OPENFABRICS ALLIANCE

2023 OFA Virtual Workshop

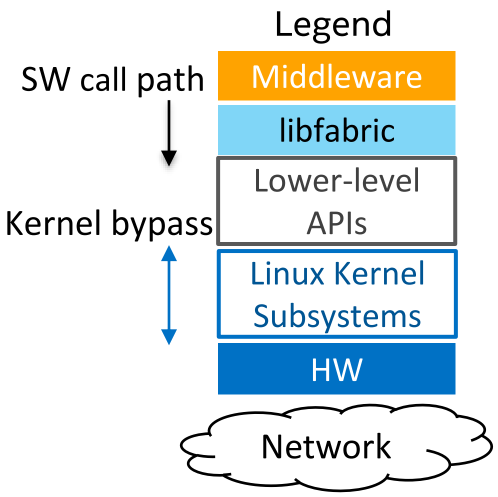
libfabric Composability: Peer Provider Architecture

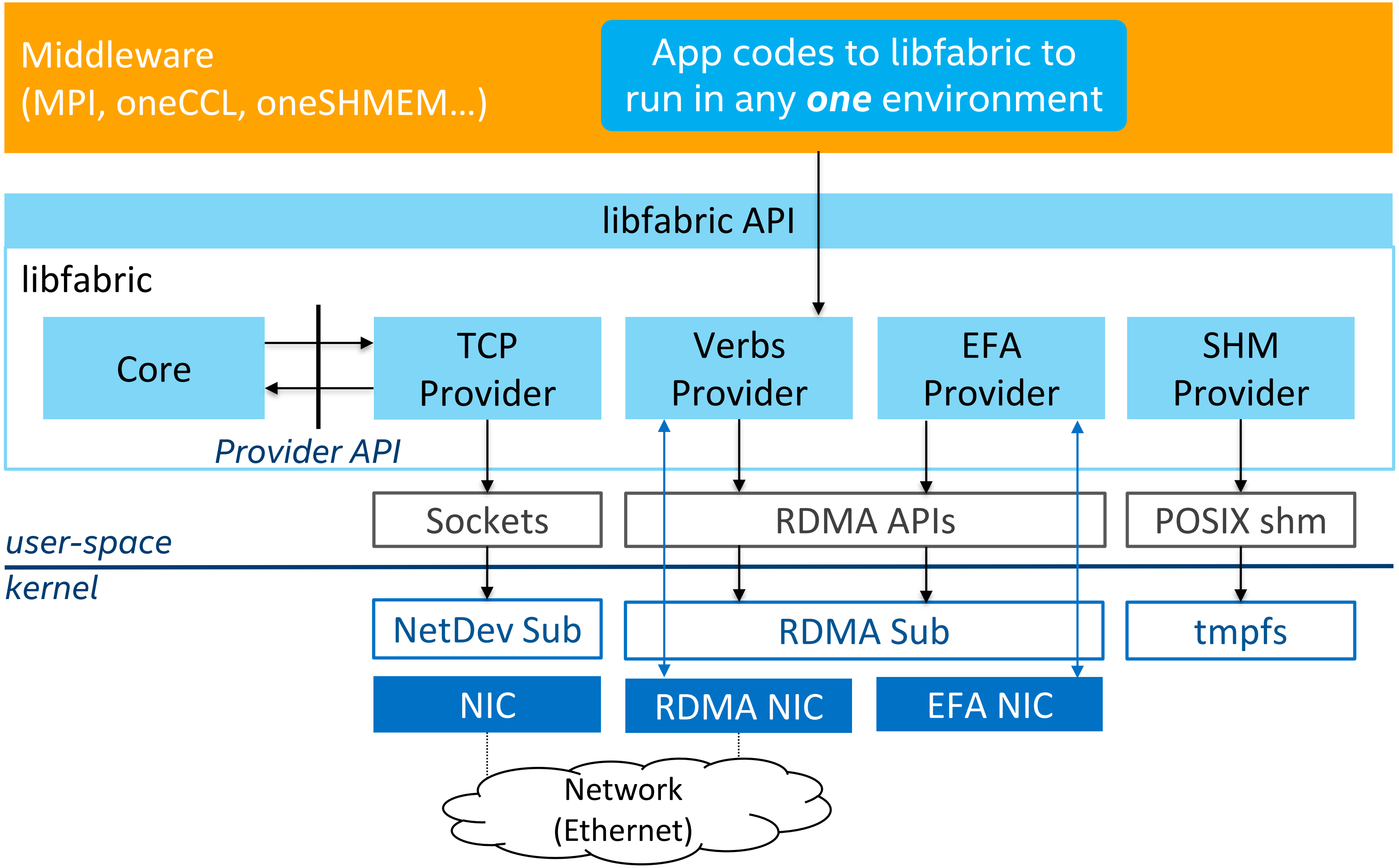
Sean Hefty

Intel Corporation

libfabric *Classic* Architecture

Illustrative Components





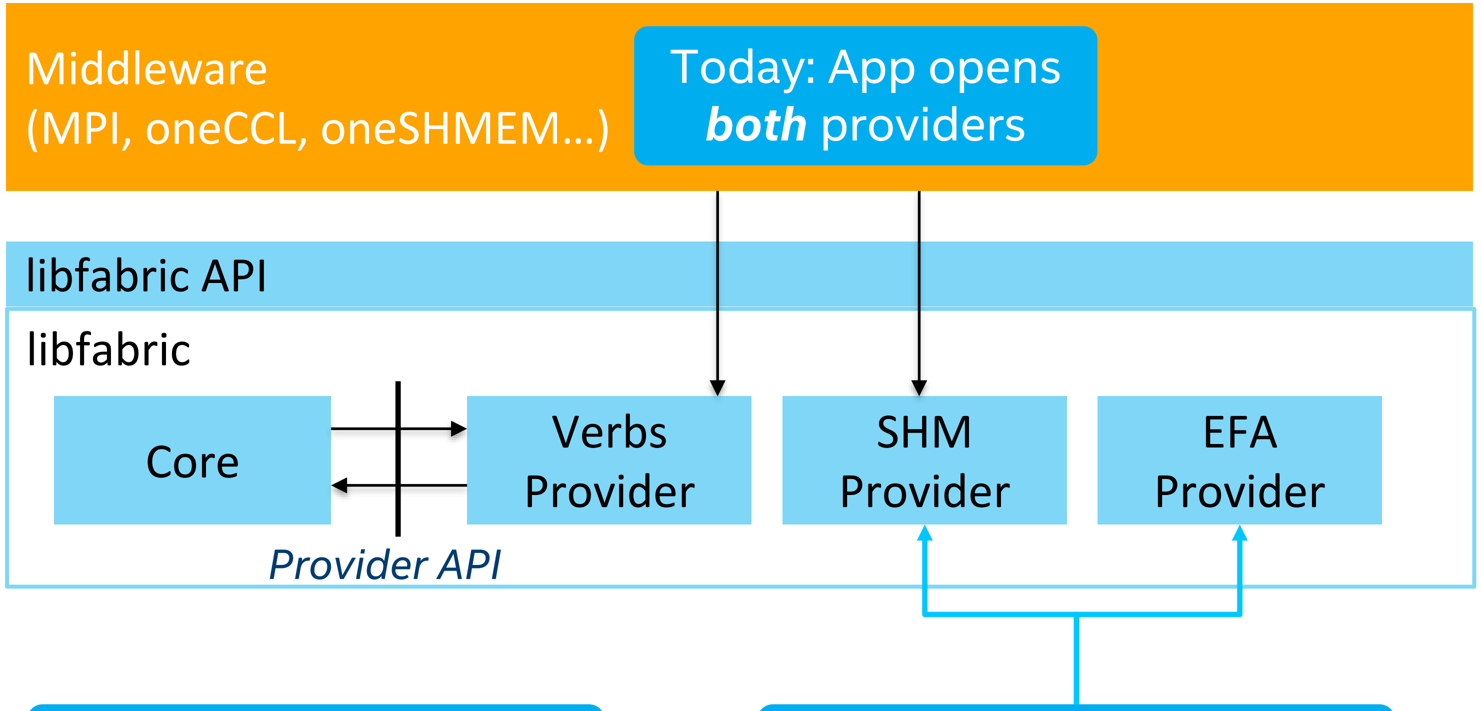
Using Multiple Providers

Example: Shared Memory + Network Provider

Legend
SW call path
Middleware
libfabric

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



2 providers

<u>Twice as many objects</u>

domains – register memory twice
endpoints – double receive buffers
completion queues – check two

