Appendix A. Programming Examples

The set of examples in this section are specific limited examples. The examples are divided into two categories.

- daemon side – which will always be the requestor (it will post the Send Request)
- client side – which will always be the responder (it will post the Receive Request, if needed).

In these examples,

- only one message will be sent
- the minimum required resources will be used
- with minimum permissions employed
  For example: if the daemon sends data to the client, the Memory Region in the daemon will not support Write permission but the Memory Region in the client will.

The following files will be examined in this chapter:

All of these examples have as the starting point the hello_world_rc_send file.

File	QP type	opcode	completion	Data Stream Direction
hello_world_rc_send.c	RC	send	polling	Daemon => Client
hello_world_rc_send_event.c	RC	send	event	Daemon => Client
hello_world_rc_write.c	RC	RDMA write	polling	Daemon => Client
hello_world_rc_read.c	RC	RDMA read	polling	Client => Daemon
hello_world_uc_send.c	UC	send	polling	Daemon => Client
hello_world_ud_send.c	UD	send	polling	Daemon => Client

A.1 hello_world_rc_send_event.c File Differences to hello_world_rc_send.c

two ways to handle completions

Polling – the instant a completion is added to the CQ the completion can be polled (read from the CQ). Where latency/ network performance is critical polling is advantageous. Polling has a more taxing effect on the CPU. Until a completion is found in the CQ CPU cycles are wasted constantly polling. Applications to use poll

Event – the thread execution is being blocked until a completion is added to the CQ. This saves CPU cycles and frees up the CPU for other tasks. the drawback is that it takes time from the completion until the thread is notified (and allowed to continue its execution).

In order to read completions using completion events one must create a completion event channel. then using the created channel when creating CQ this allows the completion events to be notified on this channel.

The user should request notification on the next completion to be created (any completion already in the CQ will not produce a completion event). It is advised to empty the CQ after requesting notification.

User should call ibv_get_cq_event() to block until there is a completion event.

Code changes

add to struct resources
Add completion channel to the structure

hello_world_rc_send.c	hello_world_rc_send_event.c
struct resources { struct ibv_device_attrdevice_attr;/* Device attributes */ struct ibv_port_attrport_attr;/* IB port attributes */ struct ibv_device**dev_list;/* device list */ struct ibv_context*ib_ctx;/* device handle */ struct ibv_pd*pd;/* PD handle */ struct ibv_cq*cq;/* CQ handle */ struct ibv_qp*qp;/* QP handle */ struct ibv_mr*mr;/* MR handle */	struct resources { struct ibv_device_attrdevice_attr;/* Device attributes */ struct ibv_port_attrport_attr;/* IB port attributes */ struct ibv_device**dev_list;/* device list */ struct ibv_context*ib_ctx;/* device handle */ struct ibv_pd*pd;/* PD handle */ struct ibv_comp_channel*comp_channel;/* completion channel */ struct ibv_cq*cq;/* CQ handle */ struct ibv_qp*qp;/* QP handle */ struct ibv_mr*mr;/* MR handle */

Changes to Function poll_completion–
Wait for a completion event, when found, ack the event, request the notification again, and empty the cq.

hello_world_rc_send.c	hello_world_rc_send_event.c
/********** * Function: poll_completion **********/ static int poll_completion(struct resources *res) { struct ibv_wc wc; unsigned long start_time_msec, cur_time_msec; struct timeval cur_time; int rc; /* poll the completion for a while before giving up of doing it .. */ gettimeofday(&cur_time, NULL); start_time_msec = (cur_time.tv_sec * 1000) + (cur_time.tv_usec / 1000); do { rc = ibv_poll_cq(res->cq, 1, &wc); if (rc < 0) { fprintf(stderr, "poll CQ failed\n"); return 1; } gettimeofday(&cur_time, NULL); cur_time_msec = (cur_time.tv_sec * 1000) + (cur_time.tv_usec / 1000); } while ((rc == 0) && ((cur_time_msec - start_time_msec) < MAX_POLL_CQ_TIMEOUT)); /* if the CQ is empty */	/********** * Function: poll_completion **********/ static int poll_completion(struct resources *res) { struct ibv_wc wc; void *ev_ctx; struct ibv_cq *ev_cq; int rc; fprintf(stdout, "waiting for completion event\n"); /* Wait for the completion event */ if (ibv_get_cq_event(res->comp_channel, &ev_cq, &ev_ctx)) { fprintf(stderr, "failed to get cq_event\n"); return 1; } fprintf(stdout, "got completion event\n"); /* Ack the event */ ibv_ack_cq_events(ev_cq, 1); /* Request notification upon the next completion event */ rc = ibv_req_notify_cq(ev_cq, 0); if (rc) { fprintf(stderr, "Couldn't request CQ notification\n"); return 1; } /* in a real program, the user should empty the CQ before waiting for the next completion event */ /* poll the completion that causes the event (if exists) */ rc = ibv_poll_cq(res->cq, 1, &wc); if (rc < 0) { fprintf(stderr, "poll CQ failed\n"); return 1; } /* check if the CQ is empty (there can be an event event when the CQ is empty, this can happen when more than one completion(s) are being created. Here we create only one completion so empty CQ means there is an error) */

(Continued) hello_world_rc_send.c	hello_world_rc_send_event.c
if (rc == 0) { fprintf(stderr, "completion wasn't found in the CQ after timeout\n"); return 1; } fprintf(stdout, "completion was found in CQ with status 0x%x\n", wc.status); /* check the completion status (here we don't care about the completion opcode */ if (wc.status != IBV_WC_SUCCESS) { fprintf(stderr, "got bad completion with status: 0x%x, vendor syndrome: 0x%x\n", wc.status, wc.vendor_err); return 1; } return 0; }	if (rc == 0) { fprintf(stderr, "completion wasn't found in the CQ after timeout\n"); return 1; } fprintf(stdout, "completion was found in CQ with status 0x%x\n", wc.status); /* check the completion status (here we don't care about the completion opcode */ if (wc.status != IBV_WC_SUCCESS) { fprintf(stderr, "got bad completion with status: 0x%x, vendor syndrome: 0x%x\n", wc.status, wc.vendor_err); return 1; } return 0; }

Changes to Function: resources_create–

Add to resources_create() here we create a completion channel and use it in the CQ creation and request notification on the created CQ (the CQ is still empty).

hello_world_rc_send.c	hello_world_rc_send_event.c
/**	/**
* Function: resources_create	* Function: resources_create
**/	**/
static int resources_create(	static int resources_create(
struct resources *res)	struct resources *res)
{	{
/* allocate Protection Domain */	/* allocate Protection Domain */
res->pd = ibv_alloc_pd(res->ib_ctx);	res->pd = ibv_alloc_pd(res->ib_ctx);
if (!res->pd) {	if (!res->pd) {
fprintf(stderr, "ibv_alloc_pd failed\n");	fprintf(stderr, "ibv_alloc_pd failed\n");
return 1;	return 1;
}	}
	res->comp_channel = ibv_create_comp_channel(res->ib_ctx);
	if (!res->comp_channel) {
	fprintf(stderr, "ibv_create_comp_channel failed\n");
	return 1;
	}
/* each side will send only one WR, so Completion Queue with 1 entry is enough */	/* each side will send only one WR, so Completion Queue with 1 entry is enough */
cq_size = 1;	cq_size = 1;
res->cq = ibv_create_cq(res->ib_ctx, cq_size, NULL, NULL, 0);	res->cq = ibv_create_cq(res->ib_ctx, cq_size, NULL, res->comp_channel, 0);
if (!res->cq) {	if (!res->cq) {
fprintf(stderr, "failed to create CQ with %u entries\n", cq_size);	fprintf(stderr, "failed to create CQ with %u entries\n", cq_size);
return 1;	return 1;
}	}
	/* Arm the CQ before any completion is expected (to prevent races) */
	rc = ibv_req_notify_cq(res->cq, 0);
	if (rc) {
	fprintf(stderr, "failed to arm the CQ\n");
	return 1;
	}
	fprintf(stdout, "CQ was armed\n");
/* allocate the memory buffer that will hold the data */	/* allocate the memory buffer that will hold the data */
size = MSG_SIZE;	size = MSG_SIZE;
res->buf = malloc(size);	res->buf = malloc(size);
if (!res->buf) {	if (!res->buf) {

