OpenFabrics Interfaces

libfabric Tutorial

Overview

- High-level Architecture
- Low-level Interface Design
- Simple Ping-pong Example
- Advanced MPI Usage
- SHMEM Usage

Overview

- This tutorial covers libfabric as of the v1.1.1 release
- Future versions might look a little different, but the v1.1 interface should remain available for a long time
- Man pages, source, presentations all available at:
  - http://ofiwg.github.io/libfabric/
  - https://github.com/ofiwg/libfabric
- Code on slides deliberately omits error checking for clarity

Developer Note

- libfabric supports a sockets provider
  - Allows it to run on most Linux systems
  - Includes virtual Linux environments
  - Also runs on OS X
    - brew install libfabric

Acknowlegements

Contributors to this tutorial not present today:

- Bob Russell, University of New Hampshire
- Sayantan Sur, Intel
- Jeff Squyres, Cisco

High-Level Architecture

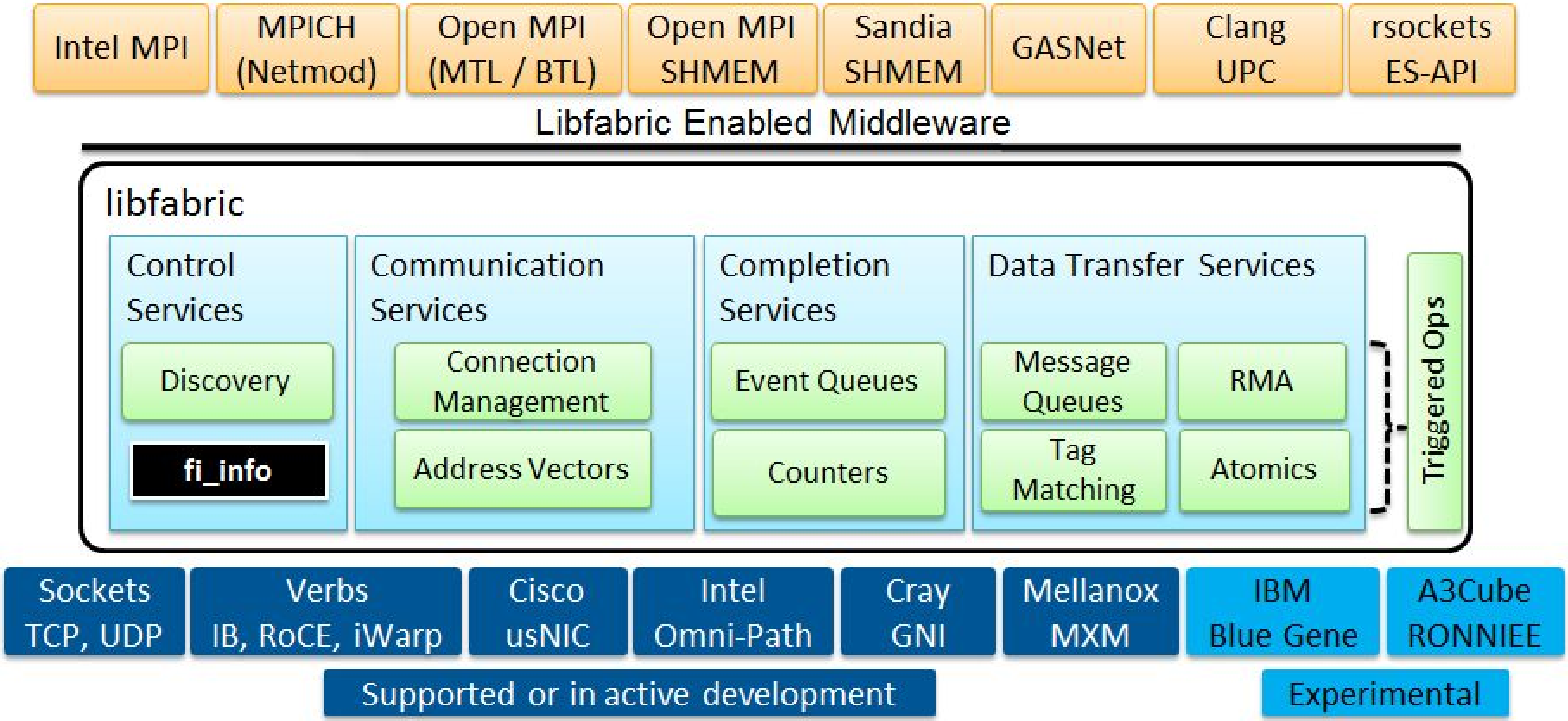
High-Level Architecture

- ➢ Interfaces and Services
- ❑ Object-Model
- ❑ Communication Models
- ❑ Endpoints

Design Guidelines

- Application driven API
- Low-level fabric services abstraction
- Extensibility built into interface
- Optimal impedance match between applications and underlying hardware
    - Minimize software overhead
    - Maximize scalability
- Implementation agnostic

Architecture



Control Services

- Discover information about types of fabric services available
- Identify most effective ways of utilizing a provider
- Request specific features
- Convey usage model to providers

Communication Services

- Setup communication between processes
- Support connection-oriented and connectionless communication
- Connection management targets ease of use
- Address vectors target high scalability

Completion Services

- Asynchronous completion support
- Event queues for detailed status
  - Configurable level of data reported
  - Separation between control versus data operations
- Low-impact counters for fast notification

Data Transfer Services

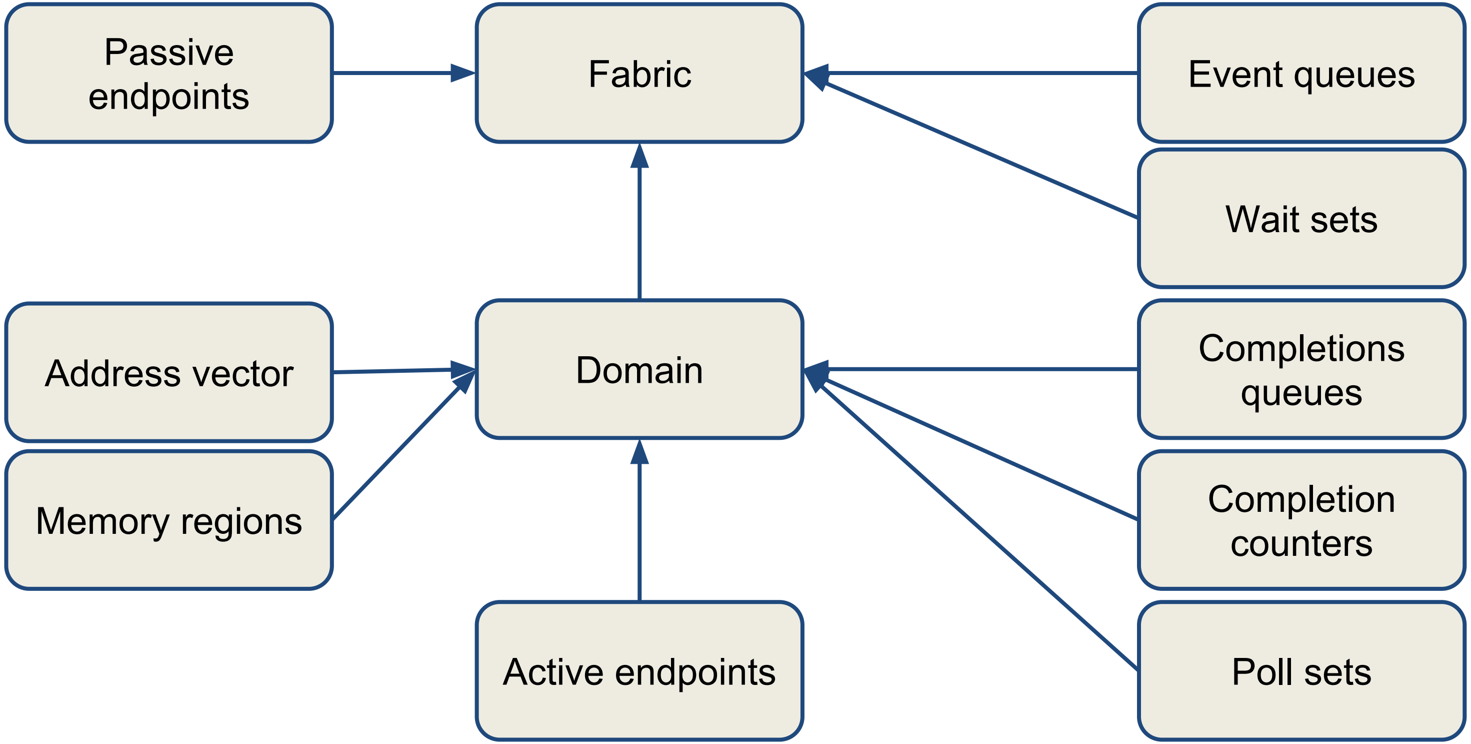
Supports different communication paradigms:

- Message Queues - send/receive FIFOs
- Tag Matching - steered message transfers
- RMA - direct memory transfers
- Atomics - direct memory manipulation

High-Level Architecture

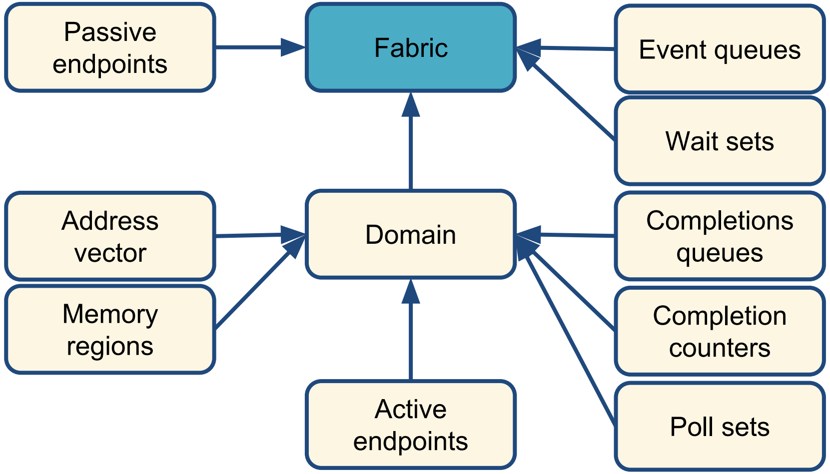
- ✓ Interfaces and Services
- ➢ Object-Model
- ❑ Communication Models
- ❑ Endpoints

Object-Model



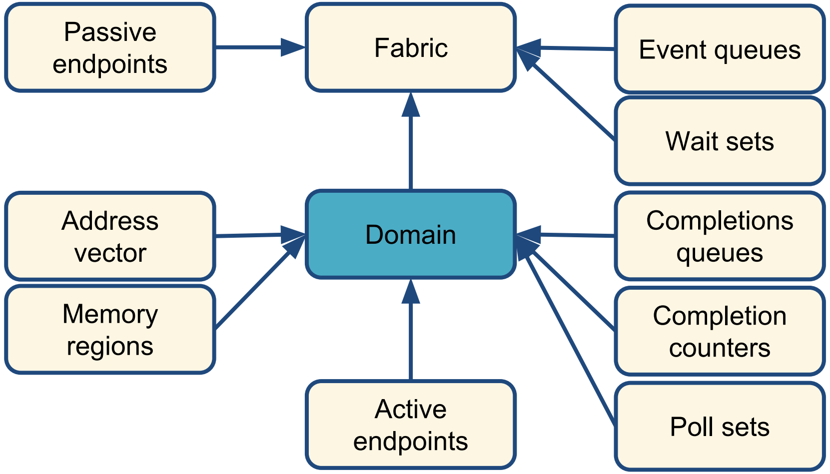
Fabric Object

- Represent a single physical or virtual network
- Shares network addresses
- May span multiple providers
- Future: topology information