Tightly coupled implementation

<u>Higher development costs</u>

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

App picks provider based on target address

Alternative: tight coupling between providers

Using Multiple Providers

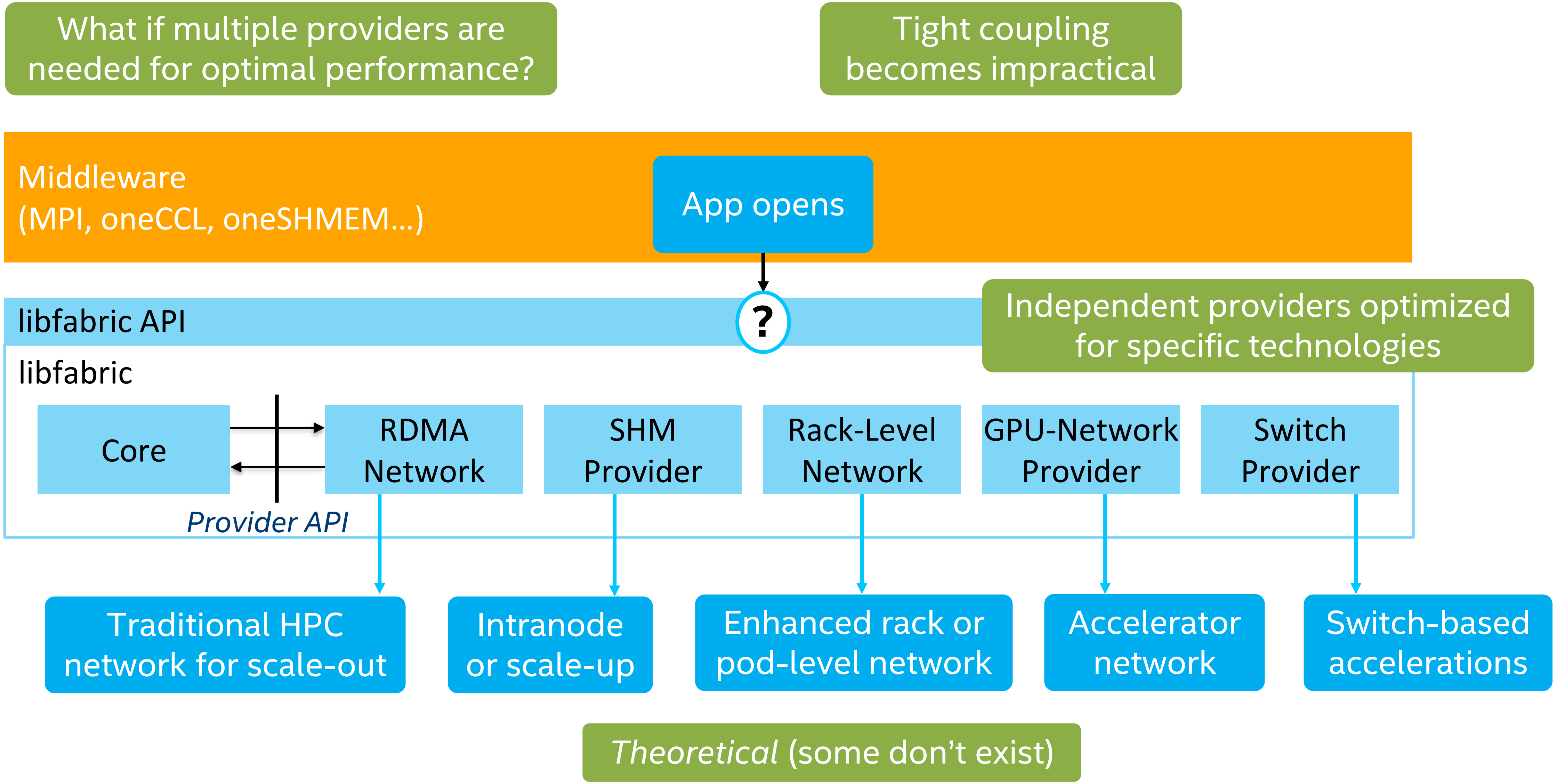
Complex, Theoretical Scenarios

Legend

SW call path

Middleware

libfabric



Using Multiple Providers

Example: Shared Memory + Network Provider

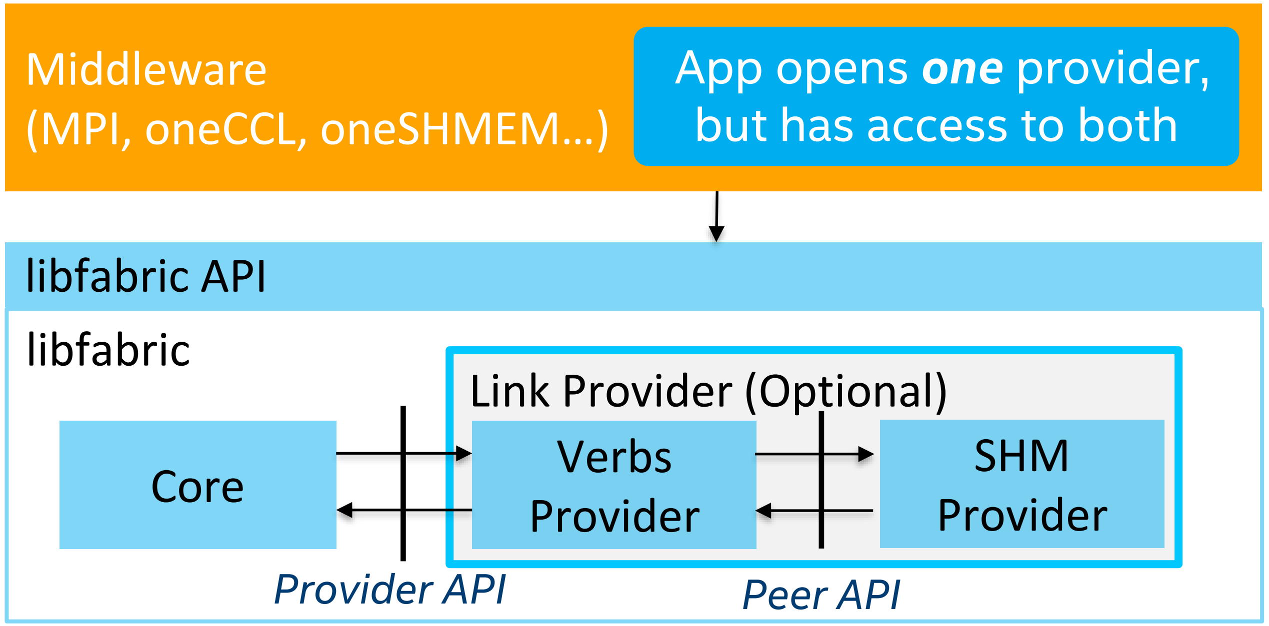
