GETTING STARTED WITH LIBFABRIC

A Beginner's Guide to OFI Programming Interface

Abstract

This guide presents the first hand introduction on how to write the very first application using the OFI libfabric library and get it running.

Xiong, Jianxin

Jianxin.xiong@intel.com

Version History:

0.1	1/21/2016	Initial version
0.2	7/7/2016	Reorder some of the steps. Show how to query address length.

Table of Contents

Introduction

About libfabric

Libfabric is an open-source fabric software library designed to provide low-latency interfaces to fabric hardware. It is developed and maintained by the OpenFabrics Interface Working Group (OFIWG).

Libfabric is designed to be flexible in order to efficiently support many different programing paradigms (MPI, PGAS, etc.) over a broad range of fabric hardware (sockets, Intel TrueScale, Intel Omni-Path, InfiniBand, Cray GNI, Cisco VIC, etc.). At the top layer, libfabric consists of a set of common interfaces to set up the communication environment and several optional communication interfaces to suit different application requirement. At the bottom layer, libfabric functions are implemented by different providers, each working over a specific fabric hardware or fabric software abstraction. Each provider is required to implement the common interface functions, but has the freedom to choose which interface functions to support. The application can query the capabilities of the providers and decide which one to use. It is even possible that multiple variations of the same provider be presented to the application, each of which has different optimization strategy for different usage model.

About this document

This is intended to be a quick, easy guide to help new libfabric users start write libfabric based code. Only a very limited set of libfabric functionalities is covered. More specifically, it only covers operations related to reliable, connectionless endpoint type (FI_RDM). So no connection manager related topics. Basic operations of message queue, tagged message queue, and RMA functions will be discussed, but atomics and triggered operations are not included. Advanced topics such as scalable endpoint are also outside the scope of this document.

For more information

Read the man pages. They contain the most authentic and complete information about libfabric programming interface. The man pages are included in the libfabric source distribution, as well as installed along with the binary. They are also available online:

```
https://github.com/ofiwg/libfabric/tree/master/man
```

Study the source code of fabtests:

```
https://github.com/ofiwg/fabtests
```

Study the source code of the OpenMPI ofi MTL:

```
https://github.com/open-mpi/ompi/tree/master/ompi/mca/mtl/ofi
```

Setting up the environment

Getting the source code

The latest source code of libfabric can be obtained from GitHub:

```
https://github.com/ofiwg/libfabric
```

Alternatively, the release source packages can be downloaded here:

```
http://downloads.openfabrics.org/downloads/ofi/
```

At the time of writing (July 2016), the latest release version is 1.3.

Building and installing the library

The library can be built and installed by simply running a few commands inside the top level directory of the source tree:

```
$ cd libfabric
$ ./autogen.sh
$ ./configure
$ make && make install
```

The VPATH function is supported, so one can put most of the building artifacts under a separate directory:

```
$ cd libfabric
$ ./autogen.sh
$ mkdir stage
$ cd stage
$ ../configure
$ make && make install
```

The “configure” script accepts some command line options, here are the most common ones:

```
--prefix=<directory_to_install_libfabric>
--enable-<provider_name>=yes
--enable-<provider_name>=no
--enable-<provider_name>=auto
--enable-<provider_name>=<directory_to_find_provider_supporting_package>
```

The “configure” script is also affected by some environment variables, most notably CC, CFLAGS, LDFLAGS, LIBS, CPPFLAGS, and CPP. For more details, try this:

```
$ ./configure --help
```

Building applications

Applications that use libfabric need to include a few header files. Most applications will need these:

```
#include <rdma/fabric.h>
#include <rdma/fi_domain.h>
#include <rdma/fi_endpoint.h>
```

A few more header files may be needed, depending on how the applications make use of libfabric functions:

```
#include <rdma/fi_cm.h>
#include <rdma/fi_tagged.h>
#include <rdma/fi_rma.h>
#include <rdma/fi_errno.h>
```

Applications need to be linked with the libfabric library. For example:

```
$ cc example.c -o example -lfabric
```

Or if libfabric is installed under “~/ofi”:

```
$ cc example.c -o example -I ~/ofi/include -L ~/ofi/lib -lfabric
```

Programming with libfabric

Hierarchy of libfabric objects

Libfabic defines various objects such as fabric, domain, endpoint, address vector, and completion queue. The corresponding man pages have the full description for these objects. However, to beginners, it would be useful to have a quick overview of the relationship of these objects, and how and in what order should they be created and initialized. Here is a summary:

Object	Depends on	Can bind to	How to get / create
fi_info (provider)			`fi_getinfo(…,&fi);`
fabric	fi_info		`fi_fabric(fi->fabric_attr,&fabric,…);`
event queue	fabric	av, ep	`fi_eq_open(fabric,…,&eq,…);`
wait set	fabric		`fi_wait_open(fabric,…,&waitset);`
domain	fabric, fi_info		`fi_domain(fabric,fi,&domain,…);`
endpoint	domain, fi_info		`fi_endpoint(domain,fi,&ep,…);`
address vector	domain	ep	`fi_av_open(domain,…,&av,…);`
completion queue	domain	ep	`fi_cq_open(domain,…,&cq,…);`
counter	domain	ep, mr	`fi_cntr_open(domain,…,&cntr,…);`
poll set	domain		`fi_poll_open(domain,…,&pollset);`
memory region	domain		`fi_mr_reg(domain,…,&mr,…);`

Provider discovery

An application should always call "fi_getinfo()" first to discover what providers are available on the system. The call can be as simple as this one:

```
struct fi_info *fi;

err = fi_getinfo(FI_VERSION(1,0), NULL, NULL, 0, NULL, &fi);
```

On success (when "err" is 0), "fi" will point to a linked list of "struct fi_info", with each item describes a provider available on the system. The application can then examine the properties of the providers, and choose the suitable one to be used thereafter.

The application can provide hints to the call to filter out those providers that doesn't meet the expectation. Since the type of the "hints" parameter has a non-flat memory layout, it is advisable to use "fi_allocinfo()" to allocate the memory. For example:

```
struct fi_info *hints, *fi;

hints = fi_allocinfo();
hints->ep_attr->type = FI_RDM;
hints->caps = FI_MSG | FI_TAGGED | FI_RMA;
hints->mode = FI_CONTEXT;
hints->fabric_attr->prov_name = strdup("psm2");
err = fi_getinfo(FI_VERSION(1,0), NULL, NULL, 0, hints, &fi);
```

This example asks for providers that support endpoint type FI_RDM, support message queue, tagged message queue, and RMA interface, and have "psm2" as the provider name (yes, it is allowed that multiple providers have the same name). This also tells the providers that the application can support FI_CONTEXT mode, i.e., the application will ensure that the "context" parameter passed to any communication call points to a valid memory space that begins with type "struct fi_context" and that part of memory is not going to be modified until the corresponding request completes. Some providers (such as the psm provider and psm2 provider) require the FI_CONTEXT mode.

It is also good practice to call "fi_freeinfo()" to free the "hints" allocated by "fi_allocinfo" and the "fi" returned from "fi_getinfo()" when they are no longer used. The use of "strdup()" in the above example is to ensure that the pointer "prov_name" can be freed via "free()", which is called automatically inside the function "fi_freeinfo()".

```
fi_freeinfo(hints);
```

Provider setup

Once a provider is selected, it needs to be initialized by creating the objects needed to use the provider for data communication. Regarding connectionless endpoint type, the following steps are generally needed:

Create fabric object

This is always the first object to create.

```
struct fid_fabric *fabric;

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

Create event queue

Event queue is used to collect asynchronous events. For connection-based endpoints, event queue is bound to endpoints to report events generated by the connection manager. For connectionless endpoints, event queue is bound to address vectors to report errors from "fi_av_insert()" call.

```
struct fi_eq_attr eq_attr;
struct fid_eq *eq;

memset(&eq_attr, 0, sizeof(eq_attr));
eq_attr.size = 64;
eq_attr.wait_obj = FI_WAIT_UNSPEC;
err = fi_eq_open(fabric, &eq_attr, &eq, NULL);
```

Create domain object

Domain is the layer of abstraction where the physical resource are shared. Most other objects belongs to a domain.

```
struct fid_domain *domain;

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

Create endpoint(s)

Endpoint is the object used for communication. Usually it is sufficient to create a single endpoint using the default attributes returned from “fi_getinfo()”.

```
struct fid_ep *ep;

err = fi_endpoint(domain, fi, &ep, NULL);
```

Depending on the provider, applications may be able to create multiple endpoints. The use of multiple endpoints is usually either for utilizing the available hardware resource (different communication ports/contexts) or for using different optimization for different capabilities and options. In the latter case, the “fi” parameter can be modified before “fi_endpoint()” is called. Notice that the modified attributes must be a subset of the original attributes returned from “fi_getinfo()”, otherwise the call may fail. For example:

```
Struct fid_ep *ep_msg, *ep_rma;

fi->caps = FI_MSG | FI_TAGGED;
err = fi_endpoint(domain, fi, &ep_msg, NULL);
fi->caps = FI_RMA;
err = fi_endpoint(domain, fi, &ep_rma, NULL);
```

Create address vector bind to endpoint and event queue

For connectionless endpoints each communication call must have a way to specify the address of the destination endpoint. Usually this address is in a different format from the address obtained from the “fi_getname()” call. It is necessary to translate between the two formats. In libfabric, this purpose is served by the address vector.