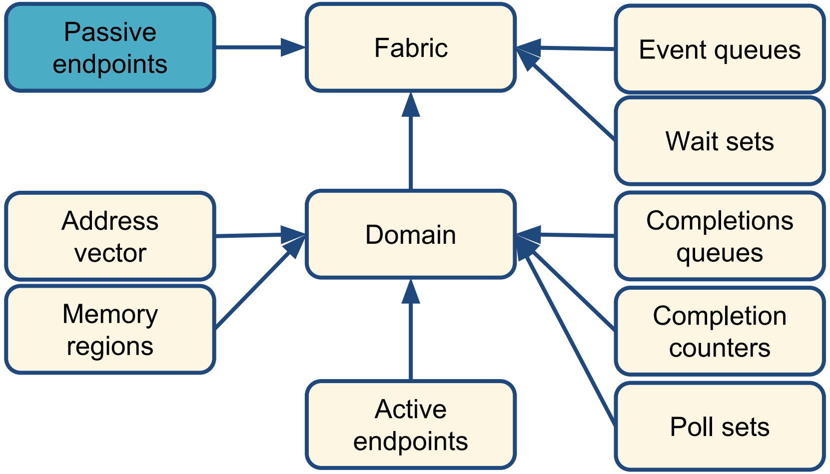
Domain Object

- Logical connection into a fabric
- Physical or virtual NIC
- Boundary for associating fabric resources



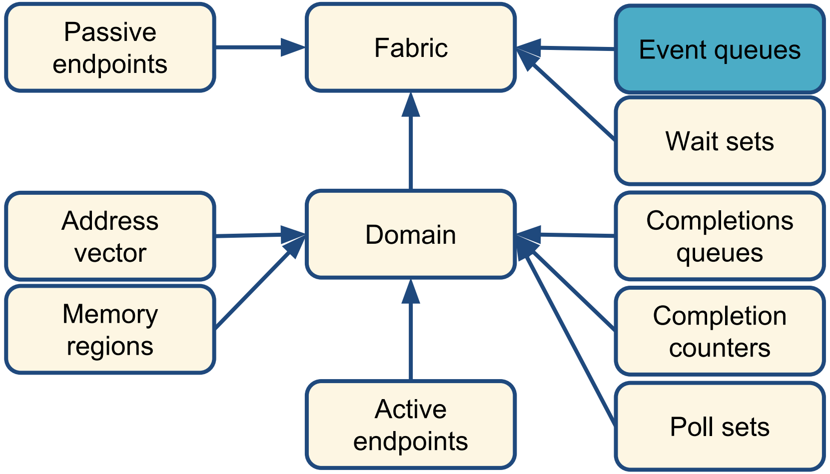
Passive Endpoint

- Used by connection-oriented protocols
- Listens for connection requests
- Often map to software constructs
- Can span multiple domains



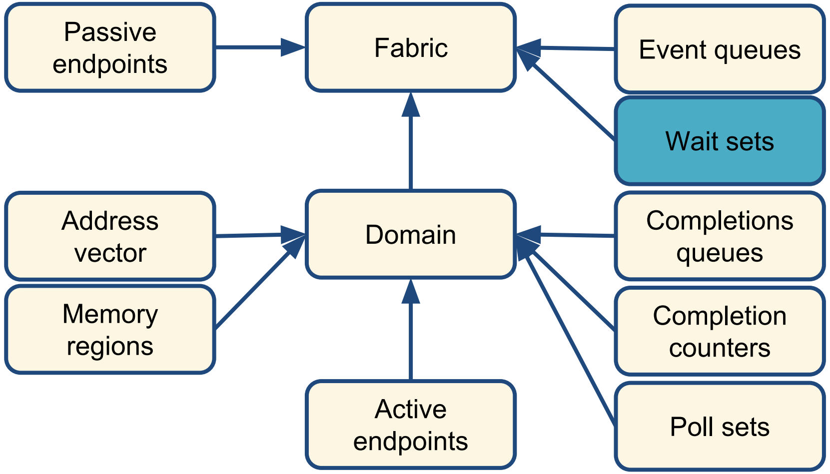
Event Queue

- Report completion of asyn-chronous control operations
- Report error and other notifications
  - Explicitly or implicitly subscribe for events
- Often mix of HW and SW support
- Usage designed for ease of use



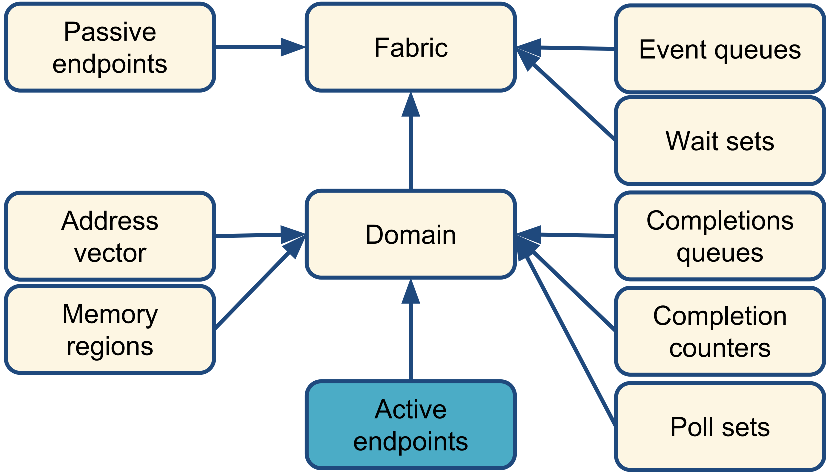
Wait Set

- Optimized method of waiting for events across multiple event queues, completion queues, and counters
- Abstraction of wait object(s)
  - Enables platform specific, high-performance wait objects
  - Uses single wait object when possible



Active Endpoint

- Data transfer communication portal
- Identified by fabric address
- Often associated with a single NIC
  - Hardware Tx/Rx command queues
- Support onload, offload, and partial offload implementations

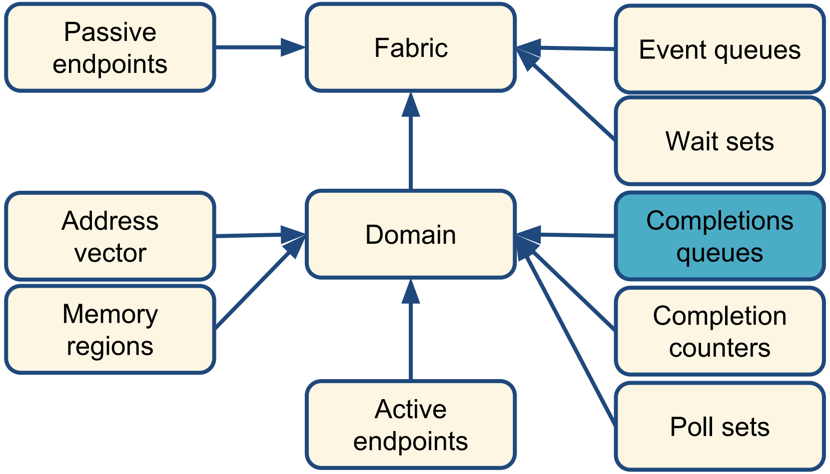


Active Endpoint Types

- `FI_EP_DGRAM`
  - Unreliable datagram
- `FI_EP_MSG`
  - Reliable, connected
- `FI_EP_RDM`
  - Reliable Datagram Message
  - Reliable, unconnected

Completion Queue

- Higher performance queues for data transfer completions
- Associated with a single domain
  - Often mapped to hardware resources
- Optimized to report successful completions
  - User-selectable completion format
  - Detailed error completions reported ‘out of band’

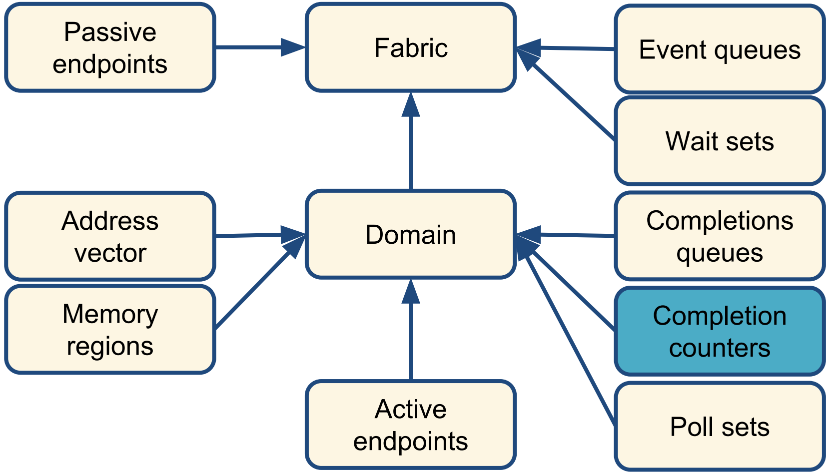


Remote CQ Data

- Application data written directly into remote completion queue
  - InfiniBand immediate data
- Support for up to 8 bytes
  - Minimum of 4 bytes, if supported

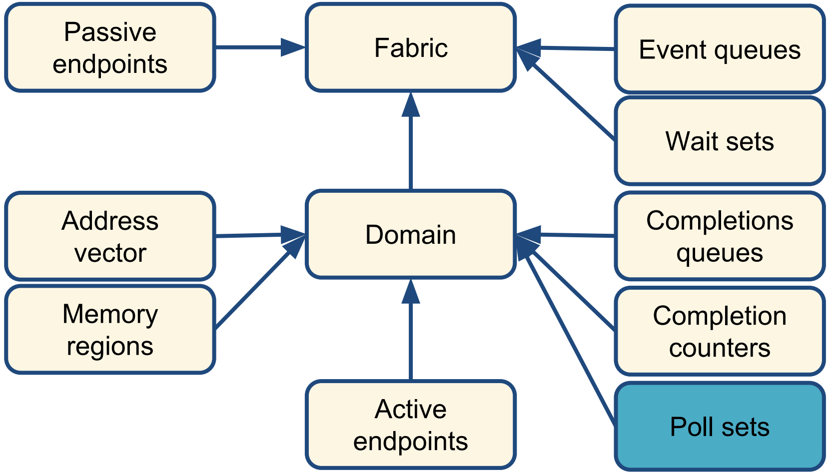
Counters

- Lightweight completion mechanism for data transfers
- Report only number of successful/error completions



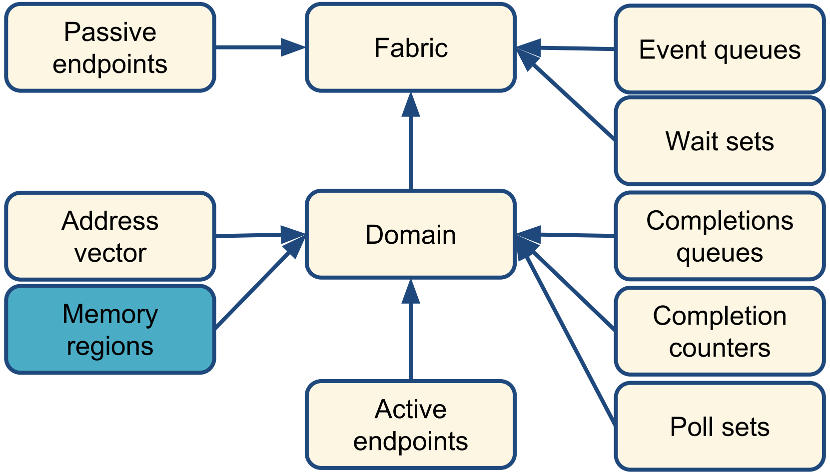
Poll Set

- Designed for providers that use the host processor to progress data transfers
  - Allows provider to use application thread
- Allows driving progress across all objects assigned to a poll set
  - Can optimize where progress occurs



Memory Region

- Local memory buffers exposed to fabric services
- Permissions control access
- Focused on desired application usage, with support for existing hardware
  - Registration of locally used buffers

