SMB-3.0 Offload Data Transfer (ODX) (fast server-side copy)

Gordon Ross
Nexenta Systems, Inc.

Sept. 2016

What's driving this? (Hyper-V)

- Hyper-V requirements for storage on SMB
  - SMB 3 protocol level
  - CA shares (or resilient handles)
  - Authenticate machine accounts
- Hyper-V "nice to have"
  - Offload Data Transfer (ODX) for fast VM deployment

Background: Copy Chunks

- Traditional server-side copy uses the "copy chunks" ioctl (etc.)
- Client-side outline: (simplified)

Open src & dst files
ioctl FSCTL_SRV_REQUEST_RESUME_KEY on src
if src is sparse
    ioctl FSCTL_QUERY_ALLOCATED_RANGES on src
build list ranges to copy
ioctl FSCTL_SRV_COPYCHUNK on dst

Why another server-side copy?

- Short answer: Allows the server to optimize.
  http://technet.microsoft.com/en-us/library/hh831628.aspx
- The *server* is in control of:
  - Representation of data being copied (the “token” is opaque to clients)
  - Length of data represented by ODX read
  - Length of data transferred by ODX write
- Much more flexible than “copy chunks”.
- Allows the *server* to be “in control”!
- Allows “T10” level copy with RSVD (optional, and not covered here)

How it works (client side)

- Client-side outline:

```
open src_file (and get size etc)
create dst_file (and set its size)
while src_file not all copied
    offload_read(src_file, &token, &rsize);
    while token not all copied (rsize > 0)
        offload_write(dst_file, token, &wsize);
```

- Two-level loop – not obvious from spec.
- The server controls read & write stride!

Details: ODX read

- Inputs:
  - src file handle
  - src file offset
  - copy_length
- Outputs:
  - xfer_length
  - ODX “token”
- Note about ODX read out.xfer_length:

  The server may return xfer_length < copy_length
  i.e. to limit number of ranges to fit in the token, or to control the read “stride”.
- … what’s that “token” thing? (next)

Details: ODX “token”

- Essentially a fixed-size “union”
- The fixed part specifies the type
- One type is pre-defined for “all zeros”
- All other types are vendor defined.
- Every token must represent the data range(s) covered by the off+len returned by ODX read.
- Server decides what the token looks like, and (more important) the read transfer length.

Example ODX token layout (as implemented in NexentaStor)

```
#define     STORAGE_OFFLOAD_TOKEN_TYPE_ZERO_DATA  0xFFFF0001  /* protocol defined */
#define     STORAGE_OFFLOAD_TOKEN_TYPE_NATIVE1      0x10001  /* implementation defined */
#define     TOKEN_TOTAL_SIZE                        512
#define     TOKEN_MAX_PAYLOAD                       504          /* 512 - 8 */

typedef struct smb_odx_token {
            uint32_t    tok_type;   /* big-endian on the wire */
            uint16_t    tok_reserved;           /* zero */
            uint16_t    tok_len;    /* big-endian on the wire */
            union {
                        uint8_t u_tok_other[TOKEN_MAX_PAYLOAD];
                        struct tok_native1 {
                                    smb2fid_t   tn1_fid;
                                    uint64_t    tn1_off;
                                    uint64_t    tn1_eof;
                        } u_tok_native1;
            } tok_u;
} smb_odx_token_t;
```

Details: ODX write

- Inputs:
  - dst file handle
  - dst file offset
  - copy_length
  - xfer offset (relative to start of token)
  - ODX “token”
- Outputs:
  - xfer_length
- Note about ODX write out.xfer_length:

  the server may return xfer_length < copy_length
  i.e. to limit the amount of I/O per ODX write request.
  The client will issue more ODX write calls with the same token until all data represented by the token is copied.

Optimizing ODX read

Easy optimizations:

- Use the “all zeros” token when the range to be copied is entirely within a “hole” (common with virtual disks, i.e. with Hyper-V).
- When there’s no more data after the requested range, set (return) the flag ..._ALL_ZERO_BEYOND which tells that client they can stop copying here.

Note that we don’t encode ranges in ODX read.
(Just encode the “src” handle, and do “sparse copy” in write.)

Optimizing ODX write

Easy optimizations:

- When given the all-zeros token, “punch a hole” (free the range with fcntl F_FREESP or equiv.)
- When given a “native” (data) token, just call a common “sparse copy” function (“hole aware”)
- Limit the amount of I/O per write, to avoid causing timeouts, and allowing the client to provide reasonable progress feed-back.

ODX copy scenarios

How this looks “on the wire”:

- When copying “all-zeros” we have a “stride” of 256MB (one ODX read and one ODX write)
- When copying data, the read “stride” is 256MB and the write “stride” is 16MB. (Client does one ODX read and then 16 ODX writes.)

[wireshark examples]

Testing ODX copy

- This part of the protocol is tricky to test thanks to the amount of control left to servers about tokens, strides etc.
- Have a simple ODX copy test for smbtorture:

  https://github.com/gwr/samba/blob/gwr-m2

  source4/torture/smb2/ioctl.c

Follow-up

- Where to see our code:
- https://github.com/Nexenta/illumos-nexenta
  usr/src/uts/common/fs/smbsrv/smb2_fsctl_odx.c
  (not published yet, but will be “soon”)
- Questions?

- Gordon Ross <gwr@nexenta.com>