Legend

SW call path

Middleware

libfabric

Needed: *efficient* cooperation of *independent* providers → Peer APIs



Orchestration handled by core provider or (future) link provider

- 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

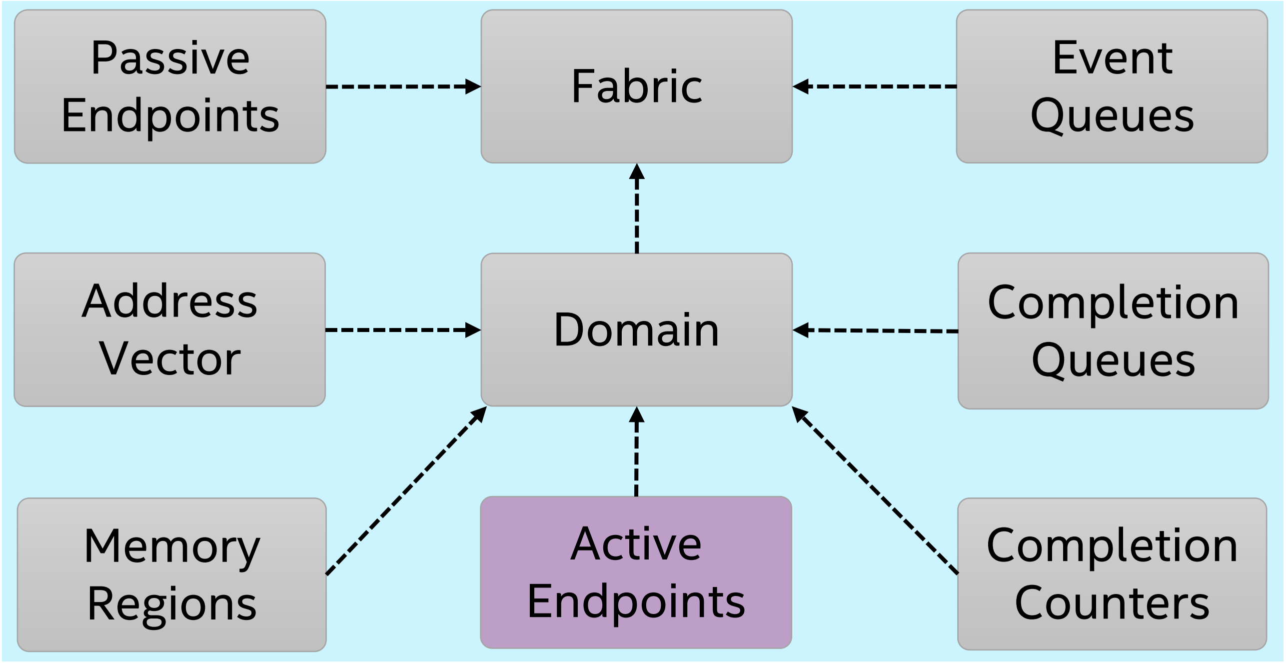
Generically combining a half dozen independently developed providers without losing performance. And, how, exactly, do you intend to accomplish this?