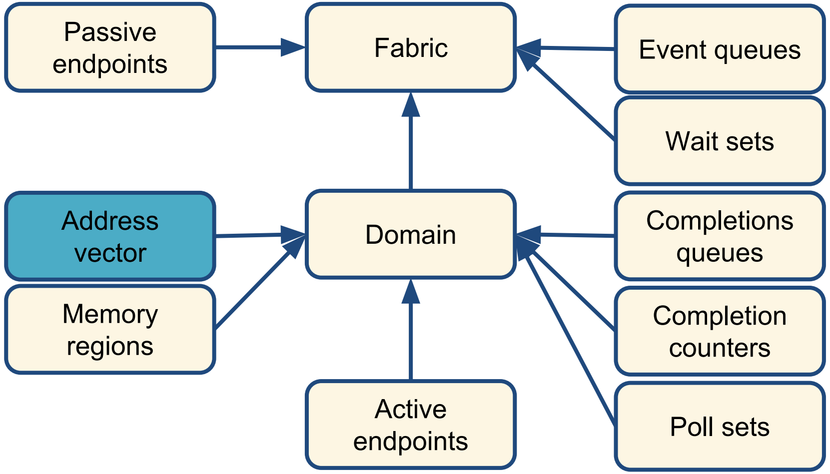


Memory Registration Modes

- `FI_MR_BASIC`
  - MR attributes selected by provider
  - Buffers identified by virtual address
  - Application must exchange MR parameters
- `FI_MR_SCALABLE`
  - MR attributes selected by application
  - Buffers accessed starting at address 0
  - Eliminates need to exchange MR parameters

Address Vector

- Store peer addresses for connectionless endpoints
- Map higher level addresses to fabric specific addresses
- Designed for high scalability
  - Enable minimal memory footprint
  - Optimized address resolution
  - Supports shared memory



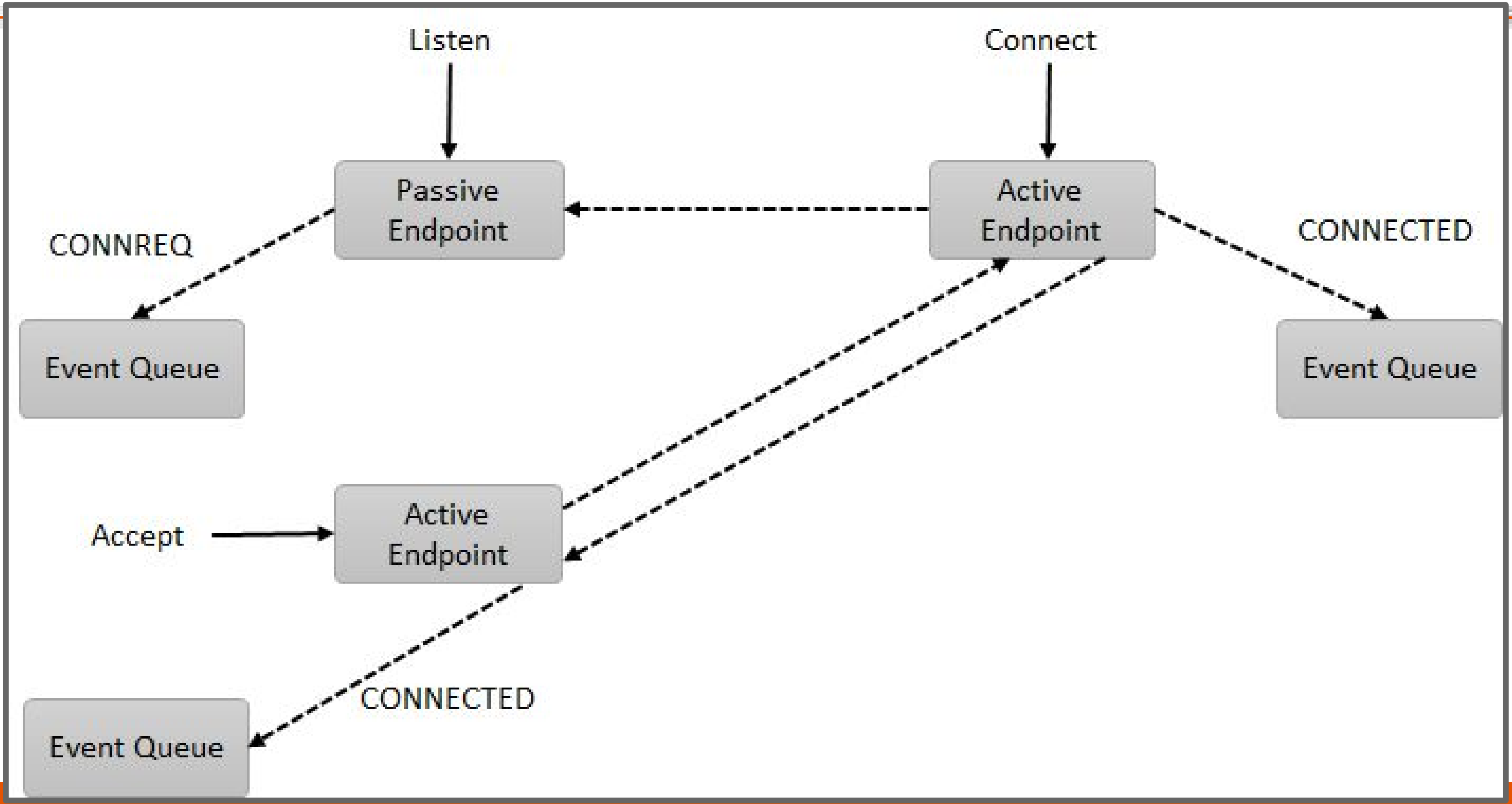
Address Vector Types

- `FI_AV_MAP`
  - Peers identified using a 64-bit **`fi_addr_t`**
  - Provider can encode fabric address directly
  - Enables direct mapping to hardware commands
- `FI_AV_TABLE`
  - Peers identified using an index
  - Minimal application memory footprint (0!)
  - May require lookup on each data transfer

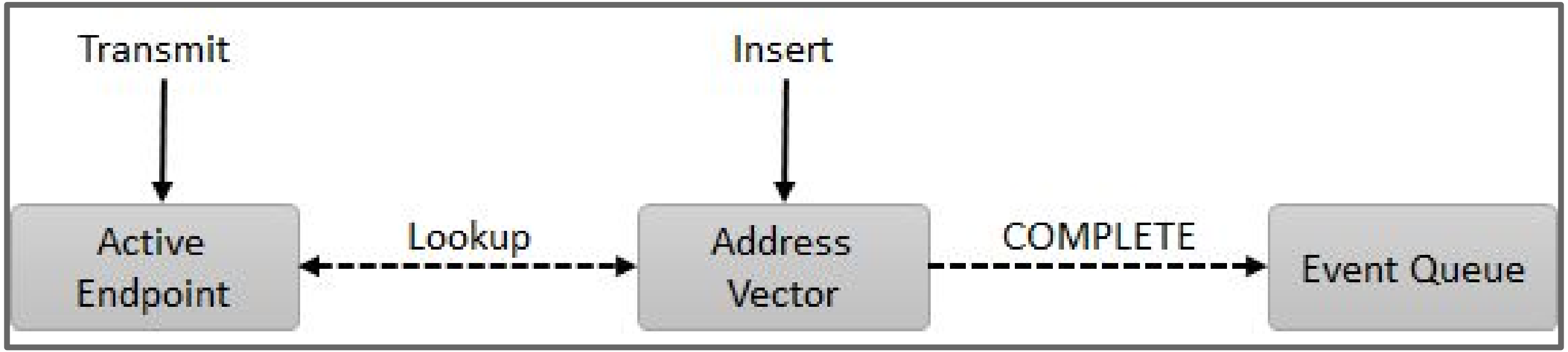
High-Level Architecture

- ✓ Services
- ✓ Object-Model
- ➢ Communication Models
- ❑ Endpoints

Connected Endpoints



Connectionless Endpoints

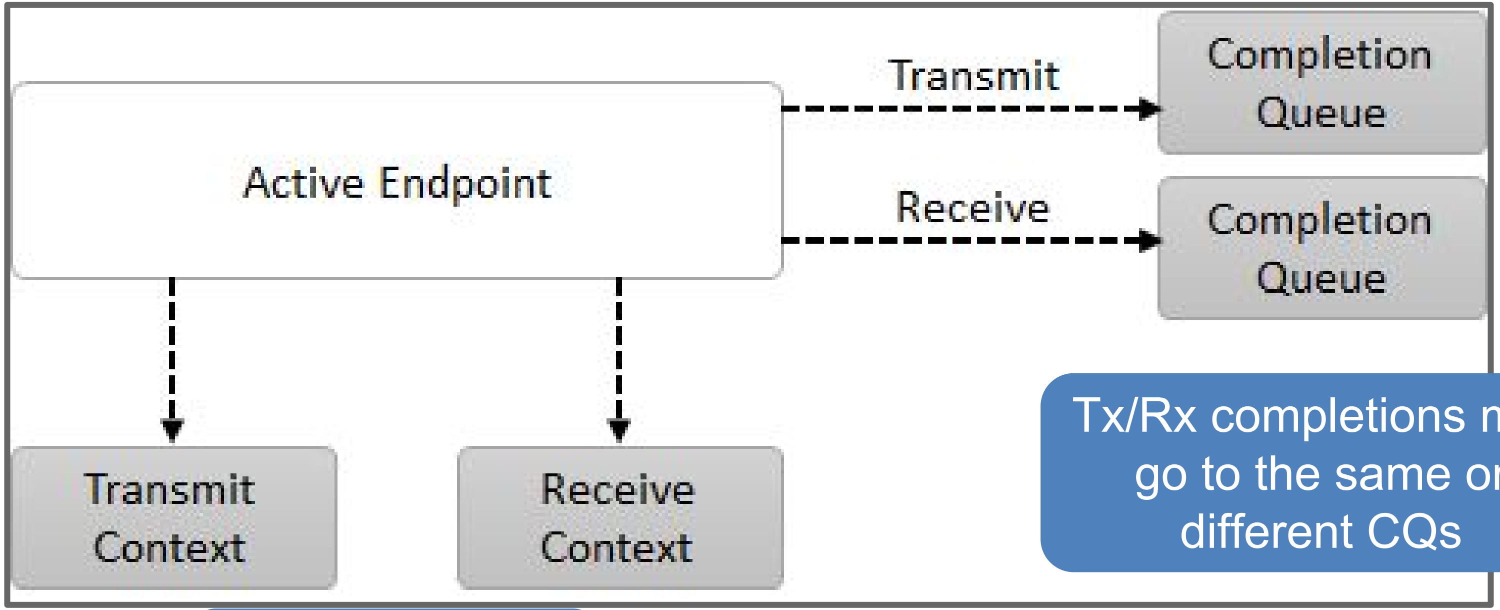


High-Level Architecture

- ✓ Services
- ✓ Object-Model
- ✓ Communication Models
- ➢ Endpoints

Basic Endpoint

Simple endpoint configuration

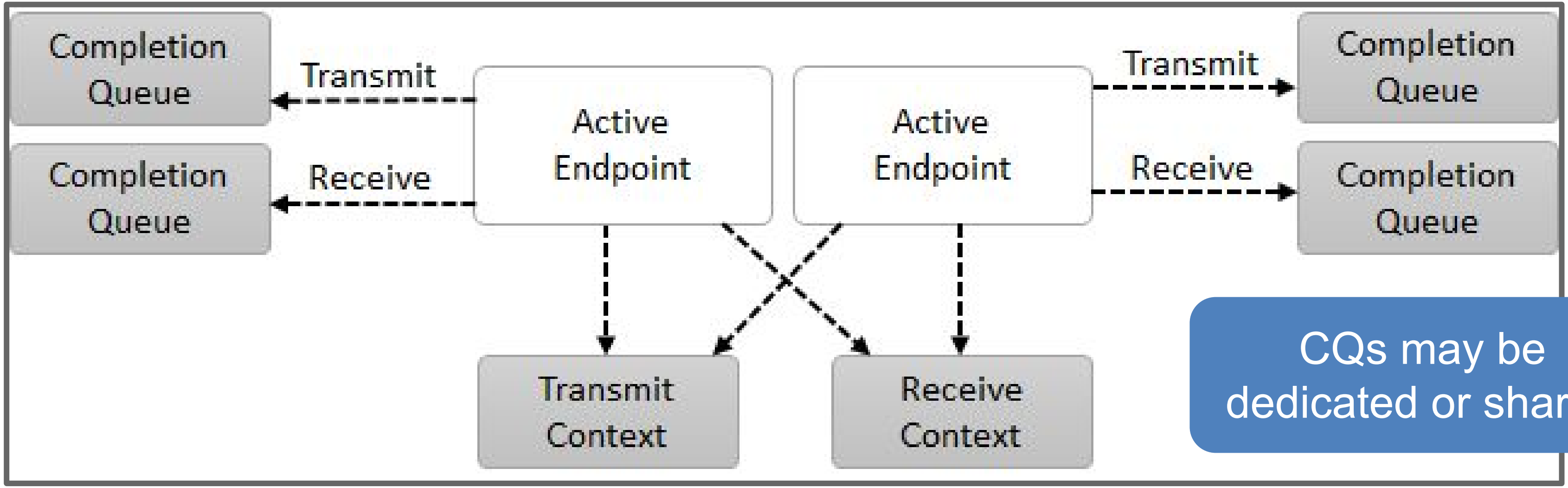


Tx/Rx completions may go to the same or different CQs

Tx/Rx command 'queues'

