libfabric Composability: Peer Provider Architecture

Sean Hefty

Intel Corporation
libfabric Classic Architecture

Illustrative Components

Middleware (MPI, oneCCL, oneSHMEM...)

App codes to libfabric to run in any **one** environment

libfabric API

libfabric

Core

TCP Provider

Verbs Provider

EFA Provider

SHM Provider

Provider API

user-space

kernel

Sockets

RDMA APIs

POSIX shm

NIC

RDMA NIC

EFA NIC

Network (Ethernet)

Legend

Middleware

libfabric

Lower-level APIs

Linux Kernel Subsystems

HW

Network

Kernel bypass

SW call path
Using Multiple Providers

Example: Shared Memory + Network Provider

What if the app wants to use 2 providers? E.g. overall performance will improve if the app can leverage shared memory for intranode communication

Middleware (MPI, oneCCL, oneSHMEM...)

Today: App opens both providers

libfabric API

Core

Verbs Provider

SHM Provider

EFA Provider

App picks provider based on target address

Alternative: tight coupling between providers

2 providers

Twice as many objects
domains – register memory twice
endpoints – double receive buffers
completion queues – check two

Tightly coupled implementation

Higher development costs

Reuse SHM core
Hide internal use of SHM
Need to repeat for other providers

E.g. overall performance will improve if the app can leverage shared memory for intranode communication

Alternative: tight coupling between providers

Reuse SHM core
Hide internal use of SHM
Need to repeat for other providers
Using Multiple Providers
Complex, Theoretical Scenarios

What if multiple providers are needed for optimal performance?

Tight coupling becomes impractical

Middleware (MPI, oneCCL, oneSHMEM...)

App opens

Independent providers optimized for specific technologies

libfabric API

libfabric

Core

RDMA Network

SHM Provider

Rack-Level Network

GPU-Network Provider

Switch Provider

Provider API

Traditional HPC network for scale-out

Intranode or scale-up

Enhanced rack or pod-level network

Accelerator network

Switch-based accelerations

Theoretical (some don't exist)
Using Multiple Providers

Example: Shared Memory + Network Provider

Needed: efficient cooperation of independent providers

Peer APIs

Middleware (MPI, oneCCL, oneSHMEM...)

App opens one provider, but has access to both

libfabric API

libfabric

Core

Verbs Provider

SHM Provider

Link Provider (Optional)

Provider API

Peer API

Keep providers highly focused

Allow independent development

Easy for providers to adopt

Once a provider is enabled for peer API support, can swap in another to its right or left

Orchestration handled by core provider or (future) link provider
Generically combining a half dozen independently developed providers without losing performance. And, how, exactly, do you intend to accomplish this?
Review: libfabric API

API defines user interface to objects

Event Queues

Completion Queues

Completion Counters

Active Endpoints

Fabric

Passive Endpoints

Address Vector

Memory Regions

Domain

Example: active endpoint

```c
struct fid_ep {
    ...
    struct fi_ops_msg *msg;
    struct fi_ops_rma *rma;
    ...
};

static inline
fi_send(ep, buf, len, ...)
{
    return ep->msg->send(ep, ...)
}
```

User invokes direct call on object
Peer Object Model
Sharable Fabric Identifiers (FIDs)

Define objects to share between providers

Peer objects define provider to provider callbacks

libfabric API

Owner Provider
Active Endpoint
Owner EP
get_rbuf()

Completion Queue
Owner CQ
write()

Peer Provider

Peer EP
start_rx()

Peer CQ
progress()

Conceptual example (details in next talks)
Don’t change that dial!
More details to follow!

THANK YOU
Inx decides if peer is remote or local

Inx adds addresses into shm and core providers’ AVs with FI_AV_USER_ID which allows peer providers to report the application fi_addrs in the CQ

core provider and shm write completions directly into Inx owned CQ and get receive buffers from Inx managed SRX
1. Owner allocates a peer cq and defines peer CQ write ops

```c
struct fid_peer_cq {
    struct fid fid;
    struct fi_ops_cq_owner *owner_ops;
};

struct fi_ops_cq_owner {
    ssize_t (*write)();
    ssize_t (*writeerr)();
};
```

2. Owner calls `fi_cq_open`, passing in the peer_cq via context indicating a peer with attr->flags | FI_PEER

```c
fi_cq_open(peer_domain, &attr, &peer_cq, peer_context);
```

3. Peer calls imported peer_cq->owner_ops in order to write an entry to the shared CQ
1. Owner creates peer_srx_context and sets owner ops

2. Owner imports SRX into peer by calling fi_srx_context passing in the peer_cq via context indicating a peer with attr->flags | FI_PEER. Peer sets peer_ops

Peer calls owner ops to get, queue, and free messages

Owner calls peer ops to start and discard unexpected messages
EXAMPLE SRX FLOW

OWNER

SRX

fi_recv()

fi_srx_context()

fi_recv()

start_msg()

PEER

msg

get_msg()

free_entry()