Address vector can be created as one of the two types: “FI_AV_MAP” or “FI_AV_TABLE”. For “FI_AV_MAP” type address vector, the translated address is returned directly to the user to be used in later communication calls. For “FI_AV_TABLE” type address vector, the translated address is stored internally and the user then use the table index in the communication calls.

Here is how to create an address vector of type “FI_AV_MAP”:

```
struct fi_av_attr av_attr;
struct fid_av *av;

memset(&av_attr, 0, sizeof(av_attr));
av_attr.type = FI_AV_MAP;
err = fi_av_open(domain, &av_attr, &av, NULL);
```

Then the address vector needs to be bound to the endpoint before that endpoint can be used for communication. With the binding, the communication calls know exactly what type the bound address vector is and can handle the address parameter accordingly.

```
err = fi_ep_bind(ep, (fid_t)av, 0);
err = fi_av_bind(av, (fid_t)eq, 0);
```

Create completion queue and bind to endpoint (optional)

Completion queue is used to report completions and errors of communication operations. Alternatively, counter can be used for similar purpose, with much less detailed information available and probably, with less overhead.

```
struct fi_cq_attr cq_attr;
struct fid_cq *cq;

memset(&cq_attr, 0, sizeof(cq_attr));
q_attr.format = FI_CQ_FORMAT_TAGGED;
cq_attr.size = 100;
err = fi_cq_open(domain, &cq_attr, &cq, NULL);
```

The format of a completion queue determines how much information is reported in the completion entries. The rule of thumb is to define to the maximum of the information that is needed and valid. For example, if the application only cares about the context of the completed operations, then the format "FI_CQ_FORMAT_CONTEXT" is enough. On the other hand, if the application needs to know the actual tag of a received message via the tagged message queue, it needs to use "FI_CQ_FORMAT_TAGGED".

A completion queue needs to be bound to an endpoint in order to report completions for operations over that endpoint. It is necessary to specify what types of operations can report completions on the queue.

```
err = fi_ep_bind(ep, (fid_t)cq, FI_TRANSMIT|FI_RECV);
```

The "FI_TRANSMIT" flag covers all the outbound data transfer requests, including message send, RMA operations, and atomic operations. The "FI_RECV" flag covers message receive operations.

Another flag, "FI_SELECTIVE_COMPLETION", can be "or"-ed with the above flags to suppress the generation of successful completions by default. Only operations with "FI_COMPLETION" flag set will generate an entry in the completion queue when finished successfully. Errors always generate entries in the completion queue regardless of the flag.

Create counter and bind to endpoint (optional)

Counter is a light-weight alternative to completion queue. Instead of having entries for each completion and error, counter just counts the number of completions and errors.

```
struct fi_cntr_attr cntr_attr;
struct fid_cntr *cntr;

memset(&cntr_attr, 0, sizeof(cntr_attr));
```

```
err = fi_cntr_open(domain, &cntr_attr, &cntr, NULL);
```

Similarly, a counter can be bound to an endpoint:

```
err = fi_ep_bind(ep, (fid_t)cntr, FI_SEND|FI_RECV|FI_READ|FI_WRITE);
```

The “FI_SEND” flag covers message send. The “FI_RECV” flag covers message receive. The “FI_READ” flag covers RMA read and atomic fetch. The “FI_WRITE” flag covers RMA write and other atomic operations. There are also “FI_REMOTE_READ” and “FI_REMOTE_WRITE” flags that cover incoming RMA and atomic operations. These two flags require the endpoint be created with the “FI_RMA_EVENT” flag and is outside the scope of this guide.