Changes to Function: resources_destroy –here we destroy the completion channel created earlier.

hello_world_rc_send.c	hello_world_rc_send_event.c
/**	/**
* Function: resources_destroy	* Function: resources_destroy
**/	**/
static int resources_destroy(	static int resources_destroy(
struct resources *res)	struct resources *res)
{	{
int test_result = 0;	int test_result = 0;
if (res->qp) {	if (res->qp) {
if (ibv_destroy_qp(res->qp)) {	if (ibv_destroy_qp(res->qp)) {
fprintf(stderr, "failed to destroy QP\n");	fprintf(stderr, "failed to destroy QP\n");
test_result = 1;	test_result = 1;
}	}
}	}
if (res->mr) {	if (res->mr) {
if (ibv_dereg_mr(res->mr)) {	if (ibv_dereg_mr(res->mr)) {
fprintf(stderr, "failed to deregister MR\n");	fprintf(stderr, "failed to deregister MR\n");
test_result = 1;	test_result = 1;
}	}
}	}
if (res->buf)	if (res->buf)
free(res->buf);	free(res->buf);
if (res->cq) {	if (res->cq) {
if (ibv_destroy_cq(res->cq)) {	if (ibv_destroy_cq(res->cq)) {
fprintf(stderr, "failed to destroy CQ\n");	fprintf(stderr, "failed to destroy CQ\n");
	test_result = 1;
	}
	}
	if (res->comp_channel) {
	if (ibv_destroy_comp_channel(res->comp_channel)) {
	fprintf(stderr, "failed to destroy completion channel\n");
test_result = 1;	test_result = 1;
}	}
}	}

A.2 hello_world_rc_send.c File Differences to hello_world_rc_write.c

This source code we will use the RDMA write opcode to send the data. In order to do this the resources in the client side (MR and QP) need to support incoming RDMA write. There is no need to post a receive request on the client side because RDMA write does not consume any.

We must make sure to exchange the address and rkey values between the daemon and the client.

When posting the send request we will use the RDMA opcode and we will use the rkey and the address of the remote side.

Note: During any RDMA operation the remote side does not post receive requests (and therefore it will not poll for a completion). The receive side is totally passive in this operation and it is up to the application to make sure the remote side knows when the operation is completed and its status.

struct cm_con_data_t {
add buffer address and remote key to the
structure struct resources {
add remote props.

hello_world_rc_send.c	hello_world_rc_write.c
#include <stdio.h> #include <stdlib.h> #include <string.h> #include <unistd.h> #include <getopt.h> #include <sys/time.h> #include <arpa/inet.h> #include <infiniband/verbs.h> #include "sock.h" /* poll CQ timeout in milisec */ #define MAX_POLL_CQ_TIMEOUT 2000 #define MSG "hello world" #define MSG_SIZE (strlen(MSG) + 1) /* structure of test parameters */ struct config_t { const char*dev_name;/* IB device name */ char*server_name;/* daemon host name */ u_int32_ttcp_port;/* daemon TCP port */ intib_port;/* local IB port to work with */ };	#include <stdio.h> #include <stdlib.h> #include <string.h> #include <unistd.h> #include <stdint.h> #include <inttypes.h> #include <endian.h> #include <byteswap.h> #include <getopt.h> #include <sys/time.h> #include <arpa/inet.h> #include <infiniband/verbs.h> #include "sock.h" /* poll CQ timeout in milisec */ #define MAX_POLL_CQ_TIMEOUT 2000 #define MSG "hello world" #define MSG_SIZE (strlen(MSG) + 1) #if __BYTE_ORDER == __LITTLE_ENDIAN static inline uint64_t htonll(uint64_t x) { return bswap_64(x); } static inline uint64_t ntohll(uint64_t x) { return bswap_64(x); } #elif __BYTE_ORDER == __BIG_ENDIAN static inline uint64_t htonll(uint64_t x) { return x; } static inline uint64_t ntohll(uint64_t x) { return x; } #else #error __BYTE_ORDER is neither __LITTLE_ENDIAN nor __BIG_ENDIAN #endif /* structure of test parameters */ struct config_t { const char*dev_name;/* IB device name */ char*server_name;/* daemon host name */ u_int32_ttcp_port;/* daemon TCP port */ intib_port;/* local IB port to work with */ };

(Continued) hello_world_rc_send.c	hello_world_rc_write.c
/* structure to exchange data which is needed to connect the QPs */ struct cm_con_data_t { uint32_t qp_num;/* QP number */ uint16_t lid;/* LID of the IB port */ } __attribute__ ((packed)); /* structure of needed test resources */ struct resources { struct ibv_device_attrdevice_attr;/* Device attributes */ struct ibv_port_attrport_attr;/* IB port attributes */ struct ibv_device**dev_list;/* device list */ struct ibv_context*ib_ctx;/* device handle */ struct ibv_pd*pd;/* PD handle */ struct ibv_cq*cq;/* CQ handle */ struct ibv_qp*qp;/* QP handle */ struct ibv_mr*mr;/* MR handle */ char*buf;/* memory buffer pointer */ intsock;/* TCP socket file descriptor */ }; struct config_t config = { "mthca0",/* dev_name */ NULL,/* server_name */ 19875,/* tcp_port */ 1/* ib_port */ };	/* structure to exchange data which is needed to connect the QPs */ struct cm_con_data_t { uint64_t addr;/* Buffer address */ uint32_t rkey;/* Remote key */ uint32_t qp_num;/* QP number */ uint16_t lid;/* LID of the IB port */ } __attribute__ ((packed)); /* structure of needed test resources */ struct resources { struct ibv_device_attrdevice_attr;/* Device attributes */ struct ibv_port_attrport_attr;/* IB port attributes */ struct cm_con_data_tremote_props;/* values to connect to remote side */ struct ibv_device**dev_list;/* device list */ struct ibv_context*ib_ctx;/* device handle */ struct ibv_pd*pd;/* PD handle */ struct ibv_cq*cq;/* CQ handle */ struct ibv_qp*qp;/* QP handle */ struct ibv_mr*mr;/* MR handle */ char*buf;/* memory buffer pointer */ intsock;/* TCP socket file descriptor */ }; struct config_t config = { "mthca0",/* dev_name */ NULL,/* server_name */ 19875,/* tcp_port */ 1/* ib_port */ };

Function Post Send

Use the RDMA Write opcode and add extra remote parameters (rkey and address needed for the RDMA operation)

hello_world_rc_send.c	hello_world_rc_write.c
/** * Function: post_send **/ static int post_send(struct resources *res) { struct ibv_send_wr sr; struct ibv_sge sge; struct ibv_send_wr *bad_wr; int rc; /* prepare the scatter/gather entry */ memset(&sge, 0, sizeof(sge)); sge.addr = (uintptr_t)res->buf; sge.length = MSG_SIZE; sge.lkey = res->mr->lkey; /* prepare the SR */ memset(&sr, 0, sizeof(sr)); sr.next = NULL; sr.wr_id = 0; sr.sg_list = &sge; sr.num_sge = 1; sr.opcode = IBV_WR_RDMA_SEND; sr.send_flags = IBV_SEND_SIGNALED; /* there is a Receive Request in the responder side, so we won't get any into RNR flow */ rc = ibv_post_send(res->qp, &sr, &bad_wr); if (rc) { fprintf(stderr, "failed to post SR\n"); return 1; } fprintf(stdout, "Send Request was posted\n"); return 0; }	/** * Function: post_send **/ static int post_send(struct resources *res) { struct ibv_send_wr sr; struct ibv_sge sge; struct ibv_send_wr *bad_wr; int rc; /* prepare the scatter/gather entry */ memset(&sge, 0, sizeof(sge)); sge.addr = (uintptr_t)res->buf; sge.length = MSG_SIZE; sge.lkey = res->mr->lkey; /* prepare the SR */ memset(&sr, 0, sizeof(sr)); sr.next = NULL; sr.wr_id = 0; sr.sg_list = &sge; sr.num_sge = 1; sr.opcode = IBV_WR_RDMA_WRITE; sr.send_flags = IBV_SEND_SIGNALED; sr.wr.rdma.remote_addr = res->remote_props.addr; sr.wr.rdma.rkey = res->remote_props.rkey; /* there is a Receive Request in the responder side, so we won't get any into RNR flow */ rc = ibv_post_send(res->qp, &sr, &bad_wr); if (rc) { fprintf(stderr, "failed to post SR\n"); return 1; } fprintf(stdout, "Send Request was posted\n"); return 0; }