Shared Contexts

Endpoints may share underlying command queues

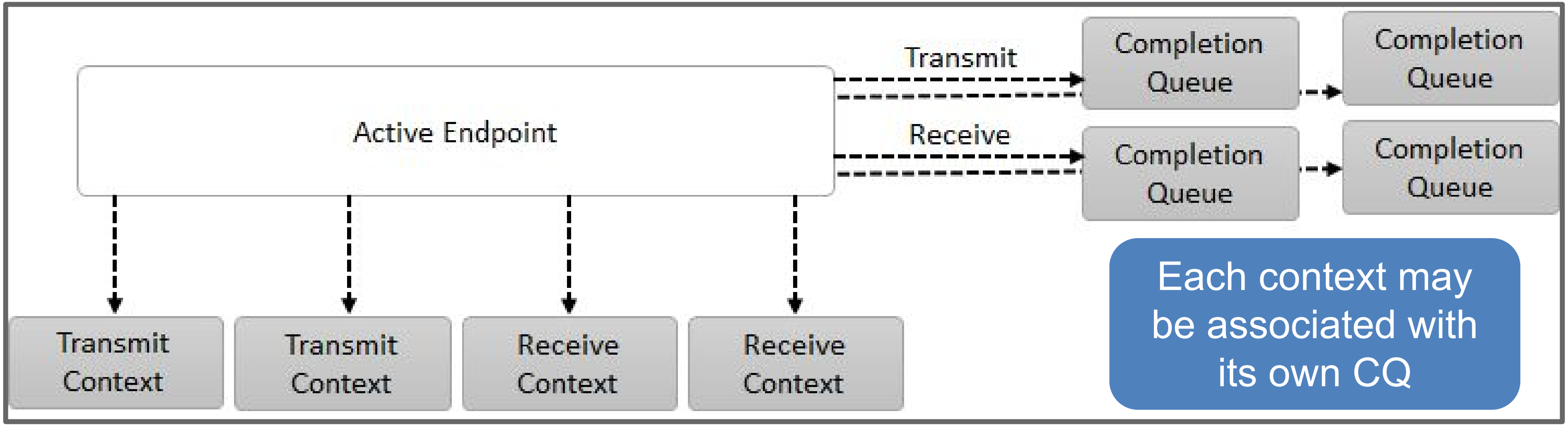


CQs may be dedicated or shared

Enables resource manager to select where resource sharing occurs

Scalable Endpoints

Targets lockless, multi-threaded usage



Each context may be associated with its own CQ

Single addressable endpoint with multiple command queues

Low-Level Interface Design

Low-Level Design

- ➢ Control Interface
- ❑ Capability and Mode Bits
- ❑ Attributes

Control Interface - getinfo

Modeled after getaddrinfo / rdma_getaddrinfo

```
int fi_getinfo(uint32_t version,
        const char *node,
        const char *service,
        uint64_t flags,
        struct fi_info *hints,
        struct fi_info **info);
```

Explicit versioning for forward compatibility

Hints used to filter output

Returns list of fabric info structures

Control Interface - fi_info

Primary structure used to query and configure interfaces

```
struct fi_info {
    struct fi_info          *next;
    uint64_t                caps;
    uint64_t                mode;
    ...
};
```

caps and ***mode*** flags provide simple mechanism to request basic fabric services

Control Interface - fi_info

Enables apps to be agnostic of fabric specific addressing (FI_FORMAT_UNSPEC) ..

```
struct fi_info {
    ...
    uint32_t        addr_format;
    size_t          src_addrlen;
    size_t          dest_addrlen;
    void            *src_addr;
    void            *dest_addr;
    ...
};
```

But also supports apps wanting to request a specific source or destination address

Apps indicate their address format for all APIs up front

Control Interface - fi_info

```
struct fi_info {
    ...
    fid_t                  handle;
    struct fi_tx_attr      *tx_attr;
    struct fi_rx_attr      *rx_attr;
    struct fi_ep_attr      *ep_attr;
    struct fi_domain_attr *domain_attr;
    struct fi_fabric_attr *fabric_attr;
};
```

Links to attributes related to the data transfer services that are being requested

Apps can use default values or request minimal attributes

Low-Level Design

- ✓ Control Interface
- ➢ Capability and Mode Bits
- ❑ Attributes

Capability Bits

Basic set of features required by application

- Desired features and services requested by application
- Primary - application must request to enable
- Secondary - application may request
  - Provider may enable if not requested

Providers enable capabilities if requested, or if it will not impact performance or security

Capabilities

Specify desired data transfer services (interfaces) to enable

- FI_MSG
- FI_RMA
- FI_TAGGED
- FI_ATOMIC

Overrides default capabilities; used to limit functionality

- FI_READ
- FI_WRITE
- FI_SEND
- FI_RECV
- FI_REMOTE_READ
- FI_REMOTE_WRITE

Capabilities

Receive oriented capabilities

- `FI_SOURCE`
  - Source address returned with completion data
  - Enabling may impact performance
- `FI_DIRECTED_RECV`
  - Use source address of an incoming message to select receive buffer
- `FI_MULTI_RECV`
  - Support for single buffer receiving multiple incoming messages
  - Enables more efficient use of receive buffers
- `FI_NAMED_RX_CTX`
  - Used with scalable endpoints
  - Allows initiator to direct transfers to a desired receive context

Capabilities

- `FI_RMA_EVENT`
  - Supports generating completion events when endpoint is the target of an RMA operation
  - Enabling can avoid sending separate message after RMA completes
- `FI_TRIGGER`
  - Supports triggered operations
  - Triggered operations are specialized use cases of existing data transfer routines
- `FI_FENCE`
  - Supports fencing operations to a given remote endpoint

Mode Bits

Provider requests to the application

- Requirements placed on the application
  - Application indicates which modes it supports
- Requests that an application implement a feature
  - Application may see improved performance
  - Cost of implementation by application is less than provider based implementation
  - Often related to hardware limitations

Mode Bits

- `FI_CONTEXT`
  - Application provides 'scratch' space for providers as part of all data transfers
  - Avoids providers needing to allocate internal structures to track requests
  - Targets providers that have a significant software component
- `FI_LOCAL_MR`
  - Provider requires that locally accessed data buffers be registered with the provider before being used
  - Supports existing iWarp and InfiniBand hardware

Mode Bits

- `FI_MSG_PREFIX`
  - Application provides buffer space before their data buffers
  - Typically used by provider to implement protocol headers
- `FI_ASYNC_IOV`
  - Indicates that IOVs must remain valid until an operation completes
  - Avoids providers needing to buffer IOVs
- `FI_RX_CQ_DATA`
  - Indicates that transfers which carry remote CQ data consume receive buffer space
  - Supports existing InfiniBand hardware

Low-Level Design

- ✓ Control Interface
- ✓ Capability and Mode Bits
- ➢ Attributes

Attributes

- Providers encode default sizes for allocated resources
- Administrator may override defaults
- Attributes reflect configured or optimal sizes
  - Not necessarily maximums
- Intent is to guide resource managers to allocate resources efficiently

Fabric Attributes

```
struct fi_fabric_attr {
	struct fid_fabric *fabric;
	char              *name;
	char              *prov_name;
	uint32_t          prov_version;
};
```

Return a reference to an already opened instance, if it exists

Framework will search for and select most recent provider version available

Domain Attributes

```
struct fi_domain_attr {
    struct fid_domain        *domain;
    char                     *name;
    enum fi_threading        threading;
    enum fi_progress         control_progress;
    enum fi_progress         data_progress;
    enum fi_resource_mgmt    resource_mgmt;
    ...
};
```

Specifies how resources (EPs, CQs, etc.) may be assigned to threads without needing locking

Indicates if application threads are used to progress operations

Indicates if provider will protect against queue overruns

Domain Attributes

```
struct fi_domain_attr {
    ...
    enum fi_av_type        av_type;
    enum fi_mr_mode        mr_mode;
    size_t                 mr_key_size;
    size_t                 cq_data_size;
    size_t                 cq_cnt;
    ...
};
```

Map or indexed address vector type

Basic or scalable memory registration

Range of MR key

Size of supported remote CQ data

Optimal number of CQs supported by domain

Domain Attributes

```
struct fi_domain_attr {
    ...
    size_t              ep_cnt;
    size_t              tx_ctx_cnt;
    size_t              rx_ctx_cnt;
    size_t              max_ep_tx_ctx;
    size_t              max_ep_rx_ctx;
    size_t              max_ep_stx_ctx;
    size_t              max_ep_srx_ctx;
};
```

Optimal endpoint resource constraints

Scalable and shared endpoint contexts

Endpoint Attributes

```
struct fi_ep_attr {
    enum fi_ep_type   type;
    uint32_t          protocol;
    uint32_t          protocol_version;
    size_t            max_msg_size;
    size_t            msg_prefix_size;
    ...
};
```

To ensure interoperability

Maximum transfer

If FI_PREFIX mode set

Endpoint Attributes

```
struct fi_ep_attr {
    ...
    size_t       max_order_raw_size;
    size_t       max_order_war_size;
    size_t       max_order_waw_size;
    uint64_t     mem_tag_format;
    size_t       tx_ctx_cnt;
    size_t       rx_ctx_cnt;
};
```

Delivery order of transport data

Tag matching support

Rx/Tx contexts

Tx/Rx Attributes

Can specify capability and mode bits per context

```
struct fi_tx_attr {
    uint64_t caps;
    uint64_t mode;
    uint64_t op_flags;
    uint64_t msg_order;
    uint64_t comp_order;
    size_t   inject_size;
    size_t   size;
    size_t   iov_limit;
    size_t   rma_iov_limit;
};
```

```
struct fi_rx_attr {
    uint64_t caps;
    uint64_t mode;
    uint64_t op_flags;
    uint64_t msg_order;
    uint64_t comp_order;
    size_t   total_buffered_recv;
    size_t   size;
    size_t   iov_limit;
};
```

Tx/Rx Attributes

- `op_flags`
  - Default flags to control operation
  - Apply to all operations where flags are not provided directly or are assumed by the call itself
- `size`
  - Minimum number of operations that may be posted to a context
  - Assumes each operation consumes the maximum amount of resources

Tx/Rx Attributes

- `inject_size`
  - *Injected* buffers may be re-used immediately on return from a function call
  - Related to `FI_INJECT` flag and `fi_inject()` call
  - Maximum size of an injected buffer
- `total_buffered_recv`
  - Total available space allocated by provider to buffer messages for which there is no matching receive
  - Handles *unexpected* messages

msg_order

- Order in which transport headers are processed
- [READ | WRITE | SEND] after [R | W | S]
- Determines how receive buffers are associated with transfers
- Necessary, but insufficient, for data ordering

comp_order

- Order in which completed requests are written to a completion object
  - FI_ORDER_NONE - no ordering defined
  - FI_ORDER_STRICT - ordered by processing
  - FI_ORDER_DATA - bytes are also written in order
- Order depends on communication type
  - Unreliable - all operations ordered
  - Reliable - ordered per remote endpoint

Simple Ping-pong Example

RDM Pingpong

Client-server test using reliable unconnected endpoints

1. Open an endpoint
2. Direct completions to selected queues
3. Setup an address vector
4. Send and receive messages

RDM Pingpong

```
struct fi_info *fi, *hints;
struct test_options opts;

int main(int argc, char **argv)
{
    init_test_options(&opts, argc, argv);
```

Assumes no failures!

