High Performance Network Programming with OFI

{% include JB/setup %}

# Introduction

OpenFabrics Interfaces, or OFI, is a framework focused on exporting fabric communication services to applications. OFI is specifically designed to meet the performance and scalability requirements of high-performance computing (HPC) applications, such as MPI, SHMEM, PGAS, DBMS, and enterprise applications, running in a tightly coupled network environment. The key components of OFI are: application interfaces, provider libraries, kernel services, daemons, and test applications.

Libfabric is a core component of OFI. It is the library that defines and exports the user-space API of OFI, and is typically the only software that applications deal with directly. Libfabric is agnostic to the underlying networking protocols, as well as the implementation of the networking devices.

The goal of OFI, and libfabric specifically, is to define interfaces that enable a tight semantic map between applications and underlying fabric services. Specifically, libfabric software interfaces have been co-designed with fabric hardware providers and application developers, with a focus on the needs of HPC users.

This guide describes the libfabric architecture and interfaces. It provides insight into the motivation for its design, and aims to instruct developers on how the features of libfabric may best be employed.

# Review of Sockets Communication

The sockets API is a widely used networking API. This guide assumes that a reader has a working knowledge of programming to sockets. It makes reference to socket based communications throughout in an effort to help explain libfabric concepts and how they relate or differ from the socket API. To be clear, there is no intent to criticize the socket API. The objective is to use sockets as a starting reference point in order to explain certain network features or limitations. The following sections provide a high-level overview of socket semantics for reference.

## Connected (TCP) Communication

The most widely used type of socket is SOCK\_STREAM. This sort of socket usually runs over TCP/IP, and as a result is often referred to as a 'TCP' socket. TCP sockets are connection-oriented, requiring an explicit connection setup before data transfers can occur. A TCP socket can only transfer data to a single peer socket.

Applications using TCP sockets are typically labeled as either a client or server. Server applications listen for connection request, and accept them when they occur. Clients, on the other hand, initiate connections to the server. After a connection has been established, data transfers between a client and server are similar. The following code segments highlight the general flow for a sample client and server. Error handling and some subtleties of the socket API are omitted for brevity.

/\* Example server code flow to initiate listen \*/  
struct addrinfo \*ai, hints;  
int listen\_fd;  
  
memset(&hints, 0, sizeof hints);  
hints.ai\_socktype = SOCK\_STREAM;  
hints.ai\_flags = AI\_PASSIVE;  
getaddrinfo(NULL, "7471", &hints, &ai);  
  
listen\_fd = socket(ai->ai\_family, SOCK\_STREAM, 0);  
bind(listen\_fd, ai->ai\_addr, ai->ai\_addrlen);  
freeaddrinfo(ai);  
  
fcntl(listen\_fd, F\_SETFL, O\_NONBLOCK);  
listen(listen\_fd, 128);

In this example, the server will listen for connection requests on port 7471 across all addresses in the system. The call to getaddrinfo() is used to form the local socket address. The node parameter is set to NULL, which result in a wild card IP address being returned. The port is hard-coded to 7471. The AI\_PASSIVE flag signifies that the address will be used by the listening side of the connection.

This example will work with both IPv4 and IPv6. The getaddrinfo() call abstracts the address format away from the server, improving its portability. Using the data returned by getaddrinfo(), the server allocates a socket of type SOCK\_STREAM, and binds the socket to port 7471.

In practice, most enterprise-level applications make use of non-blocking sockets. The fcntl() command sets the listening socket to non-blocking mode. This will affect how the server processes connection requests (shown below). Finally, the server starts listening for connection requests by calling listen. Until listen is called, connection requests that arrive at the server will be rejected by the operating system.

/\* Example client code flow to start connection \*/  
struct addrinfo \*ai, hints;  
int client\_fd;  
  
memset(&hints, 0, sizeof hints);  
hints.ai\_socktype = SOCK\_STREAM;  
getaddrinfo("10.31.20.04", "7471", &hints, &ai);  
  
client\_fd = socket(ai->ai\_family, SOCK\_STREAM, 0);  
fcntl(client\_fd, F\_SETFL, O\_NONBLOCK);  
  
connect(client\_fd, ai->ai\_addr, ai->ai\_addrlen);  
freeaddrinfo(ai);

Similar to the server, the client makes use of getaddrinfo(). Since the AI\_PASSIVE flag is not specified, the given address is treated as that of the destination. The client expects to reach the server at IP address 10.31.20.04, port 7471. For this example the address is hard-coded into the client. More typically, the address will be given to the client via the command line, through a configuration file, or from a service. Often the port number will be well-known, and the client will find the server by name, with DNS (domain name service) providing the name to address resolution. Fortunately, the getaddrinfo call can be used to convert host names into IP addresses.

Whether the client is given the server's network address directly or a name which must be translated into the network address, the mechanism used to provide this information to the client varies widely. A simple mechanism that is commonly used is for users to provide the server's address using a command line option. The problem of telling applications where its peers are located increases significantly for applications that communicate with hundreds to millions of peer processes, often requiring a separate, dedicated application to solve. For a typical client-server socket application, this is not an issue, so we will defer more discussion until later.

Using the getaddrinfo() results, the client opens a socket, configures it for non-blocking mode, and initiates the connection request. At this point, the network stack has sent a request to the server to establish the connection. Because the socket has been set to non-blocking, the connect call will return immediately and not wait for the connection to be established. As a result any attempt to send data at this point will likely fail.

/\* Example server code flow to accept a connection \*/  
struct pollfd fds;  
int server\_fd;  
  
fds.fd = listen\_fd;  
fds.events = POLLIN;  
  
poll(&fds, -1);  
  
server\_fd = accept(listen\_fd, NULL, 0);  
fcntl(server\_fd, F\_SETFL, O\_NONBLOCK);

Applications that use non-blocking sockets use select() or poll() to receive notification of when a socket is ready to send or receive data. In this case, the server wishes to know when the listening socket has a connection request to process. It adds the listening socket to a poll set, then waits until a connection request arrives (i.e. POLLIN is true). The poll() call blocks until POLLIN is set on the socket. POLLIN indicates that the socket has data to accept. Since this is a listening socket, the data is a connection request. The server accepts the request by calling accept(). That returns a new socket to the server, which is ready for data transfers.

The server sets the new socket to non-blocking mode. Non-blocking support is particularly important to applications that manage communication with multiple peers.

/\* Example client code flow to establish a connection \*/
  
struct pollfd fds;
  
int err;
  
socklen\_t len;
  
  
fds.fd = client\_fd;
  
fds.events = POLLOUT;
  
  
poll(&fds, -1);
  
  
len = sizeof err;
  
getsockopt(client\_fd, SOL\_SOCKET, SO\_ERROR, &err, &len);

The client is notified that its connection request has completed when its connecting socket is 'ready to send data' (i.e. POLLOUT is true). The poll() call blocks until POLLOUT is set on the socket, indicating the connection attempt is done. Note that the connection request may have completed with an error, and the client still needs to check if the connection attempt was successful. That is not conveyed to the application by the poll() call. The getsockopt() call is used to retrieve the result of the connection attempt. If err in this example is set to 0, then the connection attempt succeeded. The socket is now ready to send and receive data.

After a connection has been established, the process of sending or receiving data is the same for both the client and server. The examples below differ only by name of the socket variable used by the client or server application.

/\* Example of client sending data to server \*/  
struct pollfd fds;  
size\_t offset, size, ret;  
char buf[4096];  
  
fds.fd = client\_fd;  
fds.events = POLLOUT;  
  
size = sizeof(buf);  
for (offset = 0; offset < size; ) {  
 poll(&fds, -1);  
   
 ret = send(client\_fd, buf + offset, size - offset, 0);  
 offset += ret;  
}

Network communication involves buffering of data at both the sending and receiving sides of the connection. TCP uses a credit based scheme to manage flow control to ensure that there is sufficient buffer space at the receive side of a connection to accept incoming data. This flow control is hidden from the application by the socket API. As a result, stream based sockets may not transfer all the data that the application requests to send as part of a single operation.

In this example, the client maintains an offset into the buffer that it wishes to send. As data is accepted by the network, the offset increases. The client then waits until the network is ready to accept more data before attempting another transfer. The poll() operation supports this. When the client socket is ready for data, it sets POLLOUT to true. This indicates that send will transfer some additional amount of data. The client issues a send() request for the remaining amount of buffer that it wishes to transfer. If send() transfers less data than requested, the client updates the offset, waits for the network to become ready, then tries again.

/\* Example of server receiving data from client \*/  
struct pollfd fds;  
size\_t offset, size, ret;  
char buf[4096];  
  
fds.fd = server\_fd;  
fds.events = POLLIN;  
  
size = sizeof(buf);  
for (offset = 0; offset < size; ) {  
 poll(&fds, -1);  
  
 ret = recv(client\_fd, buf + offset, size - offset, 0);  
 offset += ret;  
}

The flow for receiving data is similar to that used to send it. Because of the streaming nature of the socket, there is no guarantee that the receiver will obtain all of the available data as part of a single call. The server instead must wait until the socket is ready to receive data (POLLIN), before calling receive to obtain what data is available. In this example, the server knows to expect exactly 4 KB of data from the client.

It is worth noting that the previous two examples are written so that they are simple to understand. They are poorly constructed when considering performance. In both cases, the application always precedes a data transfer call (send or recv) with poll(). The impact is even if the network is ready to transfer data or has data queued for receiving, the application will always experience the latency and processing overhead of poll(). A better approach is to call send() or recv() prior to entering the for() loops, and only enter the loops if needed.

## Connection-less (UDP) Communication

TODO

## Advantages

The socket API has two significant advantages. First, it is available on a wide variety of operating systems and platforms, and works over the vast majority of available networking hardware. It is easily the de facto networking API. This by itself makes it appealing to use.

The second key advantage is that it is relatively easy to program to. The importance of this should not be overlooked. Networking APIs that offer access to higher performing features, but are difficult to program to correctly or well, often result in lower application performance. This is not unlike coding an application in a higher-level language such as C or C++, versus assembly. Although writing directly to assembly language offers the promise of being better performing, for the vast majority of developers, their applications will perform better if written in C or C++, and using an optimized compiler. Applications should have a clear need for high-performance networking before selecting an alternative API to sockets.

## Disadvantages

When considering the problems with the socket API, we limit our discussion to the two most common sockets types: streaming (TCP) and datagram (UDP).

Most applications require that network data be sent reliably. This invariably means using a connection-oriented TCP socket. TCP sockets transfer data as a stream of bytes. However, many applications operate on messages. The result is that applications often insert headers that are simply used to convert to/from a byte stream. These headers consume additional network bandwidth and processing. The streaming nature of the interface also results in the application using loops as shown in the examples above to send and receive larger messages. The complexity of those loops can be significant if the application is managing sockets to hundreds or thousands of peers.

Another issue highlighted by the above examples deals with the asynchronous nature of network traffic. When using a reliable transport, it is not enough to place an application's data onto the network. If the network is busy, it could drop the packet, or the data could become corrupted during a transfer. The data must be kept until it has been acknowledged by the peer, so that it can be resent if needed. The socket API is defined such that the application owns the contents of its memory buffers after a socket call returns.

As an example, if we examine the socket send() call, once send() returns the application is free to modify its buffer. The network implementation has a couple of options. One option is for the send call to place the data directly onto the network. The call must then block before returning to the user until the peer acknowledges that it received the data, at which point send() can then return. The obvious problem with this approach is that the application is blocked in the send() call until the network stack at the peer can process the data and generate an acknowledgment. This can be a significant amount of time where the application is blocked and unable to process other work, such as responding to messages from other clients.

A better option is for the send() call to copy the application's data into an internal buffer. The data transfer is then issued out of that buffer, which allows retrying the operation in case of a failure. The send() call in this case is not blocked, but all data that passes through the network will result in a memory copy to a local buffer, even in the absence of any errors.

Allowing immediate re-use of a data buffer helps keep the socket API simple. However, such a feature can potentially have a negative impact on network performance. For network or memory limited applications, an alternative API may be attractive.

Because the socket API is often considered in conjunction with TCP and UDP, that is, with protocols, it is intentionally detached from the underlying network hardware implementation, including NICs, switches, and routers. Access to available network features is therefore constrained by what the API can support.

# High-Performance Networking

By analyzing the socket API in the context of high-performance networking, we can start to see some features that are desirable for a network API.

## Avoiding Memory Copies

The socket API implementation usually results in data copies occurring at both the sender and the receiver. This is a trade-off between keeping the interface easy to use, versus providing reliability. Ideally, all memory copies would be avoided when transferring data over the network. There are techniques and APIs that can be used to avoid memory copies, but in practice, the cost of avoiding a copy can often be more than the copy itself, in particular for small transfers (measured in bytes, versus kilobytes or more).

To avoid a memory copy at the sender, we need to place the application data directly onto the network. If we also want to avoid blocking the sending application, we need some way for the network layer to communicate with the application when the buffer is safe to re-use. This would allow the buffer to be re-used in case the data needs to be re-transmitted. This leads us to crafting a network interface that behaves asynchronously. The application will need to issue a request, then receive some sort of notification when the request has completed.

Avoiding a memory copy at the receiver is more challenging. When data arrives from the network, it needs to land into an available memory buffer, or it will be dropped, resulting in the sender re-transmitting the data. If we use socket recv() semantics, the only way to avoid a copy at the receiver is for the recv() to be called before the send(). Recv() would then need to block until the data has arrived. Not only does this block the receiver, it is impractical to use outside of an application with a simple request-reply protocol.

Instead, what is needed is a way for the receiving application to provide one or more buffers to the network for received data to land. The network then needs to notify the application when data is available. This sort of mechanism works well if the receiver does not care where in its memory space the data is located; it only needs to be able to process the incoming message.

As an alternative, it is possible to reverse this flow, and have the network layer hand its buffer to the application. The application would then be responsible for returning the buffer to the network layer when it is done with its processing. While this approach can avoid memory copies, it suffers from a few drawbacks. First, the network layer does not know what size of messages to expect, which can lead to inefficient memory use. Second, many would consider this a more difficult programming model to use. And finally, the network buffers would need to be mapped into the application process' memory space, which negatively impacts performance.

In addition to processing messages, some applications want to receive data and store it in a specific location in memory. For example, a database may want to merge received data records into an existing table. In such cases, even if data arriving from the network goes directly into an application's receive buffers, it may still need to be copied into its final location. It would be ideal if the network supported placing data that arrives from the network into a specific memory buffer, with the buffer determined based on the contents of the data.

### Network Buffers

Based on the problems described above, we can start to see that avoiding memory copies depends upon the ownership of the memory buffers used for network traffic. With socket based transports, the network buffers are owned and managed by the networking stack. This is usually handled by the operating system kernel. However, this results in the data 'bouncing' between the application buffers and the network buffers. By putting the application in control of managing the network buffers, we can avoid this overhead. The cost for doing so is additional complexity in the application.

Note that even though we want the application to own the network buffers, we would still like to avoid the situation where the application implements a complex network protocol. The trade-off is that the app provides the data buffers to the network stack, but the network stack continues to handle things like flow control, reliability, and segmentation and reassembly.