Create memory region (optional)

Memory region is created to allow remote memory access (RMA). Libfabric doesn’t require memory region for local buffers used in send/recv/RMA/atomic operations. However, some providers (e.g. verbs provider) require memory region for both local and remote buffers. This requirement is identified as a mode bit “FI_LOCAL_MR”. Applications that intend to use such providers should set the “FI_LOCAL_MR” mode bit in the “hints” parameter before calling “fi_getinfo()”, otherwise the provider would consider itself not suitable for the application.

```
struct fid_mr *mr;

err = fi_mr_reg(domain, buf, len, FI_REMOTE_READ|FI_REMOTE_WRITE, 0
                requested_key, 0, &mr, NULL);
```

The local descriptor and remote access key can be obtained from the memory region handle:

```
void *    desc = fi_mr_desc(mr);
uint64_t rkey = fi_mr_key(mr);
```

There are two different modes how memory region can be created. This is determined by the “mr_mode” field of “domain_attr”. Be sure to check “fi->domain_attr->mr_mode” before creating & using memory region. If a specific mr_mode is required, make sure “hints->domain_attr->mr_mode” is set properly before calling “fi_getinfo()”.

If mr_mode is “FI_MR_BASIC”, it works similar to InfiniBand MR: the keys are generated automatically and the full virtual address is used as remote address. This mode requires exchanging the keys and address information in order to access remote memory.

If mr_mode is FI_MR_SCALABLE, the keys are assigned by the user, and the offset within the memory region is used as remote access address. With this mode, applications can use pre-determined memory key and address for remote memory access.

Optionally, a counter can be bound to a memory region to count incoming remote write access to the memory region.

```
err = fi_mr_bind(mr, (fid_t)cntr, FI_REMOTE_WRITE);
```

Enable the endpoint

Although it's not necessary for every provider, but it's generally a good idea to enable the endpoint before using it for data communication.

```
err = fi_enale(ep);
```

Get endpoint address

Each endpoint has an address (or name) so that it can be addressed by remote endpoints. For connection oriented endpoints, the endpoint address is used in the connection setup calls. For connectionless endpoints, the endpoint address is needed for each communication call (see the next section: address vector).

The format of endpoint address is provider dependent and is available in the "fi->addr_format" field. Most providers that support connection-oriented endpoints support "FI_SOCKADDR". For such providers, it is usually possible to obtain remote endpoint address via regular TCP/IP address resolution mechanism. Some providers, e.g. the psm/psm2 provider, only support transport specific endpoint address format. The general practice is for each process to obtain the local endpoint addresses via the "fi_getname()" call and exchange the information with other processes in the same job using out-of-band communication channel (e.g. sockets). The end result is that each process knows the addresses of all the remote endpoints.

Since the length of endpoint address varies from provider to provider, the general practice is to query the length first. The function "fi_getname" always returns the size needed via the input/output parameter "addrlen".

```
void *addr;
size_t addrlen = 0;

fi_getname((fid_t)ep, NULL, &addrlen);

addr = malloc(addrlen);
err = fi_getname((fid_t)ep, addr, &addrlen);
```

Alternatively, one can use a statically allocated block of memory and check the returned addrlen to make sure the size is sufficient.

Perform address translation

Address translation is performed when endpoint addresses are inserted into the address vector. Multiple addresses can be inserted at the same time:

```
num_success = fi_av_insert(av, input_addrs, num_input, output_addrs, 0, NULL);
```

This function doesn't return 0 on success. If the return value is smaller than "num_input", events will be generated on the bound event queue. Each event reports the failure of one address translation. The "data" field of the error event entry (type "struct fi_eq_err_entry") will contain the index in the input array of the failed address.

Moving the data

Message queue

Sending and receiving messages are quite straight forward:

```
err = fi_send(ep, buf, len, desc, dest_addr, context);
err = fi_recv(ep, buf, len, desc, src_addr, context);
```

The parameter "desc" can be NULL if the provider doesn't require "FI_LOCAL_MR" mode, otherwise it should be the local descriptor of the memory region corresponding to "buf". If the "FI_CONTEXT" mode is required by the provider, "context" must point to a memory block with size no less than "struct fi_context" and the pointer value should not be reused until a completion has been generated for the operation. That means either "context" needs to be dynamically allocated before the send/recv call, or be pre-allocated and managed by the application.

For "fi_recv()", "src_addr" can be "FI_ADDR_UNSPEC", which means the message can come from any endpoint.

Notice that these calls don't accept flags. They use the default "op_flags" of the endpoint. To set different flags, use "fi_sendmsg()" and "fi_recvmsg()" instead. However, this may also mean less efficient implementation. There are also "fi_sendv()" and "fi_recvv()" for multi-iov send/recv. However, the support level varies from provider to provider.