Review: libfabric API

Legend

Object dependency

API defines ***user*** interface to objects



Example: active endpont

```
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

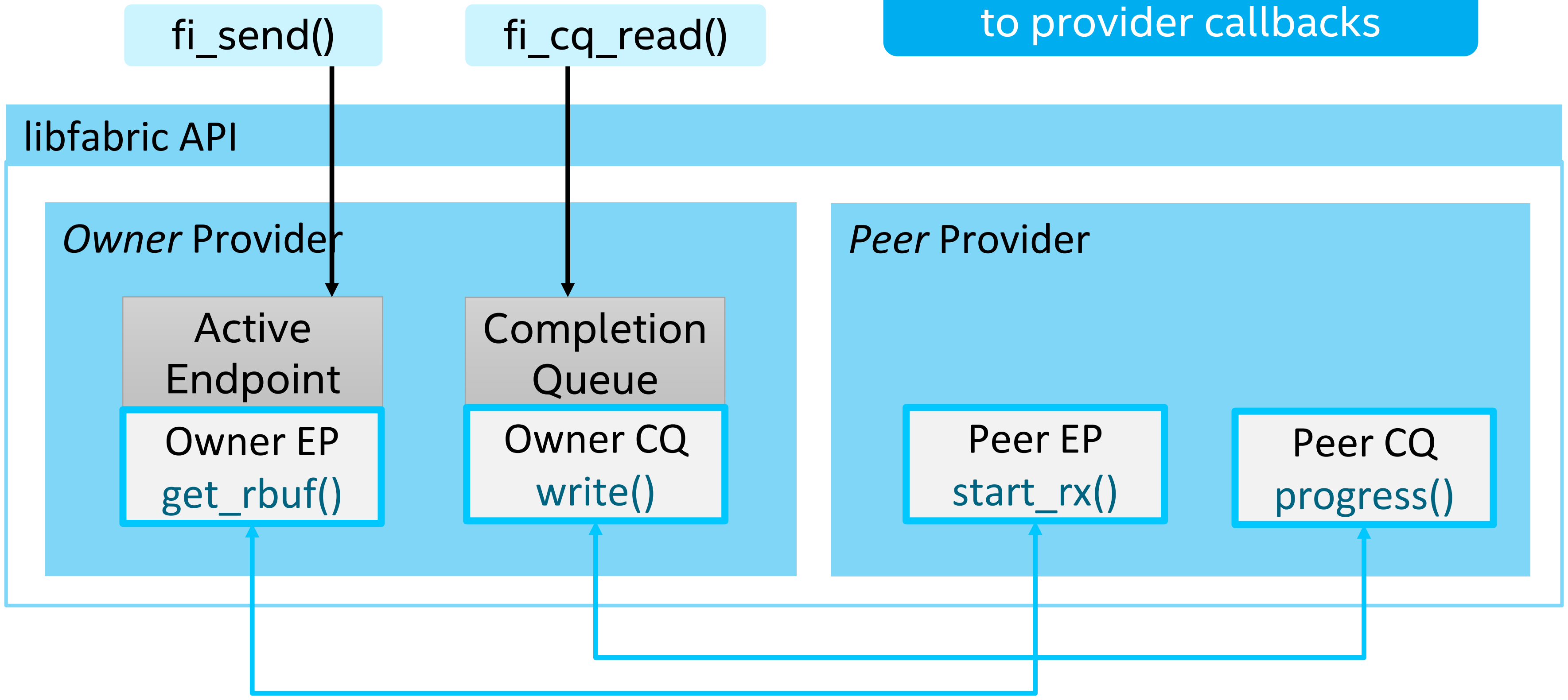
Sharable Fabric Identifiers (FIDs)

Legend

SW call path

Define objects to share between ***providers***

Peer objects define provider to provider callbacks



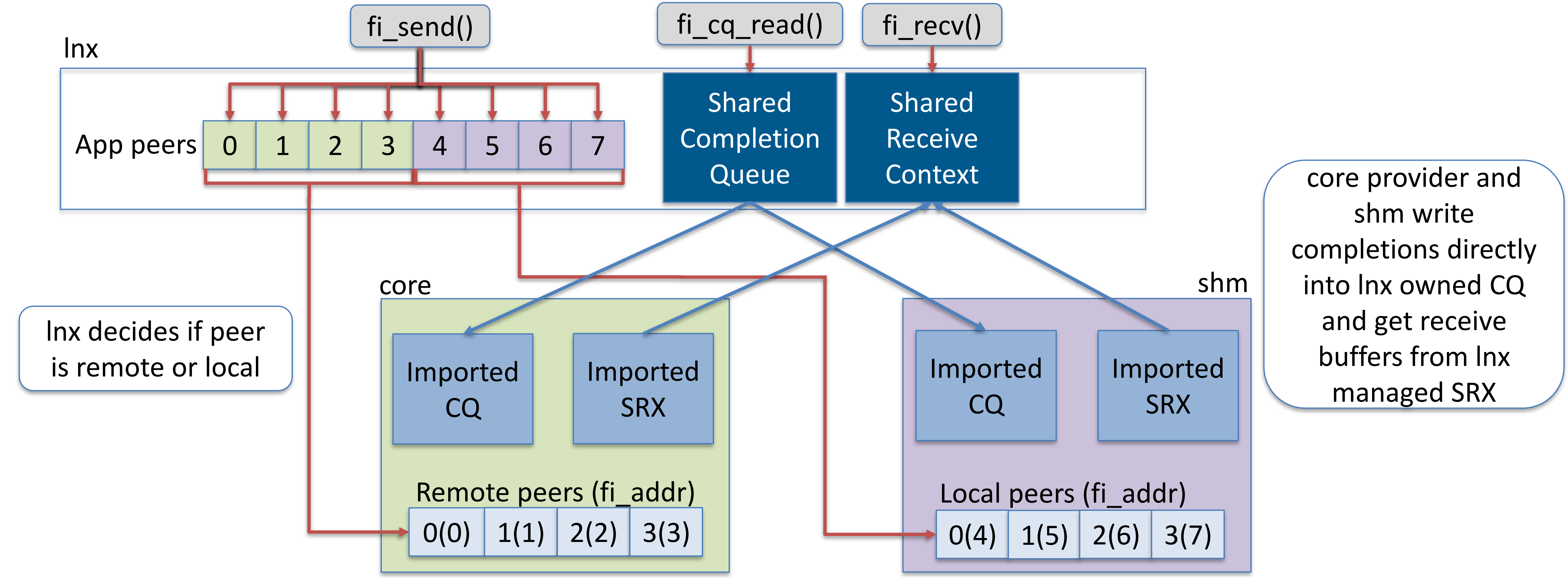
Conceptual example (details in next talks)

Don’t change that dial!
More details to follow!

How many people even
get this reference

THANK YOU

EXAMPLE - OWNER: LNX



lnx 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

SHARED COMPLETION QUEUE API

1. Owner allocates a peer cq and defines peer CQ write ops

```
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)();
        };
```

3. Peer calls imported peer_cq->owner_ops in order to write an entry to the shared CQ

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

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

                        struct fi_peer_cq_context {
                            struct fid_peer_cq *cq;
                        };
```

SHARED RECEIVE CONTEXT

```
struct fi_peer_srx_context {
     struct fid_peer_srx *srx;
};

struct fid_peer_srx {
     struct fid_ep ep_fid;
     struct fi_ops_srx_owner *owner_ops;
     struct fi_ops_srx_peer *peer_ops;
};

struct fi_ops_srx_owner {
     int (*get_msg)();
     int (*get_tag)();
     int (*queue_msg)();
     int (*queue_tag)();
     void (*free_entry)();
};

struct fi_ops_srx_peer {
     int (*start_msg)();
     int (*start_tag)();
     int (*discard_msg)();
     int (*discard_tag)();
};
```