There is no Post Receive in RDMA Write

hello_world_rc_send.c	hello_world_rc_write.c
* Function: post_receive **********/ static int post_receive(struct resources *res) { struct ibv_recv_wr rr; struct ibv_sge sge; struct ibv_recv_wr *bad_wr; int rc; /* prepare the scatter/gather entry */ memset(&sge, 0, sizeof(sge)); sge.addr = (uintptr_t)res->buf; sge.length = MSG_SIZE; sge.lkey = res->mr->lkey; /* prepare the RR */ memset(&rr, 0, sizeof(rr)); rr.next = NULL; rr.wr_id = 0; rr.sg_list = &sge; rr.num_sge = 1; /* post the Receive Request to the RQ */ rc = ibv_post_recv(res->qp, &rr, &bad_wr); if (rc) { fprintf(stderr, "failed to post RR\n"); return 1; } fprintf(stdout, "Receive Request was posted\n"); return 0; }	

Function Resources Create

Register the memory region in the client side with remote write support

hello_world_rc_send.c	hello_world_rc_write.c
/** * Function: resources_create **/ static int resources_create(struct resources *res) ---------- /*----------registerthismemorybuffer*/ mr_flags = (config.server_name) ? IBV_ACCESS_LOCAL_WRITE : 0; res->mr = ibv_reg_mr(res->pd, res->buf, size, mr_flags); if (!res->mr) { fprintf(stderr, "ibv_reg_mr failed with mr_flags=0x%x\n", mr_flags); return 1; } fprintf(stdout, "MR was registered with addr=%p, lkey=0x%x, rkey=0x%x, flags=0x%x\n", res->buf, res->mr->lkey, res->mr->rkey, mr_flags);	/** * Function: resources_create **/ static int resources_create(struct resources *res) ---------- ---------- /* register this memory buffer */ /* only the client expect to get incoming RDMA Write operation */ mr_flags = (config.server_name) ? IBV_ACCESS_LOCAL_WRITE | IBV_ACCESS_REMOTE_WRITE : 0; res->mr = ibv_reg_mr(res->pd, res->buf, size, mr_flags); if (!res->mr) { fprintf(stderr, "ibv_reg_mr failed with mr_flags=0x%x\n", mr_flags); return 1; } fprintf(stdout, "MR was registered with addr=%p, lkey=0x%x, rkey=0x%x, flags=0x%x\n", res->buf, res->mr->lkey, res->mr->rkey, mr_flags);

Function Modify QP to init

Configure the QP on the client side to support incoming remote write operation

hello_world_rc_send.c	hello_world_rc_write.c
/** * Function: modify_qp_to_init **/ static int modify_qp_to_init(struct ibv_qp *qp) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: RESET -> INIT */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_INIT; attr.port_num = config.ib_port; attr.pkey_index = 0; /* we don't do any RDMA operation, so remote operation is not permitted */ attr.qp_access_flags = 0; flags = IBV_QP_STATE | IBV_QP_PKEY_INDEX | IBV_QP_PORT | IBV_QP_ACCESS_FLAGS; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to INIT\n"); return rc; } return 0; }	/** * Function: modify_qp_to_init **/ static int modify_qp_to_init(struct ibv_qp *qp) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: RESET -> INIT */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_INIT; attr.port_num = config.ib_port; attr.pkey_index = 0; /* only the client expects to get incoming RDMA Write opertaion */ attr.qp_access_flags = (config.server_name) ? IBV_ACCESS_REMOTE_WRITE : 0; flags = IBV_QP_STATE | IBV_QP_PKEY_INDEX | IBV_QP_PORT | IBV_QP_ACCESS_FLAGS; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to INIT\n"); return rc; } return 0; }

Function Connect QP
There is not any need on the client side to post receive requests
add extra information (rkey and address) to the data which is being exchanged between the two sides

hello_world_rc_send.c	hello_world_rc_write.c
/** * Function: connect_qp **/ static int connect_qp(struct resources *res) { struct cm_con_data_t local_con_data, remote_con_data, tmp_con_data; int rc; /* modify the QP to init */ rc = modify_qp_to_init(res->qp); if (rc) { fprintf(stderr, "change QP state to INIT failed\n"); return rc; } /* let the client post RR to be prepared for incoming messages */ if (config.server_name) { rc = post_receive(res); if (rc) { fprintf(stderr, "failed to post RR\n"); return rc; } } /* exchange using TCP sockets info required to connect QPs */ local_con_data.qp_num = htonl(res->qp->qp_num); local_con_data.lid = htons(res->port_attr.lid); fprintf(stdout, "\nLocal LID = 0x%x\n", res->port_attr.lid); if (sock_sync_data(res->sock, !config.server_name, sizeof(struct cm_con_data_t), &local_con_data, &tmp_con_data) < 0) { fprintf(stderr, "failed to exchange connection data between sides\n"); return 1; } remote_con_data.qp_num = ntohl(tmp_con_data.qp_num); remote_con_data.lid = ntohs(tmp_con_data.lid); fprintf(stdout, "Remote QP number = 0x%x\n", remote_con_data.qp_num); fprintf(stdout, "Remote LID = 0x%x\n", remote_con_data.lid); /* modify the QP to RTR */	/** * Function: connect_qp **/ static int connect_qp(struct resources *res) { struct cm_con_data_t local_con_data, remote_con_data, tmp_con_data; int rc; /* modify the QP to init */ rc = modify_qp_to_init(res->qp); if (rc) { fprintf(stderr, "change QP state to INIT failed\n"); return rc; } /* exchange using TCP sockets info required to connect QPs */ local_con_data.addr = htonll((uintptr_t)res->buf); local_con_data.rkey = htonl(res->mr->rkey); local_con_data.qp_num = htonl(res->qp->qp_num); local_con_data.lid = htons(res->port_attr.lid); fprintf(stdout, "\nLocal LID = 0x%x\n", res->port_attr.lid); if (sock_sync_data(res->sock, !config.server_name, sizeof(struct cm_con_data_t), &local_con_data, &tmp_con_data) < 0) { fprintf(stderr, "failed to exchange connection data between sides\n"); return 1; } remote_con_data.addr = ntohll(tmp_con_data.addr); remote_con_data.rkey = ntohl(tmp_con_data.rkey); remote_con_data.qp_num = ntohl(tmp_con_data.qp_num); remote_con_data.lid = ntohs(tmp_con_data.lid); /* save the remote side attributes, we will need it for the post SR */ res->remote_props = remote_con_data; fprintf(stdout, "Remote address = 0x%"PRIx64"\n", remote_con_data.addr); fprintf(stdout, "Remote rkey = 0x%x\n", remote_con_data.rkey); fprintf(stdout, "Remote QP number = 0x%x\n", remote_con_data.qp_num); fprintf(stdout, "Remote LID = 0x%x\n", remote_con_data.lid); /* modify the QP to RTR */

Function Main

Expect to get a completion only on the server side. The client uses the data that was sent to it only after synchronizing with the server (otherwise the client does not know when the operation is finished and if its memory is valid.)

hello_world_rc_send.c	hello_world_rc_write.c
/********** ********** * Function: main ********** **********/ /* let the daemon post the SR */ if (!config.server_name) { if (post_send(&res)) { fprintf(stderr, "failed to post SR\n"); goto cleanup; } } /* in both sides we expect to get a completion */ if (poll_completion(&res)) { fprintf(stderr, "poll completion failed\n"); goto cleanup; } /* after polling the completion we have the message in the client buffer too */ if (config.server_name) printf("Message is '%s'\n", res.buf); if (sock_sync_ready(res.sock, !config.server_name)) { fprintf(stderr, "sync before end of test\n"); goto cleanup; } test_result = 0; cleanup: if (resources_destroy(&res)) { fprintf(stderr, "failed to destroy resources\n"); test_result = 1; } fprintf(stdout, "\ntest status is %d\n", test_result); return test_result; }	/********** ********** * Function: main ********** **********/ /* let the daemon post the SR */ if (!config.server_name) { if (post_send(&res)) { fprintf(stderr, "failed to post SR\n"); goto cleanup; } } /* we expect to get completion only in the daemon */ if (!config.server_name) { if (poll_completion(&res)) { fprintf(stderr, "poll completion failed\n"); goto cleanup; } } /* sync to make sure that: 1) the client won't close the resources before the daemon send the data 2) let the client read the message after it was written to it's memory */ if (sock_sync_ready(res.sock, !config.server_name)) { fprintf(stderr, "sync before end of test\n"); goto cleanup; } /* after polling the completion we have the message in the client buffer too */ if (config.server_name) printf("Message is '%s'\n", res.buf); test_result = 0; cleanup: if (resources_destroy(&res)) { fprintf(stderr, "failed to destroy resources\n"); test_result = 1; } fprintf(stdout, "\ntest status is %d\n", test_result); return test_result; }

A poll for completion is only on the daemon side. The data buffer can be used by the client after synchronization which is done at the end of the transaction.