### Resource Management

We define resource management to mean properly allocating network resources in order to avoid overrunning data buffers or queues. Flow control is a common aspect of resource management. Without proper flow control, a sender can overrun a slow or busy receiver. This can result in dropped packets, re-transmissions, and increased network congestion. Significant research and development has gone into implementing flow control algorithms. Because of its complexity, it is not something that an application developer should need to deal with. That said, there are some applications where flow control simply falls out of the network protocol. For example, a request-reply protocol naturally has flow control built in.

For our purposes, we expand the definition of resource management beyond flow control. Flow control typically only deals with available network buffering at a peer. We also want to be concerned about having available space in outbound data transfer queues. That is, as we issue commands to the local NIC to send data, that those commands can be queued at the NIC. When we consider reliability, this means tracking outstanding requests until they have been acknowledged. Resource management will need to ensure that we do not overflow that request queue.

Additionally, supporting asynchronous operations (described in detail below) will introduce potential new queues. Those queues must not overflow as well.

## Asynchronous Operations

Arguably, the key feature of achieving high-performance is supporting asynchronous operations. The socket API supports asynchronous transfers with its non-blocking mode. However, because the API itself operates synchronously, the result is additional data copies. For an API to be asynchronous, an application needs to be able to submit work, then later receive some sort of notification that the work is done. In order to avoid extra memory copies, the application must agree not to modify its data buffers until the operation completes.

There are two main ways to notify an application that it is safe to re-use its data buffers. One mechanism is for the network layer to invoke some sort of callback or send a signal to the application that the request is done. Some asynchronous APIs use this mechanism. The drawback of this approach is that signals interrupt an application's processing. This can negatively impact the CPU caches, plus requires interrupt processing. Additionally, it is often difficult to develop an application that can handle processing a signal that can occur at anytime.

An alternative mechanism for supporting asynchronous operations is to write events into some sort of completion queue when an operation completes. This provides a way to indicate to an application when a data transfer has completed, plus gives the application control over when and how to process completed requests. For example, it can process requests in batches to improve code locality and performance.

### Interrupts and Signals

Interrupts are a natural extension to supporting asynchronous operations. However, when dealing with an asynchronous API, they can negatively impact performance. Interrupts, even when directed to a kernel agent, can interfere with application processing.

If an application has an asynchronous interface with completed operations written into a completion queue, it is often sufficient for the application to simply check the queue for events. As long as the application has other work to perform, there is no need for it to block. This alleviates the need for interrupt generation. A NIC merely needs to write an entry into the completion queue and update a tail pointer to signal that a request is done.

If we follow this argument, then it can be beneficial to give the application control over when interrupts should occur and when to write events to some sort of wait object. By having the application notify the network layer that it will wait until a completion occurs, we can better manage the number and type of interrupts that are generated.

### Event Queues

As outlined above, there are performance advantages to having an API that reports completions or provides other types of notification using an event queue. A very simple type of event queue merely tracks completed operations. As data is received or a send completes, an entry is written into the event queue.

## Direct Hardware Access

When discussing the network layer, most software implementations refer to kernel modules responsible for implementing the necessary transport and network protocols. However, if we want network latency to approach sub-microsecond speeds, then we need to remove as much software between the application and its access to the hardware as possible. One way to do this is for the application to have direct access to the network interface controller's command queues. Similarly, the NIC requires direct access to the application's data buffers and control structures, such as the above mentioned completion queues.

Note that when we speak about an application having direct access to network hardware, we're referring to the application process. Naturally, an application developer is highly unlikely to code for a specific hardware NIC. That work would be left to some sort of network library specifically targeting the NIC. The actual network layer, which implements the network transport, could be part of the network library or offloaded onto the NIC's hardware or firmware.

### Kernel Bypass

Kernel bypass is a feature that allows the application to avoid calling into the kernel for data transfer operations. This is possible when it has direct access to the NIC hardware. Complete kernel bypass is impractical because of security concerns and resource management constraints. However, it is possible to avoid kernel calls for what are called 'fast-path' operations, such as send or receive.

For security and stability reasons, operating system kernels cannot rely on data that comes from user space applications. As a result, even a simple kernel call often requires acquiring and releasing locks, coupled with data verification checks. If we can limit the effects of a poorly written or malicious application to its own process space, we can avoid the overhead that comes with kernel validation without impacting system stability.

### Direct Data Placement

Direct data placement means avoiding data copies when sending and receiving data, plus placing received data into the correct memory buffer where needed. On a broader scale, it is part of having direct hardware access, with the application and NIC communicating directly with shared memory buffers and queues.

Direct data placement is often thought of by those familiar with RDMA - remote direct memory access. RDMA is a technique that allows reading and writing memory that belongs to a peer process that is running on a node across the network. Advanced RDMA hardware is capable of accessing the target memory buffers without involving the execution of the peer process. RDMA relies on offloading the network transport onto the NIC in order to avoid interrupting the target process.

The main advantages of supporting direct data placement is avoiding memory copies and minimizing processing overhead.

# Designing Interfaces for Performance

We want to design a network interface that can meet the requirements outlined above. Moreover, we also want to take into account the performance of the interface itself. It is often not obvious how an interface can adversely affect performance, versus performance being a result of the underlying implementation. The following sections describe how interface choices can impact performance. Of course, when we begin defining the actual APIs that an application will use, we will need to trade off raw performance for ease of use where it makes sense.

When considering performance goals for an API, we need to take into account the target application use cases. For the purposes of this discussion, we want to consider applications that communicate with thousands to millions of peer processes. Data transfers will include millions of small messages per second per peer, and large transfers that may be up to gigabytes of data. At such extreme scales, even small optimizations are measurable, in terms of both performance and power. If we have a million peers sending a millions messages per second, eliminating even a single instruction from the code path quickly multiplies to saving billions of instructions per second from the overall execution, when viewing the operation of the entire application.

We once again refer to the socket API as part of this discussion in order to illustrate how an API can affect performance.

/\* Notable socket function prototypes \*/  
/\* "control" functions \*/  
int socket(int domain, int type, int protocol);  
int bind(int socket, const struct sockaddr \*addr, socklen\_t addrlen);  
int listen(int socket, int backlog);  
int accept(int socket, struct sockaddr \*addr, socklen\_t \*addrlen);  
int connect(int socket, const struct sockaddr \*addr, socklen\_t addrlen);  
int shutdown(int socket, int how);  
int close(int socket);   
  
/\* "fast path" data operations - send only (receive calls not shown) \*/  
ssize\_t send(int socket, const void \*buf, size\_t len, int flags);  
ssize\_t sendto(int socket, const void \*buf, size\_t len, int flags,  
 const struct sockaddr \*dest\_addr, socklen\_t addrlen);  
ssize\_t sendmsg(int socket, const struct msghdr \*msg, int flags);  
ssize\_t write(int socket, const void \*buf, size\_t count);  
ssize\_t writev(int socket, const struct iovec \*iov, int iovcnt);  
  
/\* "indirect" data operations \*/  
int poll(struct pollfd \*fds, nfds\_t nfds, int timeout);  
int select(int nfds, fd\_set \*readfds, fd\_set \*writefds,  
 fd\_set \*exceptfds, struct timeval \*timeout);

Examining this list, there are a couple of features to note. First, there are multiple calls that can be used to send data, as well as multiple calls that can be used to wait for a non-blocking socket to become ready. This will be discussed in more detail further on. Second, the operations have been split into different groups (terminology is ours). Control operations are those functions that an application seldom invokes during execution. They often only occur as part of initialization.

Data operations, on the other hand, may be called hundreds to millions of times during an application's lifetime. They deal directly or indirectly with transferring or receiving data over the network. Data operations can be split into two groups. Fast path calls interact with the network stack to immediately send or receive data. In order to achieve high bandwidth and low latency, those operations need to be as fast as possible. Non-fast path operations that still deal with data transfers are those calls, that while still frequently called by the application, are not as performance critical. For example, the select() and poll() calls are used to block an application thread until a socket becomes ready. Because those calls suspend the thread execution, performance is a lesser concern. (Performance of those operations is still of a concern, but the cost of executing the operating system scheduler often swamps any but the most substantial performance gains.)

## Call Setup Costs

The amount of work that an application needs to perform before issuing a data transfer operation can affect performance, especially message rates. Obviously, the more parameters an application must push on the stack to call a function increases its instruction count. However, replacing stack variables with a single data structure does not help to reduce the setup costs.

Suppose that an application wishes to send a single data buffer of a given size to a peer. If we examine the socket API, the best fit for such an operation is the write() call. That call takes only those values which are necessary to perform the data transfer. The send() call is a close second, and send() is a more natural function name for network communication, but send() requires one extra argument over write(). Other functions are even worse in terms of setup costs. The sendmsg() function, for example, requires that the application format a data structure, the address of which is passed into the call. This requires significantly more instructions from the application if done for every data transfer.

Even though all other send functions can be replaced by sendmsg(), it is useful to have multiple ways for the application to issue send requests. Not only are the other calls easier to read and use (which lower software maintenance costs), but they can also improve performance.

## Branches and Loops

When designing an API, developers rarely consider how the API impacts the underlying implementation. However, the selection of API parameters can require that the underlying implementation add branches or use control loops. Consider the difference between the write() and writev() calls. The latter passes in an array of I/O vectors, which may be processed using a loop such as this:

/\* Sample implementation for processing an array \*/  
for (i = 0; i < iovcnt; i++) {  
 ...  
}

In order to process the iovec array, the natural software construct would be to use a loop to iterate over the entries. Loops result in additional processing. Typically, a loop requires initializing a loop control variable (e.g. i = 0), adds ALU operations (e.g. i++), and a comparison (e.g. i < iovcnt). This overhead is necessary to handle an arbitrary number of iovec entries. If the common case is that the application wants to send a single data buffer, write() is a better option.

In addition to control loops, an API can result in the implementation needing branches. Branches can change the execution flow of a program, impacting processor pipe-lining techniques. Processor branch prediction helps alleviate this issue. However, while branch prediction can be correct nearly 100% of the time while running a micro-benchmark, such as a network bandwidth or latency test, with more realistic network traffic, the impact can become measurable.

We can easily see how an API can introduce branches into the code flow if we examine the send() call. Send() takes an extra flags parameter over the write() call. This allows the application to modify the behavior of send(). From the viewpoint of implementing send(), the flags parameter must be checked. In the best case, this adds one additional check (flags are non-zero). In the worst case, every valid flag may need a separate check, resulting in potentially dozens of checks.

Overall, the sockets API is well designed considering these performance implications. It provides complex calls where they are needed, with simpler functions available that can avoid some of the overhead inherent in other calls.

## Command Formatting

The ultimate objective of invoking a network function is to transfer or receive data from the network. In this section, we're dropping to the very bottom of the software stack to the component responsible for directly accessing the hardware. This is usually referred to as the network driver, and its implementation is often tied to a specific piece of hardware, or a series of NICs by a single hardware vendor.

In order to signal a NIC that it should read a memory buffer and copy that data onto the network, the software driver usually needs to write some sort of command to the NIC. To limit hardware complexity and cost, a NIC may only support a couple of command formats. This differs from the software interfaces that we've been discussing, where we can have different APIs of varying complexity in order to reduce overhead. There can be significant costs associated with formatting the command and posting it to the hardware.

With a standard NIC, the command is formatted by a kernel driver. That driver sits at the bottom of the network stack servicing requests from multiple applications. It must typically format each command only after a request has passed through the network stack.

With devices that are directly accessible by a single application, there are opportunities to use pre-formatted command structures. The more of the command that can be initialized prior to the application submitting a network request, the more streamlined the process, and the better the performance.

As an example, a NIC needs to have the destination address as part of a send operation. If an application is sending to a single peer, that information can be cached and be part of a pre-formatted network header. This is only possible if the NIC driver knows that the destination will not change between sends. The closer that the driver can be to the application, the greater the chance for optimization. An optimal approach is for the driver to be part of a library that executes entirely within the application process space.

## Memory Footprint

Memory footprint concerns are most notable among high-performance computing (HPC) applications that communicate with thousands of peers. Excessive memory consumption impacts application scalability, limiting the number of peers that can operate in parallel to solve problems. There is often a trade-off between minimizing the memory footprint needed for network communication, application performance, and ease of use of the network interface.

As we discussed with the socket API semantics, part of the ease of using sockets comes from the network layering copying the user's buffer into an internal buffer belonging to the network stack. The amount of internal buffering that's made available to the application directly correlates with the bandwidth that an application can achieve. In general, larger internal buffering increases network performance, with a cost of increasing the memory footprint consumed by the application. This memory footprint exists independent of the amount of memory allocated directly by the application. Eliminating network buffering not only helps with performance, but also scalability, by reducing the memory footprint needed to support the application.

While network memory buffering increases as an application scales, it can often be configured to a fixed size. The amount of buffering needed is dependent on the number of active communication streams being used at any one time. That number is often significantly lower than the total number of peers that an application may need to communicate with. The amount of memory required to *address* the peers, however, usually has a linear relationship with the total number of peers.

With the socket API, each peer is identified using a struct sockaddr. If we consider a UDP based socket application using IPv4 addresses, a peer is identified by the following address.

/\* IPv4 socket address - with typedefs removed \*/  
struct sockaddr\_in {  
 uint16\_t sin\_family; /\* AF\_INET \*/  
 uint16\_t sin\_port;  
 struct {  
 uint32\_t sin\_addr;  
 } in\_addr;  
};

In total, the application requires 8-bytes of addressing for each peer. If the app communicates with a million peers, that explodes to roughly 8 MB of memory space that is consumed just to maintain the address list. If IPv6 addressing is needed, then the requirement increases by a factor of 4.

Luckily, there are some tricks that can be used to help reduce the addressing memory footprint, though doing so will introduce more instructions into code path to access the network stack. For instance, we can notice that all addresses in the above example have the same sin\_family value (AF\_INET). There's no need to store that for each address. This potentially shrinks each address from 8 bytes to 6. (We may be left with unaligned data, but that's a trade-off to reducing the memory consumption). Depending on how the addresses are assigned, further reduction may be possible. For example, if the application uses the same set of port addresses at each node, then we can eliminate storing the port, and instead calculate it from some base value. This type of trick can be applied to the IP portion of the address if the app is lucky enough to run across sequential IP addresses.

The main issue with this sort of address reduction is that it is difficult to achieve. It requires that each application check for and handle address compression, exposing the application to the addressing format used by the networking stack. It should be kept in mind that TCP/IP and UDP/IP addresses are logical addresses, not physical. When running over Ethernet, the addresses that appear at the link layer are MAC addresses, not IP addresses. The IP to MAC address association is managed by the network software. We would like to provide addressing that is simple for an application to use, but at the same time can provide a minimal memory footprint.

## Communication Resources

We need to take a brief detour in the discussion in order to delve deeper into the network problem and solution space. Instead of continuing to think of a socket as a single entity, with both send and receive capabilities, we want to consider its components separately. A network socket can be viewed as three basic constructs: a transport level address, a send or transmit queue, and a receive queue. Because our discussion will begin to pivot away from pure socket semantics, we will refer to our network 'socket' as an endpoint.

In order to reduce an application's memory footprint, we need to consider features that fall outside of the socket API. So far, much of the discussion has been around sending data to a peer. We now want to focus on the best mechanisms for receiving data.

