Kernel OpenFabrics Interface

Data Transfer

Stan Smith Intel SSG/DPD

April, 2015

Steps

- The Big Picture
  - Initialization
  - Server connection setup
  - Client connection setup
  - Connection Finalization
  - Data transfer / CQ events*
  - Atomic operations
  - Shutdown

- Current State
  - Server and client endpoints are InfiniBand RC connected.
  - Both server and client have allocated send & receive CQs for bi-directional transfers.

- Client data transfer:
  - fi_send() IB Tx example

- Server server data transfer:
  - fi_recv() IB Rx example

Data Completion Events

fi_cq_sread() synchronous/blocking read CQ event(s).

fi_cq_read() non-blocking read CQ event(s).

fi_readfrom(struct fid_cq *cq, void *buf, size_t count, fi_addr_t *src);
// src points to the ‘count’ address(es) of who sent the datagram(s).

fi_cq_error() retrieve ‘CQ’ data transfer error information

Callback Semantics:

- Callback function invoked when a CQ entry is inserted into the specified CQ.

Do Eqs (not to be confused with a brand of beer) also get callbacks?

CQ Event Callback

```
struct fi_cq_attr {
        size_t                  size;
        uint64_t                flags;    // FI_CB_CTX_INTR
        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;
        void (*cq_event_handler) (struct fid_cq *cq, void *context);
};

ret = fi_cq_open(ctx->domain, &cq_attr, &ctx->send_cq, context);
```

CQ Event Callback

// CQ entry available

void *cq_handler(struct fid_cq *cq, void *context)

{

- Callback function execution context is interrupt (top-half)
- fi_cq_read() or fi_cq_sread() permissible only if provider allows.
- Function invoked when a CQ entry is place into the specified Completion Queue.

}

ret = fi_cq_open(ctx->domain, &cq_attr, &ctx->send_cq, context);

kOFI connected msg reception

```
// all connected receive functions return ‘ssize_t’
// ‘src_addr’ must be NULL or FI_ADDR_UNSPEC; provider ignores

fi_recv(struct fid_ep *ep, void *buf, size_t len, void *desc,
        fi_addr_t src_addr, void *context);

fi_recvv(struct fid_ep *ep, const struct kvec *iov, void *desc,
         size_t count, fi_addr_t src_addr, void *context);

fi_recvmsg(struct fid_ep *ep, const struct fi_msg *msg, uint64_t flags);
```

kOFI struct fi_msg

```
struct fi_msg {
   const struct kvec *msg_iov; /* scatter-gather array */
   void              **desc;   /* local request descriptors */
   size_t            iov_count;/* # elements in iov */
   fi_addr_t         addr;     /* optional endpoint address */
   void              *context; /* user-defined context */
   uint64_t          data;     /* optional Tx message data */
};
```

fi_sendmsg() & fi_recvmsg()

kOFI message reception

fi_recvmsg(struct fid_ep *ep, const struct fi_msg *msg, uint64_t flags);

flags specified with fi_recvmsg / fi_sendmsg override most flags previously configured with the endpoint, except where noted (see fi_endpoint).

FI_REMOTE_CQ_DATA - Applies to fi_sendmsg and fi_senddata. Indicates that remote CQ data is available and should be sent as part of the request. See fi_getinfo for additional details. (Immediate Data)

FI_COMPLETION - Indicates that a completion entry should be generated for the specified operation. The endpoint must be bound to an event queue with FI_COMPLETION that corresponds to the specified operation, or this flag is ignored.

kOFI message reception

FI_MORE - Indicates that the user has additional requests that will immediately be posted after the current call returns. Use of this flag may improve performance by enabling the provider to optimize its access to the fabric hardware.

FI_REMOTE_SIGNAL - Indicates that a completion event at the target process should be generated for the given operation. The remote endpoint must be configured with FI_REMOTE_SIGNAL, or this flag will be ignored by the target.

FI_INJECT - Applies to fi_sendmsg. Indicates that the outbound data buffer should be returned to user immediately after the send call returns, even if the operation is handled asynchronously. This may require that the underlying provider implementation copy the data into a local buffer and transfer out of that buffer.

kOFI message reception

FI_MULTI_RECV - Applies to posted receive operations. This flag allows the user to post a single buffer that will receive multiple incoming messages. Received messages will be packed into the receive buffer until the buffer has been consumed. Use of this flag may cause a single posted receive operation to generate multiple events as messages are placed into the buffer. The placement of received data into the buffer may be subjected to provider specific alignment restrictions. The buffer will be freed from the endpoint when the available buffer space falls below the network's MTU size (see FI_OPT_MIN_MULTI_RECV).