1. Owner creates peer_srx_context and sets owner ops

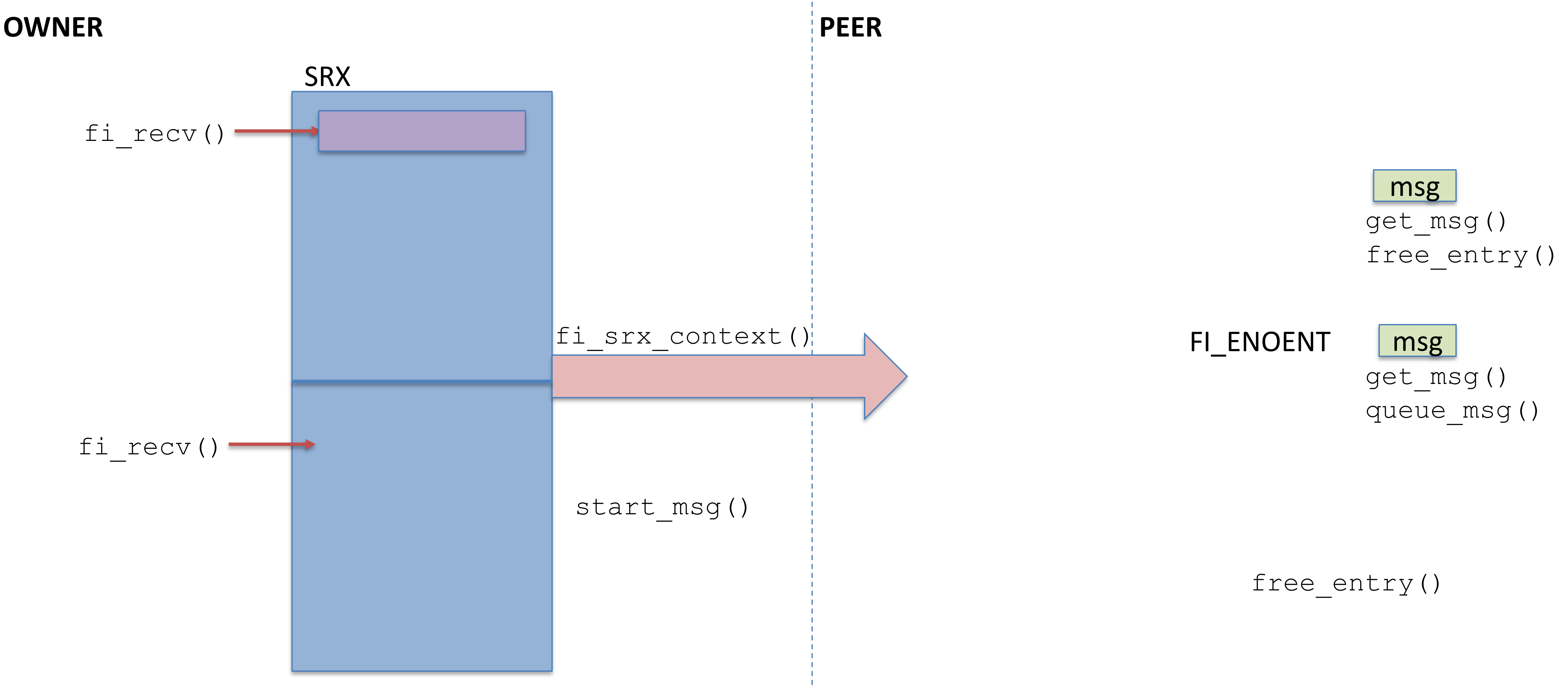
2. Owner imports SRX into peer by calling fi_srx_context passing in the peer_cq via context indicating a peer with attr->flags | FI_PEER. Peer sets peer_ops

```
fi_srx_context(peer_domain, &attr, &srx_fid, peer_srx_context);
```

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

Owner calls peer ops to start and discard unexpected messages

EXAMPLE SRX FLOW



2023 OFA Virtual Workshop

THANK YOU

Alexia Ingerson

Intel Corporation

OPENFABRICS ALLIANCE

2023 OFA Virtual Workshop

OFI PROVIDER FOR COLLECTIVE OFFLOAD

Jianxin Xiong

Intel Corporation

OFI COLLECTIVE API

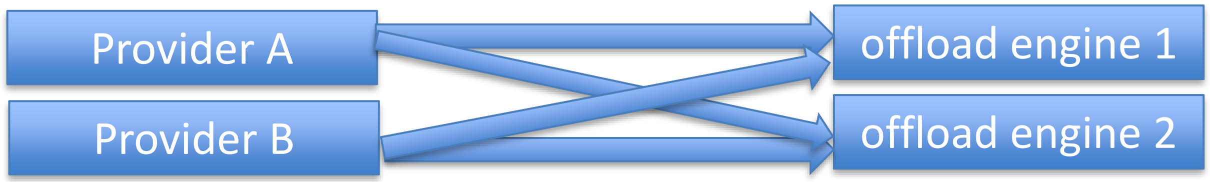
- **API summary**
  - Asynchronous
  - Defined in `<rdma/fi_collective.h>`
  - Supported ops: barrier, broadcast, alltoall, allreduce, allgather, reduce, reduce_scatter, scatter, gather
  - Wrapper functions: `fi_barrier(), fi_broadcast(), ……`
  - Collective groups: av_set
    - A set of addresses (`fi_addr_t`) 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**

```
struct fid_ep {
    ……
    struct fi_ops_collective collective;
}
```

IMPLEMENTATION CONSIDERATIONS

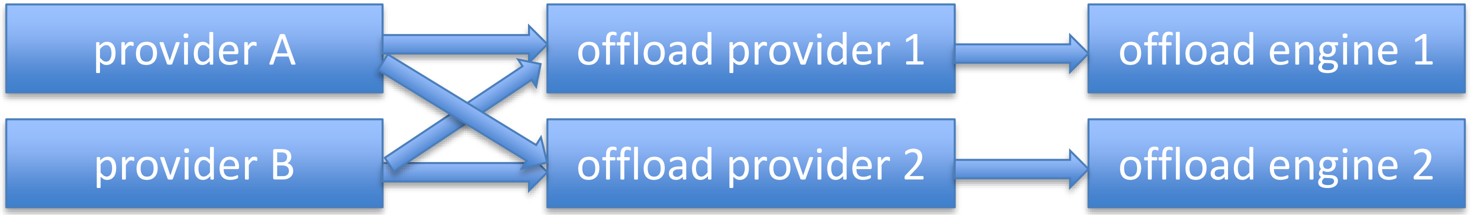
- **Goal: Efficiently enable multiple providers over multiple collective offload engines**