A.3 hello_world_rc_send.c File Differences to hello_world_rc_read.c

This source code we will use the RDMA read opcode to exchange the data. In order to do this the resources in the client side (MR and QP) need to support incoming RDMA read. There is no need to post a receive request on the client side because RDMA read does not consume any.

We must make sure to exchange the address and rkey values between the daemon and the client.

When posting the send request we will use the RDMA opcode and we will use the rkey and the address of the remote side.

> Note: During any RDMA operation the remote side does not post receive requests (and therefore it will not poll for a completion). The receive side is totally passive in this operation and it is up to the application to make sure the remote side knows when the operation is completed and its status

struct cm_con_data_t {
add buffer address and remote key to the
structure struct resources {
add remote props.

hello_world_rc_send.c	hello_world_rc_read.c
#include <stdio.h>	#include <stdio.h>
#include <stdlib.h>	#include <stdlib.h>
#include <string.h>	#include <string.h>
#include <unistd.h>	#include <unistd.h>
	#include <stdint.h>
	#include <inttypes.h>
	#include <endian.h>
	#include <byteswap.h>
#include <getopt.h>	#include <getopt.h>
#include <sys/time.h>	#include <sys/time.h>
#include <arpa/inet.h>	#include <arpa/inet.h>
#include <infiniband/verbs.h>	#include <infiniband/verbs.h>
#include "sock.h"	#include "sock.h"
/* poll CQ timeout in milisec */	/* poll CQ timeout in milisec */
#define MAX_POLL_CQ_TIMEOUT 2000	#define MAX_POLL_CQ_TIMEOUT 2000
#define MSG "hello world"	#define MSG "hello world"
#define MSG_SIZE (strlen(MSG) + 1)	#define MSG_SIZE (strlen(MSG) + 1)
	#if __BYTE_ORDER == __LITTLE_ENDIAN
	static inline uint64_t htonll(uint64_t x) { return bswap_64(x); }
	static inline uint64_t ntohll(uint64_t x) { return bswap_64(x); }
	#elif __BYTE_ORDER == __BIG_ENDIAN
	static inline uint64_t htonll(uint64_t x) { return x; }
	static inline uint64_t ntohll(uint64_t x) { return x; }
	#else
	#error __BYTE_ORDER is neither __LITTLE_ENDIAN nor __BIG_ENDIAN
	#endif
/* structure of test parameters */	/* structure of test parameters */
struct config_t {	struct config_t {
const char*dev_name;/* IB device name */	const char*dev_name;/* IB device name */
char*server_name;/* daemon host name */	char*server_name;/* daemon host name */
u_int32_ttcp_port;/* daemon TCP port */	u_int32_ttcp_port;/* daemon TCP port */
intib_port;/* local IB port to work with */	intib_port;/* local IB port to work with */
};	};
/* structure to exchange data which is needed to connect the QPs */	/* structure to exchange data which is needed to connect the QPs */
struct cm_con_data_t {	struct cm_con_data_t {
	uint64_t addr;/* Buffer address */
	uint32_t rkey;/* Remote key */
uint32_t qp_num;/* QP number */	uint32_t qp_num;/* QP number */
uint16_t lid;/* LID of the IB port */	uint16_t lid;/* LID of the IB port */
} __attribute__ ((packed));	} __attribute__ ((packed));
/* structure of needed test resources */	/* structure of needed test resources */
struct resources {	struct resources {
struct ibv_device_attrdevice_attr;/* Device attributes */	struct ibv_device_attrdevice_attr;/* Device attributes */
struct ibv_port_attrport_attr;/* IB port attributes */	struct ibv_port_attrport_attr;/* IB port attributes */
	struct cm_con_data_tremote_props;/* values to connect to remote side */
struct ibv_device**dev_list;/* device list */	struct ibv_device**dev_list;/* device list */
struct ibv_context*ib_ctx;/* device handle */	struct ibv_context*ib_ctx;/* device handle */
struct ibv_pd*pd;/* PD handle */	struct ibv_pd*pd;/* PD handle */
struct ibv_cq*cq;/* CQ handle */	struct ibv_cq*cq;/* CQ handle */
struct ibv_qp*qp;/* QP handle */	struct ibv_qp*qp;/* QP handle */
struct ibv_mr*mr;/* MR handle */	struct ibv_mr*mr;/* MR handle */
char*buf;/* memory buffer pointer */	char*buf;/* memory buffer pointer */
intsock;/* TCP socket file descriptor */	intsock;/* TCP socket file descriptor */

(Continued) hello_world_rc_send.c	hello_world_rc_read.c
}; struct config_t config = { "mthca0",/* dev_name */ NULL,/* server_name */ 19875,/* tcp_port */ 1/* ib_port */ };	}; struct config_t config = { "mthca0",/* dev_name */ NULL,/* server_name */ 19875,/* tcp_port */ 1/* ib_port */ };

In the Post Send Function
Set the opcode to RDMA read and add the remote props

hello_world_rc_send.c	hello_world_rc_read.c
/*** * Function: post_send ***/ static int post_send(struct resources *res) { struct ibv_send_wr sr; struct ibv_sge sge; struct ibv_send_wr *bad_wr; int rc; /* prepare the scatter/gather entry */ memset(&sge, 0, sizeof(sge)); sge.addr = (uintptr_t)res->buf; sge.length = MSG_SIZE; sge.lkey = res->mr->lkey; /* prepare the SR */ memset(&sr, 0, sizeof(sr)); sr.next = NULL; sr.wr_id = 0; sr.sg_list = &sge; sr.num_sge = 1; sr.opcode = IBV_WR_SEND; sr.send_flags = IBV_SEND_SIGNALED; /* there is a Receive Request in the responder side, so we won't get any into RNR flow */ rc = ibv_post_send(res->qp, &sr, &bad_wr); if (rc) { fprintf(stderr, "failed to post SR\n"); return 1; } fprintf(stdout, "Send Request was posted\n"); return 0; }	/*** * Function: post_send ***/ static int post_send(struct resources *res) { struct ibv_send_wr sr; struct ibv_sge sge; struct ibv_send_wr *bad_wr; int rc; /* prepare the scatter/gather entry */ memset(&sge, 0, sizeof(sge)); sge.addr = (uintptr_t)res->buf; sge.length = MSG_SIZE; sge.lkey = res->mr->lkey; /* prepare the SR */ memset(&sr, 0, sizeof(sr)); sr.next = NULL; sr.wr_id = 0; sr.sg_list = &sge; sr.num_sge = 1; sr.opcode = IBV_WR_RDMA_READ; sr.send_flags = IBV_SEND_SIGNALED; sr.wr.rdma.remote_addr = res->remote_props.addr; sr.wr.rdma.rkey = res->remote_props.rkey; /* there is a Receive Request in the responder side, so we won't get any into RNR flow */ rc = ibv_post_send(res->qp, &sr, &bad_wr); if (rc) { fprintf(stderr, "failed to post SR\n"); return 1; } fprintf(stdout, "Send Request was posted\n"); return 0; }