With sockets, when an app has data to receive (indicated, for example, by a POLLIN event), we call recv(). The network stack copies the receive data into its buffer and returns. If we want to avoid the data copy on the receive side, we need a way for the application to post its buffers to the network stack *before* data arrives.

Arguably, a natural way of extending the socket API to support this feature is to have each call to recv() simply post the buffer to the network layer. As data is received, the receive buffers are removed in the order that they were posted. Data is copied into the posted buffer and returned to the user. It would be noted that the size of the posted receive buffer may be larger (or smaller) than the amount of data received. If the available buffer space is larger, hypothetically, the network layer could wait a short amount of time to see if more data arrives. If nothing more arrives, the receive completes with the buffer returned to the application.

This raises an issue regarding how to handle buffering on the receive side. So far, with sockets we've mostly considered a streaming protocol. However, many applications deal with messages which end up being layered over the data stream. If they send an 8 KB message, they want the receiver to receive an 8 KB message. Message boundaries need to be maintained.

If an application sends and receives a fixed sized message, buffer allocation becomes trivial. The app can post X number of buffers each of an optimal size. However, if there is a wide mix in message sizes, difficulties arise. It is not uncommon for an app to have 80% of its messages be a couple hundred of bytes or less, but 80% of the total data that it sends to be in large transfers that are, say, a megabyte or more. Pre-posting receive buffers in such a situation is challenging.

A commonly used technique used to handle this situation is to implement one application level protocol for smaller messages, and use a separate protocol for transfers that are larger than some given threshold. This would allow an application to post a bunch of smaller messages, say 4 KB, to receive data. For transfers that are larger than 4 KB, a different communication protocol is used, possibly over a different socket or endpoint.

### Shared Receive Queues

If an application pre-posts receive buffers to a network queue, it needs to balance the size of each buffer posted, the number of buffers that are posted to each queue, and the number of queues that are in use. With a socket like approach, each socket would maintain an independent receive queue where data is placed. If an application is using 1000 endpoints and posts 100 buffers, each 4 KB, that results in 400 MB of memory space being consumed to receive data. (We can start to realize that by eliminating memory copies, one of the trade offs is increased memory consumption.) While 400 MB seems like a lot of memory, there is less than half a megabyte allocated to a single receive queue. At today's networking speeds, that amount of space can be consumed within milliseconds. The result is that if only a few endpoints are in use, the application will experience long delays where flow control will kick in and back the transfers off.

There are a couple of observations that we can make here. The first is that in order to achieve high scalability, we need to move away from a connection-oriented protocol, such as streaming sockets. Secondly, we need to reduce the number of receive queues that an application uses.

A shared receive queue is a network queue that can receive data for many different endpoints at once. With shared receive queues, we no longer associate a receive queue with a specific transport address. Instead network data will target a specific endpoint address. As data arrives, the endpoint will remove an entry from the shared receive queue, place the data into the application's posted buffer, and return it to the user. Shared receive queues can greatly reduce the amount of buffer space needed by an applications. In the previous example, if a shared receive queue were used, the app could post 10 times the number of buffers (1000 total), yet still consume 100 times less memory (4 MB total). This is far more scalable. The drawback is that the application must now be aware of receive queues and shared receive queues, rather than considering the network only at the level of a socket.

### Multi-Receive Buffers

Shared receive queues greatly improve application scalability; however, it still results in some inefficiencies as defined so far. We've only considered the case of posting a series of fixed sized memory buffers to the receive queue. As mentioned, determining the size of each buffer is challenging. Transfers larger than the fixed size require using some other protocol in order to complete. If transfers are typically much smaller than the fixed size, then the extra buffer space goes unused.

Again referring to our example, if the application posts 1000 buffers, then it can only receive 1000 messages before the queue is emptied. At data rates measured in millions of messages per second, this will introduce stalls in the data stream. An obvious solution is to increase the number of buffers posted. The problem is dealing with variable sized messages, including some which are only a couple hundred bytes in length. For example, if the average message size in our case is 256 bytes or less, then even though we've allocated 4 MB of buffer space, we only make use of 6% of that space. The rest is wasted in order to handle messages which may only occasionally be up to 4 KB.

A second optimization that we can make is to fill up each posted receive buffer as messages arrive. So, instead of a 4 KB buffer being removed from use as soon as a single 256 byte message arrives, it can instead receive up to 16, 256 byte, messages. We refer to such a feature as 'multi-receive' buffers.

With multi-receive buffers, instead of posting a bunch of smaller buffers, we instead post a single larger buffer, say the entire 4 MB, at once. As data is received, it is placed into the posted buffer. Unlike TCP streams, we still maintain message boundaries. The advantages here are twofold. Not only is memory used more efficiently, allowing us to receive more smaller messages at once and larger messages overall, but we reduce the number of function calls that the application must make to maintain its supply of available receive buffers.

When combined with shared receive queues, multi-receive buffers help support optimal receive side buffering and processing. The main drawback to supporting multi-receive buffers are that the application will not necessarily know up front how many messages may be associated with a single posted memory buffer. This is rarely a problem for applications.

## Optimal Hardware Allocation

As part of scalability considerations, we not only need to consider the processing and memory resources of the host system, but also the allocation and use of the NIC hardware. We've referred to network endpoints as combination of transport addressing, transmit queues, and receive queues. The latter two queues are often implemented as hardware command queues. Command queues are used to signal the NIC to perform some sort of work. A transmit queue indicates that the NIC should transfer data. A transmit command often contains information such as the address of the buffer to transmit, the length of the buffer, and destination addressing data. The actual format and data contents vary based on the hardware implementation.

NICs have limited resources. Only the most scalable, high-performance applications likely need to be concerned with utilizing NIC hardware optimally. However, such applications are an important and specific focus of OFI. Managing NIC resources is often handled by a resource manager application, which is responsible for allocating systems to competing applications, among other activities.

Supporting applications that wish to make optimal use of hardware requires that hardware related abstractions be exposed to the application. Such abstractions cannot require a specific hardware implementation, and care must be taken to ensure that the resulting API is still usable by developers unfamiliar with dealing with such low level details. Exposing concepts such as shared receive queues is an example of giving an application more control over how hardware resources are used.

### Sharing Command Queues

By exposing the transmit and receive queues to the application, we open the possibility for the application that makes use of multiple endpoints to determine how those queues might be shared. We talked about the benefits of sharing a receive queue among endpoints. The benefits of sharing transmit queues are not as obvious.

An application that uses more addressable endpoints than there are transmit queues will need to share transmit queues among the endpoints. By controlling which endpoint uses which transmit queue, the application can prioritize traffic. A transmit queue can also be configured to optimize for a specific type of data transfer, such as large transfers only.

From the perspective of a software API, sharing transmit or receive queues implies exposing those constructs to the application, and allowing them to be associated with different endpoint addresses.

### Multiple Queues

The opposite of a shared command queue are endpoints that have multiple queues. An application that can take advantage of multiple transmit or receive queues can increase parallel handling of messages without synchronization constraints. Being able to use multiple command queues through a single endpoint has advantages over using multiple endpoints. Multiple endpoints require separate addresses, which increases memory use. A single endpoint with multiple queues can continue to expose a single address, while taking full advantage of available NIC resources.

## Progress Model Considerations

One aspect of the sockets programming interface that developers often don't consider is the location of the protocol implementation. This is usually managed by the operating system kernel. The network stack is responsible for handling flow control messages, timing out transfers, re-transmitting unacknowledged transfers, processing received data, and sending acknowledgments. This processing requires that the network stack consume CPU cycles. Portions of that processing can be done within the context of the application thread, but much must be handled by kernel threads dedicated to network processing.

By moving the network processing directly into the application process, we need to be concerned with how network communication makes forward progress. For example, how and when are acknowledgments sent? How are timeouts and message re-transmissions handled? The progress model defines this behavior, and it depends on how much of the network processing has been offloaded onto the NIC.

More generally, progress is the ability of the underlying network implementation to complete processing of an asynchronous request. In many cases, the processing of an asynchronous request requires the use of the host processor. For performance reasons, it may be undesirable for the provider to allocate a thread for this purpose, which will compete with the application thread(s). We can avoid thread context switches if the application thread can be used to make forward progress on requests -- check for acknowledgments, retry timed out operations, etc. Doing so requires that the application periodically call into the network stack.

## Ordering

Network ordering is a complex subject. With TCP sockets, data is sent and received in the same order. Buffers are re-usable by the application immediately upon returning from a function call. As a result, ordering is simple to understand and use. UDP sockets complicate things slightly. With UDP sockets, messages may be received out of order from how they were sent. In practice, this often doesn't occur, particularly, if the application only communicates over a local area network, such as Ethernet.

With our evolving network API, there are situations where exposing different order semantics can improve performance. These details will be discussed further below.

### Messages

UDP sockets allow messages to arrive out of order because each message is routed from the sender to the receiver independently. This allows packets to take different network paths, to avoid congestion or take advantage of multiple network links for improved bandwidth. We would like to take advantage of the same features in those cases where the application doesn't care in which order messages arrive.

Unlike UDP sockets, however, our definition of message ordering is more subtle. UDP messages are small, MTU sized packets. In our case, messages may be gigabytes in size. We define message ordering to indicate whether the start of each message is processed in order or out of order. This is related to, but separate from the order of how the message payload is received.

An example will help clarify this distinction. Suppose that an application has posted two messages to its receive queue. The first receive points to a 4 KB buffer. The second receive points to a 64 KB buffer. The sender will transmit a 4 KB message followed by a 64 KB message. If messages are processed in order, then the 4 KB send will match with the 4 KB received, and the 64 KB send will match with the 64 KB receive. However, if messages can be processed out of order, then the sends and receives can mismatch, resulting in the 64 KB send being truncated.

In this example, we're not concerned with what order the data is received in. The 64 KB send could be broken in 64 1-KB transfers that take different routes to the destination. So, bytes 2k-3k could be received before bytes 1k-2k. Message ordering is not concerned with ordering *within* a message, only *between* messages. With ordered messages, the messages themselves need to be processed in order.

The more relaxed message ordering can be the more optimizations that the network stack can use to transfer the data. However, the application must be aware of message ordering semantics, and be able to select the desired semantic for its needs. For the purposes of this section, messages refers to transport level operations, which includes RDMA and similar operations (some of which have not yet been discussed).

### Data

Data ordering refers to the receiving and placement of data both *within and between* messages. Data ordering is most important to messages that can update the same target memory buffer. For example, imagine an application that writes a series of database records directly into a peer memory location. Data ordering, combined with message ordering, ensures that the data from the second write updates memory after the first write completes. The result is that the memory location will contain the records carried in the second write.

Enforcing data ordering between messages requires that the messages themselves be ordered. Data ordering can also apply within a single message, though this level of ordering is usually less important to applications. Intra-message data ordering indicates that the data for a single message is received in order. Some applications use this feature to 'spin' reading the last byte of a receive buffer. Once the byte changes, the application knows that the operation has completed and all earlier data has been received. (Note that while such behavior is interesting for benchmark purposes, using such a feature in this way is strongly discouraged. It is not portable between networks or platforms.)

### Completions

Completion ordering refers to the sequence that asynchronous operations report their completion to the application. Typically, unreliable data transfer will naturally complete in the order that they are submitted to a transmit queue. Each operation is transmitted to the network, with the completion occurring immediately after. For reliable data transfers, an operation cannot complete until it has been acknowledged by the peer. Since ack packets can be lost or possibly take different paths through the network, operations can be marked as completed out of order. Out of order acks is more likely if messages can be processed out of order.

Asynchronous interfaces require that the application track their outstanding requests. Handling out of order completions can increase application complexity, but it does allow for optimizing network utilization.

# OFI Architecture

Libfabric is well architected to support the previously discussed features, with specific focus on exposing direct network access to an application. Direct network access, sometimes referred to as RDMA, allows an application to access network resources without operating system intervention. Data transfers can occur between networking hardware and application memory with minimal software overhead. Although libfabric supports scalable network solutions, it does not mandate any implementation. And the APIs have been defined specifically to allow multiple implementations.

The following diagram highlights the general architecture of the interfaces exposed by libfabric. For reference, the diagram shows libfabric in reference to a NIC.

## Framework versus Provider

OFI is divided into two separate components. The main component is the OFI framework, which defines the interfaces that applications use. The OFI framework provides some generic services; however, the bulk of the OFI implementation resides in the providers. Providers plug into the framework and supply access to fabric hardware and services. Providers are often associated with a specific hardware device or NIC. Because of the structure of the OFI framework, applications access the provider implementation directly for most operations, in order to ensure the lowest possible software latency.

One important provider is referred to as the sockets provider. This provider implements the libfabric API over TCP sockets. A primary objective of the sockets provider is to support development efforts. Developers can write and test their code over the sockets provider on a small system, possibly even a laptop, before debugging on a larger cluster. The sockets provider can also be used as a fallback mechanism for applications that wish to target libfabric features for high-performance networks, but which may still need to run on small clusters connected, for example, by Ethernet.

The UDP provider has a similar goal, but implements a much smaller feature set than the sockets provider. The UDP provider is implemented over UDP sockets. It only implements those features of libfabric which would be most useful for applications wanting unreliable, unconnected communication. The primary goal of the UDP provider is to provide a simple building block upon which the framework can construct more complex features, such as reliability. As a result, a secondary objective of the UDP provider is to improve application scalability when restricted to using native operating system sockets.

The final generic (not associated with a specific network technology) provider is often referred to as the utility provider. The utility provider is a collection of software modules that can be used to extend the feature coverage of any provider. For example, the utility provider layers over the UDP provider to implement connection-oriented and reliable endpoint types. It can similarly layer over a provider that only supports connection-oriented communication to expose reliable, connection-less (aka reliable datagram) semantics.

Other providers target specific network technologies and systems, such as InfiniBand, Cray Aries networks, or Intel Omni-Path Architecture.

## Control services

Control services are used by applications to discover information about the types of communication services available in the system. For example, discovery will indicate what fabrics are reachable from the local node, and what sort of communication each provides.

In terms of implementation, control services are handled primarily by a single API, fi\_getinfo(). Modeled very loosely on getaddrinfo(), it is used not just to discover what features are available in the system, but also how they might best be used by an application desiring maximum performance.

Control services themselves are not considered performance critical. However, the information exchanged between an application and the providers must be expressive enough to indicate the most performant way to access the network. Those details must be balanced with ease of use. As a result, the fi\_getinfo() call provides the ability to access complex network details, while allowing an application to ignore them if desired.

## Communication Services

Communication interfaces are used to setup communication between nodes. It includes calls to establish connections (connection management), as well as functionality used to address connection-less endpoints (address vectors).

The best match to socket routines would be connect(), bind(), listen(), and accept(). In fact the connection management calls are modeled after those functions, but with improved support for the asynchronous nature of the calls. For performance and scalability reasons, connection-less endpoints use a unique model, that is not based on sockets or other network interfaces. Address vectors are discussed in detail later, but target applications needing to talk with potentially thousands to millions of peers. For applications communicating with a handful of peers, address vectors can slightly complicate initialization for connection-less endpoints. (Connection-oriented endpoints may be a better option for such applications).

## Completion Services

OFI exports asynchronous interfaces. Completion services are used to report the results of submitted data transfer operations. Completions may be reported using the cleverly named completions queues, which provide details about the operation that completed. Or, completions may be reported using lower-impact counters that simply return the number of operations that have completed.