FI_REMOTE_COMPLETE - Applies to fi_sendmsg. Indicates that a completion should not be generated until the operation has completed on the remote side.

kOFI message reception

FI_FENCE - Applies to transmits. Indicates that the requested operation, also known as the fenced operation, be deferred until all previous operations targeting the same target endpoint have completed.

kOFI connected msg Tx

```
// all send/transmission functions return ‘ssize_t’
// ‘dest_addr’ must be NULL or FI_ADDR_UNSPEC; provider ignores.

fi_send( struct fid_ep *ep, void *buf, size_t len, void *desc,
            fi_addr_t dest_addr, void *context );

fi_sendv( struct fid_ep *ep, const struct kvec *iov, void **desc, size_t count,
             fi_addr_t dest_addr, void *context );

fi_senddata( struct fid_ep *ep, void *buf, size_t len, void *desc,
                uint64_t imm_data, fi_addr_t dest_addr, void *context );

fi_sendmsg( struct fid_ep *ep, const struct fi_msg *msg, uint64_t flags );

fi_inject( struct fid_ep *ep, void *buf, size_t len, fi_addr_t dest_addr );
```

kOFI RDMA Transfer

- fi_write() post RDMA write.
- fi_read() post RDMA read.
- fi_writev() post RDMA write (kvec).
- fi_readv() post RDMA read (kvec).
- fi_writemsg() post RDMA write (kvec + immediate data).
- fi_readmsg() post RDMA read (kvec + immediate data).

RMA for connected EP: src_addr and dest_addr ignored.

RMA for a connectionless (RD) EP:

src_addr - source node address for an RMA read operation

dest_addr - destination node address for an RMA write operation

‘addr’ arg is the RMA target memory starting address (kva).

kOFI RDMA Transfer

```
// all RMA ops return ssize_t

fi_write(struct fid_ep *ep, const void *buf, size_t len, void *desc,
        fi_addr_t dest_addr, uint64_t addr, uint64_t key, void *context);

fi_writev(struct fid_ep *ep, const struct kvec *iov, void **desc, size_t count,
        fi_addr_t dest_addr, uint64_t addr, uint64_t key, void *context);

fi_read(struct fid_ep *ep, void *buf, size_t len, void *desc, fi_addr_t src_addr,
        uint64_t addr, uint64_t key, void *context);

fi_readv(struct fid_ep *ep, const struct kvec *iov, void **desc, size_t count,
        fi_addr_t src_addr, uint64_t addr, uint64_t key, void *context);
```

kOFI RDMA Transfer

fi_inject_write (struct fid_ep *ep, const void *buf, size_t len, fi_addr_t dest_addr,
uint64_t addr, uint64_t key);

fi_writedata(struct fid_ep *ep, const void *buf, size_t len, void *desc,
uint64_t data, fi_addr_t dest_addr, uint64_t addr, uint64_t key,
void *context);

fi_writemsg(struct fid_ep *ep, const struct fi_msg_rma *msg, uint64_t flags);
(kvec + immediate data).

fi_readmsg(struct fid_ep *ep, const struct fi_msg_rma *msg, uint64_t flags);
(kvec + immediate data).

kOFI RDMA Transfer

```
struct fi_rma_iov {
        uint64_t            addr;
        size_t              len;
        uint64_t            key;
};
struct fi_msg_rma {
        const struct kvec       *msg_iov;
        void                    **desc;
        size_t                  iov_count;
        fi_addr_t               addr;
        const struct fi_rma_iov *rma_iov;
        size_t                  rma_iov_count;
        void                    *context;
        uint64_t                data;
};
```

Reliable Datagram data transfer

Why do the connected EndPoint and reliable-datagram EP Rx and Tx calls have the same format?

Specifically, what happened to fi_sendto() and fi_recvfrom()?

OFI implementors wanted to reduce the number of OFI API calls.

fi_sendto() was merged into fi_send()

fi_recvfrom() was merged into fi_recv().

