] Open-iSCSI Project, http://www.open-iscsi.org/

[10] Linux-iSCSI Project, http://linux-iscsi.sourceforge.net/