As per the Note the post receive is completely removed

hello_world_rc_send.c	hello_world_rc_read.c
/** * Function: post_receive **/ static int post_receive(struct resources *res) { struct ibv_recv_wr rr; struct ibv_sge sge; struct ibv_recv_wr *bad_wr; int rc; /* prepare the scatter/gather entry */ memset(&sge, 0, sizeof(sge)); sge.addr = (uintptr_t)res->buf; sge.length = MSG_SIZE; sge.lkey = res->mr->lkey; /* prepare the RR */ memset(&rr, 0, sizeof(rr)); rr.next = NULL; rr.wr_id = 0; rr.sg_list = &sge; rr.num_sge = 1; /* post the Receive Request to the RQ */ rc = ibv_post_recv(res->qp, &rr, &bad_wr); if (rc) { fprintf(stderr, "failed to post RR\n"); return 1; } fprintf(stdout, "Receive Request was posted\n"); return 0; }	

Function: resources_create()
The client prepares the data to be sent because the daemon will read the data from the clients buffer.
The memory region is being created with remote read permission in the client side.

hello_world_rc_send.c	hello_world_rc_read.c
/** * Function: resources_create **/ /* allocate the memory buffer that will hold the data */ size = MSG_SIZE; res->buf = malloc(size); if (!res->buf) { fprintf(stderr, "failed to malloc %Zu bytes to memory buffer\n", size); return 1; } /* only in the daemon side put the message in the memory buffer */ if (!config.server_name) { strcpy(res->buf, MSG); fprintf(stdout, "going to send the message: '%s'\n", res->buf); } else memset(res->buf, 0, size); /* register this memory buffer */ mr_flags = (config.server_name) ? IBV_ACCESS_LOCAL_WRITE : 0; res->mr = ibv_reg_mr(res->pd, res->buf, size, mr_flags); if (!res->mr) { fprintf(stderr, "ibv_reg_mr failed with mr_flags=0x%x\n", mr_flags); return 1; } fprintf(stdout, "MR was registered with addr=%p, lkey=0x%x, rkey=0x%x, flags=0x%x\n", res->buf, res->mr->lkey, res->mr->rkey, mr_flags);	/** * Function: resources_create **/ /* allocate the memory buffer that will hold the data */ size = MSG_SIZE; res->buf = malloc(size); if (!res->buf) { fprintf(stderr, "failed to malloc %Zu bytes to memory buffer\n", size); return 1; } /* only in the client side put the message in the memory buffer */ if (config.server_name) { strcpy(res->buf, MSG); fprintf(stdout, "going to send the message: '%s'\n", res->buf); } else memset(res->buf, 0, size); /* register this memory buffer */ /* only the client expect to get incoming RDMA Read operation */ mr_flags = (config.server_name) ? IBV_ACCESS_REMOTE_READ : IBV_ACCESS_LOCAL_WRITE; res->mr = ibv_reg_mr(res->pd, res->buf, size, mr_flags); if (!res->mr) { fprintf(stderr, "ibv_reg_mr failed with mr_flags=0x%x\n", mr_flags); return 1; } fprintf(stdout, "MR was registered with addr=%p, lkey=0x%x, rkey=0x%x, flags=0x%x\n", res->buf, res->mr->lkey, res->mr->rkey, mr_flags);

Function: modify_qp_to_init()
The QP is being configured with remote read permission in the client side.

hello_world_rc_send.c	hello_world_rc_read.c
/*** * Function: modify_qp_to_init ***/ static int modify_qp_to_init(struct ibv_qp *qp) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: RESET -> INIT */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_INIT; attr.port_num = config.ib_port; attr.pkey_index = 0; /* we don't do any RDMA operation, so remote operation is not permitted */ attr.qp_access_flags = 0; flags = IBV_QP_STATE | IBV_QP_PKEY_INDEX | IBV_QP_PORT | IBV_QP_ACCESS_FLAGS; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to INIT\n"); return rc; } return 0; }	/*** * Function: modify_qp_to_init ***/ static int modify_qp_to_init(struct ibv_qp *qp) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: RESET -> INIT */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_INIT; attr.port_num = config.ib_port; attr.pkey_index = 0; /* only the client expects to get incoming RDMA READ opertaion */ attr.qp_access_flags = (config.server_name) ? IBV_ACCESS_REMOTE_READ : 0; flags = IBV_QP_STATE | IBV_QP_PKEY_INDEX | IBV_QP_PORT | IBV_QP_ACCESS_FLAGS; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to INIT\n"); return rc; } return 0; }

Function: modify_qp_to_rtr()
The QP is being configured to support 1 incoming RDMA read on the client side.

hello_world_rc_send.c	hello_world_rc_read.c
/*** * Function: modify_qp_to_rtr ***/ static int modify_qp_to_rtr(struct ibv_qp *qp, uint32_t remote_qpn, uint16_t dlid) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: INIT -> RTR */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_RTR; attr.path_mtu = IBV_MTU_256; attr.dest_qp_num = remote_qpn; attr.rq_psn = 0; attr.max_dest_rd_atomic = 0; attr.min_rnr_timer = 0x12; attr.ah_attr.is_global = 0; attr.ah_attr.dlid = dlid; attr.ah_attr.sl = 0; attr.ah_attr.src_path_bits = 0; attr.ah_attr.port_num = config.ib_port; flags = IBV_QP_STATE | IBV_QP_AV | IBV_QP_PATH_MTU | IBV_QP_DEST_QPN | IBV_QP_RQ_PSN | IBV_QP_MAX_DEST_RD_ATOMIC | IBV_QP_MIN_RNR_TIMER; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to RTR\n"); return rc; } return 0; }	/*** * Function: modify_qp_to_rtr ***/ static int modify_qp_to_rtr(struct ibv_qp *qp, uint32_t remote_qpn, uint16_t dlid) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: INIT -> RTR */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_RTR; attr.path_mtu = IBV_MTU_256; attr.dest_qp_num = remote_qpn; attr.rq_psn = 0; /* the client need to be responder to incoming RDMA Read */ attr.max_dest_rd_atomic = (config.server_name) ? 1 : 0; attr.min_rnr_timer = 0x12; attr.ah_attr.is_global = 0; attr.ah_attr.dlid = dlid; attr.ah_attr.sl = 0; attr.ah_attr.src_path_bits = 0; attr.ah_attr.port_num = config.ib_port; flags = IBV_QP_STATE | IBV_QP_AV | IBV_QP_PATH_MTU | IBV_QP_DEST_QPN | IBV_QP_RQ_PSN | IBV_QP_MAX_DEST_RD_ATOMIC | IBV_QP_MIN_RNR_TIMER; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to RTR\n"); return rc; } return 0; }

Function: modify_qp_to_rts()
The QP is being configured to support 1 outgoing RDMA read on the daemon side.

hello_world_rc_send.c	hello_world_rc_read.c
/*** * Function: modify_qp_to_rts ***/ static int modify_qp_to_rts(struct ibv_qp *qp) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: RTR -> RTS */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_RTS; attr.timeout = 0x12; attr.retry_cnt = 6; attr.rnr_retry = 0; attr.sq_psn = 0; attr.max_rd_atomic = 0; flags = IBV_QP_STATE | IBV_QP_TIMEOUT | IBV_QP_RETRY_CNT | IBV_QP_RNR_RETRY | IBV_QP_SQ_PSN | IBV_QP_MAX_QP_RD_ATOMIC; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to RTS\n"); return rc; } return 0; }	/*** * Function: modify_qp_to_rts ***/ static int modify_qp_to_rts(struct ibv_qp *qp) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: RTR -> RTS */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_RTS; attr.timeout = 0x12; attr.retry_cnt = 6; attr.rnr_retry = 0; attr.sq_psn = 0; /* the daemon need to be initiator of incoming RDMA Read */ attr.max_rd_atomic = (!config.server_name) ? 1 : 0; flags = IBV_QP_STATE | IBV_QP_TIMEOUT | IBV_QP_RETRY_CNT | IBV_QP_RNR_RETRY | IBV_QP_SQ_PSN | IBV_QP_MAX_QP_RD_ATOMIC; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to RTS\n"); return rc; } return 0; }

Function: connect_qp()
The connect QP does not post RR
Add the address and rkey to the exchange data.