Completion services are designed with high-performance, low-latency in mind. The calls map directly into the providers, and data structures are defined to minimize memory writes and cache impact. Completion services do not have corresponding socket APIs. (For Windows developers, they are similar to IO completion ports).

## Data Transfer Services

Applications have needs of different data transfer semantics. The data transfer services in OFI are designed around different communication paradigms. Although shown outside the data transfer services, triggered operations are strongly related to the data transfer operations.

There are four basic data transfer interface sets. Message queues expose the ability to send and receive data with message boundaries being maintained. Message queues act as FIFOs, with sent messages matched with receive buffers in the order that messages are received. The message queue APIs are derived from the socket data transfer APIs, such as send(). sendto(), sendmsg(), recv(), recvmsg(), etc.

Tag matching is similar to message queues in that it maintains message boundaries. Tag matching differs from message queues in that received messages are directed into buffers based on small steering tags that are carried in the sent message. This allows a receiver to post buffers labeled 1, 2, 3, and so forth, with sends labeled respectively. The benefit is that send 1 will match with receive buffer 1, independent of how send operations may be transmitted or re-ordered by the network.

RMA stands for remote memory access. RMA transfers allow an application to write data directly into a specific memory location in a target process, or to read memory from a specific address at the target process and return the data into a local buffer. RMA is essentially equivalent to RDMA; the exception being that RDMA originally defined a specific transport implementation of RMA.

Atomic operations are often viewed as a type of extended RMA transfer. They permit direct access to the memory on the target process. The benefit of atomic operations is that they allow for manipulation of the memory, such as incrementing the value found at the target buffer. So, where RMA can write the value X to a remote memory buffer, atomics can change the value of the remote memory buffer, say Y, to Y + 1. Because RMA and atomic operations provide direct access to a process’s memory buffers, additional security synchronization is needed.

## Memory Registration

Memory registration is the security mechanism used to grant a remote peer access to local memory buffers. Registered memory regions associate memory buffers with permissions granted for access by fabric resources. A memory buffer must be registered before it can be used as the target of an RMA or atomic data transfer. Memory registration supports a simple protection mechanism. After a memory buffer has been registered, that registration request (buffer's address, buffer length, and access permission) is given a registration key. Peers that issue RMA or atomic operations against that memory buffer must provide this key as part of their operation. This helps protects against unintentional accesses to the region. (Memory registration can help guard against malicious access, but it is often too weak by itself to ensure system isolation. Other, fabric specific, mechanisms protect against malicious access. Those mechanisms are currently outside of the scope of the libfabric API.)

Memory registration often plays a secondary role with high-performance networks. In order for a NIC to read or write application memory directly, it must access the physical memory pages that back the application's address space. Modern operating systems employ page files that swap out virtual pages from one process with the virtual pages from another. As a result, a physical memory page may map to different virtual addresses depending on when it is accessed. Furthermore, when a virtual page is swapped in, it may be mapped to a new physical page. If a NIC attempts to read or write application memory without being linked into the virtual address manager, it could access the wrong data, possibly corrupting an application's memory. Memory registration can be used to avoid this situation from occurring. For example, registered pages can be marked such that the operating system locks the virtual to physical mapping, avoiding any possibility of the virtual page being paged out or remapped.

# Object Model

Interfaces exposed by OFI are associated with different objects. The following diagram shows a high-level view of the parent-child relationships.

## Fabric

A fabric represents a collection of hardware and software resources that access a single physical or virtual network. For example, a fabric may be a single network subnet or cluster. All network ports on a system that can communicate with each other through the fabric belong to the same fabric. A fabric shares network addresses and can span multiple providers.

Fabrics are the top level object from which other objects are allocated.

## Domain

A domain represents a logical connection into a fabric. For example, a domain may correspond to a physical or virtual NIC. Because domains often correlate to a single NIC, a domain defines the boundary within which other resources may be associated. Objects such as completion queues and active endpoints must be part of the same domain in order to be related to each other.

## Passive Endpoint

Passive endpoints are used by connection-oriented protocols to listen for incoming connection requests. Passive endpoints often map to software constructs and may span multiple domains. They are best represented by a listening socket. Unlike the socket API, however, in which an allocated socket may be used with either a connect() or listen() call, a passive endpoint may only be used with a listen call.

## Event Queues

EQs are used to collect and report the completion of asynchronous operations and events. Event queues handle *control* events, which are not directly associated with data transfer operations. The reason for separating control events from data transfer events is for performance reasons. Control events usually occur during an application's initialization phase, or at a rate that's several orders of magnitude smaller than data transfer events. Event queues are most commonly used by connection-oriented protocols for notification of connection request or established events. A single event queue may combine multiple hardware queues with a software queue and expose them as a single abstraction.

## Wait Sets

The intended objective of a wait set is to reduce system resources used for signaling events. For example, a wait set may allocate a single file descriptor. All fabric resources that are associated with the wait set will signal that file descriptor when an event occurs. The advantage is that the number of opened file descriptors is greatly reduced. The closest operating system semantic would be the Linux epoll construct. The difference is that a wait set does not merely multiplex file descriptors to another file descriptor, but allows for their elimination completely. Wait sets allow a single underlying wait object to be signaled whenever a specified condition occurs on an associated event queue, completion queue, or counter.

## Active Endpoint

Active endpoints are data transfer communication portals. Active endpoints are used to perform data transfers, and are conceptually similar to a connected TCP or UDP socket. Active endpoints are often associated with a single hardware NIC, with the data transfers partially or fully offloaded onto the NIC.

## Completion Queue

Completion queues are high-performance queues used to report the completion of data transfer operations. Unlike event queues, completion queues are often associated with a single hardware NIC, and may be implemented entirely in hardware. Completion queue interfaces are designed to minimize software overhead.

## Completion Counter

Completion queues are used to report information about which request has completed. However, some applications use this information simply to track how many requests have completed. Other details are unnecessary. Completion counters are optimized for this use case. Rather than writing entries into a queue, completion counters allow the provider to simply increment a count whenever a completion occurs.

## Poll Set

OFI allows providers to use an application’s thread to process asynchronous requests. This can provide performance advantages for providers that use software to progress the state of a data transfer. Poll sets allow an application to group together multiple objects, such that progress can be driven across all associated data transfers. In general, poll sets are used to simplify applications where a manual progress model is employed.

## Memory Region

Memory regions describe application’s local memory buffers. In order for fabric resources to access application memory, the application must first grant permission to the fabric provider by constructing a memory region. Memory regions are required for specific types of data transfer operations, such as RMA and atomic operations.

## Address Vectors

Address vectors are used by connection-less endpoints. They map higher level addresses, such as IP addresses, which may be more natural for an application to use, into fabric specific addresses. The use of address vectors allows providers to reduce the amount of memory required to maintain large address look-up tables, and eliminate expensive address resolution and look-up methods during data transfer operations.

# Communication Model

OFI supports three main communication endpoint types: reliable-connected, unreliable datagram, and reliable-unconnected. (The fourth option, unreliable-connected is unused by applications, so is not included as part of the current implementation). Communication setup is based on whether the endpoint is connected or unconnected. Reliability is a feature of the endpoint's data transfer protocol.

## Connected Communications

The following diagram highlights the general usage behind connection-oriented communication. Connected communication is based on the flow used to connect TCP sockets, with improved asynchronous support.

Connections require the use of both passive and active endpoints. In order to establish a connection, an application must first create a passive endpoint and associate it with an event queue. The event queue will be used to report the connection management events. The application then calls listen on the passive endpoint. A single passive endpoint can be used to form multiple connections.

The connecting peer allocates an active endpoint, which is also associated with an event queue. Connect is called on the active endpoint, which results in sending a connection request (CONNREQ) message to the passive endpoint. The CONNREQ event is inserted into the passive endpoint’s event queue, where the listening application can process it.

Upon processing the CONNREQ, the listening application will allocate an active endpoint to use with the connection. The active endpoint is bound with an event queue. Although the diagram shows the use of a separate event queue, the active endpoint may use the same event queue as used by the passive endpoint. Accept is called on the active endpoint to finish forming the connection. It should be noted that the OFI accept call is different than the accept call used by sockets. The differences result from OFI supporting process direct I/O.

OFI does not define the connection establishment protocol, but does support a traditional three-way handshake used by many technologies. After calling accept, a response is sent to the connecting active endpoint. That response generates a CONNECTED event on the remote event queue. If a three-way handshake is used, the remote endpoint will generate an acknowledgment message that will generate a CONNECTED event for the accepting endpoint. Regardless of the connection protocol, both the active and passive sides of the connection will receive a CONNECTED event that signals that the connection has been established.

## Connection-less Communications

Connection-less communication allows data transfers between active endpoints without going through a connection setup process. The diagram below shows the basic components needed to setup connection-less communication. Connection-less communication setup differs from UDP sockets in that it requires that the remote addresses be stored with libfabric.

OFI requires the addresses of peer endpoints be inserted into a local addressing table, or address vector, before data transfers can be initiated against the remote endpoint. Address vectors abstract fabric specific addressing requirements and avoid long queuing delays on data transfers when address resolution is needed. For example, IP addresses may need to be resolved into Ethernet MAC addresses. Address vectors allow this resolution to occur during application initialization time. OFI does not define how an address vector be implemented, only its conceptual model.

Because address vector setup is considered a control operation, and often occurs during an application's initialization phase, they may be used both synchronously and asynchronously. When used synchronously, calls to insert new addresses into the AV block until the resolution completes. When an address vector is used asynchronously, it must be associated with an event queue. With the asynchronous model, after an address has been inserted into the AV and the fabric specific details have been resolved, a completion event is generated on the event queue. Data transfer operations against that address are then permissible on active endpoints that are associated with the address vector.

All connection-less endpoints that transfer data must be associated with an address vector.

# Endpoints

Endpoints represent communication portals, and all data transfer operations are initiated on endpoints. OFI defines the conceptual model for how endpoints are exposed to applications, as demonstrated in the diagrams below.

Endpoints are usually associated with a transmit context and a receive context. Transmit and receive contexts are often implemented using hardware queues that are mapped directly into the process’s address space, though OFI does not require this implementation. Although not shown, an endpoint may be configured only to transmit or receive data. Data transfer requests are converted by the underlying provider into commands that are inserted into transmit and/or receive contexts.

Endpoints are also associated with completion queues. Completion queues are used to report the completion of asynchronous data transfer operations. An endpoint may direct completed transmit and receive operations to separate completion queues, or the same queue (not shown)

## Shared Contexts

A more advanced usage model of endpoints that allows for resource sharing is shown below.

Because transmit and receive contexts may be associated with limited hardware resources, OFI defines mechanisms for sharing contexts among multiple endpoints. The diagram above shows two endpoints each sharing transmit and receive contexts. However, endpoints may share only the transmit context or only the receive context or neither. Shared contexts allow an application or resource manager to prioritize where resources are allocated and how shared hardware resources should be used.

Completions are still associated with the endpoints, with each endpoint being associated with their own completion queue(s).

### Receive Contexts

TODO

### Transmit Contexts

TODO

## Scalable Endpoints

The final endpoint model is known as a scalable endpoint. Scalable endpoints allow a single endpoint to take advantage of multiple underlying hardware resources.

Scalable endpoints have multiple transmit and/or receive contexts. Applications can direct data transfers to use a specific context, or the provider can select which context to use. Each context may be associated with its own completion queue. Scalable contexts allow applications to separate resources to avoid thread synchronization or data ordering restrictions.

# Data Transfers

Obviously, the goal of network communication is to transfer data between systems. In the same way that sockets defines different data transfer semantics for TCP versus UDP sockets (streaming versus datagram messages), OFI defines different data transfer semantics. However, unlike sockets, OFI allows different semantics over a single endpoint, even when communicating with the same peer.

OFI defines separate API sets for the different data transfer semantics; although, there are strong similarities between the API sets. The differences are the result of the parameters needed to invoke each type of data transfer.

## Message transfers

Message transfers are most similar to UDP datagram transfers. The sender requests that data be transferred as a single transport operation to a peer. Even if the data is referenced using an I/O vector, it is treated as a single logical unit. The data is placed into a waiting receive buffer at the peer. Unlike UDP sockets, message transfers may be reliable or unreliable, and many providers support message transfers that are gigabytes in size.

Message transfers are usually invoked using API calls that contain the string "send" or "recv". As a result they may be referred to simply as sends or receives.

Message transfers involve the target process posting memory buffers to the receive context of its endpoint. When a message arrives from the network, a receive buffer is removed from the Rx context, and the data is copied from the network into the receive buffer. Messages are matched with posted receives in the order that they are received. Note that this may differ from the order that messages are sent, depending on the transmit side's ordering semantics. Furthermore, received messages may complete out of order. For instance, short messages could complete before larger messages, especially if the messages originate from different peers. Completion ordering semantics indicate the order that posted receive operations complete.

Conceptually, on the transmit side, messages are posted to a transmit context. The network processes messages from the Tx context, packetizing the data into outbound messages. Although many implementations process the Tx context in order (i.e. the Tx context is a true queue), ordering guarantees determine the actual processing order. For example, sent messages may be copied to the network out of order if targeting different peers.

In the default case, OFI defines ordering semantics such that messages 1, 2, 3, etc. from the sender are received in the same order at the target. Relaxed ordering semantics is an optimization technique that applications can opt into in order to improve network performance and utilization.

## Tagged messages

Tagged messages are similar to message transfers except that the messages carry one additional piece of information, a message tag. Tags are application defined values that are part of the message transfer protocol and are used to route packets at the receiver. At a high level, they are roughly similar to sequence numbers or message ids. The difference is that tag values are set by the application, may be any value, and duplicate tag values are allowed.

Each sent message carries a single tag value, which is used to select a receive buffer into which the data is copied. On the receiving side, message buffers are also marked with a tag. Messages that arrive from the network search through the posted receive messages until a matching tag is found. Tags allow messages to be placed into overlapping groups.

Tags are often used to identify virtual communication groups or roles. For example, one tag value may be used to identify a group of systems that contain input data into a program. A second tag value could identify the systems involved in the processing of the data. And a third tag may identify systems responsible for gathering the output from the processing. (This is purely a hypothetical example for illustrative purposes only). Moreover, tags may carry additional data about the type of message being used by each group. For example, messages could be separated based on whether the context carries control or data information.

In practice, message tags are typically divided into fields. For example, the upper 16 bits of the tag may indicate a virtual group, with the lower 16 bits identifying the message purpose. The tag message interface in OFI is designed around this usage model. Each sent message carries exactly one tag value, specified through the API. At the receiver, buffers are associated with both a tag value and a mask. The mask is applied to both the send and receive tag values (using a bit-wise AND operation). If the resulting values match, then the tags are said to match. The received data is then placed into the matched buffer.

For performance reasons, the mask is specified as 'ignore' bits. Although this is backwards from how many developers think of a mask (where the bits that are valid would be set to 1), the definition ends up mapping well with applications. The actual operation performed when matching tags is:

send\_tag | ignore == recv\_tag | ignore  
/\* this is equivalent to:  
 \* send\_tag & ~ignore == recv\_tag & ~ignore  
 \*/

Tagged messages are equivalent of message transfers if a single tag value is used. But tagged messages require that the receiver perform the matching operation at the target, which can impact performance versus untagged messages.