  - 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**

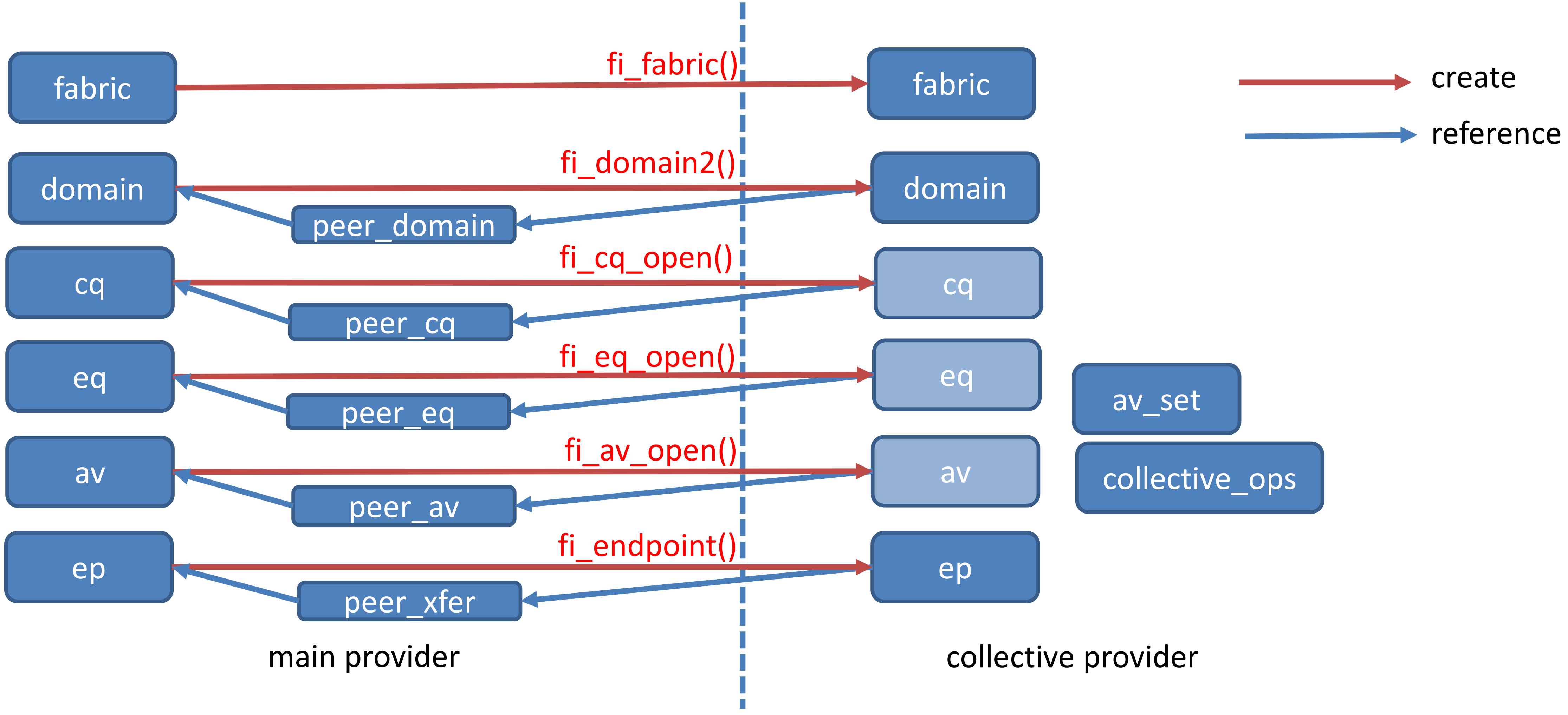
COLLECTIVE OFFLOAD WITH PEER PROVIDER

- **Implement a collective-only provider for each offload engine**

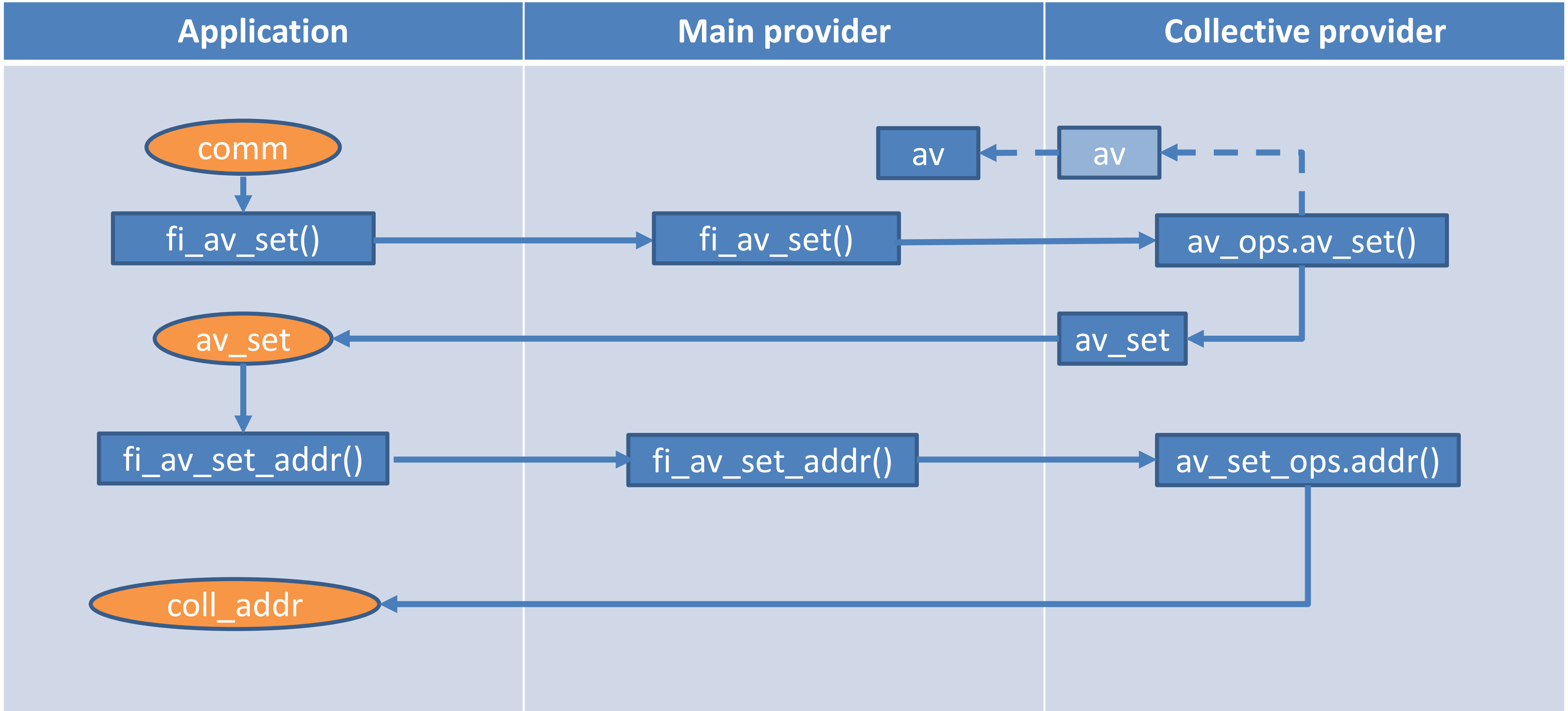


- Act as a peer provider to the “main” provider
- The main provider shares necessary data structure (domain, cq, eq, av, etc) via the peer-provider API
  - Eliminate the needs of creating duplicated queues / tables
  - The collective provider reports completions / events directly to the main provider
- 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

