Porting UNH EXS from verbs to OFI

Patrick MacArthur <pmacarth@iol.unh.edu>
UNH InterOperability Laboratory
#OFADevWorkshop
Acknowledgements

The author would like to thank the University of New Hampshire InterOperability Laboratory for the use of their RDMA cluster for the development, maintenance, and testing of UNH EXS.

This material is based upon work supported by the National Science Foundation under Grant No. OCI-1127228 and by the National Science Foundation Graduate Research Fellowship Program under award number DGE-0913620.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.
Background
UNH EXS (Extended Sockets)

https://www.iol.unh.edu/expertise/unh-exs

- Based on ES-API (Extended Sockets API) published by the Open Group
- Extensions to sockets API to provide asynchronous, zero-copy transfers
  - Memory registration (exs_mregister(), exs_mderegister())
  - Event queues for completion of asynchronous events (exs_qcreate(), exs_qdequeue(), exs_qdelete())
  - Asynchronous operations (exs_send(), exs_recv(), exs_accept(), exs_connect())
- UNH EXS supports SOCK SEQPACKET (reliable message-oriented) and SOCK STREAM (reliable stream-oriented) modes
- No SOCK DGRAM (unreliable datagram) mode (yet)
Motivation

• Enable porting UNH EXS to future non-IB fabrics
• Prepare for future Windows Network Direct port
• Battle-test implementation of libfabric providers
Status of OFI port

• Successfully runs over OFI verbs provider and OFI sockets provider
• Still some missing functionality (due to missing functionality in both providers)
Connection Establishment Issues
EXS Connection Establishment

• ES-API specifies asynchronous `exs_accept()` and `exs_connect()` functions

• How to create a socket not specified by ES-API—intention was to rely on existing sockets API functions
  – `socket()`, `bind()`, `listen()`

• UNH EXS provides `exs_socket()`, `exs_bind()`, `exs_listen()` with same interface as POSIX
Server connected socket setup

**POSIX Sockets**

```c
struct addrinfo *ai;
hints.flags = AI_PASSIVE;
getaddrinfo(name, service, &hints, &ai);
lfd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
bind(lfd, ai->ai_addr, ai->ai_addrlen);
listen(lfd, 0);

afd = accept(lfd, &peer_addr, &peer_addrlen);
```

**UNH EXS**

```c
exs_init(EXS_VERSION1);
struct addrinfo *ai;
hints.flags = AI_PASSIVE;
getaddrinfo(name, service, &hints, &ai);
fd = exs_socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
exs_bind(lfd, ai->ai_addr, ai->ai_addrlen);
exs_listen(lfd, 0);
accept_queue = exs_qcreate(n);
exs_accept(lfd, &av, n, 0, accept_queue);
/* ... */
exs_qdequeue(accept_queue, &events, n, NULL);
afd = EXS_EVT_NEW_SOCKET(events[m]);
```
Client connected socket setup

### POSIX Sockets

```c
getaddrinfo(name, service, &hints, &ai);
fd = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
bind(fd, ai->ai_addr, ai->ai_addrlen);
connect(fd, ai->ai_addr, ai->ai_addrlen);
```

### UNH EXS

```c
exs_init(EXS_VERSION1);
getaddrinfo(name, service, &hints, &ai);
fd = exs_socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol);
exs_bind(fd, ai->ai_addr, ai->ai_addrlen);
connect_queue = exs_qcreate(n);
exs_connect(fd, ai->ai_addr, ai->ai_addrlen, 0, NULL, connect_queue, &ctx);
/* ... */
exs_qdequeue(connect_queue, &events, n, NULL);
```
POSIX/EXS: getaddrinfo()

• POSIX-defined function used to perform name resolution in protocol-agnostic fashion
  – Not part of original sockets API, came in with IPv6
  – Use of getaddrinfo() is optional in sockets

• Arguments
  – Node and service strings
  – Hints structure limiting returned entries

• returns linked list of struct addrinfo
  – Elements of this structure are passed to socket(), bind(), and connect()
  – No POSIX/EXS function takes struct addrinfo as input
OFI: fi_getinfo()

- Functionally analogous to POSIX getaddrinfo() and verbs rdma_getaddrinfo()
- Address of local/remote host specified as either:
  - node and service strings
  - src_addr and dst_addr fields of hints structure
- Returns struct fi_info which is directly passed to OFI “constructor” calls
  - Users required to call fi_getinfo() before any other OFI function
    - Different from sockets (POSIX and EXS), in which no call takes struct addrinfo as a parameter
- How to deal with this requirement?
fi_getinfo(): Obvious Strategy