Sometimes it may be desirable to just send a short message and forget about it. There is no context to allocate and no completion to track, and the user buffer can be reused right after the call returns. This can be achieved with the "fi_inject()" call.

```
err = fi_inject(ep, buf, len, dest_addr);
```

There is also no need to create a memory region, even with the "FI_LOCAL_MR" mode set.

Tagged message queue

Tagged messaging is similar to regular send/receive, with tag-matching added as an additional means for multiplexing a single message queue.

```
err = fi_tsend(ep, buf, len, desc, dest_addr, stag, context);
err = fi_trecv(ep, buf, len, desc, src_addr, rtag, mask, context);
```

The tags are said to be matched if "stag" and "rtag" match on all the unmasked bits, i.e.:

```
(stag & ~mask) == (rtag & ~mask)
```

Other aspects of the calls are similar to the non-tagged version, including the “inject” call:

```
err = fi_tinject(ep, buf, len, dest_addr, tag);
```

Remote memory access

Remote memory access (RMA) is the interface for one-sided operations. Instead of “send”,”recv”,”inject”, we have “write”, “read”, and “inject_write”:

```
err = fi_write(ep, buf, len, desc, dest_addr, addr, key, context);
err = fi_read(ep, buf, len, desc, src_addr, addr, key, context);
err = fi_inject_write(ep, buf, len, dest_addr, addr, key);
```

The parameters “addr” and “key” are the remote access address and remote access key for the memory region registered on the remote side. Other aspects of the calls are similar to the message queue operations.

Completions

If completions are reported through completion queue, “fi_cq_read()” is used to pull completions off the queue. If an error entry is encountered, “fi_cq_read()” sets an internal error state and return “-FI_EAVAIL”. When this happens, the application must pull the error entry with “fi_cq_readerr()” which also clears the error state. This design is to allow smaller entry size for regular completions and larger entry size for errors. The following piece of code waits for N successful completions:

```
struct fi_cq_tagged_entry entry[N];
struct fi_cq_err_entry error;
int ret, completed = 0;

while (completed < N) {
    ret = fi_cq_read(cq, entry, N);
    if (ret == -FI_EAGAIN) {
        continue;
    } else if (ret == -FI_EAVAIL) {
        fi_cq_readerr(cq, &error, NULL);
          /* handle operation error */
    } else if (ret < 0) {
          /* handle error */
    } else {
          completed += ret;
    }
}
```

If completions are reported through counter, “fi_cntr_read()” and “fi_cntr_readerr()” is used to get the number of operations completed successfully and the number of operations completed in error, respectively. The following code waits for N completions (successful or error):

```
uint64_t succ0, succ, error0, error;
uint64_t completed = 0;

succ0 = fi_cntr_read(cntr);
```

```
error0 = fi_cntr_readerr(cntr);
while (completed < N) {
    succ = fi_cntr_read(cntr);
    error = fi_cntr_readerr(cntr);
    completed = (succ - succ0) + (error - error0);
}
```

Error handling

The error numbers recognized by libfabric are defined in “fi_errno.h”. Although majority of these error number are just redefinition the linux standard error numbers, some of them are specific to libfabric (256 and above). The function “fi_strerror()” can be used to translate the error numbers to a more user friendly strings.

```
err = fi_ep_bind(ep, (fid_t)av, 0);
if (err)
    fprintf(stderr, “fi_ep_bind returns %d (%s)\n”, err, fi_strerror(-err));
```

Runtime parameters

Some environment variables are recognized by libfabric runtime for debugging purpose or for fine tuning the behavior of the common framework or individual providers. Here are some commonly used variables:

```
FI_LOG_LEVEL=warn                                 only show warning messages
FI_LOG_LEVEL=trace                                more verbose than warn
FI_LOG_LEVEL=info                                 more verbose than trace
FI_LOG_LEVEL=debug                                most verbose
FI_PROVIDER=psm2                                  use psm2 provider only
FI_PROVIDER=^sockets                              don't use sockets provider
FI_PSM2_UUID=12345678-8765-4321-1234-567887654321 set PSM2 UUID
```

The full set of environment variables are documented in the man pages. The list can also be displayed by the "fi_info" command:

```
$ fi_info -env
# 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)

……

# FI_SOCKETS_PE_AFFINITY: String
# sockets: If specified, bind the progress thread to the indicated range(s) of
Linux virtual processor ID(s). This option is currently not supported on OS X.
Usage: id_start[-id_end[:stride]][,]
```