hello_world_rc_send.c	hello_world_rc_read.c
/** * Function: connect_qp **/ static int connect_qp(struct resources *res) { struct cm_con_data_t local_con_data, remote_con_data, tmp_con_data; int rc; /* modify the QP to init */ rc = modify_qp_to_init(res->qp); if (rc) { fprintf(stderr, "change QP state to INIT failed\n"); return rc; } /* let the client post RR to be prepared for incoming messages */ if (config.server_name) { rc = post_receive(res); if (rc) { fprintf(stderr, "failed to post RR\n"); return rc; } } /* exchange using TCP sockets info required to connect QPs */ local_con_data.qp_num = htonl(res->qp->qp_num); local_con_data.lid = htons(res->port_attr.lid); fprintf(stdout, "\nLocal LID = 0x%x\n", res->port_attr.lid); if (sock_sync_data(res->sock, !config.server_name, sizeof(struct cm_con_data_t), &local_con_data, &tmp_con_data) < 0) { fprintf(stderr, "failed to exchange connection data between sides\n"); return 1; } remote_con_data.qp_num = ntohl(tmp_con_data.qp_num); remote_con_data.lid = ntohs(tmp_con_data.lid); fprintf(stdout, "Remote QP number = 0x%x\n", remote_con_data.qp_num); fprintf(stdout, "Remote LID = 0x%x\n", remote_con_data.lid); /* modify the QP to RTR */ rc = modify_qp_to_rtr(res->qp, remote_con_data.qp_num, remote_con_data.lid); if (rc) { fprintf(stderr, "failed to modify QP state from RESET to RTS\n"); return rc; }	/** * Function: connect_qp **/ static int connect_qp(struct resources *res) { struct cm_con_data_t local_con_data, remote_con_data, tmp_con_data; int rc; /* modify the QP to init */ rc = modify_qp_to_init(res->qp); if (rc) { fprintf(stderr, "change QP state to INIT failed\n"); return rc; } /* exchange using TCP sockets info required to connect QPs */ local_con_data.addr = htonll((uintptr_t)res->buf); local_con_data.rkey = htonl(res->mr->rkey); local_con_data.qp_num = htonl(res->qp->qp_num); local_con_data.lid = htons(res->port_attr.lid); fprintf(stdout, "\nLocal LID = 0x%x\n", res->port_attr.lid); if (sock_sync_data(res->sock, !config.server_name, sizeof(struct cm_con_data_t), &local_con_data, &tmp_con_data) < 0) { fprintf(stderr, "failed to exchange connection data between sides\n"); return 1; } remote_con_data.addr = ntohll(tmp_con_data.addr); remote_con_data.rkey = ntohl(tmp_con_data.rkey); remote_con_data.qp_num = ntohl(tmp_con_data.qp_num); remote_con_data.lid = ntohs(tmp_con_data.lid); /* save the remote side attributes, we will need it for the post SR */ res->remote_props = remote_con_data; fprintf(stdout, "Remote address = 0x%"PRIx64"\n", remote_con_data.addr); fprintf(stdout, "Remote rkey = 0x%x\n", remote_con_data.rkey); fprintf(stdout, "Remote QP number = 0x%x\n", remote_con_data.qp_num); fprintf(stdout, "Remote LID = 0x%x\n", remote_con_data.lid); /* modify the QP to RTR */ rc = modify_qp_to_rtr(res->qp, remote_con_data.qp_num, remote_con_data.lid); if (rc) { fprintf(stderr, "failed to modify QP state from RESET to RTS\n"); return rc; }

Function main()
A poll for completion is only on the daemon side. The data buffer can be used by the daemon side after synchronization which is done at the end of the transaction.

hello_world_rc_send.c	hello_world_rc_.c
/********** ********** * Function: main ********** **********/ /* connect the QPs */ if (connect_qp(&res)) { fprintf(stderr, "failed to connect QPs\n"); goto cleanup; } /* let the daemon post the SR */ if (!config.server_name) { if (post_send(&res)) { fprintf(stderr, "failed to post SR\n"); goto cleanup; } } /* in both sides we expect to get a completion */ if (poll_completion(&res)) { fprintf(stderr, "poll completion failed\n"); goto cleanup; } /* after polling the completion we have the message in the client buffer too */ if (config.server_name) printf("Message is '%s'\n", res.buf); if (sock_sync_ready(res.sock, !config.server_name)) { fprintf(stderr, "sync before end of test\n"); goto cleanup; } test_result = 0;	/********** ********** * Function: main ********** **********/ /* connect the QPs */ if (connect_qp(&res)) { fprintf(stderr, "failed to connect QPs\n"); goto cleanup; } /* let the daemon post the SR */ if (!config.server_name) { if (post_send(&res)) { fprintf(stderr, "failed to post SR\n"); goto cleanup; } } /* we expect to get completion only in the daemon */ if (!config.server_name) { if (poll_completion(&res)) { fprintf(stderr, "poll completion failed\n"); goto cleanup; } } /* sync to make sure that: the client won't close the resources before the daemon read the data */ if (sock_sync_ready(res.sock, !config.server_name)) { fprintf(stderr, "sync before end of test\n"); goto cleanup; } /* after polling the completion we have the message in the daemon buffer too */ if (!config.server_name) printf("Message is '%s'\n", res.buf); test_result = 0;

A.4 hello_world_rc_send.c File Differences to hello_world_uc_send.c

Using a UC QP with a send opcode is almost the same as using an RC QP. The only differences are that we create a UC QP instead of an RC QP and different parameters are used in the QP connection.

Function: resources_create()
Create a QP of the UC type instead of the RC type

hello_world_rc_send.c	hello_world_uc_send.c
/********** * Function: resources_create **********/ static int resources_create(struct resources *res) { /* create the Queue Pair */ memset(&qp_init_attr, 0, sizeof(qp_init_attr)); qp_init_attr.qp_type = IBV_QPT_RC; qp_init_attr.sq_sig_all = 1; qp_init_attr.send_cq = res->cq; qp_init_attr.recv_cq = res->cq; qp_init_attr.cap.max_send_wr = 1; qp_init_attr.cap.max_recv_wr = 1; qp_init_attr.cap.max_send_sge = 1; qp_init_attr.cap.max_recv_sge = 1; res->qp = ibv_create_qp(res->pd, &qp_init_attr); if (!res->qp) { fprintf(stderr, "failed to create QP\n"); return 1; } fprintf(stdout, "QP was created, QP number=0x%x\n", res->qp->qp_num); return 0; }	/********** * Function: resources_create **********/ static int resources_create(struct resources *res) { /* create the Queue Pair */ memset(&qp_init_attr, 0, sizeof(qp_init_attr)); qp_init_attr.qp_type = IBV_QPT_UC; qp_init_attr.sq_sig_all = 1; qp_init_attr.send_cq = res->cq; qp_init_attr.recv_cq = res->cq; qp_init_attr.cap.max_send_wr = 1; qp_init_attr.cap.max_recv_wr = 1; qp_init_attr.cap.max_send_sge = 1; qp_init_attr.cap.max_recv_sge = 1; res->qp = ibv_create_qp(res->pd, &qp_init_attr); if (!res->qp) { fprintf(stderr, "failed to create QP\n"); return 1; } fprintf(stdout, "QP was created, QP number=0x%x\n", res->qp->qp_num); return 0; }