• Implement new exs_getaddrinfo() in terms of fi_getinfo()
  – Pass arguments directly to fi_getinfo()
  – Embed corresponding struct fi_info in each returned struct addrinfo
  – Allows user some limited choice of fabric provider

• Problem: fi_info structure needed to perform exs_listen()/exs_connect() calls, but struct addrinfo not passed in
  – Makes this solution untenable without new EXS API functions
fi_getinfo(): Actual Strategy

- Call `fi_getinfo()` within `exs_listen()` and `exs_connect()` that take `sockaddr` parameter
- Pass `struct sockaddr` via hints to `fi_getinfo()`
- `fi_info` struct stored as part of connection state
- **Disadvantage: hides decision of which fabric provider to use from user**
  - Current policy is to use first `fi_info` entry for which `listen/connect` succeeds
OFI: Endpoints

- Listening and connecting sockets both created with socket() system call
  - EXS retains this behavior
  - Verbs mimics behavior with rdma_create_id()
- OFI: Listening (passive) and connecting endpoints are completely separate types! (This is good API design)
- Cannot associate socket with OFI endpoint at time of exs_socket() call
Implementation Issues
### exs_socket() implementation

**Problem:** need a unique fd to return to user

<table>
<thead>
<tr>
<th>Existing Verbs</th>
<th>Libfabric</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn-&gt;channel =</td>
<td>dummy_fd = socket(...)</td>
</tr>
<tr>
<td>rdma_create_event_channel();</td>
<td>conn-&gt;hints = fi_allocinfo();</td>
</tr>
<tr>
<td>rdma_create_id(conn-&gt;channel, ...);</td>
<td>/* initialize hints */</td>
</tr>
<tr>
<td>return conn-&gt;channel-&gt;fd;</td>
<td>return dummy_fd;</td>
</tr>
</tbody>
</table>

- RDMA CM **event channel** and **cm_id** are provider independent
- Return event channel fd as the fd of the socket

- Event queues and endpoint structures provider-dependent
- **Does not allocate any fabric resources yet**
- Create dummy socket and return its fd
exs_bind() implementation

Existing Verbs

```
rdma_bind_addr(conn->cm_id, address);
```

libfabric

```
new_conn->hints->src_addrlen = address_len;
memcpy(new_conn->hints->src_addr, address, address_len);
```

Libfabric implementation does not actually bind socket.

This means that exs_getsockname() on bound but not listening/connected socket will not return ephemeral port number—incompatibility with Linux sockets
exs_listen() implementation

Existing Verbs

libfabric

```c
fi_getinfo(EXS_FI_VERSION, NULL, NULL,
           0, &new_conn->hints,
           &all_info);
for (auto &info : all_info) {
    fi_fabric(info->fabric_attr,
              &new_conn->fabric, new_conn);
    fi_passive_ep(fabric, info,
                 &new_conn->pep, new_conn);
    fi_eq_open(fabric, eq_attr,
               &new_conn->cm_eq, new_conn);
    fi_pep_bind(new_conn->pep,
                &new_conn->cm_eq->fid, 0);
    rdma_listen(conn->cm_id, backlog);
    fi_listen(new_conn->pep);
    break;
}
```
exs_connect() implementation

Existing Verbs

/* User thread */
ret = rdma_resolve_addr(conn->cm_id, 
                        address, 2000);
return ret;

/* EXS internal thread */
rdma_get_cm_event(conn->event_channel, &event);
rdma_resolve_route(conn->cm_id, 2000); 
rdma_get_cm_event(conn->event_channel, &event);
/* Set up CQ, QP, etc. */
rdma_connect(conn->cm_id, ...);

libfabric

fi_getinfo(EXS_FI_VERSION, NULL, NULL, 0, &new_conn->hints, &info);
fi_fabric(info->fabric_attr, &new_conn->fabric, new_conn);
fi_domain(new_conn->fabric, info, &new_conn->domain, new_conn);
fi_endpoint(new_conn->domain, info, &new_conn->ep, new_conn);

/* Set up/bind CQ, EQ, etc. */
fi_connect(new_conn->ep, info->dest_addr, ...);
Connection Establishment: Summary of Differences