Use command line to pass in source/destination address

RDM Pingpong

```
hints = fi_allocinfo();
hints->ep_attr->type = FI_EP_RDM;
hints->caps = FI_MSG;

if (opts.dest_addr) {
    /* "client" */
    fi_getinfo(FI_VERSION(1,1), opts.dest_addr,
               opts.dest_port, 0, hints, &fi);
```

Send/receive messages over RDM endpoint

Explicitly define version supported by app

Never use `FI_MAJOR_VERSION` or `FI_MINOR_VERSION` defines!

RDM Pingpong

```
} else {
    /* "server" */
    fi_getinfo(FI_VERSION(1,1), opts.src_addr,
                    opts.src_port, FI_SOURCE, hints, &fi);
}
```

Node and service parameters are local addresses

We assume both sides get the same fabric name and endpoint protocol

We could use the hints to force this

RDM Pingpong

fi_getinfo returns optimal options first

In the absence of any hints, provider returns attributes most suited to their implementation

```
fi_fabric(fi->fabric_attr, &fabric, NULL);

fi_domain(fabric, fi, &domain, NULL);
```

Open fabric and domain using returned defaults

RDM Pingpong

Create completion queue for transmit context

```
struct fi_cq_attr cq_attr = {};

cq_attr.wait_obj = FI_WAIT_NONE;
cq_attr.format = FI_CQ_FORMAT_CONTEXT;
cq_attr.size = fi->tx_attr->size;

fi_cq_open(domain, &cq_attr, &tx_cq, NULL);
```

We will never block waiting for a completion

Size the CQ the same as the transmit context

Only provide request context for each completion

RDM Pingpong

Create completion queue for receive context

Only adjust the size

```
cq_attr.size = fi->rx_attr->size;

fi_cq_open(domain, &cq_attr, &rx_cq, NULL);
```

Since this is a pingpong test, we really only need CQ sizes of 1

RDM Pingpong

Create address vector

```
struct fi_av_attr av_attr = {};

av_attr.type = fi->domain_attr->av_type;
av_attr.count = 1;

fi_av_open(domain, fi, &av_attr, &av, NULL);
```

Use AV type optimal for provider

By default, addresses inserted into the AV will be resolved synchronously. We can obtain asynchronous operation by binding the AV with an event queue

RDM Pingpong

Create the endpoint

```
fi_endpoint(domain, fi, &ep, NULL)
```

Use the default attributes specified by the provider

RDM Pingpong

Associate the endpoint with the other resources

```
fi_ep_bind(ep, av, 0);
fi_ep_bind(ep, tx_cq, FI_TRANSMIT);
fi_ep_bind(ep, rx_cq, FI_RECV);

fi_enable(ep);
```

And enable it for data transfers

RDM Pingpong

Client is given server address through command line

```
fi_recv(ep, rx_buf, MAX_CTRL_MSG_SIZE, 0, 0, NULL);
```

Client will send its address to the server as its first message

Server will ack when it is ready

RDM Pingpong

Define function to retrieve a completion

```
int wait_for_comp(struct fid_cq *cq)
{
    struct fi_cq_entry entry;
    int ret;

    while (1) {
        ret = fi_cq_read(cq, &entry, 1);
        if (ret > 0)
            return 0;
```

CQ entry based on configured format (i.e. `FI_CQ_FORMAT_CONTEXT`)

Return success if we have a completion

RDM Pingpong

The operation failed

```
            if (ret != -FI_EAGAIN) {
                struct fi_cq_err_entry err_entry;
                fi_cq_readerr(cq, &err_entry, 0);
                printf("%s %s\n", fi_strerror(err_entry.err),
                       fi_cq_strerror(cq, err_entry.prov_errno,
                                      err_entry.err_data,
                                      NULL, 0);
                return ret;
            }
        }
    }
```

Print some error information

RDM Pingpong

```
size_t addrlen = MAX_CTRL_MSG_SIZE;

if (opts.dest_addr) {
    fi_av_insert(av, fi->dest_addr, 1, &remote_addr,
                 0, NULL);

    fi_getname(&ep->fid, tx_buf, &addrlen);
```

Synchronously insert server address into client's AV

Get client address to send to server

RDM Pingpong

```
        fi_send(ep, tx_buf, addrlen, NULL, remote_addr,
                NULL);

        wait_for_comp(rx_cq);

        fi_recv(ep, rx_buf, opts.size, 0, 0, NULL);
        wait_for_comp(tx_cq);
    } else {
```

Wait for server to ack that it’s ready

RDM Pingpong

```
    wait_for_comp(rx_cq);

    fi_av_insert(av, rx_buf, 1, &remote_addr,
                 0, NULL);

    fi_recv(ep, rx_buf, opts.size, 0, 0, NULL);
    fi_send(ep, tx_buf, 1, NULL, remote_addr, NULL);
    wait_for_comp(tx_cq);
}
```

Server waits for message from client

Insert client address

Ack that we're ready

RDM Pingpong

Exchange messages

```
for (i = 0; i < opts.iterations; i++) {
    if (opts.dest_addr) {
        fi_send(ep, tx_buf, opts.size, NULL,
                remote_addr, NULL);
        wait_for_comp(tx_cq);
        wait_for_comp(rx_cq);
        fi_recv(ep, rx_buf, opts.size, 0, 0, NULL);
    } else {
```

RDM Pingpong

```
                wait_for_comp(rx_cq);
                fi_recv(ep, rx_buf, opts.size, 0, 0, NULL);
                fi_send(ep, tx_buf, opts.size, NULL,
                        remote_addr, NULL);
                wait_for_comp(tx_cq);
            }
        }
        /* done */
    }
```

Advanced MPI Usage

MPI: Choosing an OFI mapping

This tutorial:
- Presents a possible mapping of MPI to OFI.
- Designed for providers with a semantic match to MPI

Counters | CQ

Implements

MPI progress

Examples of this mapping:
- OpenMPI MTL
- MPICH Netmod

Tagged

Implements

MPI Two-sided

RMA | MSG

Implements

MPI-3 RMA

One size does not fit all!
- OFI is a set of building blocks
- Customize for your hardware

Domain | Endpoint

Implements

MPI Initialization

MPI_Init

- Initializes processes for communication
- Sets up OFI data structures
- exchanges information necessary to communicate
- Establishes an MPI *communicator*
  - Processes are addressable by MPI *rank*
  - Initial “world” communicator is all processes in a job

MPI_Init

```
int MPI_init(/* … */){
   /* Initialize MPI and OFI                                         */
   /* Use process manager for job state (job size, rank, etc)       */
   /* fi_getinfo(): Query and request OFI capabilities              */
   /* Map OFI capabilities to MPI API set.                           */
   /* Open OFI objects (fabric, domain, endpoint, counter, cq, av)*/
   /* Exchange addresses via process manager                        */
   /* Bind OFI objects                                               */
}
```

OFI initialization is designed so critical communication code path will be lightweight.

There are *multiple* ways to map MPI semantics to OFI semantics

MPI_Init: Data structures

```
typedef struct global_t {
    /* … */
    struct fid_domain   *domain;
    struct fid_fabric   *fabric;
    struct fid_endpoint *ep;
    struct fid_cq       *p2p_cq;
    struct fid_cntr     *rma_ctr;
    struct fid_mr       *mr;
    struct fid_av       *av;
    /* … */
} global_t;
global_t gbl;
```

Endpoint

Completion Queue

Counter

Memory Region

Address Vector

Global State Object

MPI_Init: Map MPI to OFI

```
int MPI_init() {
…
    hints = fi_allocinfo();
    assert(hints != NULL);

    hints->mode = FI_CONTEXT;
    hints->caps = FI_TAGGED;        /* Implements MPI tagged (2-sided) p2p  */
    hints->caps |= FI_MSG;          /* Implements control messages          */
    hints->caps |= FI_MULTI_RECV;   /* Ring buffer for control              */
    hints->caps |= FI_RMA;          /* Implements MPI-3 RMA                 */
    hints->caps |= FI_ATOMIC;       /* Implements MPI-3 RMA atomics         */
…
}
```

Allocate and clear an info struct

MPI provides context via MPI Requests

Request Capabilities

MPI_Init: OFI hints

```
int MPI_init() {
    /* MPI handles locking, OFI should not lock */
    hints->domain_attr->threading = FI_THREAD_ENDPOINT;

    /* OFI handles progress:  Note that this choice may be provider dependent */
    hints->domain_attr->control_progress = FI_PROGRESS_AUTO;
    hints->domain_attr->data_progress = FI_PROGRESS_AUTO;

    /* OFI handles flow control*/
    hints->domain_attr->resource_mgmt = FI_RM_ENABLED;

    /* MPI does not want to exchange memory regions */
    hints->domain_attr->mr_mode = FI_MR_SCALABLE;

    /* Completions indicate data is in memory at the target */
    hints->tx_attr->op_flags = FI_DELIVERY_COMPLETE|FI_COMPLETION;
}
```

MPI_Init: Query Provider

```
int MPI_init() {
…
ret = fi_getinfo(fi_version, NULL, NULL,
                 OULL, hints, &prov));
prov_use = choose_prov_from_list(prov);

max_buffered_send  = prov_use->tx_attr->inject_size;
max_buffered_write = prov_use->tx_attr->inject_size;
max_send           = prov_use->ep_attr->max_msg_size;
max_write          = prov_use->ep_attr->max_msg_size;
…
}
```

OFI returns a list of suitable providers based on hints

Query OFI limits

MPI_Init: Single Basic endpoint

```
int MPI_init(/* … */)
{ …
/* Create the endpoint */
struct fid_endpoint *ep;
fi_endpoint(domain, prov_use, &ep, NULL);
gbl.ep = ep;
/* Bind the MR, CQs, counters, and AV to the endpoint object */
/* In this MPI model, we have 1 endpoint, 1 counter, and 1 completion queue */
fi_ep_bind(ep, (fid_t)gbl.p2p_cq, FI_SEND|FI_RECV|FI_SELECTIVE_COMPLETION);
fi_ep_bind(ep, (fid_t)gbl.rma_ctr, FI_READ | FI_WRITE);
fi_ep_bind(ep, (fid_t)gbl.av, 0ULL);
fi_enable(ep);
fi_ep_bind(ep, (fid_t)mr, FI_REMOTE_READ | FI_REMOTE_WRITE);
…
}
```

MPI_Init: Address Exchange

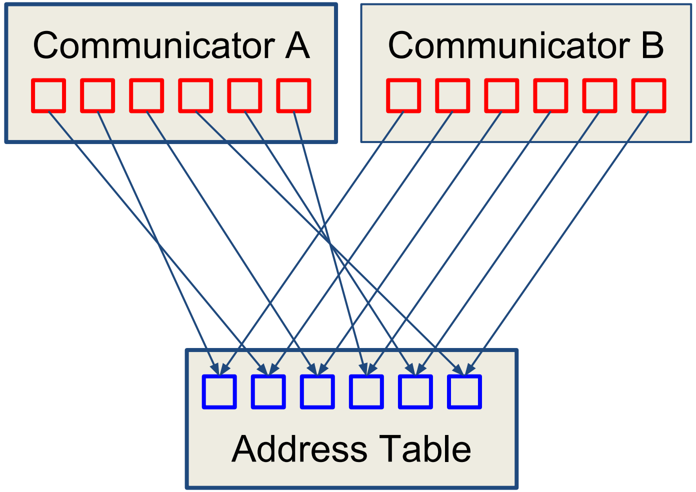
```
int MPI_init(/* … */)
{ …
    /* Get our endpoint name and publish  */
    /* the socket to the KVS              */
    addrnamelen = FI_NAME_MAX;
    fi_getname((fid_t)gbl.ep, addrname,&addrnamelen);

    allgather_addresses(addrname, &all_addrnames);

    /* Names are exchanged:  Create an address vector and */
    /* optionally add a table of mapped addresses         */
    fi_av_open(gbl.domain, av_attr, &gbl.av,  NULL);
    fi_av_insert(gbl.av, all_addrnames, job_size, mapped_table, 0ULL, NULL);
…
}
```

MPI Communicators

- MPI communicators remap a per-communicator MPI rank to a global canonical process (often referenced by process rank in `MPI_COMM_WORLD`)
- An OFI Address vector is a logical container for a list of network addresses
- The MPI implementation must map logical per-communicator ranks to a network address to communicate
- There are several ways that communicators can be mapped

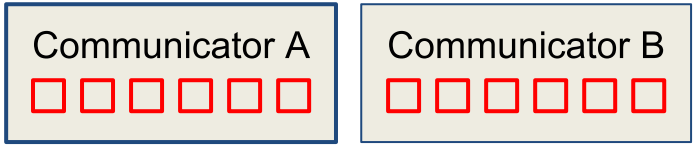
How should MPI use address vectors?