Function: modify_qp_to_rtr()
Use only the required attributes for a UC QP connection

hello_world_rc_send.c	hello_world_uc_send.c
/*** * Function: modify_qp_to_rtr ***/ static int modify_qp_to_rtr(struct ibv_qp *qp, uint32_t remote_qpn, uint16_t dlid) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: INIT -> RTR */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_RTR; attr.path_mtu = IBV_MTU_256; attr.dest_qp_num = remote_qpn; attr.rq_psn = 0; attr.max_dest_rd_atomic = 0; attr.min_rnr_timer = 0x12; attr.ah_attr.is_global = 0; attr.ah_attr.dlid = dlid; attr.ah_attr.sl = 0; attr.ah_attr.src_path_bits = 0; attr.ah_attr.port_num = config.ib_port; flags = IBV_QP_STATE | IBV_QP_AV | IBV_QP_PATH_MTU | IBV_QP_DEST_QPN | IBV_QP_RQ_PSN | IBV_QP_MAX_DEST_RD_ATOMIC | IBV_QP_MIN_RNR_TIMER; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to RTR\n"); return rc; } return 0; }	/*** * Function: modify_qp_to_rtr ***/ static int modify_qp_to_rtr(struct ibv_qp *qp, uint32_t remote_qpn, uint16_t dlid) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: INIT -> RTR */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_RTR; attr.path_mtu = IBV_MTU_256; attr.dest_qp_num = remote_qpn; attr.rq_psn = 0; attr.ah_attr.is_global = 0; attr.ah_attr.dlid = dlid; attr.ah_attr.sl = 0; attr.ah_attr.src_path_bits = 0; attr.ah_attr.port_num = config.ib_port; flags = IBV_QP_STATE | IBV_QP_AV | IBV_QP_PATH_MTU | IBV_QP_DEST_QPN | IBV_QP_RQ_PSN; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to RTR\n"); return rc; } return 0; }

Function: modify_qp_to_rts()
Use only the required attributes for a UC QP connection

hello_world_rc_send.c	hello_world_uc_send.c
/********** * Function: modify_qp_to_rts **********/ static int modify_qp_to_rts(struct ibv_qp *qp) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: RTR -> RTS */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_RTS; attr.timeout = 0x12; attr.retry_cnt = 6; attr.rnr_retry = 0; attr.sq_psn = 0; attr.max_rd_atomic = 0; flags = IBV_QP_STATE | IBV_QP_TIMEOUT | IBV_QP_RETRY_CNT | IBV_QP_RNR_RETRY | IBV_QP_SQ_PSN | IBV_QP_MAX_QP_RD_ATOMIC; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to RTS\n"); return rc; } return 0; }	/********** * Function: modify_qp_to_rts **********/ static int modify_qp_to_rts(struct ibv_qp *qp) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: RTR -> RTS */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_RTS; attr.sq_psn = 0; flags = IBV_QP_STATE | IBV_QP_SQ_PSN; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to RTS\n"); return rc; } return 0; }

A.5 hello_world_rc_send.c File Differences to hello_world_ud_send.c

struct resources {
Add Remote props

Add Address handle

hello_world_rc_send.c	hello_world_ud_send.c
/* poll CQ timeout in milisec */ #define MAX_POLL_CQ_TIMEOUT 2000 #define MSG "hello world" #define MSG_SIZE (strlen(MSG) + 1)	/* poll CQ timeout in milisec */ #define MAX_POLL_CQ_TIMEOUT 2000 #define MSG "hello world" #define MSG_SIZE (strlen(MSG) + 1) /* qkey value that we will use */ #define DEF_QKEY 0x12345 /* Global Routing Header size */ #define GRH_SIZE 40
/* structure of test parameters */ struct config_t { const char*dev_name;/* IB device name */ char*server_name;/* daemon host name */ u_int32_ttcp_port;/* daemon TCP port */ intib_port;/* local IB port to work with */ };	/* structure of test parameters */ struct config_t { const char*dev_name;/* IB device name */ char*server_name;/* daemon host name */ u_int32_ttcp_port;/* daemon TCP port */ intib_port;/* local IB port to work with */ };
/* structure to exchange data which is needed to connect the QPs */ struct cm_con_data_t { uint32_t qp_num;/* QP number */ uint16_t lid;/* LID of the IB port */ } __attribute__ ((packed));	/* structure to exchange data which is needed to connect the QPs */ struct cm_con_data_t { uint32_t qp_num;/* QP number */ uint16_t lid;/* LID of the IB port */ } __attribute__ ((packed));
/* structure of needed test resources */ struct resources { struct ibv_device_attrdevice_attr;/* Device attributes */ struct ibv_port_attrport_attr;/* IB port attributes */ struct ibv_device**dev_list;/* device list */ struct ibv_context*ib_ctx;/* device handle */ struct ibv_pd*pd;/* PD handle */ struct ibv_cq*cq;/* CQ handle */ struct ibv_qp*qp;/* QP handle */ struct ibv_mr*mr;/* MR handle */ char*buf;/* memory buffer pointer */ intsock;/* TCP socket file descriptor */ }; struct config_t config = { "mthca0",/* dev_name */ NULL,/* server_name */ 19875,/* tcp_port */ 1/* ib_port */ };	/* structure of needed test resources */ struct resources { struct ibv_device_attrdevice_attr;/* Device attributes */ struct ibv_port_attrport_attr;/* IB port attributes */ struct cm_con_data_tremote_props;/* values to connect to remote side */ struct ibv_device**dev_list;/* device list */ struct ibv_context*ib_ctx;/* device handle */ struct ibv_pd*pd;/* PD handle */ struct ibv_cq*cq;/* CQ handle */ struct ibv_qp*qp;/* QP handle */ struct ibv_ah*ah;/* AH handle */ struct ibv_mr*mr;/* MR handle */ char*buf;/* memory buffer pointer */ intsock;/* TCP socket file descriptor */ }; struct config_t config = { "mthca0",/* dev_name */ NULL,/* server_name */ 19875,/* tcp_port */ 1/* ib_port */ };

Function: post_send()
Add to the send request: the Address handle to the remote side, and add the remote QP attributes (the qkey and the qp_num).

hello_world_rc_send.c	hello_world_ud_send.c
/********** * Function: post_send **********/ static int post_send(struct resources *res) { struct ibv_send_wr sr; struct ibv_sge sge; struct ibv_send_wr *bad_wr; int rc; /* prepare the scatter/gather entry */ memset(&sge, 0, sizeof(sge)); sge.addr = (uintptr_t)res->buf; sge.length = MSG_SIZE; sge.lkey = res->mr->lkey; /* prepare the SR */ memset(&sr, 0, sizeof(sr)); sr.next = NULL; sr.wr_id = 0; sr.sg_list = &sge; sr.num_sge = 1; sr.opcode = IBV_WR_SEND; sr.send_flags = IBV_SEND_SIGNALED; /* there is a Receive Request in the responder side, so we won't get any into RNR flow */ rc = ibv_post_send(res->qp, &sr, &bad_wr); if (rc) { fprintf(stderr, "failed to post SR\n"); return 1; } fprintf(stdout, "Send Request was posted\n"); return 0; }	/********** * Function: post_send **********/ static int post_send(struct resources *res) { struct ibv_send_wr sr; struct ibv_sge sge; struct ibv_send_wr *bad_wr; int rc; /* prepare the scatter/gather entry */ memset(&sge, 0, sizeof(sge)); sge.addr = (uintptr_t)res->buf; sge.length = MSG_SIZE; sge.lkey = res->mr->lkey; /* prepare the SR */ memset(&sr, 0, sizeof(sr)); sr.next = NULL; sr.wr_id = 0; sr.sg_list = &sge; sr.num_sge = 1; sr.opcode = IBV_WR_SEND; sr.send_flags = IBV_SEND_SIGNALED; sr.wr.ud.ah = res->ah; sr.wr.ud.remote_qpn = res->remote_props.qp_num; sr.wr.ud.remote_qkey = DEF_QKEY; /* there is a Receive Request in the responder side, so we won't get any into RNR flow */ rc = ibv_post_send(res->qp, &sr, &bad_wr); if (rc) { fprintf(stderr, "failed to post SR\n"); return 1; } fprintf(stdout, "Send Request was posted\n"); return 0; }

Function: post_receive()

Always make sure that the buffer is large enough to hold the message and the grh because every incoming UD message may contain a grh.