## RMA

RMA operations are architected such that they can require no processing at the RMA target. NICs which offload transport functionality can perform RMA operations without impacting host processing. RMA write operations transmit data from the initiator to the target. The memory location where the data should be written is carried within the transport message itself.

RMA read operations fetch data from the target system and transfer it back to the initiator of the request, where it is copied into memory. This too can be done without involving the host processor at the target system when the NIC supports transport offloading.

The advantage of RMA operations is that they decouple the processing of the peers. Data can be placed or fetched whenever the initiator is ready without necessarily impacting the peer process.

Because RMA operations allow a peer to directly access the memory of a process, additional protection mechanisms are used to prevent unintentional or unwanted access. RMA memory that is updated by a write operation or is fetched by a read operation must be registered for access with the correct permissions specified.

## Atomic operations

Atomic transfers are used to read and update data located in remote memory regions in an atomic fashion. Conceptually, they are similar to local atomic operations of a similar nature (e.g. atomic increment, compare and swap, etc.). The benefit of atomic operations is they enable offloading basic arithmetic capabilities onto a NIC. Unlike other data transfer operations, atomics require knowledge of the format of the data being accessed.

A single atomic function may operate across an array of data, applying an atomic operation to each entry, but the atomicity of an operation is limited to a single data type or entry. OFI defines a wide variety of atomic operations across all common data types. However support for a given operation is dependent on the provider implementation.

# Fabric Interfaces

A full description of the libfabric API is documented in the relevant man pages. This section provides an introduction to select interfaces, including how they may be used. It does not attempt to capture all subtleties or use cases, nor describe all possible data structures or fields.

## Using fi\_getinfo

The fi\_getinfo() call is one of the first calls that most applications will invoke. It is designed to be easy to use for simple applications, but extensible enough to configure a network for optimal performance. It serves several purposes. First, it abstracts away network implementation and addressing details. Second, it allows an application to specify which features they require of the network. Last, it provides a mechanism for a provider to report how an application can use the network in order to achieve the best performance.

/\* API prototypes \*/  
struct fi\_info \*fi\_allocinfo(void);  
  
int fi\_getinfo(int version, const char \*node, const char \*service,  
 uint64\_t flags, struct fi\_info \*hints, struct fi\_info \*\*info);

/\* Sample initialization code flow \*/  
struct fi\_info \*hints, \*info;  
  
hints = fi\_allocinfo();  
  
/\* hints will point to a cleared fi\_info structure  
 \* Initialize hints here to request specific network capabilities  
 \*/  
  
fi\_getinfo(FI\_VERSION(1, 4), NULL, NULL, 0, hints, &info);  
fi\_freeinfo(hints);  
  
/\* Use the returned info structure to allocate fabric resources \*/

The hints parameter is the key for requesting fabric services. The fi\_info structure contains several data fields, plus pointers to a wide variety of attributes. The fi\_allocinfo() call simplifies the creation of an fi\_info structure. In this example, the application is merely attempting to get a list of what providers are available in the system and the features that they support. Note that the API is designed to be extensible. Versioning information is provided as part of the fi\_getinfo() call. The version is used by libfabric to determine what API features the application is aware of. In this case, the application indicates that it can properly handle any feature that was defined for the 1.4 release (or earlier).

Applications should *always* hard code the version that they are written for into the fi\_getinfo() call. This ensures that newer versions of libfabric will provide backwards compatibility with that used by the application.

Typically, an application will initialize the hints parameter to list the features that it will use.

/\* Taking a peek at the contents of fi\_info \*/  
struct fi\_info {  
 struct fi\_info \*next;  
 uint64\_t caps;  
 uint64\_t mode;  
 uint32\_t addr\_format;  
 size\_t src\_addrlen;  
 size\_t dest\_addrlen;  
 void \*src\_addr;  
 void \*dest\_addr;  
 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;  
};

The fi\_info structure references several different attributes, which correspond to the different OFI objects that an application allocates. Details of the various attribute structures are defined below. For basic applications, modifying or accessing most attribute fields are unnecessary. Many applications will only need to deal with a few fields of fi\_info, most notably the capability (caps) and mode bits.

On success, the fi\_getinfo() function returns a linked list of fi\_info structures. Each entry in the list will meet the conditions specified through the hints parameter. The returned entries may come from different network providers, or may differ in the returned attributes. For example, if hints does not specify a particular endpoint type, there may be an entry for each of the three endpoint types. As a general rule, libfabric returns the list of fi\_info structures in order from most desirable to least. High-performance network providers are listed before more generic providers, such as the socket or UDP providers.

### Capabilities

The fi\_info caps field is used to specify the features and services that the application requires of the network. This field is a bit-mask of desired capabilities. There are capability bits for each of the data transfer services mentioned above: FI\_MSG, FI\_TAGGED, FI\_RMA, and FI\_ATOMIC. Applications should set each bit for each set of operations that it will use. These bits are often the only bits set by an application.

In some cases, additional bits may be used to limit how a feature will be used. For example, an application can use the FI\_SEND or FI\_RECV bits to indicate that it will only send or receive messages, respectively. Similarly, an application that will only initiate RMA writes, can set the FI\_WRITE bit, leaving FI\_REMOTE\_WRITE unset. The FI\_SEND and FI\_RECV bits can be used to restrict the supported message and tagged operations. By default, if FI\_MSG or FI\_TAGGED are set, the resulting endpoint will be enabled to both send and receive messages. Likewise, FI\_READ, FI\_WRITE, FI\_REMOTE\_READ, FI\_REMOTE\_WRITE can restrict RMA and atomic operations.

Capabilities are grouped into two general categories: primary and secondary. Primary capabilities must explicitly be requested by an application, and a provider must enable support for only those primary capabilities which were selected. Secondary capabilities may optionally be requested by an application. If requested, a provider must support a capability if it is asked for or fail the fi\_getinfo request. A provider may optionally report non-selected secondary capabilities if doing so would not compromise performance or security. That is, a provider may grant an application a secondary capability, whether the application requests it or not.

All of the capabilities discussed so far are primary. Secondary capabilities mostly deal with features desired by highly scalable, high-performance applications. For example, the FI\_MULTI\_RECV secondary capability indicates if the provider can support the multi-receive buffers feature described above.

Because different providers support different sets of capabilities, applications that desire optimal network performance may need to code for a capability being either present or absent. When present, such capabilities can offer a scalability or performance boost. When absent, an application may prefer to adjust its protocol or implementation to work around the network limitations. Although providers can often emulate features, doing so can impact overall performance, including the performance of data transfers that otherwise appear unrelated to the feature in use. For example, if a provider needs to insert protocol headers into the message stream in order to implement a given capability, the appearance of that header could negatively impact the performance of all transfers. By exposing such limitations to the application, the app has better control over how to best emulate the feature or work around its absence.

It is recommended that applications code for only those capabilities required to achieve the best performance. If a capability would have little to no effect on overall performance, developers should avoid using such features as part of an initial implementation. This will allow the application to work well across the widest variety of hardware. Application optimizations can then add support for less common features. To see which features are supported by which providers, see the libfabric [Provider Feature Maxtrix](https://github.com/ofiwg/libfabric/wiki/Provider-Feature-Matrix) for the relevant release.

### Mode Bits

Where capability bits represent features desired by applications, mode bits correspond to behavior requested by the provider. That is, capability bits are top down requests, whereas mode bits are bottom up restrictions. Mode bits are set by the provider to request that the application use the API in a specific way in order to achieve optimal performance. Mode bits often imply that the additional work needed by the application will be less overhead than forcing that same implementation down into the provider. Mode bits arise as a result of hardware implementation restrictions.

An application developer decides which mode bits they want to or can easily support as part of their development process. Each mode bit describes a particular behavior that the application must follow to use various interfaces. Applications set the mode bits that they support when calling fi\_getinfo(). If a provider requires a mode bit that isn't set, that provider will be skipped by fi\_getinfo(). If a provider does not need a mode bit that is set, it will respond to the fi\_getinfo() call, with the mode bit cleared. This indicates that the application does not need to perform the action required by the mode bit.

One of the most common mode bits needed by providers is FI\_CONTEXT. This mode bit requires that applications pass in a libfabric defined data structure (struct fi\_context) into any data transfer function. That structure must remain valid and unused by the application until the data transfer operation completes. The purpose behind this mode bit is that the struct fi\_context provides "scratch" space that the provider can use to track the request. For example, it may need to insert the request into a linked list, or track the number of times that an outbound transfer has been retried. Since many applications already track outstanding operations with their own data structure, by embedding the struct fi\_context into that same structure, overall performance can be improved. This avoids the provider needing to allocate and free internal structures for each request.

Continuing with this example, if an application does not already track outstanding requests, then it would leave the FI\_CONTEXT mode bit unset. This would indicate that the provider needs to get and release its own structure for tracking purposes. In this case, the costs would essentially be the same whether it were done by the application or provider.

For the broadest support of different network technologies, applications should attempt to support as many mode bits as feasible. Most providers attempt to support applications that cannot support any mode bits, with as small an impact as possible. However, implementation of mode bit avoidance in the provider will often impact latency tests.

# FIDs

FID stands for fabric identifier. It is the conceptual equivalent to a file descriptor. All fabric resources are represented by a fid structure, and all fid's are derived from a base fid type. In object-oriented terms, a fid would be the parent class. The contents of a fid are visible to the application.

/\* Base FID definition \*/  
enum {  
 FI\_CLASS\_UNSPEC,  
 FI\_CLASS\_FABRIC,  
 FI\_CLASS\_DOMAIN,  
 ...  
};  
  
struct fi\_ops {  
 size\_t size;  
 int (\*close)(struct fid \*fid);  
 ...  
};  
  
/\* All fabric interface descriptors must start with this structure \*/  
struct fid {  
 size\_t fclass;  
 void \*context;  
 struct fi\_ops \*ops;  
};

The fid structure is designed as a trade-off between minimizing memory footprint versus software overhead. Each fid is identified as a specific object class. Examples are given above (e.g. FI\_CLASS\_FABRIC). The context field is an application defined data value. The context field is usually passed as a parameter into the call that allocates the fid structure (e.g. fi\_fabric() or fi\_domain()). The use of the context field is application specific. Applications often set context to a corresponding structure that they've allocated. The context field is the only field that applications are recommended to access directly. Access to other fields should be done using defined function calls.

The ops field points to a set of function pointers. The fi\_ops structure defines the operations that apply to that class. The size field in the fi\_ops structure is used for extensibility, and allows the fi\_ops structure to grow in a backward compatible manner as new operations are added. The fid deliberately points to the fi\_ops structure, rather than embedding the operations directly. This allows multiple fids to point to the same set of ops, which minimizes the memory footprint of each fid. (Internally, providers usually set ops to a static data structure, with the fid structure dynamically allocated.)

Although it's possible for applications to access function pointers directly, it is strongly recommended that the static inline functions defined in the man pages be used instead. This is required by applications that may be built using the FABRIC\_DIRECT library feature. (FABRIC\_DIRECT is a compile time option that allows for highly optimized builds by tightly coupling an application with a specific provider. See the man pages for more details.)

Other OFI classes are derived from this structure, adding their own set of operations.

/\* Example of deriving a new class for a fabric object \*/  
struct fi\_ops\_fabric {  
 size\_t size;  
 int (\*domain)(struct fid\_fabric \*fabric, struct fi\_info \*info,  
 struct fid\_domain \*\*dom, void \*context);  
 ...  
};  
  
struct fid\_fabric {  
 struct fid fid;  
 struct fi\_ops\_fabric \*ops;  
};

Other fid classes follow a similar pattern as that shown for fid\_fabric. The base fid structure is followed by zero or more pointers to operation sets.

# Fabric

The top-level object that applications open is the fabric identifier. The fabric can mostly be viewed as a container object by applications, though it does identify which provider applications use. (Future extensions are likely to expand methods that apply directly to the fabric object. An example is adding topology data to the API.)

Opening a fabric is usually a straightforward call after calling fi\_getinfo().

int fi\_fabric(struct fi\_fabric\_attr \*attr, struct fid\_fabric \*\*fabric, void \*context);

The fabric attributes can be directly accessed from struct fi\_info. The newly opened fabric is returned through the 'fabric' parameter. The 'context' parameter appears in many operations. It is a user-specified value that is associated with the fabric. It may be used to point to an application specific structure and is retrievable from struct fid\_fabric.

## Attributes

The fabric attributes are straightforward.

struct fi\_fabric\_attr {  
 struct fid\_fabric \*fabric;  
 char \*name;  
 char \*prov\_name;  
 uint32\_t prov\_version;  
};

The only field that applications are likely to use directly is the prov\_name. This is a string value that can be used by hints to select a specific provider for use. On most systems, there will be multiple providers available. Only one is likely to represent the high-performance network attached to the system. Others are generic providers that may be available on any system, such as the TCP socket and UDP providers.

The fabric field is used to help applications manage open fabric resources. If an application has already opened a fabric that can support the returned fi\_info structure, this will be set to that fabric. The contents of struct fid\_fabric is visible to applications. It contains a pointer to the application's context data that was provided when the fabric was opened.

## Environment Variables

Environment variables are used by providers to configure internal options for optimal performance or memory consumption. Libfabric provides an interface for querying which environment variables are usable, along with an application to display the information to a command window. Although environment variables are usually configured by an administrator, an application can query for variables programmatically.

/\* APIs to query for supported environment variables \*/  
enum fi\_param\_type {  
 FI\_PARAM\_STRING,  
 FI\_PARAM\_INT,  
 FI\_PARAM\_BOOL  
};  
  
struct fi\_param {  
 /\* The name of the environment variable \*/  
 const char \*name;  
 /\* What type of value it stores: string, integer, or boolean \*/  
 enum fi\_param\_type type;  
 /\* A description of how the variable is used \*/  
 const char \*help\_string;  
 /\* The current value of the variable \*/  
 const char \*value;  
};  
  
int fi\_getparams(struct fi\_param \*\*params, int \*count);  
void fi\_freeparams(struct fi\_param \*params);

The modification of environment variables is typically a tuning activity done on larger clusters. However there are a few values that are useful for developers. These can be seen by executing the fi\_info command.

$ fi\_info -e  
# FI\_LOG\_LEVEL: String  
# Specify logging level: warn, trace, info, debug (default: warn)  
  
# FI\_LOG\_PROV: String  
# Specify specific provider to log (default: all)  
  
# FI\_LOG\_SUBSYS: String  
# Specify specific subsystem to log (default: all)  
  
# FI\_PROVIDER: String  
# Only use specified provider (default: all available)

Full documentation for these variables is available in the man pages. Variables beyond these may only be documented directly in the library itself, and available using the 'fi\_info -e' command.

The FI\_LOG\_LEVEL can be used to increase the debug output from libfabric and the providers. Note that in the release build of libfabric, debug output from data path operations (transmit, receive, and completion processing) may not be available. The FI\_PROVIDER variable can be used to enable or disable specific providers. This is useful to ensure that a given provider will be used.

# Domains

Domains usually map to a specific local network interface adapter. A domain may either refer to the entire NIC, a port on a multi-port NIC, or a virtual device exposed by a NIC. From the viewpoint of the application, a domain identifies a set of resources that may be used together.