Method 1: AV_MAP

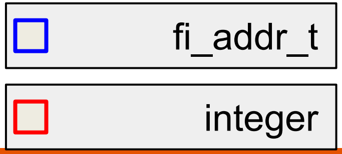



```
int MPI_Send(comm, rank){
  fi_addr_t addr =
    addr_table[comm->table[rank]];
  fi_tsend(gbl.ep, …, addr, …);
}
```

Method 2: AV_TABLE




```
int MPI_Send(comm, rank){
  int addr = comm->table[rank];
  fi_tsend(gbl.ep, …, addr, …);
}
```



How should MPI use address vectors?

Method 3: AV_MAP

Communicator A

Communicator B

```
int MPI_Send(comm, rank){
  fi_addr_t addr =
    comm->table[rank];
  fi_send(gbl.ep, …, addr, …);
}
```

Method 4: AV_TABLE (per comm)

Communicator A

Communicator B

```
int MPI_Send(comm, rank){
    int addr = rank;
    fi_send(comm->ep, …, addr, …);
}
```

O(1) communicator storage is possible

- Requires a new AV bound to an endpoint per communicator.

fi_addr_t

integer

MPI Communication

- Endpoints have been created and bound to resources
- Addresses have been exchanged
- Data can be sent/received
- Send operations
  - `MPI_Send`: blocking send of a buffer to a rank in a communicator
  - `MPI_Isend`: non-blocking send of a buffer to a rank in a communicator

MPI_Send

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype,
             int rank, int tag, MPI_Comm comm) {
…
/* We can implement a lightweight send if certain conditions are met  */
/* Lightweight send maps to fi_tinject                                 */
if (datatype_is_contiguous && (data_sz <= max_buffered_send))
    mpi_errno = send_lightweight(buf, data_sz, rank, tag, comm);
else
    mpi_errno = send_normal(buf, count, datatype, rank, tag, comm)
…
}
```

MPI_Send: send_lightweight

```
int send_lightweight(const void *buf, size_t data_sz,
                     int rank, int tag, Comm *comm)
{
    int mpi_errno = MPI_SUCCESS;
    uint64_t match_bits;
    ssize_t ret;

    /* Convert MPI rank to address, initialize the tag, inject! */
    /* Tagged inject is buffered, no need to wait for completion*/
    match_bits = init_sendtag(comm->comm_id, comm->rank, tag);
    ret = fi_tinject(ep, buf, data_sz,
                     RANK_TO_FIADDR(comm, rank),
                     match_bits);
    if(ret != 0) mpi_errno = handle_mpi_error(ret);
    return mpi_errno;
}
```

MPI Send: Matching

- MPI enforces message order based on {rank, tag, communicator}.
- OFI uses a 64-bit integer for matching.
- The match bits must pack this information

```
/* match/ignore bit manipulation
 *
 * 0123 4567 01234567 0123 4567 01234567 0123 4567 01234567 01234567 01234567
 *      |                  |                  |
 * ^    |    context id    |       source     |       message tag
 * |    |   (communicator) |                  |
 * +---- protocol
 */
```

MPI Send: Matching Alternative

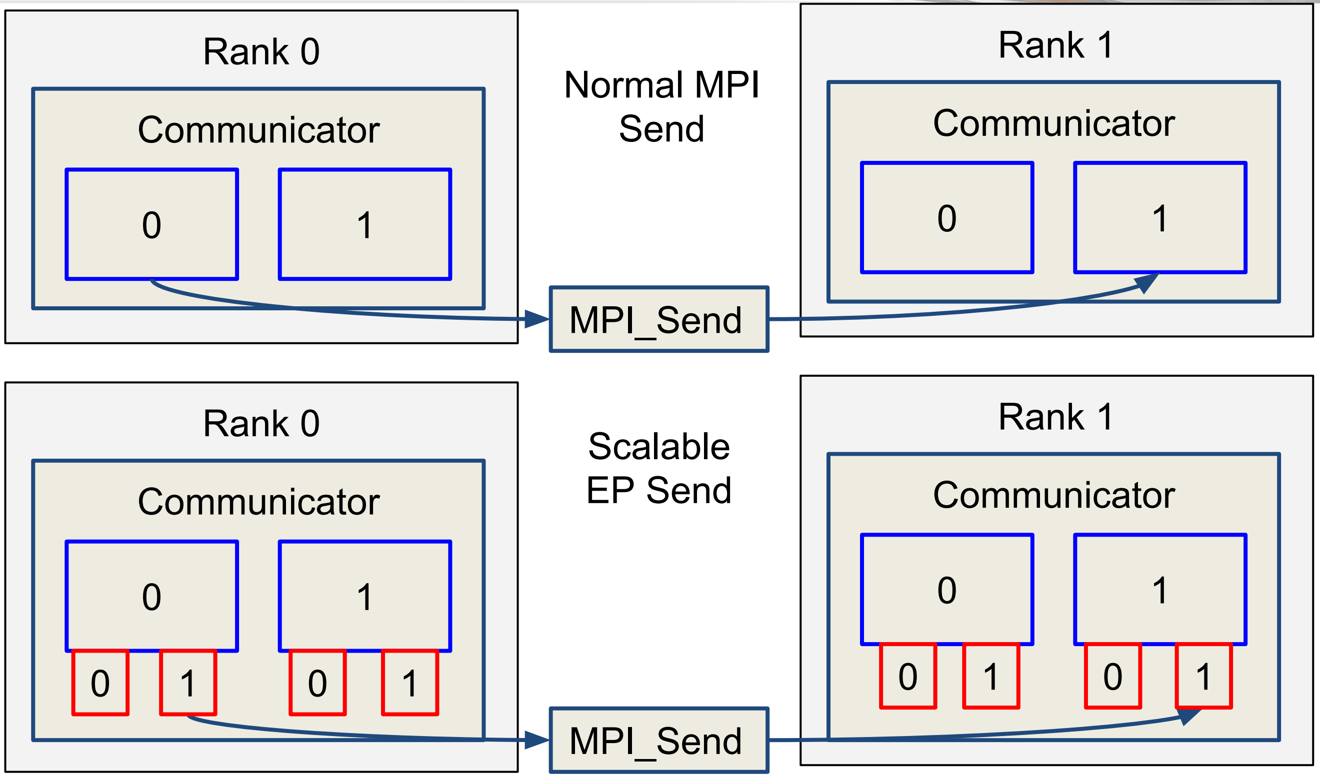
- Use `fi_tsenddata/fi_tinjectdata` to send immediate data
- Use `FI_DIRECTED_RECEIVE` to accept messages from a specific destination address
- Send source rank in immediate data

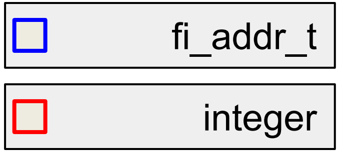
```
/* match/ignore bit manipulation
 *
 * 0123 4567 01234567 01234567 01234567 01234567 01234567 01234567 01234567
 *      |                                |
 * ^    |     context id                 |                  message tag
 * |    |   (communicator)               |
 * +---- protocol
 */
```

MPI_Send: send_normal

```
int send_normal(const void *buf, size_t data_sz,
                int rank, int tag, Comm *comm, Request **req) { …
    /* Create a request, handle datatype processing, send  */
    /* The MPI request object contains the OFI state via   */
    /* The fi_context field in the request object          */
    *req = create_and_setup_mpi_request();
    … /* Other MPI processing.  Example, datatypes  */
    match_bits = init_sendtag(comm->comm_id, comm->rank, tag);
    ret = fi_tsend(ep, dt_buf, dt_sz,
                   RANK_TO_FIADDR(comm, rank),
                   match_bits, &((*r)->ofi_context) );
    if(ret != 0) mpi_errno = handle_mpi_error(ret);
    /* Block until send is complete  */
    while((*req)->state != DONE)
        PROGRESS();
    return mpi_errno;
}
```

Advanced MPI_Send: Scalable EP





Why use Scalable EP's?:

- Parallelism
- Reduce lock contention
- Different attributes (like ordering)
- Exploit independent hardware resources

MPI_Init: Scalable endpoints

```
int MPI_init(/* … */)
{ …
/* Create the transmit context using scalable endpoints */
struct fi_tx_attr tx_attr;
fi_scalable_ep(gbl.domain, prov_use, ep, NULL);
/* For Tagged MPI Point to Point */
tx_attr.caps = FI_TAGGED;
fi_tx_context(gbl.ep, index, &tx_attr, &g_txc(index), NULL);
fi_ep_bind(g_txc_tag(index), (fid_t)p2p_cq, FI_SEND);
/* For request based MPI RMA */
tx_attr.caps = FI_RMA|FI_ATOMIC;
fi_tx_context(gbl.ep, index+1, &tx_attr, &g_txc_rma(index), NULL);
fi_ep_bind(g_txc_rma(index), (fid_t) p2p_cq, FI_SEND);
/* For non-request based MPI RMA */
tx_attr.caps = FI_RMA|FI_ATOMIC;
fi_tx_context(gbl.ep, index+3, &tx_attr, &g_txc_cntr(index), NULL);
fi_ep_bind(g_txc_cntr(index), (fid_t) rma_ctr, FI_WRITE | FI_READ);
}
```

Tagged transmit context: shared completion queue

RMA transmit context: shared completion queue

RMA transmit context: completion counter

Advanced MPI_Send: Scalable EP

```
ret = fi_tsend(g_txc(src_tx_ctx_index), dt_buf, dt_sz,
               RANK_TO_FIADDR_SEP(comm, rank, dest_rx_ctx_index),
               match_bits, &(*r)->ofi_context);
```

Endpoint:

- Use a transmit context instead of endpoint.
- transmit context is an endpoint created with `fi_tx_context`

Addressing:

- Use EP version of RANK_TO_FIADDR
- “endpoint” is an index (offset) into an array of receive contexts

```
fi_addr_t RANK_TO_FIADDR_SEP(Comm *comm, int rank,
                             int endpoint) {
    return fi_rx_addr(RANK_TO_FI_ADDR(comm, rank),
                      endpoint,
                      MAX_ENDPOINT_BITS);
/* MAX_ENDPOINT_BITS is an address vector attribute */
}
```

Which endpoint is right for my MPI?

- **Basic Endpoint**
  - Possible to implement all of MPI
- **Scalable endpoint + contexts**
  - Useful for internal threading modes, separation of resources, software parallelization
  - Adding capabilities to RMA windows (such as ordering restrictions)
  - Binding resources at window creation time (memory regions)
- **Shared endpoint + contexts**
  - Useful on shared or oversubscribed hardware
- **Use them all!**
  - Start with basic, and customize/specialize MPI with a mix of endpoint types

MPI Progress

- MPI_Send has initiated a blocking send operation that may take time to complete. MPI will need to read the completion queues to complete the request.
- MPI must handle errors (fatal and recoverable) in the progress loop.

MPI Progress