FI_ENOENT

msg

get_msg()

queue_msg()

free_entry()
THANK YOU

Alexia Ingerson

Intel Corporation
OFI PROVIDER FOR COLLECTIVE OFFLOAD

Jianxin Xiong

Intel Corporation
OFI COLLECTIVE API

- **API summary**
  - Asynchronous
  - Defined in `<rdma/fi_collective.h>`
  - Supported ops: barrier, broadcast, alltoall, allreduce, allgather, reduce, reduce_scatter, scatter, gather
  - Wrapper functions: `fi_barrier()`, `fi_broadcast()`, …
  - Collective groups: `av_set`
    - A set of addresses (`fi_addr_t`) representing group members
    - Can perform set operations: insert, remove, intersect, union, diff
    - Similar to multicast group, join via the same `fi_join()` call, but with `FI_COLLECTIVE` flag

- **Collective ops can be defined for each endpoint**

```c
struct fid_ep {
    // ...
    struct fi_ops_collective collective;
};
```
IMPLEMENTATION CONSIDERATIONS

- **Goal**: Efficiently enable multiple providers over multiple collective offload engines

- An example of offload engines is switch with collective support

- **Option 1 -- fully independent implementations**
  - Each provider implements collective ops for each offload engine
  - Pros: good separation between providers and between offload engines
  - Cons: a lot of duplicated efforts

- **Option 2 -- collective functions as utility code**
  - Pros: reduce code duplication
  - Cons: utility code enforce common basic data structures (domain, ep, cq, etc) to be used by providers

- **Peer-provider provides a better option**
COLLECTIVE OFFLOAD WITH PEER PROVIDER

- Implement a collective-only provider for each offload engine

  - Act as a peer provider to the “main” provider
  - The main provider shares necessary data structure (domain, cq, eq, av, etc) via the peer-provider API
    - Eliminate the needs of creating duplicated queues / tables
    - The collective provider reports completions / events directly to the main provider
  - Pros:
    - Reduce code duplication
    - Separation between the main provider and the offload provider – interface via peer-provider API only
  - Cons:
    - The provider-to-provider workflow must be coordinated and well-defined
# COLLECTIVE GROUP CREATION

<table>
<thead>
<tr>
<th>Application</th>
<th>Main provider</th>
<th>Collective provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>fi_av_set()</td>
<td>fi_av_set()</td>
<td>av_ops.av_set()</td>
</tr>
<tr>
<td>av_set</td>
<td>av_set</td>
<td>av_set</td>
</tr>
<tr>
<td>fi_av_set_addr()</td>
<td>fi_av_set_addr()</td>
<td>av_set_ops.addr()</td>
</tr>
<tr>
<td>coll_addr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Diagram:**
- **Application:**
  - comm
  - fi_av_set()
  - av_set
  - fi_av_set_addr()
  - coll_addr
- **Main provider:**
  - av
  - fi_av_set()
  - fi_av_set_addr()
- **Collective provider:**
  - av
  - av_ops.av_set()
  - av_set_ops.addr()
JOIN COLLECTIVE GROUP

<table>
<thead>
<tr>
<th>Application</th>
<th>Main provider</th>
<th>Collective provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>fi_join_collective()</td>
<td>fi_join()</td>
<td>cm_ops.join()</td>
</tr>
<tr>
<td>coll_addr</td>
<td>av_set</td>
<td>schedule offload work</td>
</tr>
<tr>
<td>fi_eq_read</td>
<td>eq</td>
<td>fi_eq_write()</td>
</tr>
<tr>
<td>mc</td>
<td>ret</td>
<td>completion</td>
</tr>
</tbody>
</table>
Offload collective engine may require a small set of out of band collectives for bootstrapping. Can be implemented in the main provider using pt2pt communication.

```
Main provider

coll_ops.barrier(FI_PEER_XFER)

peer_xfer_ops.complete()

Collective provider

fi_barrier2(FI_PEER_XFER)

ret

status
```
- Pt2pt based collectives can be moved to its own provider

<table>
<thead>
<tr>
<th>Util coll provider</th>
<th>Main provider</th>
<th>Offload coll provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>coll_ops.barrier2()</td>
<td>fi_barrier2()</td>
<td>bootstrap</td>
</tr>
<tr>
<td>send/recv(FI_PEER_XFER)</td>
<td>send/recv</td>
<td>fi_barrier2(FI_PEER_XFER)</td>
</tr>
<tr>
<td>peer_xfer_ops.complete()</td>
<td>send/recv completion</td>
<td>peer_xfer_ops.complete()</td>
</tr>
<tr>
<td>send/recv status</td>
<td></td>
<td></td>
</tr>
<tr>
<td>coll completion</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CONCLUSION AND FUTURE WORK

- Peer provider provides a mechanism for implementing “functional” providers w/o duplicating important data structures. Collective offload is one such function that suits this model well.

- As a proof-of-concept, a utility collective provider has been implemented to provide software-based collective functionality.
  - The rxml provider now uses this utility collective provider for default collective support instead of the old “shared utility code” based implementation.
  - Enables other providers to leverage the pt2pt based collective implementation more easily.

- Future work will have offload collective provider(s) implemented for popular collective offload engine(s). That’s when upper layer middleware can start taking advantage of OFI collectives.
THANK YOU

Jianxin Xiong

Intel Corporation