Similar to a fabric, opening a domain is straightforward after calling fi\_getinfo().

int fi\_domain(struct fid\_fabric \*fabric, struct fi\_info \*info,  
 struct fid\_domain \*\*domain, void \*context);

The fi\_info structure returned from fi\_getinfo() can be passed directly to fi\_domain() to open a new domain.

## Attributes

A domain defines the relationship between data transfer services (endpoints) and completion services (completion queues and counters). Many of the domain attributes describe that relationship and its impact to the application.

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;  
 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;  
 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

Details of select attributes and their impact to the application are described below.

## Threading

OFI defines a unique threading model. The libfabric design is heavily influenced by object-oriented programming concepts. A multi-threaded application must determine how libfabric objects (domains, endpoints, completion queues, etc.) will be allocated among its threads, or if any thread can access any object. For example, an application may spawn a new thread to handle each new connected endpoint. The domain threading field provides a mechanism for an application to identify which objects may be accessed simultaneously by different threads. This in turn allows a provider to optimize or, in some cases, eliminate internal synchronization and locking around those objects.

The threading is best described as synchronization levels. As threading levels increase, greater potential parallelism is achieved. For example, an application can indicate that it will only access an endpoint from a single thread. This allows the provider to avoid acquiring locks around data transfer calls, knowing that there cannot be two simultaneous calls to send data on the same endpoint. The provider would only need to provide serialization if separate endpoints accessed the same shared software or hardware resources.

Threading defines where providers could optimize synchronization primitives. However, providers may still implement more serialization than is needed by the application. (This is usually a result of keeping the provider implementation simpler).

Various threading models are described in detail in the man pages. Developers should study the fi\_domain man page and available threading options, and select a mode that is best suited for how the application was designed. If an application leaves the value undefined, providers will report the highest (most parallel) threading level that they support.

## Progress

As previously discussed, progress models are a result of using the host processor in order to perform some portion of the transport protocol. In order to simplify development, OFI defines two progress models: automatic or manual. It does not attempt to identify which specific interface features may be offloaded, or what operations require additional processing by the application's thread.

Automatic progress means that an operation initiated by the application will eventually complete, even if the application makes no further calls into the libfabric API. The operation is either offloaded entirely onto hardware, the provider uses an internal thread, or the operating system kernel may perform the task. The use of automatic progress may increase system overhead and latency in the latter two cases. For control operations, this is usually acceptable. However, the impact to data transfers may be measurable, especially if internal threads are required to provide automatic progress.

The manual progress model can avoid this overhead for providers that do not offload all transport features into hardware. With manual progress the provider implementation will handle transport operations as part of specific libfabric functions. For example, a call to fi\_cq\_read() which retrieves a list of completed operations may also be responsible for sending ack messages to notify peers that a message has been received. Since reading the completion queue is part of the normal operation of an application, there is little impact to the application and additional threads are avoided.

Applications need to take care when using manual progress, particularly if they link into libfabric multiple times through different code paths or library dependencies. If application threads are used to drive progress, such as responding to received data with ACKs, then it is critical that the application thread call into libfabric in a timely manner.

OFI defines wait and poll set objects that are specifically designed to assist with driving manual progress.

## Memory Registration

RMA and atomic operations can both read and write memory that is owned by a peer process, and neither require the involvement of the target processor. Because the memory can be modified over the network, an application must opt into exposing its memory to peers. This is handled by the memory registration process. Registered memory regions associate memory buffers with permissions granted for access by fabric resources. A memory buffer must be registered before it can be used as the target of a remote RMA or atomic data transfer. Additionally, a fabric provider may require that data buffers be registered before being used in local transfers. The latter is necessary to ensure that the virtual to physical page mappings do not change.

Although there are a few different attributes that apply to memory registration, OFI groups those attributes into one of two different modes (for application simplicity).

### Basic Memory Registration Mode

Basic memory registration mode is defined around supporting the InfiniBand, RoCE, and iWarp architectures, which maps well to a wide variety of RMA capable hardware. In basic mode, registration occurs on allocated memory buffers, and the MR attributes are selected by the provider. The application must only register allocated memory, and the protection keys that are used to access the memory are assigned by the provider. The impact of using basic registration is that the application must inform any peer that wishes to access the region the local virtual address of the memory buffer, along with the key to use when accessing it. Peers must provide both the key and the target's virtual address as part of the RMA operation.

Although not part of the basic memory registration mode definition, hardware that supports this mode frequently requires that all data buffers used for network communication also be registered. This includes buffers posted to send or receive messages, *source* RMA and atomic buffers, and tagged message buffers. This restriction is indicated using the FI\_LOCAL\_MR mode bit. This restriction is needed to ensure that the virtual to physical address mappings do not change between a request being submitted and the hardware processing it.

### Scalable Memory Registration Mode

Scalable memory registration targets highly parallel, high-performance applications. Such applications often have an additional level of security that allows the peers to operate in a more trusted environment where memory registration is employed. In scalable mode, registration occurs on memory address ranges, and the MR attributes are selected by the user. There are two notable differences with scalable mode.

First is that the address ranges do not need to map to allocated memory buffers at the time the registration call is made. (Virtual memory must back the ranges before they are accessed as part of any data transfer operation.) This allows, for example, for an application to expose all or a significant portion of its address space to peers. When combined with a symmetric memory allocator, this feature can eliminate a process from needing to store the target addresses of its peers. Second, the application selects the protection key for the region. Target addresses and keys can be hard-coded or determined algorithmically, reducing the memory footprint and avoiding network traffic associated with registration.

### Memory Region APIs

The following APIs highlight how to allocate and access a registered memory region. Note that this is not a complete list of memory region (MR) calls, and for full details on each API, readers should refer directly to the man pages.

int fi\_mr\_reg(struct fid\_domain \*domain, const void \*buf, size\_t len,  
 uint64\_t access, uint64\_t offset, uint64\_t requested\_key, uint64\_t flags,  
 struct fid\_mr \*\*mr, void \*context);  
  
void \* fi\_mr\_desc(struct fid\_mr \*mr);  
uint64\_t fi\_mr\_key(struct fid\_mr \*mr);

By default, memory regions are associated with a domain. A MR is accessible by any endpoint that is opened on that domain. A region starts at the address specified by 'buf', and is 'len' bytes long. The 'access' parameter are permission flags that are OR'ed together. The permissions indicate which type of operations may be invoked against the region (e.g. FI\_READ, FI\_WRITE, FI\_REMOTE\_READ, FI\_REMOTE\_WRITE). The 'buf' parameter must point to allocated virtual memory when using basic registration mode.

If scalable registration is used, the application can specify the desired MR key through the 'requested\_key' parameter. The 'offset' and 'flags' parameters are not used and reserved for future use.

A MR is associated with local and remote protection keys. The local key is referred to as a memory descriptor and may be retrieved by calling fi\_mr\_desc(). This call is only needed if the FI\_LOCAL\_MR mode bit has been set. The memory descriptor is passed directly into data transfer operations, for example:

/\* fi\_mr\_desc() example using fi\_send() \*/  
fi\_send(ep, buf, len, fi\_mr\_desc(mr), 0, NULL);

The remote key, or simply MR key, is used by the peer when targeting the MR with an RMA or atomic operation. If scalable registration is used, the MR key will be the same as the 'requested\_key'. Otherwise, it is a provider selected value. The key must be known to the peer. If basic registration is used, this means that the key will need to be sent in a separate message to the initiating peer. (Some applications exchange the key as part of connection setup).

The API is designed to handle MR keys that are at most 64-bits long. The size of the actual key is reported as a domain attribute. Typical sizes are either 32 or 64 bits, depending on the underlying fabric. Support for keys larger than 64-bits is possible but requires using extended calls not discussed here.

# Endpoints

Endpoints are transport level communication portals. Opening an endpoint is trivial after calling fi\_getinfo(), however, there are different open calls, depending on the type of endpoint to allocate. There are separate calls to open active, passive, and scalable endpoints.

## Active

Active endpoints may be connection-oriented or connection-less. The data transfer interfaces – messages (fi\_msg), tagged messages (fi\_tagged), RMA (fi\_rma), and atomics (fi\_atomic) – are associated with active endpoints. In basic configurations, an active endpoint has transmit and receive queues. In general, operations that generate traffic on the fabric are posted to the transmit queue. This includes all RMA and atomic operations, along with sent messages and sent tagged messages. Operations that post buffers for receiving incoming data are submitted to the receive queue.

Active endpoints are created in the disabled state. They must transition into an enabled state before accepting data transfer operations, including posting of receive buffers. The fi\_enable call is used to transition an active endpoint into an enabled state. The fi\_connect and fi\_accept calls will also transition an endpoint into the enabled state, if it is not already enabled. An endpoint may immediately be allocated after opening a domain, using the same fi\_info structure that was returned from fi\_getinfo().

int fi\_endpoint(struct fid\_domain \*domain, struct fi\_info \*info,  
 struct fid\_ep \*\*ep, void \*context);

### Enabling

In order to transition an endpoint into an enabled state, it must be bound to one or more fabric resources. An endpoint that will generate asynchronous completions, either through data transfer operations or communication establishment events, must be bound to appropriate completion queues or event queues, respectively, before being enabled. Unconnected endpoints must be bound to an address vector.

/\* Example to enable an unconnected endpoint \*/  
  
/\* Allocate an address vector and associated it with the endpoint \*/  
fi\_av\_open(domain, &av\_attr, &av, NULL);  
fi\_ep\_bind(ep, &av->fid, 0);  
  
/\* Allocate and associate completion queues with the endpoint \*/  
fi\_cq\_open(domain, &tx\_cq\_attr, &tx\_cq, NULL);  
fi\_ep\_bind(ep, &tx\_cq->fid, FI\_TRANSMIT);  
  
fi\_cq\_open(domain, &rx\_cq\_attr, &rx\_cq, NULL);  
fi\_ep\_bind(ep, &rx\_cq->fid, FI\_RECV);  
  
fi\_enable(ep);

In the above example, we allocate an address vector and transmit and receive completion queues. The attributes for the address vector and completion queue are omitted (additional discussion below). Those are then associated with the endpoint through the fi\_ep\_bind() call. After all necessary resources have been assigned to the endpoint, we enable it. Enabling the endpoint indicates to the provider that it should allocate any hardware and software resources and complete the initialization for the endpoint.

The fi\_enable() call is always called for unconnected endpoints. Connected endpoints may be able to skip calling fi\_enable(), since fi\_connect() and fi\_accept() will enable the endpoint automatically. However, applications may still call fi\_enable() prior to calling fi\_connect() or fi\_accept(). Doing so allows the application to post receive buffers to the endpoint, which ensures that they are available to receive data in the case where the peer endpoint sends messages immediately after it establishes the connection.

## Passive

Passive endpoints are used to listen for incoming connection requests. Passive endpoints are of type FI\_EP\_MSG, and may not perform any data transfers. An application wishing to create a passive endpoint typically calls fi\_getinfo() using the FI\_SOURCE flag, often only specifying a 'service' address. The service address corresponds to a TCP port number.

Passive endpoints are associated with event queues. Event queues report connection requests from peers. Unlike active endpoints, passive endpoints are not associated with a domain. This allows an application to listen for connection requests across multiple domains.

/\* Example passive endpoint listen \*/  
fi\_passive\_ep(fabric, info, &pep, NULL);  
  
fi\_eq\_open(fabric, &eq\_attr, &eq, NULL);  
fi\_pep\_bind(pep, &eq->fid, 0);  
  
fi\_listen(pep);

A passive endpoint must be bound to an event queue before calling listen. This ensures that connection requests can be reported to the application. To accept new connections, the application waits for a request, allocates a new active endpoint for it, and accepts the request.

/\* Example accepting a new connection \*/  
  
/\* Wait for a CONNREQ event \*/  
fi\_eq\_sread(eq, &event, &cm\_entry, sizeof cm\_entry, -1, 0);  
assert(event == FI\_CONNREQ);  
  
/\* Allocate an new endpoint for the connection \*/  
if (!cm\_entry.info->domain\_attr->domain)  
 fi\_domain(fabric, cm\_entry.info, &domain, NULL);  
fi\_endpoint(domain, cm\_entry.info, &ep, NULL);  
  
/\* See the resource binding section below for details on associated fabric objects \*/  
fi\_ep\_bind(ep, &eq->fid, 0);  
fi\_cq\_open(domain, &tx\_cq\_attr, &tx\_cq, NULL);  
fi\_ep\_bind(ep, &tx\_cq->fid, FI\_TRANSMIT);  
fi\_cq\_open(domain, &rx\_cq\_attr, &rx\_cq, NULL);  
fi\_ep\_bind(ep, &rx\_cq->fid, FI\_RECV);  
  
fi\_enable(ep);  
fi\_recv(ep, rx\_buf, len, NULL, 0, NULL);  
  
fi\_accept(ep, NULL, 0);  
fi\_eq\_sread(eq, &event, &cm\_entry, sizeof cm\_entry, -1, 0);  
assert(event == FI\_CONNECTED);

The connection request event (FI\_CONNREQ) includes information about the type of endpoint to allocate, including default attributes to use. If a domain has not already been opened for the endpoint, one must be opened. Then the endpoint and related resources can be allocated. Unlike the unconnected endpoint example above, a connected endpoint does not have an AV, but does need to be bound to an event queue. In this case, we use the same EQ as the listening endpoint. Once the other EP resources (e.g. CQs) have been allocated and bound, the EP can be enabled.

To accept the connection, the application calls fi\_accept(). Note that because of thread synchronization issues, it is possible for the active endpoint to receive data even before fi\_accept() can return. The posting of receive buffers prior to calling fi\_accept() handles this condition, which avoids network flow control issues occurring immediately after connecting.

The fi\_eq\_sread() calls are blocking (synchronous) read calls to the event queue. These calls wait until an event occurs, which in this case are connection request and establishment events.

## Scalable

For most applications, an endpoint consists of a transmit and receive context associated with a single address. The transmit and receive contexts often map to hardware command queues. For multi-threaded applications, access to these hardware queues requires serialization, which can lead to them becoming bottlenecks. Scalable endpoints were created to address this.

A scalable endpoint is an endpoint that has multiple transmit and/or receive contexts associated with it. As an example, consider an application that allocates a total of four processing threads. By assigning each thread its own transmit context, the application can avoid serializing (i.e. locking) access to hardware queues.

The advantage of using a scalable endpoint over allocating multiple traditional endpoints is reduced addressing footprint. A scalable endpoint has a single address, regardless of how many transmit or receive contexts it may have.

Support for scalable endpoints is provider specific, with support indicated by the domain attributes:

struct fi\_domain\_attr {  
 ...  
 size\_t max\_ep\_tx\_ctx;  
 size\_t max\_ep\_rx\_ctx;  
 ...

The above fields indicates the maximum number of transmit and receive contexts, respectively, that may be associated with a single endpoint. One or both of these values will be greater than one if scalable endpoints are supported. Applications can configure and allocate a scalable endpoint using the fi\_scalable\_ep call:

/\* Set the required number of transmit of receive contexts  
 \* These must be <= the domain maximums listed above.  
 \* This will usually be set prior to calling fi\_getinfo  
 \*/  
struct fi\_info \*hints, \*info;  
struct fid\_domain \*domain;  
struct fid\_ep \*scalable\_ep, \*tx\_ctx[4], \*rx\_ctx[2];  
  
hints = fi\_allocinfo();  
...  
  