• CM event queues
  – Verbs: provider independent
  – OFI: provider-specific

• Address resolution
  – Verbs: rdma_getaddrinfo optional
  – OFI: fi_info struct required

• Listening endpoint
  – Verbs: same type as connecting endpoint
  – OFI: listening and connecting endpoint distinct types with distinct constructors

• Client connection establishment
  – Verbs: requires multiple asynchronous operations in sequence
  – OFI: single fi_connect operation
Verbs Inline Data vs. OFI Injected Data

• Both copy data into HW memory at post time; remove need to register memory

• OFI Injected data:
  – **FI_INJECT flag** to fi_sendmsg, fi_writemsg: Behaves identically to verbs IBV_SEND_INLINE flag to ibv_post_send
  – **fi_inject call**: Injects data and suppresses completion, **even if completions were requested for all operations**!
  – fi_inject call may lead to CQ overrun unless application maintains and checks counter on every send
Write with remote CQ data

- Verbs: incoming RDMA WRITE with immediate data consumes a posted receive WR
  - This makes no sense semantically
- OFI: optional to consume a posted receive WR
  - If no recv WR consumed, op_context field of completion entry will be NULL
  - **Missing feature:** detect this at initialization time, to avoid creating “dummy” buffers/receive work requests
    - GitHub: libfabric issue #666
fi_shutdown() vs. rdma_disconnect()

• rdma_disconnect()
  – Transitions QP to error state
  – Flushes all pending WRs to CQ
  – Causes completion event on completion channel
  – In UNH EXS: wakes up completion thread and signals connection shutdown

• fi_shutdown()
  – Behavior for outstanding operations not specified
  – No guaranteed wakeup for thread blocked on completion queue
    • EXS Workaround: use timeout on blocking CQ read call
Performance
Performance Tests

- Using Mellanox ConnectX-3 FDR InfiniBand HCAs
  - Connected via Mellanox SX6036 FDR InfiniBand switch
- Scientific Linux 6.4 with OFED 3.5-2
  - libibverbs 1.1.7
  - librdmacm 1.0.17
  - libfabric git master
- OFI verbs provider vs. existing Verbs
- Message-oriented sockets
- Tests performed: blast (throughput), ping (latency)
Throughput—little difference
Latency—big difference
Conclusions
Conclusions

• Successfully ported UNH EXS to OFI verbs, sockets providers
• Porting UNH EXS uncovered many bugs and missing features in providers
• Revealed some differences between OFI and Verbs:
  – OFI distinguishes between listening and connecting endpoints, Verbs doesn’t
  – OFI “constructors” take fi_info as a parameter, Verbs don’t
  – OFI event queues, wait sets, etc. are per-provider, Verbs are system-wide
  – OFI received immediate data may or may not consume a receive WR, Verbs always does
  – OFI doesn’t guarantee wakeup from blocking EQ/CQ calls on connection shutdown, Verbs does
Thank You
Backup
UNH EXS Classification

- Middleware for legacy applications
- Use of multiple providers (possibly at same time)
- Limited to reliable connected endpoints for now
- Required data transfer operations:
  - SEND/RECV (for control messages)
  - RDMA WRITE WITH IMM (for data)
Status of OFI port

- OFI port on separate branch; mainline still uses Verbs
  - Plan to merge OFI support into mainline when complete
  - OFI (libfabric) or Verbs (libibverbs + librdmacm) will be selectable at compile time
EXS Data Transfer Protocols

Direct Transfer (Message and Stream Sockets)

- User send buffer
- exs_send
- sender
- ADVERT
- DIRECT
- RDMA WRITE WITH IMM
- receiver
- exs_recv
- User receive buffer

Indirect Transfer (Stream sockets only)

- User send buffer
- exs_send
- sender
- INDIRECT
- RDMA WRITE WITH IMM
- receiver
- exs_recv
- Intermediate Receive Buffer
- COPY
- User receive buffer
exs_shutdown()/exs_close()

- We wish to ensure that all messages arrive at destination endpoint prior to disconnect
- Verbs EXS shutdown: EOF message exchange
  - User calls exs_close()
    - Local fd invalidated
    - Returns immediately; completes asynchronously
  - Local endpoint completes outstanding sends
  - Local endpoint sends EOF message
  - On receive EOF, remote endpoint sends EOF reply
  - On receive EOF reply completion, local endpoint calls rdma_disconnect()
    - Disconnected CM event fires and all WRs flushed
  - Once socket refcount == 0, close event posted