Do we follow form….or add fi_sendto() & fi_recvfrom()?

Reliable Datagram Data Transfer

ssize_t
fi_recv(struct fid_ep *ep, void *buf, size_t len, void *desc,
        fi_addr_t src_addr, void *context);

src_addr == NULL or FI_ADDR_UNSPEC
  then receive from any sender.
otherwise recvfrom() the specified addr ‘src_addr’ only; MPI filtering; see fi_cq_readfrom() to determine sender’s address.

OK with src_addr semantics or do we want an explicit ‘fi_recvfrom()’?
Side note #1: fi_recv() could lose the src_addr argument?
Side note #2: OFI vs. kOFI source level compatibity requirements?

Reliable Datagram data transfer

ssize_t
fi_send(struct fid_ep *ep, void *buf, size_t len, void *desc,
fi_addr_t dst_addr, void *context);

dst_addr == (NULL or FI_ADDR_UNSPEC) then error, otherwise sendto() ‘dst_addr’.

OK with dst_addr semantics? do we want an explicit ‘fi_sendto()’ call?

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;            // set during connection establishment
        struct fid_pep          *pep;
        struct fid_eq           *eq;            / set during connect
        struct fid_cq           *send_cq;
        struct fid_cq           *recv_cq;
        struct fid_mr           *mr;
        char                    *buf;
} application_context_t;

application_context_t       ctx = { 0 };
```

IB connected Client Tx example

```
 // Allocate plus IB register a Tx buffer
ctx.buf = (char*) kzalloc(len, GFP_KERNEL);
 if (!ctx.buf) {
       print_err("kalloc failed!\n");
       return -ENOMEM;
}
ret = fi_mr_reg(ctx.domain, ctx.buf, len, 0, 0, 0, 0, &ctx.mr, NULL);

if (ret < 0)    // handle error

 ret = fi_send(ctx.ep, ctx.buf, len, fi_mr_desc(ctx.mr), 0, buf.ctx);

if (ret < 0)    // handle error
```

IB connected Client Tx example

```
// reap send completion(s)

struct fi_cq_msg_entry  comp;

do {
        ret = fi_cq_read(ctx.scq, &comp, 1);
        if (ret == -EAGAIN ) {   // 0 ?
                if ( --eagain_cnt <= 0 ) {
                        eagain_cnt = EAGAIN_TRIES;
                        schedule();
                }
        }
        if (kthread_should_stop())
                return -EINTR;
} while( ret == -EAGAIN );
```

IB connected Client Tx example

```
// Handle any TX CQ error
if (ret < 0) {
        struct fi_cq_err_entry cqe = { 0 };

        rc = fi_cq_readerr(ctx.scq, &cqe, 0);
        if (rc) {
                char buf[64];

                print_err("fi_cq_readerr() Tx err '%s'(%d)\n", fi_strerror(cqe.err), cqe.err);
                print_err("fi_cq_readerr() provider_err %s'(%d)\n",
                        fi_cq_strerror(ctx.scq, cqe.prov_errno, cqe.err_data, buf, sizeof(buf)),
                                        cqe.prov_errno);
        }
        return ret;
}
print_msg("Tx '%s'\n", (char*) comp.op_context);    // context from fi_send()
```

IB connected Rx example

```
// Post a receive

ret = fi_recv(ctx->ep, ctx.buf, len, fi_mr_desc(ctx->mr),
                    FI_ADDR_UNSPEC, (void*) ctx.buf);

if (ret < 0) {
      print_err("ERR: fi_recv() '%s'\n",fi_strerror(ret));
      return ret;
}
```

IB connected Server Rx example

```
// reap fi_recv() CQ completion

struct fi_cq_msg_entry  comp;

do {
        ret = fi_cq_read(ctx.scq, &comp, 1);
        if (ret == -EAGAIN ) {  // 0 ?
                if ( --eagain_cnt <= 0 ) {
                        eagain_cnt = EAGAIN_TRIES;
                        schedule();
                }
        }
        if (kthread_should_stop())
                return -EINTR;
} while( ret == -EAGAIN );
```

IB connected server Rx example