/\* A scalable endpoint requires > 1 Tx or Rx queue \*/  
hints->ep\_attr->tx\_ctx\_cnt = 4;  
hints->ep\_attr->rx\_ctx\_cnt = 2;  
  
/\* Call fi\_getinfo and open fabric, domain, etc. \*/  
...  
  
fi\_scalable\_ep(domain, info, &sep, NULL);

The above example opens an endpoint with four transmit and two receive contexts. However, a scalable endpoint only needs to be scalable in one dimension -- transmit or receive. For example, it could use multiple transmit contexts, but only require a single receive context. It could even use a shared context, if desired.

Submitting data transfer operations to a scalable endpoint is more involved. First, if the endpoint only has a single transmit context, then all transmit operations are posted directly to the scalable endpoint, the same as if a traditional endpoint were used. Likewise, if the endpoint only has a single receive context, then all receive operations are posted directly to the scalable endpoint. An additional step is needed before posting operations to one of many contexts, that is, the 'scalable' portion of the endpoint. The desired context must first be retrieved:

/\* Retrieve the first (index 0) transmit and receive contexts \*/  
fi\_tx\_context(scalable\_ep, 0, info->tx\_attr, &tx\_ctx[0], &tx\_ctx[0]);  
fi\_rx\_context(scalable\_ep, 0, info->rx\_attr, &rx\_ctx[0], &rx\_ctx[0]);

Data transfer operations are then posted to the tx\_ctx or rx\_ctx. It should be noted that although the scalable endpoint, transmit context, and receive context are all of type fid\_ep, attempting to submit a data transfer operation against the wrong object will result in an error.

By default all transmit and receive contexts belonging to a scalable endpoint are similar with respect to other transmit and receive contexts. However, applications can request that a context have fewer capabilities than what was requested for the scalable endpoint. This allows the provider to configure its hardware resources for optimal performance. For example, suppose a scalable endpoint has been configured for tagged message and RMA support. An application can open a transmit context with only tagged message support, and another context with only RMA support.

## Resource Bindings

Before an endpoint can be used for data transfers, it must be associated with other resources, such as completion queues, counters, address vectors, or event queues. Resource bindings must be done prior to enabling an endpoint. All active endpoints must be bound to completion queues. Unconnected endpoints must be associated with an address vector. Passive and connection-oriented endpoints must be bound to an event queue. The resource binding requirements are cumulative: for example, an RDM endpoint must be bound to completion queues and address vectors.

As shown in previous examples, resources are associated with endpoints using a bind operation:

int fi\_ep\_bind(struct fid\_ep \*ep, struct fid \*fid, uint64\_t flags);  
int fi\_scalable\_ep\_bind(struct fid\_ep \*sep, struct fid \*fid, uint64\_t flags);  
int fi\_pep\_bind(struct fid\_pep \*pep, struct fid \*fid, uint64\_t flags);

The bind functions are similar to each other (and map to the same fi\_bind call internally). Flags are used to indicate how the resources should be associated. The passive endpoint section above shows an example of binding passive and active endpoints to event and completion queues.

## EP Attributes

The properties of an endpoint are specified using endpoint attributes. These may be set as hints passed into the fi\_getinfo call. Unset values will be filled out by the provider.

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;  
 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;  
};

A full description of each field is available in the libfabric man pages, with selected details listed below.

### Endpoint Type

This indicates the type of endpoint: reliable datagram (FI\_EP\_RDM), reliable-connected (FI\_EP\_MSG), or unreliable datagram (DGRAM). Nearly all applications will need to specify the endpoint type as a hint passed into fi\_getinfo, as most applications will only be coded to support a single endpoint type.

### Maximum Message Size

This size is the maximum size for any data transfer operation that goes over the endpoint. For unreliable datagram endpoints, this is often the MTU of the underlying network. For reliable endpoints, this value is often a restriction of the underlying transport protocol. Applications that require transfers larger than the maximum reported size are required to break up a single, large transfer into multiple operations.

Providers expose their hardware or network limits to the applications, rather than segmenting large transfers internally, in order to minimize completion overhead. For example, for a provider to support large message segmentation internally, it would need to emulate all completion mechanisms (queues and counters) in software, even if transfers that are larger than the transport supported maximum were never used.

### Message Order Size

This field specifies data ordering. It defines the delivery order of transport data into target memory for RMA and atomic operations. Data ordering requires message ordering.

For example, suppose that an application issues two RMA write operations to the same target memory location. (The application may be writing a time stamp value every time a local condition is met, for instance). Message ordering indicates that the first write as initiated by the sender is the first write processed by the receiver. Data ordering indicates whether the *data* from the first write updates memory before the second write updates memory.

The max\_order\_xxx\_size fields indicate how large a message may be while still achieving data ordering. If a field is 0, then no data ordering is guaranteed. If a field is the same as the max\_msg\_size, then data order is guaranteed for all messages.

It is common for providers to support data ordering up to max\_msg\_size for back to back operations that are the same. For example, an RMA write followed by an RMA write may have data ordering regardless of the size of the data transfer (max\_order\_waw\_size = max\_msg\_size). Mixed operations, such as a read followed by a write, are often more restricted. This is because RMA read operations may require acknowledgments from the *initiator*, which impacts the re-transmission protocol.

For example, consider an RMA read followed by a write. The target will process the read request, retrieve the data, and send a reply. While that is occurring, a write is received that wants to update the same memory location accessed by the read. If the target processes the write, it will overwrite the memory used by the read. If the read response is lost, and the read is retried, the target will be unable to re-send the data. To handle this, the target either needs to: defer handling the write until it receives an acknowledgment for the read response, buffer the read response so it can be re-transmitted, or indicate that data ordering is not guaranteed.

Because the read or write operation may be gigabytes in size, deferring the write may add significant latency, and buffering the read response may be impractical. The max\_order\_xxx\_size fields indicate how large back to back operations may be with ordering still maintained. In many cases, read after write and write and read ordering may be significantly limited, but still usable for implementing specific algorithms, such as a global locking mechanism.

## Rx/Tx Context Attributes

The endpoint attributes define the overall abilities for the endpoint; however, attributes that apply specifically to receive or transmit contexts are defined by struct fi\_rx\_attr and fi\_tx\_attr, respectively:

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;  
};  
  
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;  
};

Context capabilities must be a subset of the endpoint capabilities. For many applications, the default attributes returned by the provider will be sufficient, with the application only needing to specify endpoint attributes.

Both context attributes include an op\_flags field. This field is used by applications to specify the default operation flags to use with any call. For example, by setting the transmit context’s op\_flags to FI\_INJECT, the application has indicated to the provider that all transmit operations should assume ‘inject’ behavior is desired. (I.e. the buffer provided to the call must be returned to the application upon return from the function). The op\_flags applies to all operations that do not provide flags as part of the call (e.g. fi\_sendmsg). A common use of op\_flags is to specify the default completion semantic desired (discussed next) by the application.

It should be noted that some attributes are dependent upon the peer endpoint having supporting attributes in order to achieve correct application behavior. For example, message order must be the compatible between the initiator’s transmit attributes and the target’s receive attributes. Any mismatch may result in incorrect behavior that could be difficult to debug.

# Completions

Data transfer operations complete asynchronously. Libfabric defines two mechanism by which an application can be notified that an operation has completed: completion queues and counters.

Regardless of which mechanism is used to notify the application that an operation is done, developers must be aware of what a completion indicates.

In all cases, a completion indicates that it is safe to reuse the buffer(s) associated with the data transfer. This completion mode is referred to as inject complete and corresponds to the operational flags FI\_INJECT\_COMPLETE. However, a completion may also guarantee stronger semantics.

Although libfabric does not define an implementation, a provider can meet the requirement for inject complete by copying the application’s buffer into a network buffer before generating the completion. Even if the transmit operation is lost and must be retried, the provider can resend the original data from the copied location. For large transfers, a provider may not mark a request as inject complete until the data has been acknowledged by the target. Applications, however, should only infer that it is safe to reuse their data buffer for an inject complete operation.

Transmit complete is a completion mode that provides slightly stronger guarantees to the application. The meaning of transmit complete depends on whether the endpoint is reliable or unreliable. For an unreliable endpoint (FI\_EP\_DGRAM), a transmit completion indicates that the request has been delivered to the network. That is, the message has left the local NIC. For reliable endpoints, a transmit complete occurs when the request has reached the target endpoint. Typically, this indicates that the target has acked the request. Transmit complete maps to the operation flag FI\_TRANSMIT\_COMPLETE.

A third completion mode is defined to provide guarantees beyond transmit complete. With transmit complete, an application knows that the message is no longer dependent on the local NIC or network (e.g. switches). However, the data may be buffered at the remote NIC and has not necessarily been written to the target memory. As a result, data sent in the request may not be visible to all processes. The third completion mode is delivery complete.

Delivery complete indicates that the results of the operation are available to all processes on the fabric. The distinction between transmit and delivery complete is subtle, but important. It often deals with *when* the target endpoint generates an acknowledgment to a message. For providers that offload transport protocol to the NIC, support for transmit complete is common. Delivery complete guarantees are more easily met by providers that implement portions of their protocol on the host processor. Delivery complete corresponds to the FI\_DELIVERY\_COMPLETE operation flag.

Applications can request a default completion mode when opening an endpoint by setting one of the above mentioned complete flags as an op\_flags for the context’s attributes. However, it is usually recommended that application use the provider’s default flags for best performance, and amend its protocol to achieve its completion semantics. For example, many applications will perform a ‘finalize’ or ‘commit’ procedure as part of their operation, which synchronizes the processing of all peers and guarantees that all previously sent data has been received.

## CQs

Completion queues often map directly to provider hardware mechanisms, and libfabric is designed around minimizing the software impact of accessing those mechanisms. Unlike other objects discussed so far (fabrics, domains, endpoints), completion queues are not part of the fi\_info structure or involved with the fi\_getinfo() call.

All active endpoints must be bound with one or more completion queues. This is true even if completions will be suppressed by the application (e.g. using the FI\_SELECTIVE\_COMPLETION flag). Completion queues are needed to report operations that complete in error.

Transmit and receive contexts are each associated with their own completion queue. An endpoint may direct transmit and receive completions to separate CQs or to the same CQ. For applications, using a single CQ reduces system resource utilization. While separating completions to different CQs could simplify code maintenance or improve multi-threading execution. A CQ may be shared among multiple endpoints.

CQs are allocated separately from endpoints and are associated with endpoints through the fi\_ep\_bind() function.

### Attributes

The properties of a completion queue are specified using the fi\_cq\_attr structure:

struct fi\_cq\_attr {  
 size\_t size;  
 uint64\_t flags;  
 enum fi\_cq\_format format;  
 enum fi\_wait\_obj wait\_obj;  
 int signaling\_vector;  
 enum fi\_cq\_wait\_cond wait\_cond;  
 struct fid\_wait \*wait\_set;  
};

Select details are described below.

#### CQ Size

The CQ size is the number of entries that the CQ can store before being overrun. If resource management is disabled, then the application is responsible for ensuring that it does not submit more operations than the CQ can store. When selecting an appropriate size for a CQ, developers should consider the size of all transmit and receive contexts that insert completions into the CQ.

Because CQs often map to hardware constructs, their size may be limited to a pre-set maximum. Applications should be prepared to allocate multiple CQs if they make use of a lot of endpoints –- a connection-oriented server application, for example. Applications should size the CQ correctly to avoid wasting system resources, while still protecting against queue overruns.

#### CQ Format

In order to minimize the amount of data that a provider must report, the type of completion data written back to the application is select-able. This limits the number of bytes the provider writes to memory, and allows necessary completion data to fit into a compact structure. Each CQ format maps to a specific completion structure. Developers should analyze each structure, select the smallest structure that contains all of the data it requires, and specify the corresponding enum value as the CQ format.

For example, if an application only needs to know which request completed, along with the size of a received message, it can select the following:

cq\_attr->format = FI\_CQ\_FORMAT\_MSG;  
  
struct fi\_cq\_msg\_entry {  
 void \*op\_context;  
 uint64\_t flags;  
 size\_t len;  
};

Once the format has been selected, the underlying provider will assume that read operations against the CQ will pass in an array of the corresponding structure. The CQ data formats are designed such that a structure that reports more information can be cast to one that reports less.

#### CQ Wait Object

Wait objects are a way for an application to suspend execution until it has been notified that a completion is ready to be retrieved from the CQ. The use of wait objects is recommended over busy waiting (polling) techniques for most applications. CQs include calls (fi\_cq\_sread() – for synchronous read) that will block until a completion occurs. Applications that will only use the libfabric blocking calls should select FI\_WAIT\_UNSPEC as their wait object. This allows the provider to select an object that is optimal for its implementation.

Applications that need to wait on other resources, such as open file handles or sockets, can request that a specific wait object be used. The most common alternative to FI\_WAIT\_UNSPEC is FI\_WAIT\_FD. This associates a file descriptor with the CQ. The file descriptor may be retrieved from the CQ using an fi\_control() operation, and can be passed to standard operating system calls, such as select() or poll().

### Reading Completions

Completions may be read from a CQ by using one of the non-blocking calls, fi\_cq\_read / fi\_cq\_readfrom, or one of the blocking calls, fi\_cq\_sread / fi\_cq\_sreadfrom. Regardless of which call is used, applications pass in an array of completion structures based on the selected CQ format. The CQ interfaces are optimized for batch completion processing, allowing the application to retrieve multiple completions from a single read call. The difference between the read and readfrom calls is that readfrom returns source addressing data, if available. The readfrom derivative of the calls is only useful for unconnected endpoints, and only if the corresponding endpoint has been configured with the FI\_SOURCE capability.

FI\_SOURCE requires that the provider use the source address available in the raw completion data, such as the packet's source address, to retrieve a matching entry in the endpoint’s address vector. Applications that carry some sort of source identifier as part of their data packets can avoid the overhead associated with using FI\_SOURCE.

### Retrieving Errors

Because the selected completion structure is insufficient to report all data necessary to debug or handle an operation that completes in error, failed operations are reported using a separate fi\_cq\_readerr() function. This call takes as input a CQ error entry structure, which allows the provider to report more information regarding the reason for the failure.

/\* read error prototype \*/  
fi\_cq\_readerr(struct fid\_cq \*cq, struct fi\_cq\_err\_entry \*buf, uint64\_t flags);  
  
/\* error data structure \*/  
struct fi\_cq\_err\_entry {  
 void \*op\_context;  
 uint64\_t flags;  
 size\_t len;  
 void \*buf;  
 uint64\_t data;  
 uint64\_t tag;  
 size\_t olen;  
 int err;  
 int prov\_errno;  
 void \*err\_data;  
};  
  
/\* Sample error handling \*/  
struct fi\_cq\_msg\_entry entry;  
struct fi\_cq\_err\_entry err\_entry;  
int ret;  
  
ret = fi\_cq\_read(cq, &entry, 1);  
if (ret == -FI\_EAVAIL)  
 ret = fi\_cq\_readerr(cq, &err\_entry, 0);

As illustrated, if an error entry has been inserted into the completion queue, then attempting to read the CQ will result in the read call returning -FI\_EAVAIL (error available). This indicates that the application must use the fi\_cq\_readerr() call to remove the failed operation's completion information before other completions can be reaped from the CQ.