```
int internal_progress(){
    int                     mpi_errno;
    ssize_t                 ret;
    struct fi_cq_tagged_entry  wc[NUM_CQ_ENTRIES];

    ret = fi_cq_read(cq, (void *) wc, NUM_CQ_ENTRIES);
    if(likely(ret > 0))
        mpi_errno = handle_cq_entries(wc, ret);
    else if (ret == -FI_EAGAIN)
        mpi_errno = MPI_SUCCESS;
    else
        mpi_errno = handle_cq_error(ret);
    return mpi_errno;
}
```

Optimized “Good” path

Empty Poll path

Error path

MPI Progress

```
static inline MPI_Request *context_to_req(void *context){
    char *base = (char *)context;
    return (Request *)container_of(base, Request, context);
}
static inline int handle_cq_entries(cq_tagged_entry_t * wc, ssize_t num){
    int i;
    Request *req;
    for (i = 0; i < num; i++) {
        req = context_to_req(wc[i].op_context);
        dispatch_function(&wc[i],req);
    }
    return MPI_SUCCESS;
}
```

Handles multiple completions

Embedded OFI context

- Embedding helps to prevent double allocations (provider and app)
- OFI can also allocate context

MPI Progress: dispatch

```
static inline int dispatch_function(cq_tagged_entry_t *wc, MPI_Request *req)
{
    int mpi_errno;
    switch (request->event)) {
    case EVENT_SEND:
        mpi_errno = send_done_event(wc, req);
        break;
    case EVENT_RECV:
        mpi_errno = recv_done_event(wc, req);
        break;
…
  };
}
```

Marks MPI send request complete. MPI test/wait routines watch the completion state.

Marks MPI receive request complete and populates the MPI status object

MPI_Recv

```
int MPI_Recv(const void *buf, size_t data_sz,int rank, int tag, Comm * comm,
             Request *req){
…
    *req = create_and_setup_mpi_request();
    match_bits = init_recvtag(&mask_bits, comm, rank, tag);
…   /* Other MPI processing.  Example, datatypes  */
    fi_trecv(ep,recv_buf,data_sz,NULL,
         (MPI_ANY_SOURCE == rank) ? FI_ADDR_UNSPEC : RANK_TO_FIADDR(comm, rank),
         match_bits, mask_bits, &req->context)
…
    /* check completion queue for match */
}
```

Receive always takes a context

Mask bits tell receive to ignore ANY_SOURCE match bits if rank == MPI_ANY_SOURCE

MPI_Probe: check for inbound msg

```
int MPI_Probe(int rank, int tag, Comm * comm,MPI_Status *status){
    match_bits = init_recvtag(&mask_bits, comm, rank, tag);
    … /* Other MPI processing  */
    msg.addr     = remote_proc;
    msg.tag      = match_bits;
    msg.ignore   = mask_bits;
    msg.context = req->context;
    while(!req->done) {
        ret = fi_trecvmsg(ep, &msg, FI_PEEK|FI_COMPLETION);
        if(ret == 0)
            PROGRESS_WHILE(!req.done);
        else if(ret == -FI_ENOMSG)
            continue;
        else
            error();
    }
    /* Fill out status and return  */
}
```

add `FI_CLAIM` to flags when implementing MPI_Mprobe

Peek matches a message but does not dequeue it.

Progress to complete message

Probe/Receive completion events

```
int probe_event(cq_tagged_entry_t *wc,
                Request           *rreq){
    rreq->done                 = TRUE;
    rreq->status.MPI_SOURCE = get_source(wc->tag);
    rreq->status.MPI_TAG       = get_tag(wc->tag);
    rreq->status.MPI_ERROR  = MPI_SUCCESS;
    rreq->status.count         = wc->len;
    return MPI_SUCCESS;
}
```

Probe

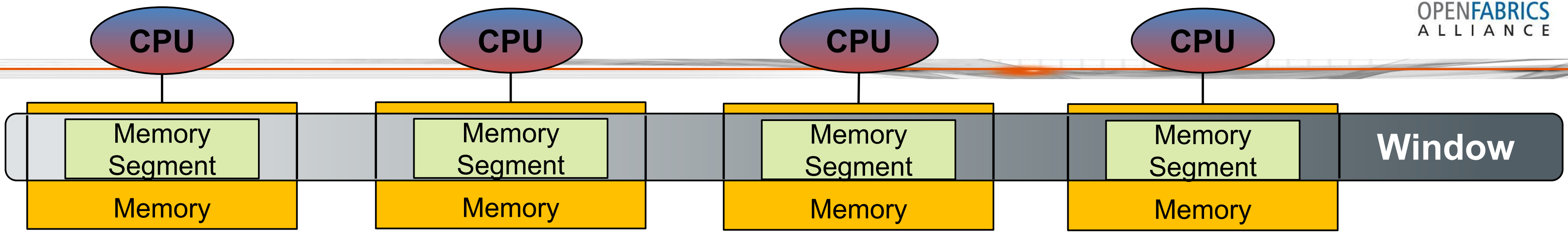
- Fills out status to be returned to the user
- get_tag/get_source are macros that decode the tag
- length is data bytes, needs to be converted to MPI count

Recv

- Fills out status to be returned to the user
- Unpacks data and/or handles the datatypes
- Handles any protocol acking required (like ssend acks)

```
int recv_event(cq_tagged_entry_t *wc,
               Request           *rreq){
    rreq->done                 = TRUE;
    rreq->status.MPI_SOURCE = get_source(wc->tag);
    rreq->status.MPI_TAG       = get_tag(wc->tag);
    rreq->status.MPI_ERROR  = MPI_SUCCESS;
    rreq->status.count         = wc->len;
    /* Other MPI processing: unpack, request completion */
    return MPI_SUCCESS;
}
```

MPI 3 RMA Desired Semantics



- Memory exposed to incoming read/write by all targets
- Offset based addressing within a window O(1) storage is ideal
- Synchronization per MPI window
- Local and request based completion requirement at the origin for certain ops
- Non-contiguous support
- Hardware accelerated atomics
- Asynchronous progress

Memory Regions and Windows

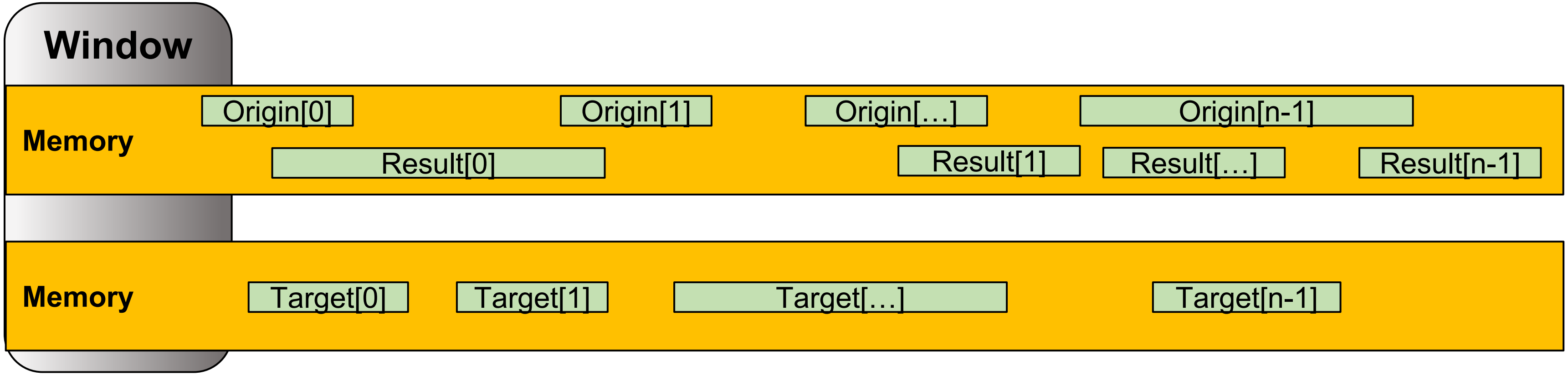
- Possible Mappings (Using FI_SCALABLE_MR):
  - **Mapping 1**: Global endpoint, user defined key, map all of memory
  - **Mapping 2:** TX context per window, user defined key
  - **Mapping 3:** TX/RX context per window, no key (offsets embedded in RX)
  - **Common**: Fallback to mapping 1 when resources constrained
- Use symmetric heap if possible
- O(1) if resources and MPI parameters permit, otherwise:
  - Displacement unit (if necessary)
  - Window bases if Mapping 1 and no symmetric heap.

Synchronization

- Counters used for synchronization (e.g., `MPI_Win_flush`)
- Completion queues used to signal request-based "R" variants (e.g., `MPI_Rput`)
- `MPI_Win_lock/unlock` uses message queue API for protocol

Non-contiguous Data

- Desirable to not pack the data or send datatype, possible mappings:
  - Send a series of RMA operations that align contiguous chunks
  - Generate iovec lists that correspond to OFI hardware limits
  - Handle datatypes natively with OFI



MPI_Put

```
int MPI_Put() {
…
/* We can implement a lightweight put if conditions are met */
if (origin_contig && target_contig && other_conditions &&
    origin_bytes <= max_buffered_write)) {
        /* Increment counter to synchronize with fi_cntr_read */
        global_cntr++;
        fi_inject_write(ep, (char *)origin_addr, target_bytes,
                        RANK_TO_FIADDR(win->comm, target_rank),
                        target_address, win->memory_key );
    }
}
```

Target address can be offset or VA based

Keys can be:

- Exchanged
- App provided: (FI_MR_SCALABLE)

MPI_Get