```
 // Handle any Rx CQ error
if (ret < 0) {
        struct fi_cq_err_entry cqe = { 0 };

        rc = fi_cq_readerr(ctx.scq, &cqe, 0);
        if (rc) {
                char buf[64];

                print_err("fi_cq_readerr() Rx err '%s'(%d)\n", fi_strerror(cqe.err), cqe.err);
                print_err("fi_cq_readerr() provider_err %s'(%d)\n",
                        fi_cq_strerror(ctx.scq, cqe.prov_errno, cqe.err_data, buf, sizeof(buf)),
                                        cqe.prov_errno);
        }
        return ret;
}
print_msg(“Rx '%s'\n", (char*) comp.op_context);    // context from fi_recv()
```

Remote Atomic operations

Base atomic operations:

fi_atomic / fi_atomicv / fi_atomicmsg / fi_inject_atomic :
Initiate an atomic operation to remote memory (ret ssize_t)

```
fi_atomic(struct fid_ep *ep, const void *buf, size_t count, void *desc,
          fi_addr_t dest_addr, uint64_t addr, uint64_t key,
          enum fi_datatype datatype,
          enum fi_op op,
          void *context);
```

fi_ops - FI_MIN, FI_MAX, FI_SUM, FI_PROD, FI_LOR, FI_LAND, FI_BOR, FI_BAND, FI_LXOR, FI_BXOR, FI_ATOMIC_READ, FI_ATOMIC_WRITE.

Remote atomic fetch

fi_fetch_atomic / fi_fetch_atomicv / fi_fetch_atomicmsg :

Initiate an atomic operation to remote memory, retrieving the initial value.

The difference between the fetch and base atomic calls are the fetch atomic routines return the initial value that was stored at the target to the user.

```
ssize_t
fi_fetch_atomic(struct fid_ep *ep, const void *buf, size_t count, void *desc,
                void *result, void *result_desc, fi_addr_t dest_addr, uint64_t addr,
                uint64_t key, enum fi_datatype datatype,
                enum fi_op op, void *context);
```

Supported atomic fetch operations: FI_MIN, FI_MAX, FI_SUM, FI_PROD, FI_LOR, FI_LAND, FI_BOR, FI_BAND, FI_LXOR, FI_BXOR, FI_ATOMIC_READ, and FI_ATOMIC_WRITE.

Remote atomic compare

fi_compare_atomic / fi_compare_atomicv / fi_compare_atomicmsg :

Initiates an atomic compare-n-swap operation to remote memory, retrieving the initial value.

```
ssize_t
fi_compare_atomic(struct fid_ep *ep, const void *buf, size_t count, void *desc,
                  const void *compare, void *compare_desc, void *result,
                  void *result_desc, fi_addr_t dest_addr, uint64_t addr,
                  uint64_t key, enum fi_datatype datatype,
                  enum fi_op op, void *context);
```

The compare atomic functions are used for operations that require comparing the target data against a value before performing a swap operation.

Supported compare atomic functions: FI_CSWAP, FI_CSWAP_NE, FI_CSWAP_LE, FI_CSWAP_LT, FI_CSWAP_GE, FI_CSWAP_GT, FI_M(asked)SWAP.

Remote atomic op valid

fi_atomic_valid / fi_fetch_atomic_valid / fi_compare_atomic_valid :

Indicates if a provider supports a specific atomic operation

```
int fi_atomicvalid(struct fid_ep *ep, enum fi_datatype datatype, enum fi_op op,
                   size_t count);
```

Zero return value indicates the specified op is support.

Connection shutdown

fi_shutdown will gracefully disconnect an endpoint from its peer.

Any posted endpoint operations will complete or be flushed (rx).

Shutdown flags:

- 0 - the endpoint is fully disconnected, and no additional data transfers will be possible.
- FI_WRITE – outbound data transfers will be disconnected.
- FI_READ – inbound data transfers will be disconnected.

An FI_SHUTDOWN event will be generated for an endpoint when the remote peer issues a disconnect using fi_shutdown or abruptly closes the endpoint.

int fi_shutdown(struct fid_ep *ep, uint64_t flags);

Close kOFI objects

Synchronously close kOFI objects. All objects begin with a ‘struct fid’.

Close implies releasing all resources associated with the object (aka destroy).

```
int  fi_close(struct fid *fid)
```

```
// endpoint close example:
ret = fi_close(&ep->fid);
```

Finito