A fabric error code regarding the failure is reported as the err field in the fi\_cq\_err\_entry structure. A provider specific error code is also available through the prov\_errno field. This field can be decoded into a displayable string using the fi\_cq\_strerror() routine. The err\_data field is provider specific data that assists the provider in decoding the reason for the failure.

## Counters

Completion counters are conceptually very simple completion mechanisms that return the number of completions that have occurred on an endpoint. No other details about the completion is available. Counters work well for connection-oriented applications that make use of strict completion ordering (rx/tx attribute comp\_order = FI\_ORDER\_STRICT), or applications that need to collect a specific number of responses from peers.

An endpoint has more flexibility with how many counters it can use relative to completion queues. Different types of operations can update separate counters. For instance, sent messages can update one counter, while RMA writes can update another. This allows for simple, yet powerful usage models, as control message completions can be tracked independently from large data transfers. Counters are associated with active endpoints using the fi\_ep\_bind() call:

/\* Example binding a counter to an endpoint.  
 \* The counter will update on completion of any transmit operation.  
 \*/  
fi\_ep\_bind(ep, cntr, FI\_SEND | FI\_WRITE | FI\_READ);

Counters are defined such that they can be implemented either in hardware or in software, by layering over a hardware completion queue. Even when implemented in software, counter use can improve performance by reducing the amount of completion data that is reported. Additionally, providers may be able to optimize how a counter is updated, relative to an application counting the same type of events. For example, a provider may be able to compare the head and tail pointers of a queue to determine the total number of completions that are available, allowing a single write to update a counter, rather than repeatedly incrementing a counter variable once for each completion.

### Attributes

Most counter attributes are a subset of the CQ attributes:

struct fi\_cntr\_attr {  
 enum fi\_cntr\_events events;  
 enum fi\_wait\_obj wait\_obj;  
 struct fid\_wait \*wait\_set;  
 uint64\_t flags;  
};

The sole exception is the events field, which must be set to FI\_CNTR\_EVENTS\_COMP, indicating that completion events are being counted. (This field is defined for future extensibility). A completion counter is updated according to the completion model that was selected by the endpoint. For example, if an endpoint is configured to for transmit complete, the counter will not be updated until the transfer has been received by the target endpoint.

### Counter Values

A completion counter is actually comprised of two different values. One represents the number of operations that complete successfully. The other indicates the number of operations which complete in error. Counters do not provide any additional information about the type of error, nor indicate which operation failed. Details of errors must be retrieved from a completion queue.

Reading a counter’s values is straightforward:

uint64\_t fi\_cntr\_read(struct fid\_cntr \*cntr);  
uint64\_t fi\_cntr\_readerr(struct fid\_cntr \*cntr);

# Address Vectors

A primary goal of address vectors is to allow applications to communicate with thousands to millions of peers while minimizing the amount of data needed to store peer addressing information. It pushes fabric specific addressing details away from the application to the provider. This allows the provider to optimize how it converts addresses into routing data, and enables data compression techniques that may be difficult for an application to achieve without being aware of low-level fabric addressing details. For example, providers may be able to algorithmically calculate addressing components, rather than storing the data locally. Additionally, providers can communicate with resource management entities or fabric manager agents to obtain quality of service or other information about the fabric, in order to improve network utilization.

An equally important objective is ensuring that the resulting interfaces, particularly data transfer operations, are fast and easy to use. Conceptually, an address vector converts an endpoint address into an fi\_addr\_t. The fi\_addr\_t (fabric interface address datatype) is a 64-bit value that is used in all ‘fast-path’ operations – data transfers and completions.

Address vectors are associated with domain objects. This allows providers to implement portions of an address vector, such as quality of service mappings, in hardware.

## AV Attributes

Address vectors are created with the following attributes. Select attribute details are discussed below:

struct fi\_av\_attr {  
 enum fi\_av\_type type;  
 int rx\_ctx\_bits;  
 size\_t count;  
 size\_t ep\_per\_node;  
 const char \*name;  
 void \*map\_addr;  
 uint64\_t flags;  
};

### AV Type

There are two types of address vectors. The type refers to the format of the returned fi\_addr\_t values for addresses that are inserted into the AV. With type FI\_AV\_TABLE, returned addresses are simple indices, and developers may think of the AV as an array of addresses. Each address that is inserted into the AV is mapped to the index of the next free array slot. The advantage of FI\_AV\_TABLE is that applications can refer to peers using a simple index, eliminating an application’s need to store any addressing data. I.e. the application can generate the fi\_addr\_t values themselves. This type maps well to applications, such as MPI, where a peer is referenced by rank.

The second type is FI\_AV\_MAP. This type does not define any specific format for the fi\_addr\_t value. Applications that use type map are required to provide the correct fi\_addr\_t for a given peer when issuing a data transfer operation. The advantage of FI\_AV\_MAP is that a provider can use the fi\_addr\_t to encode the target’s address, which avoids retrieving the data from memory. As a simple example, consider a fabric that uses TCP/IPv4 based addressing. An fi\_addr\_t is large enough to contain the address, which allows a provider to copy the data from the fi\_addr\_t directly into an outgoing packet.

### AV Rx Context Bits

The rx\_ctx\_bits field is only used with scalable endpoints with named received contexts, and is best described using an example. A peer process has allocated a scalable endpoint with two receive contexts. The first receive context will be used for control message, with data messages targeting the second context. Named contexts is a feature that allows the initiator to select which context will receive a message. If the initiating application wishes to send a data message, it must indicate that the message should be steered to the second context.

The rx\_ctx\_bits allocates a specific number of bits out of the fi\_addr\_t value that will be used to indicate which context a given operation will target. Applications should reserve a number of bits large enough to indicate any context at the target. For example, two bits is sufficient to target one of four different receive contexts.

The function fi\_rx\_addr() converts a target fi\_addr\_t address, along with the requested receive context index, into a new fi\_addr\_t value that may be used to transfer data to a scalable endpoint’s receive context.

fi\_addr\_t fi\_rx\_addr(fi\_addr\_t fi\_addr, int rx\_index, int rx\_ctx\_bits);

This call is simply a wrapper that writes the rx\_index into the space that was reserved in the fi\_addr\_t value.

### Sharing AVs Between Processes

Large scale parallel programs typically run with multiple processes allocated on each node. Because these processes communicate with the same set of peers, the addressing data needed by each process is the same. Libfabric defines a mechanism by which processes running on the same node may share their address vectors. This allows a system to maintain a single copy of addressing data, rather than one copy per process.

Although libfabric does not require any implementation for how an address vector is shared, the interfaces map well to using shared memory. Address vectors which will be shared are given an application specific name. How an application selects a name that avoid conflicts with unrelated processes, or how it communicates the name with peer processes is outside the scope of libfabric.

In addition to having a name, a shared AV also has a base map address -- map\_addr. Use of map\_addr is only important for address vectors that are of type FI\_AV\_MAP, and allows applications to share fi\_addr\_t values. From the viewpoint of the application, the map\_addr is the base value for all fi\_addr\_t values. A common use for map\_addr is for the process that creates the initial address vector to request a value from the provider, exchange the returned map\_addr with its peers, and for the peers to open the shared AV using the same map\_addr. This allows the fi\_addr\_t values to be stored in shared memory that is accessible by all peers.

# Wait and Poll Sets

As mentioned, most libfabric operations involve asynchronous processing, with completions reported to event queues, completion queues, and counters. Wait sets and poll sets were created to help manage and optimize checking for completed requests across multiple objects.

A poll set is a collection of event queues, completion queues, and counters. Applications use a poll set to check if a new completion event has arrived on any of its associated objects. When events occur infrequently or to one of several completion reporting objects, using a poll set can improve application efficiency by reducing the number of calls that the application makes into the libfabric provider. The use of a poll set should be considered by apps that use at least two completion reporting structures, and it is likely that checking them will find that no new events have occurred.

A wait set is similar to a poll set, and is often used in conjunction with one. In ideal implementations, a wait set is associated with a single wait object, such as a file descriptor. All event / completion queues and counters associated with the wait set will be configured to signal that wait object when an event occurs. This minimizes the system resources that are necessary to support applications waiting for events.

## Poll Set

The poll set API is fairly straightforward.

int fi\_poll\_open(struct fid\_domain \*domain, struct fi\_poll\_attr \*attr,  
 struct fid\_poll \*\*pollset);  
int fi\_poll\_add(struct fid\_poll \*pollset, struct fid \*event\_fid,  
 uint64\_t flags);  
int fi\_poll\_del(struct fid\_poll \*pollset, struct fid \*event\_fid,  
 uint64\_t flags);  
int fi\_poll(struct fid\_poll \*pollset, void \*\*context, int count);

Applications call fi\_poll\_open() to allocate a poll set. The attribute structure is a placeholder for future extensions and contains a single flags field, which is reserved. To add and remove event queues, completion queues, and counters, the fi\_poll\_add() and fi\_poll\_del() calls are used. As with open, the flags parameter is for extensibility and should be 0. Once objects have been associated with the poll set, an app may call fi\_poll() to retrieve a list of objects that may have new events available.

struct my\_cq \*tx\_cq, \*rx\_cq; /\* embeds fid\_cq, configured separately \*/  
struct fid\_poll \*pollset;  
struct fi\_poll\_attr attr = {};  
void \*cq\_context;  
  
/\* Allocate and add CQs to poll set \*/  
fi\_poll\_open(domain, &attr, &pollset);  
fi\_poll\_add(pollset, &tx\_cq->cq.fid, 0);  
fi\_poll\_add(pollset, &rx\_cq->cq.fid, 0);  
  
/\* Check for events \*/  
ret = fi\_poll(pollset, &cq\_context, 1);  
if (ret == 1) {  
 /\* CQ had an event \*/  
 struct my\_cq \*cq = cq\_context;  
 struct fi\_cq\_msg\_entry entry;  
 fi\_cq\_read(&cq->cq, &entry, 1);  
}

It’s worth noting that fi\_poll() returns a list of objects that have experienced some level of activity since they were last checked. However, an object appearing in the poll output does not guarantee that an event is actually available. For example, fi\_poll() may return the context associated with a completion queue, but an app may find that queue empty when reading it. This behavior is permissible by the API and is the result of potential provider implementation details.

One reason this can occur is if an entry is added to the completion queue, but that entry should not be reported to the application. For example, the completion may correspond to a message sent or received as part the provider’s protocol, and may not correspond to an application operation. The fi\_poll() routine simply reports whether the queue is empty or not, and is not intended for event processing, which is deferred until the queue can be read in order to avoid additional software queuing overhead.

## Wait Sets

The wait set API is actually smaller than the poll set.

int fi\_wait\_open(struct fid\_fabric \*fabric, struct fi\_wait\_attr \*attr,  
 struct fid\_wait \*\*waitset);  
int fi\_wait(struct fid\_wait \*waitset, int timeout);  
  
struct fi\_wait\_attr {  
 enum fi\_wait\_obj wait\_obj;  
 uint64\_t flags;  
};

The type of wait object that the wait set should use is specified through the wait attribute structure. Unlike poll sets, a wait set is associated with event queues, completion queues, and counters during their creation. This is necessary so that system resources, such as file descriptors, can be properly allocated and configured. Applications can block until the wait set’s wait object is signaled using the fi\_wait() call. Or an application can use an fi\_control() call to retrieve the native wait object for use directly with system calls, such as poll() and select().

A wait set is signaled whenever an event is added to one of its associated objects which would trigger the signal. In many cases, the wait set is signaled when any new event occurs; however, some objects will delay signaling the wait object until a threshold is crossed.

Because wait sets are responsible for linking completion reporting objects with wait objects, they can only indicate when a wait object has been signaled. A wait set cannot identify which object was responsible for signaling the wait object. Once a wait has been satisfied, applications are responsible for checking all completion structures for events. One simple way to accomplish this is to place all objects sharing a wait set into a peer poll set.

## Using Native Wait Objects: TryWait

There is an important difference between using libfabric completion objects, versus sockets, that may not be obvious from the discussions so far. With sockets, the object that is signaled is the same object that abstracts the queues, namely the file descriptor. When data is received on a socket, that data is placed in a queue associated directly with the fd. Reading from the fd retrieves that data. If an application wishes to block until data arrives on a socket, it calls select() or poll() on the fd. The fd is signaled when a message is received, which releases the blocked thread, allowing it to read the fd.

By associating the wait object with the underlying data queue, applications are exposed to an interface that is easy to use and race free. If data is available to read from the socket at the time select() or poll() is called, those calls simply return that the fd is readable.

There are a couple of significant disadvantages to this approach, which have been discussed previously, but from different perspectives. The first is that every socket must be associated with its own fd. There is no way to share a wait object among multiple sockets. (This is a main reason for the development of epoll semantics). The second is that the queue is maintained in the kernel, so that the select() and poll() calls can check them.

Libfabric separates the wait object from the queues. For applications that use libfabric interfaces to wait for events, such as fi\_cq\_sread and fi\_wait, this separation is mostly hidden from the application. The exception is that applications may receive a signal (e.g. fi\_wait() returns success), but no events are retrieved when a queue is read. This separation allows the queues to reside in the application's memory space, while wait objects may still use kernel components. A reason for the latter is that wait objects may be signaled as part of system interrupt processing, which would go through a kernel driver.

Applications that want to use native wait objects (e.g. file descriptors) directly in operating system calls must perform an additional step in their processing. In order to handle race conditions that can occur between inserting an event into a completion or event object and signaling the corresponding wait object, libfabric defines a ‘trywait’ function. The fi\_trywait implementation is responsible for handling potential race conditions which could result in an application either losing events or hanging. The following example demonstrates the use of fi\_trywait.

/\* Get the native wait object -- an fd in this case \*/  
fi\_control(&cq->fid, FI\_GETWAIT, (void \*) &fd);  
FD\_ZERO(&fds);  
FD\_SET(fd, &fds);  
  
while (1) {  
 ret = fi\_trywait(fabric, &cq->fid, 1);  
 if (ret == FI\_SUCCESS) {  
 /\* It’s safe to block on the fd \*/  
 select(fd + 1, &fds, NULL, &fds, &timeout);  
 } else if (ret == -FI\_EAGAIN) {  
 /\* Read and process all completions from the CQ \*/  
 do {  
 ret = fi\_cq\_read(cq, &comp, 1);  
 } while (ret > 0);  
 } else {  
 /\* something really bad happened \*/  
 }  
}

In this example, the application has allocated a CQ with an fd as its wait object. It calls select() on the fd. Before calling select(), the application must call fi\_trywait() successfully (return code of FI\_SUCCESS). Success indicates that a blocking operation can now be invoked on the native wait object without fear of the application hanging or events being lost. If fi\_trywait() returns –FI\_EAGAIN, it usually indicates that there are queued events to process.

# Putting It All Together

## MSG EP pingpong

## RDM EP pingpong

此页面的语言为英语  
  
翻译为中文（简体）  
  
   
  
随时将英语翻译为中文（简体）PRO  
一律不翻译英语  
一律不翻译raw.githubusercontent.com