```
int MPI_Get() {
…
/* We can implement a lighter weight get if conditions are met */
if (origin_contig && target_contig && other_conditions &&
    origin_bytes <= max_msg_size)) {
        /* Increment counter to synchronize with fi_cntr_read */
        global_cntr++;
        fi_read(ep, (char *)origin_addr, target_bytes,
                RANK_TO_FIADDR(win->comm, target_rank),
                target_address, win->memory_key );
    }
}
```

Target address can be offset or VA based

Keys can be:

- Exchanged
- App provided: (`FI_MR_SCALABLE`)

MPI Atomics

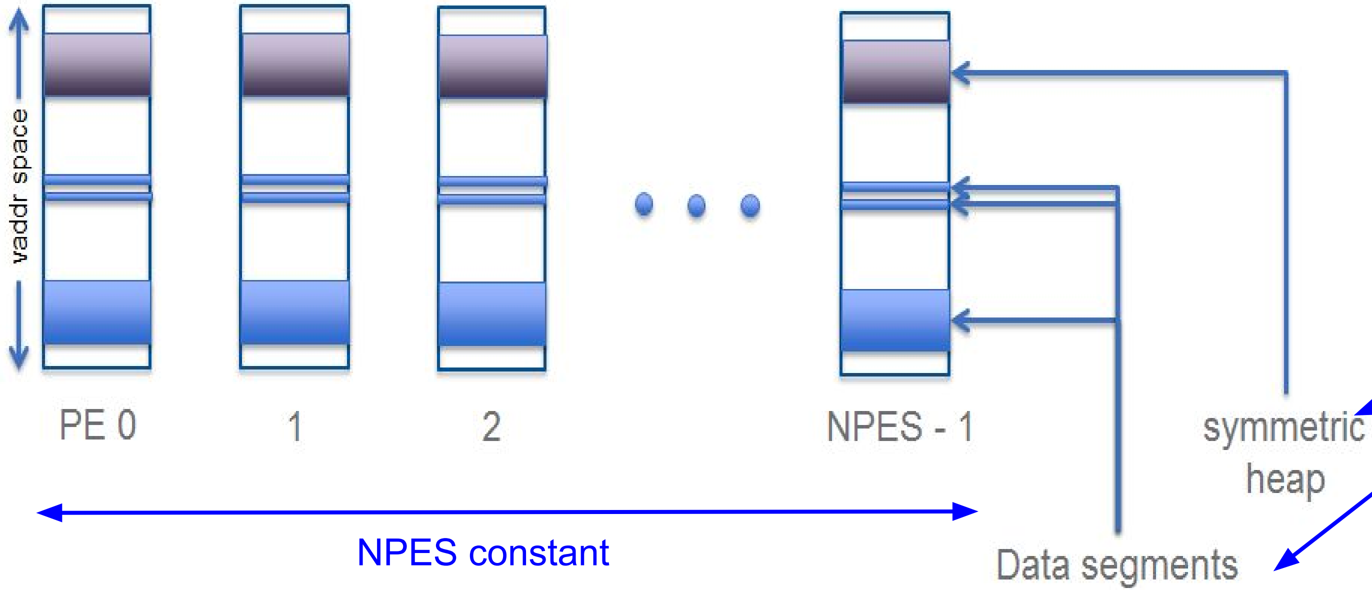
- Natively supported in libfabric
  - `FI_ATOMIC` capability for `fi_getinfo`
  - see `fi_atomic` functions in the [fi_atomic(3) man page]
- Query MPI datatype and MPI op and use valid table
  - Fall back to message queue API emulation if hardware atomic is not available
  - Provide optimized versions of single element atomics
- At window creation, determine MPI ordering info key values and create the scalable context with corresponding `FI_ORDER_xAy` flags
- We'll discuss more later in the tutorial

OpenSHMEM Example

Content

- OpenSHMEM program model in a nutshell
- Mapping to libfabric constructs
  - Endpoint types
  - Address vectors
  - Memory Registration
  - Completion Queues and Counters
- Example Code walkthrough

Program Model in a Nutshell



Certain segments of PE address space remotely accessible in a one-sided fashion

OpenSHMEM - FI_EP_RDM endpoints

- FI_EP_RDM likely best choice for OpenShmem
  - one endpoint can be used to put/get, etc. to all PEs in job
  - relatively simple connection setup, but does require some sort of out-of-band to exchange endpoint *names*
  - Does require an *Address Vector* instance

OpenSHMEM - Which AV Type?

- Two types of address vectors -
  - FI_AV_MAP - using this type means the library must internally keep a mechanism for mapping a PE to a fabric *fi_addr_t*
  - FI_AV_TABLE - supports a simple indexing scheme to be used in place of *fi_addr_t*. Can support using the PE rank as the address in data transfer operations (this is likely the better choice for OpenSHMEM)

OpenSHMEM - Memory Registration

- libfabric supports two memory registration modes - *basic* and *scalable*
- FI_LOCAL_MR mode bit indicates whether local buffers need to be registered

OpenSHMEM - Memory Registration(2)

- Scalable memory registration model simpler to use
  - does not require O(NPES) bookkeeping of memory keys.
  - simplifies the implementation of a growable symmetric heap.
- Basic memory registration model is likely to be supported by more providers
- Probably a good idea to be able to support both models at least for the medium term

OpenSHMEM - Completion Queues, Counters

- With current OpenSHMEM api, no need to track data transfers on a per operation basis
- But for *shmem_quiet/shmem_fence* do need to count outstanding data transfers
- *fi_cntr*'s smart to use here

- To support *shmem_quiet* semantics, want to use `FI_DELIVERY_COMPLETE` for `tx_attr` op_flags

Example walkthrough

- *shmem_init*
- data transfer examples
  - *shmem_put*
  - *shmem_iput* (done two ways)
  - *shmem_double_swap*

shmem_init (1)

```
void shmem_init(void) *dest, const void *src, size_t nelems, int pe)
{
    uint32_t version = FI_VERSION(1,0);  /* api version we use */
    struct fi_tx_attr tx_attr = {0};
    struct fi_rx_attr rx_attr = {0};
    struct fi_ep_attr ep_attr = {0};
    struct fi_domain_attr dom_attr = {0};
    struct fi_info hints, *p_info = NULL;
    hints = fi_allocinfo();

    hints->caps = FI_RMA;          /* one sided, got to have that        */
    hints->caps |= FI_ATOMIC;      /* for shmem_fadd, etc.               */
    hints->caps |= FI_MSG;         /* may be useful for control messages */
…
```

shmem_init (2)

```
    hints.ep_attr = &ep_attr;
    hints.ep_attr->type = FI_EP_RDM;    /* specify EP type     */
    hints.tx_attr = &tx_attr;
    hints.tx_attr->op_flags =
        FI_DELIVERY_COMPLETE;        /* shmem_quiet visibility guarantee */
    hints.rx_attr = &rx_attr;
    hints.rx_attr->op_flags = 0;
    hints.domain_attr = &dom_attr;
    hints.domain_attr->data_progress =
        FI_PROGRESS_AUTO;       /* no to shmem_progress */
#ifdef USE_SCALABLE_MR
    hints.domain_attr->mr_mode = FI_MR_SCALABLE;  /* optionally try scalable mr
*/
#endif
    fi_getinfo(version, NULL, 0, 0, &hints, &p_info);
```

…

shmem_init (3)

```
fi_fabric(p_info->fabric_attr,        /* get a fab desc     */
          &fab_desc, NULL);
fi_domain(fab_desc,p_info, &dom_desc, NULL);   /* get a dom desc */
fi_endpoint(dom_desc, p_info, &ep_desc, NULL);  /* get a ep desc */
cntr_attr.events = FI_CNTR_EVENTS_COMP;
fi_cntr_open(dom, &cnt_attr, &putcntr);         /* open a put cntr */
fi_ep_bind(ep_desc, &putcntr->fid, FI_WRITE);   /* bind to ep */
fi_cntr_open(dom, &cnt_attr, &getcntr);         /* open a put cntr */
fi_ep_bind(ep_desc, &getcntr->fid, FI_READ);    /* bind to ep */

av_attr.type = FI_AV_TABLE;
fi_av_open(dom, &av_attr, &av_desc, NULL);      /* get an av desc */
fi_ep_bind(ep_desc, &av->fid, 0);               /* bind to ep */

/* also open CQ and bind to ep (not shown) */
```

…

shmem_init (4)

```
#ifdef USE_SCALABLE_MR
    fi_mr_reg(dom_desc, 0, UINT64_MAX,  /* register entire addr space */
              FI_REMOTE_READ | FI_REMOTE_WRITE,
              0,                         /* zero-offset */
              0ULL,                      /* pick 0 as memory key  */
              0,
              &mr_desc,
              NULL);
#else
    fi_mr_reg(dom_desc, bss_base, bss_len,
              FI_REMOTE_READ | FI_REMOTE_WRITE,
              0,0ULL,0,&bss_mr_desc, NULL);
    bss_mr_desc_key = fi_mr_key(bss_mr_desc);
     /* same for symmetric heap*/
#endif
…
```

shmem_init (5)

```
fi_enable(ep_desc);                           /* enable ep for data transfers */
len = sizeof(getname_buf);
fi_getname(ep_desc,getname_buf, &len);  /* get ep name */

all_ep_names = (char *)malloc(len * n_pes);
out_of_band_xchg(getname_buf, all_ep_names);  /* oob exchange of ep names */

n = fi_av_insert(av_desc,     /* add entries to av table */
                 all_ep_names,
                 npes,
                 NULL,       /* don't need vec of fi_addr's for FI_AV_TABLE
*/
                 0,
                 NULL);
free(all_ep_names);
/* for !USE_SCALABLE_MR also need to exchange memory keys */
}
```

shmem_put/iput

shmem_put

```
void shmem_put64(void *dest, const void *src, size_t nelems, int pe)
{
    extern uint64_t put_count;
    uint64_t key = 0ULL;
#if !USE_SCALABLE_MR
    key = key_is_bss_or_symheap(dest);  /* assumes sym heap at same VADDR */
#endif
    fi_write(ep_desc,
             src,
             nelems * sizeof(long),
             NULL,                      /* FI_LOCAL_MR */
             (fi_addr_t)pe,
             (uint64_t)dest,
             key,
             NULL);
    put_count++;
    fi_cntr_wait(putcntr, put_count, -1);  /* wait till src can be reused*/
}
```

shmem_iput - using FI_MORE(1)

```
void shmem_iput64(void *dest, const void *src,
                  ptrdiff_t tst, ptrdiff_t sst, size_t nelems, int pe)
{
    extern uint64_t put_count;
    uint64_t key = 0ULL;
    uint64_t i;
    struct fi_msg_rma msg_rma;
    struct iov  s_msg_iov,
    struct fi_rma_iov t_msg_iov;

    s_msg_iov.iov_len = sizeof(long); t_msg_iov.len = sizeof(long);
    msg_rma.msg_iov = &s_msg_iov;
    msg_rma.rma_iov = &t_msg_iov;
    msg_rma.desc = NULL;     /* assumes FI_LOCAL_MR */
    msg_rma.iov_count = msg_rma.rma_iov_count = 1;
    msg_rma.addr = (fi_addr_t)pe;
    t_msg_iov.key = 0ULL;     /* USE_SCALABLE_MR */

. . .
```

shmem_iput - using FI_MORE (2)

```
. . .
    for (i=0;i<nelems-1;i++) {
        s_msg_iov.iov_base = src + i * sst * sizeof(long);
        t_msg_iov.addr = (uint64_t)dest + i * tst * sizeof(long);
        fi_writemsg(ep_desc,
                    &msg_rma,
                    FI_MORE);
    }

    s_msg_iov.iov_base = src + (nelems - 1) * sst * sizeof(long);
    t_msg_iov.addr = (uint64_t)dest + (nelems - 1) * tst * sizeof(long);
    fi_writemsg(ep_desc,
                &msg_rma,
                0);
    put_count += nelems;
    fi_cntr_wait(putcntr, put_count, -1);  /* wait till src can be reused*/
}
```

shmem_swap

```
long shmem_long_swap(long *target, long value, int pe)
{
    extern uint64_t get_count;
    uint64_t key = 0ULL, result;
    const uint64_t mask = ~0ULL;

#if !USE_SCALABLE_MR
    key = key_is_bss_or_symheap(dest); /* assumes sym heap at same VADDR */
#endif

    fi_compare_atomic(ep_desc, &value, 1, NULL,
                      &mask, NULL,
                      &result, NULL,
                      (fi_addr_t)pe,
                      (uint64_t)target, key,
                      FI_UINT64, FI_MSWAP, NULL);
    get_count++;
    fi_cntr_wait(getcntr, get_count, -1);  /* wait till data has returned*/
    return result;
}
```

For more information:

OFIWG BoF - Tuesday 1:30 - 3:00 PM

2016 International OpenFabrics Alliance Workshop

Monterey, CA

April 4-8

https://www.openfabrics.org/index.php/blogs/80-2016-international-openfabrics-alliance-workshop.html

Mail list - ofiwg@lists.openfabrics.org

Backup slides

shmem_put

```
void shmem_put64(void *dest, const void *src, size_t nelems, int pe)
{
    extern uint64_t put_count;
    uint64_t key = 0ULL;
    struct fi_msg_rma msg_rma;
    struct iov  s_msg_iov,
    struct fi_rma_iov t_msg_iov;

#if !USE_SCALABLE_MR
    key = key_is_bss_or_symheap(dest); /* assumes sym heap at same VADDR */
#endif

    s_msg_iov.iov_base = src;
    s_msg_iov.iov_len = sizeof(long) * nelems;
    t_msg_iov.addr = (uint64_t)dest;
    t_msg_iov.len = sizeof(long) * nelems;
. . .
```

shmem_put (2)

```
...
    t_msg_iov.key = key;

    msg_rma.msg_iov = &s_msg_iov;
    msg_rma.rma_iov = &t_msg_iov;
    msg_rma.desc = NULL;      /* assumes FI_LOCAL_MR */
    msg_rma.iov_count = msg_rma.rma_iov_count = 1;
    msg_rma.addr = (uint64_t)pe;

    fi_writemsg(ep_desc,
                &msg_rma,
                FI_DELIVERY_COMPLETE);

    put_count++;
    fi_cntr_wait(putcntr, put_count, -1);  /* wait till src can be reused*/

}
```