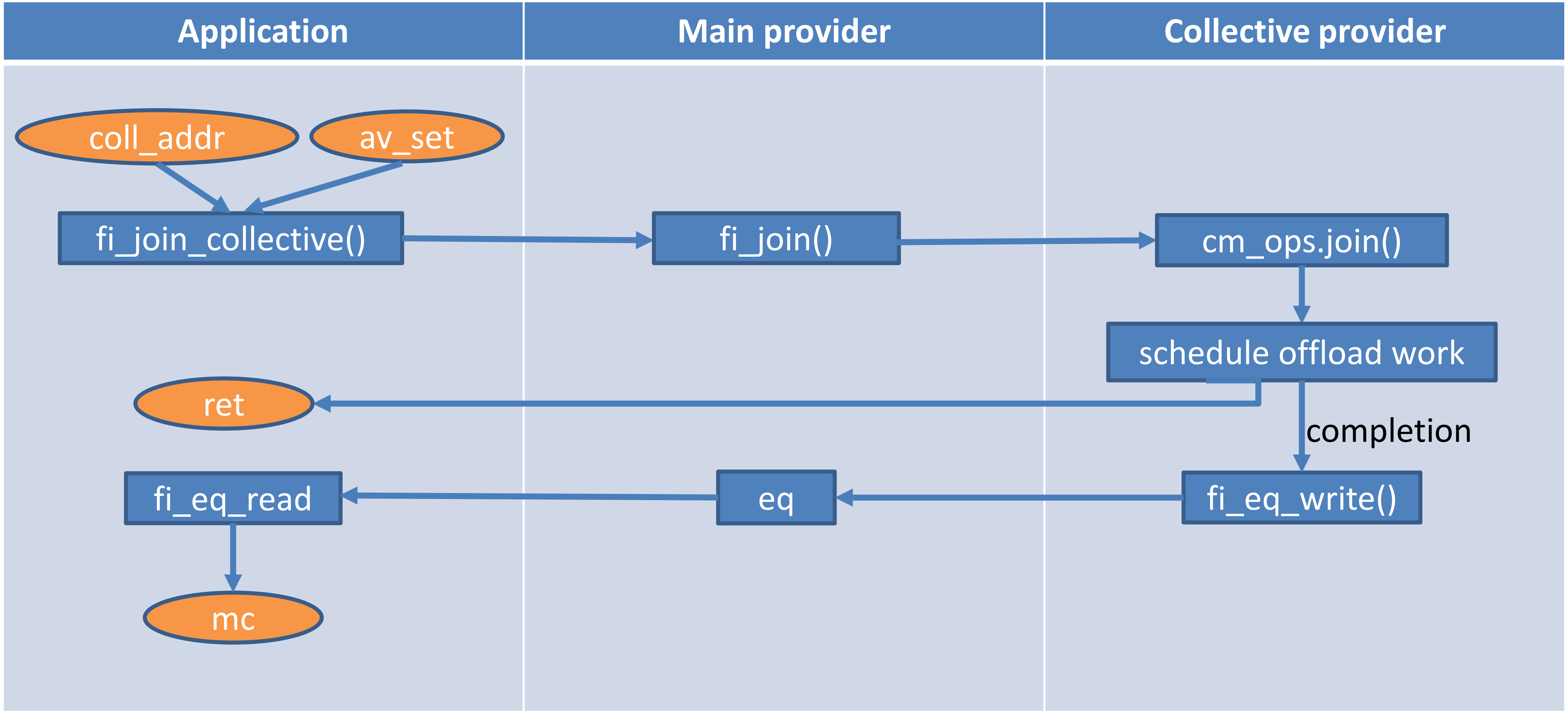
DESIGN OVERVIEW



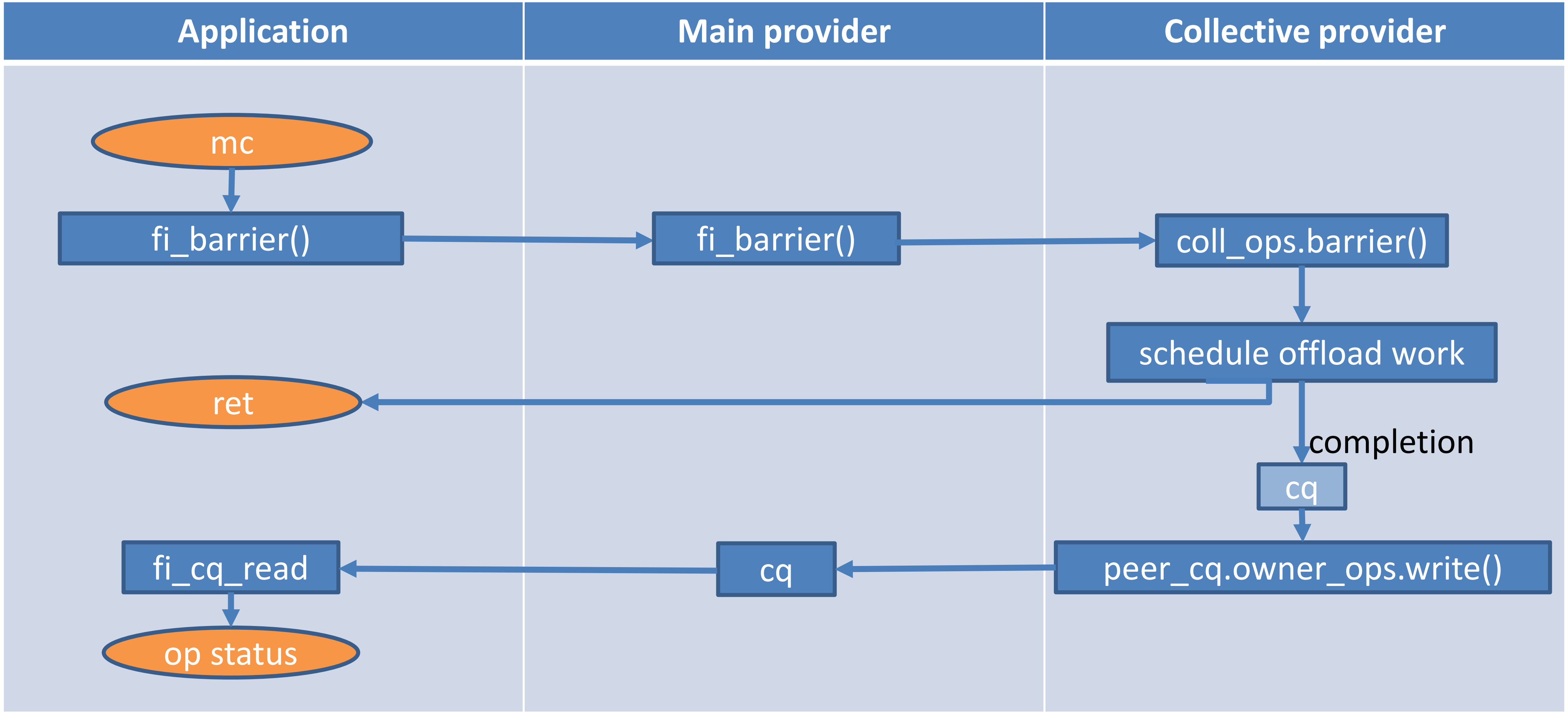
COLLECTIVE GROUP CREATION



JOIN COLLECTIVE GROUP

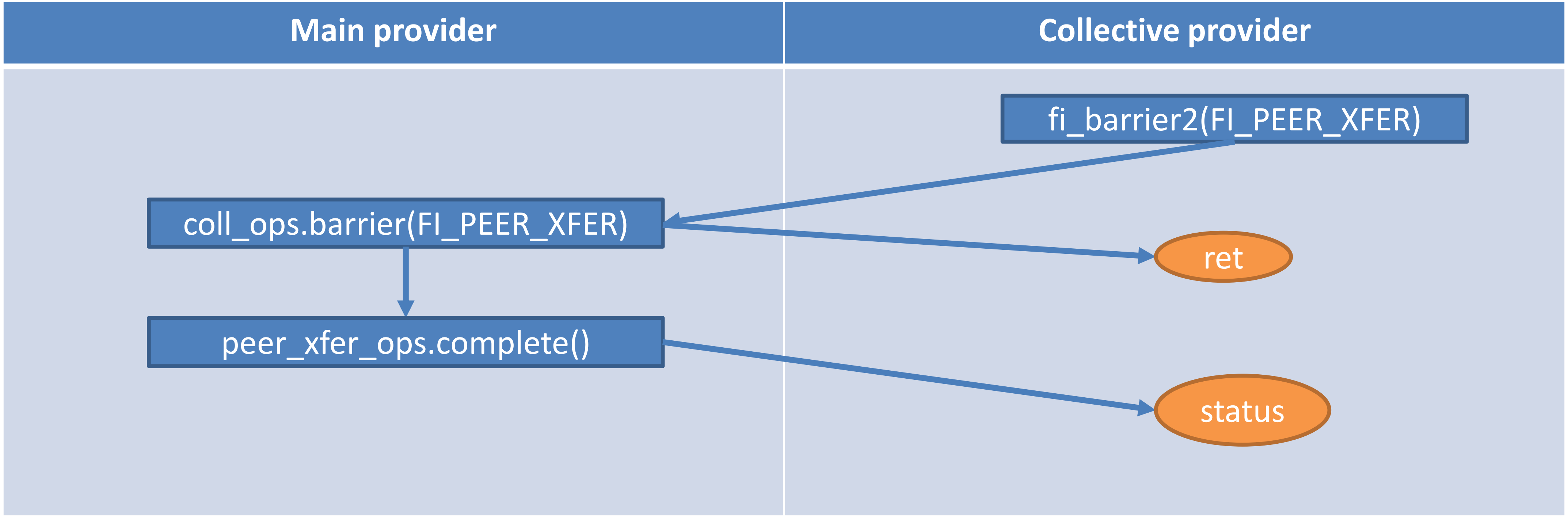


COLLECTIVE OPS



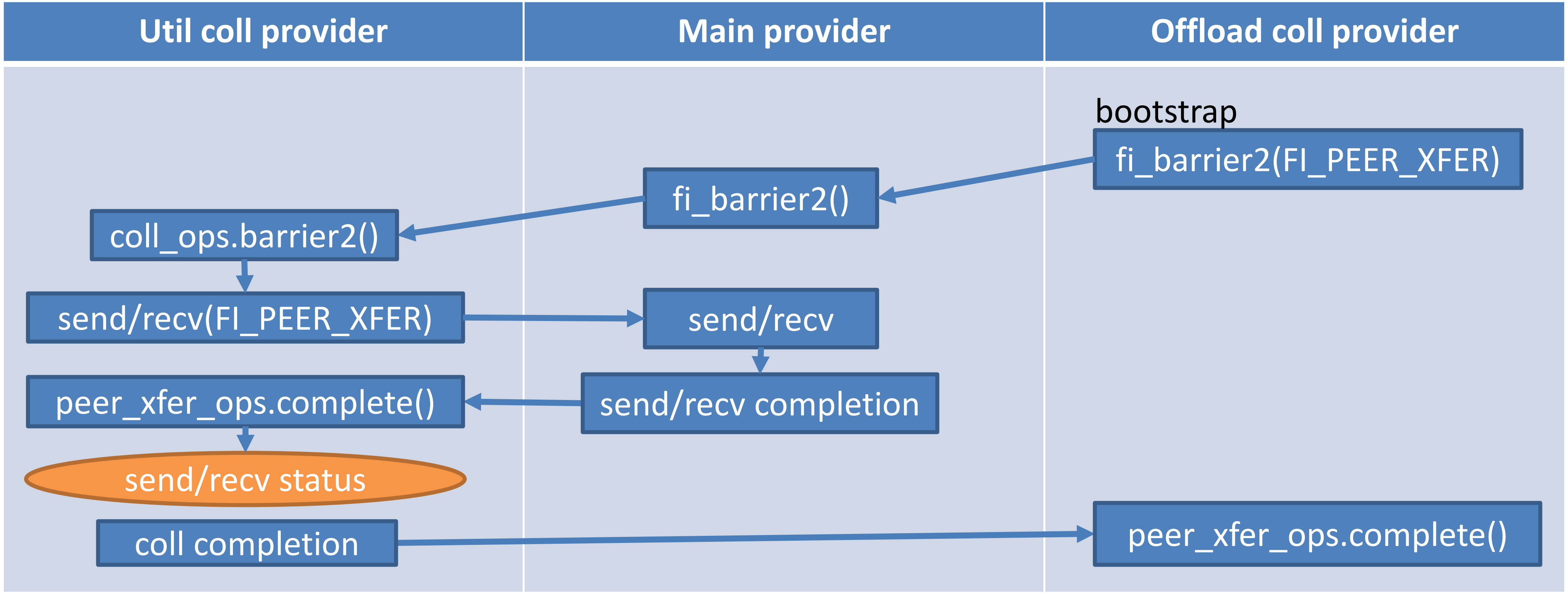
BOOTSTRAP COLLECTIVE

- **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**



UTILITY COLLECTIVE PROVIDER

- **Pt2pt based collectives can be moved to its own provider**



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 rxm 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.**

2023 OFA Virtual Workshop

THANK YOU

Jianxin Xiong

Intel Corporation