OPENFABRICS ALLIANCE

Kernel OpenFabrics Interface

Client connection setup

Stan Smith Intel SSG/DPD

March, 2015

Steps

- The Big Picture
  - Initialization
  - Server connection setup
  - Client connection setup*
  - Connection Finalization
  - Data transfer
  - Shutdown
- Current State
  - Completed: Initialization, Server connection setup waiting for conn request
- Client connection setup:
  - fi_endpoint() *ep – create an active endpoint
  - fi_eq_open() *event_q – where connection completion events occur.
  - fi_cq_open() *send_cq – data transmission completion events occur.
  - fi_eq_bind() bind EQ to active end-point.
  - fi_ep_bind() bind send CQ to active end-point.
  - fi_enable(ep) Transition the endpoint to an active state.
  - fi_connect(ep) connect to passive ep.

fi_endpoint()

Endpoints are transport level communication portals: active or passive.

Active endpoints belong to access domains and can perform data transfers after being enabled.

Active endpoints may be connection-oriented or connectionless, and may provide data reliability.

```
struct fid_ep  *ep;

int fi_endpoint( struct fid_fabric *fabric, struct fi_info *info,
                 struct fid_ep **ep, void *context);
```

Arguments:

- *fabric - fabric instance.
- *info – application selected fi_info record.
- **ep (out) endpoint pointer.
- *context – domain event context.

fi_eq_open()

Create an event queue used to validate an active connection request completed successfully.

int fi_eq_open(struct fid_fabric *fabric, struct fi_eq_attr *attr, struct fid_eq **eq, void *context)

Arguments:

*fabric – opened fabric instance

*attr – EQ attributes

**eq – output event queue ptr.

*context – application supplied value, returned in EQ events.

Return:

0 == success, otherwise negative ERRNO.

struct fi_eq_attr

```
struct fi_eq_attr {
        size_t              size;       # min EQ depth, 0 implies a provider default.
        uint64_t            flags;      # operational flags: FI_WRITE, can write to EQ
        enum fi_wait_obj    wait_obj;       # wait object type:
                                            FI_WAIT_NONE async
                                            FI_WAIT_UNSPEC(blocking)
                                            FI_WAIT_SET, use existing wait_set.
        int                 signaling_vector;  # ?
        struct fid_wait     *wait_set;      # existing wait_set pointer;
                                            # requires FI_WAIT_SET
}
```

fi_cq_open()

Create a data transfer completion queue.

int fi_cq_open(struct fid_domain *domain, struct fi_cq_attr *attr, struct fid_cq **cq,
void *context)

Arguments:

*domain– opened domain instance

*attr – CQ attributes

**cq – output completion queue ptr.

*context – application supplied value, returned in EQ events.

Return:

0 == success, otherwise negative ERRNO.

struct fi_cq_attr

```
struct fi_cq_attr {
        size_t                  size;       # min CQ depth, 0 allows provider to choose default value
        uint64_t                flags;      # config flags: FI_WRITE, _REMOTE_SIGNAL
        enum fi_cq_format       format;     # FI_CQ_FORMAT_CONTEXT, _MSG, _DATA
        enum fi_wait_obj        wait_obj;   # FI_WAIT_NONE(async), FI_WAIT_UNSPEC(blocking)
        int                     signaling_vector; # core where CQ interrupts are directed.
        struct fi_cq_wait_cond wait_cond;
                FI_CQ_COND_NONE – No wait condition.
                FI_CQ_COND_THREASHOLD – CQ.cond field is wait for ‘cond’ entries.
        struct fid_wait         *wait_set;      # existing wait_set pointer; requires FI_WAIT_SET
}
```

fi_ep_bind()

Associate an active endpoint with an event queue

int fi_ep_bind(struct fid_ep *ep, struct fid *fid, uint64_t flags)

Arguments:

*ep – existing passive endpoint

*fid – fid to bind to the active endpoint.

flags – operational flags: FI_READ, FI_WRITE, FI_RDMA, etc. (limit to)

Return:

0 == success, otherwise negative ERRNO.

fi_connect()

Connect to a passive endpoint.

int fi_connect(struct fid_ep *ep, void *addr, void *param, size_t param_len)

Arguments:

*ep – existing active endpoint

*addr – target address, format defined in ep creation via fi_info.addr_format = FI_SOCKADDR_IN (aka struct sockaddr_in).

*param – IB private data pointer.

param_len – IB: private data length

Return:

0 == success, otherwise negative ERRNO.

fi_eq_sread()

Blocking read/peek one event from specified event queue.

int fi_eq_sread(struct fid_eq *eq, uint32_t *event, void *buffer, size_t len,
int timeout, uint64_t flags)

Arguments:

*eq – opened Event Queue

*event – reported event ID.

*buffer – event format defined via fi_info->ep_attr.protocol

len – sizeof(eq event).

timeout – default units (milliseconds) or flags defines units.

flags – operational flags:

FI_PEEK – read without consuming the event.

FI_TIME_US – micro-seconds

FI_TIME_MS – milli-seconds

Return: number of bytes written to buffer; at most 1 event.

Infiniband Client context example

```
typedef struct {
        struct fi_context       context;
        struct fi_info          *prov;
        struct fid_fabric       *fabric;        // set during initialization steps
        struct fid_domain       *domain;        // set during initialization steps
        struct fid_ep           *ep;
        struct fid_pep          *pep;
        struct fid_eq           *eq;
        struct fid_cq           *send_cq;
        struct fid_cq           *recv_cq;
        struct fid_mr           *mr;
        char                    *buf;
} application_context_t;

application_context_t       ctx = { 0 };
```

Infiniband Client RC example

```
struct fi_eq_attr       eq_attr = {10, 0, FI_WAIT_UNSPEC, 0, NULL };
struct fi_cq_attr       cq_attr = { 0, 0, FI_CQ_FORMAT_MSG, 0, 0,
                                  FI_CQ_COND_NONE , 0 };
struct fi_fid_cq        *send_cq;

provider->ep_attr->tx_ctx_cnt = (size_t) (post_depth + 1);      /* set QP WR depth */
provider->ep_attr->rx_ctx_cnt = (size_t) (post_depth + 1);

provider->tx_attr->iov_limit = 1;       // set ScatterGather max depth
provider->rx_attr->iov_limit = 1;
provider->tx_attr->inject_size = 0; // no INLINE support

ret = fi_endpoint(domain, provider, &ctx.ep, application_context );
ret = fi_eq_open(fabric, &attr, &ctx.eq, 0);
ret = fi_cq_open(domain, &cq_attr, &ctx.send_cq, 0);
```

Infiniband Client RC example II

```
uint32_t                event_id;
struct fi_eq_cm_entry   event;
char                    *param = “Private Data”
struct sockaddr_in      addr = {  0  };

ret = fi_ep_bind( &ctx.ep->fid, &ctx.send_cq->fid, 0);
ret = fi_ep_bind( &ctx.ep->fid, &ctx.eq->fid, 0);

ret = fi_enable(ep);  // transition endpoint to the ‘active’ state.

addr.sin_family = AF_INET;
addr.sin_port   = htons(PORT);
addr.sin_addr   = htonl(target);

ret = fi_connect( ctx.ep, (void *)addr, (void*)param, sizeof(*param) );

      // fi_connect() is async
```

OPENFABRICS ALLIANCE

Connection finalization is the next step


That's all Folks!
kalilak