hello_world_rc_send.c	hello_world_ud_send.c
/********** * Function: post_receive **********/ static int post_receive(struct resources *res) { struct ibv_recv_wr rr; struct ibv_sge sge; struct ibv_recv_wr *bad_wr; int rc; /* prepare the scatter/gather entry */ memset(&sge, 0, sizeof(sge)); sge.addr = (uintptr_t)res->buf; sge.length = MSG_SIZE; sge.lkey = res->mr->lkey; /* prepare the RR */ memset(&rr, 0, sizeof(rr)); rr.next = NULL; rr.wr_id = 0; rr.sg_list = &sge; rr.num_sge = 1; /* post the Receive Request to the RQ */ rc = ibv_post_recv(res->qp, &rr, &bad_wr); if (rc) { fprintf(stderr, "failed to post RR\n"); return 1; } fprintf(stdout, "Receive Request was posted\n"); return 0; }	/********** * Function: post_receive **********/ static int post_receive(struct resources *res) { struct ibv_recv_wr rr; struct ibv_sge sge; struct ibv_recv_wr *bad_wr; int rc; /* prepare the scatter/gather entry */ memset(&sge, 0, sizeof(sge)); sge.addr = (uintptr_t)res->buf; sge.length = MSG_SIZE + GRH_SIZE; sge.lkey = res->mr->lkey; /* prepare the RR */ memset(&rr, 0, sizeof(rr)); rr.next = NULL; rr.wr_id = 0; rr.sg_list = &sge; rr.num_sge = 1; /* post the Receive Request to the RQ */ rc = ibv_post_recv(res->qp, &rr, &bad_wr); if (rc) { fprintf(stderr, "failed to post RR\n"); return 1; } fprintf(stdout, "Receive Request was posted\n"); return 0; }

Function: resources_create
add enough place for the GRH in the client side memory buffer and memory region Create a QP of the UD type instead of the RC type

hello_world_rc_send.c	hello_world_ud_send.c
/** * Function: resources_create **/ static int resources_create(struct resources *res) /* each side will send only one WR, so Completion Queue with 1 entry is enough */ cq_size = 1; res->cq = ibv_create_cq(res->ib_ctx, cq_size, NULL, NULL, 0); if (!res->cq) { fprintf(stderr, "failed to create CQ with %u entries\n", cq_size); return 1; } /* allocate the memory buffer that will hold the data */ size = MSG_SIZE; res->buf = malloc(size); if (!res->buf) { fprintf(stderr, "failed to malloc %Zu bytes to memory buffer\n", size); return 1; } /* create the Queue Pair */ memset(&qp_init_attr, 0, sizeof(qp_init_attr)); qp_init_attr.qp_type = IBV_QPT_RC; qp_init_attr.sq_sig_all = 1; qp_init_attr.send_cq = res->cq; qp_init_attr.recv_cq = res->cq; qp_init_attr.cap.max_send_wr = 1; qp_init_attr.cap.max_recv_wr = 1; qp_init_attr.cap.max_send_sge = 1; qp_init_attr.cap.max_recv_sge = 1;	/** * Function: resources_create **/ static int resources_create(struct resources *res) /* each side will send only one WR, so Completion Queue with 1 entry is enough */ cq_size = 1; res->cq = ibv_create_cq(res->ib_ctx, cq_size, NULL, NULL, 0); if (!res->cq) { fprintf(stderr, "failed to create CQ with %u entries\n", cq_size); return 1; } /* allocate the memory buffer that will hold the data */ size = MSG_SIZE; /* add enough place for the GRH in the client side */ if (config.server_name) size += GRH_SIZE; res->buf = malloc(size); if (!res->buf) { fprintf(stderr, "failed to malloc %Zu bytes to memory buffer\n", size); return 1; } /* create the Queue Pair */ memset(&qp_init_attr, 0, sizeof(qp_init_attr)); qp_init_attr.qp_type = IBV_QPT_UD; qp_init_attr.sq_sig_all = 1; qp_init_attr.send_cq = res->cq; qp_init_attr.recv_cq = res->cq; qp_init_attr.cap.max_send_wr = 1; qp_init_attr.cap.max_recv_wr = 1; qp_init_attr.cap.max_send_sge = 1; qp_init_attr.cap.max_recv_sge = 1;

Function: modify_qp_to_init(
In UD QP there in no need to configure the remote operations (RDMA is not supported) qkey needs to be configured

hello_world_rc_send.c	hello_world_ud_send.c
/*** * Function: modify_qp_to_init ***/ static int modify_qp_to_init(struct ibv_qp *qp) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: RESET -> INIT */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_INIT; attr.port_num = config.ib_port; attr.pkey_index = 0; /* we don't do any RDMA operation, so remote operation is not permitted */ attr.qp_access_flags = 0; flags = IBV_QP_STATE | IBV_QP_PKEY_INDEX | IBV_QP_PORT | IBV_QP_ACCESS_FLAGS; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to INIT\n"); return rc; } return 0; }	/*** * Function: modify_qp_to_init ***/ static int modify_qp_to_init(struct ibv_qp *qp) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: RESET -> INIT */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_INIT; attr.port_num = config.ib_port; attr.pkey_index = 0; attr.qkey = DEF_QKEY; flags = IBV_QP_STATE | IBV_QP_PKEY_INDEX | IBV_QP_PORT | IBV_QP_QKEY; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to INIT\n"); return rc; } return 0; }

Function: modify_qp_to_rtr()
Use only the required attributes for a UC QP connection

hello_world_rc_send.c	hello_world_ud_send.c
/*** * Function: modify_qp_to_rtr ***/ static int modify_qp_to_rtr(struct ibv_qp *qp, uint32_t remote_qpn, uint16_t dlid) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: INIT -> RTR */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_RTR; attr.path_mtu = IBV_MTU_256; attr.dest_qp_num = remote_qpn; attr.rq_psn = 0; attr.max_dest_rd_atomic = 0; attr.min_rnr_timer = 0x12; attr.ah_attr.is_global = 0; attr.ah_attr.dlid = dlid; attr.ah_attr.sl = 0; attr.ah_attr.src_path_bits = 0; attr.ah_attr.port_num = config.ib_port; flags = IBV_QP_STATE | IBV_QP_AV | IBV_QP_PATH_MTU | IBV_QP_DEST_QPN | IBV_QP_RQ_PSN | IBV_QP_MAX_DEST_RD_ATOMIC | IBV_QP_MIN_RNR_TIMER; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to RTR\n"); return rc; } return 0; }	/*** * Function: modify_qp_to_rtr ***/ static int modify_qp_to_rtr(struct ibv_qp *qp) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: INIT -> RTR */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_RTR; flags = IBV_QP_STATE; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to RTR\n"); return rc; } return 0; }

Function: modify_qp_to_rts()
Use only the required attributes for a UC QP connection

hello_world_rc_send.c	hello_world_ud_send.c
/*** * Function: modify_qp_to_rts ***/ static int modify_qp_to_rts(struct ibv_qp *qp) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: RTR -> RTS */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_RTS; attr.timeout = 0x12; attr.retry_cnt = 6; attr.rnr_retry = 0; attr.sq_psn = 0; attr.max_rd_atomic = 0; flags = IBV_QP_STATE | IBV_QP_TIMEOUT | IBV_QP_RETRY_CNT | IBV_QP_RNR_RETRY | IBV_QP_SQ_PSN | IBV_QP_MAX_QP_RD_ATOMIC; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to RTS\n"); return rc; } return 0; }	/*** * Function: modify_qp_to_rts ***/ static int modify_qp_to_rts(struct ibv_qp *qp) { struct ibv_qp_attr attr; int flags; int rc; /* do the following QP transition: RTR -> RTS */ memset(&attr, 0, sizeof(attr)); attr.qp_state = IBV_QPS_RTS; attr.sq_psn = 0; flags = IBV_QP_STATE | IBV_QP_SQ_PSN; rc = ibv_modify_qp(qp, &attr, flags); if (rc) { fprintf(stderr, "failed to modify QP state to RTS\n"); return rc; } return 0; }

Function: connect_qp()
Create in the client side the address handle to be able to send messages to the daemon.

hello_world_rc_send.c	hello_world_ud_send.c
/** * Function: connect_qp **/ static int connect_qp(struct resources *res) /* exchange using TCP sockets info required to connect QPs */ local_con_data.qp_num = htonl(res->qp->qp_num); local_con_data.lid = htons(res->port_attr.lid); fprintf(stdout, "\nLocal LID = 0x%x\n", res->port_attr.lid); if (sock_sync_data(res->sock, !config.server_name, sizeof(struct cm_con_data_t), &local_con_data, &tmp_con_data) < 0) { fprintf(stderr, "failed to exchange connection data between sides\n"); return 1; } remote_con_data.qp_num = ntohl(tmp_con_data.qp_num); remote_con_data.lid = ntohs(tmp_con_data.lid);	/** * Function: connect_qp **/ static int connect_qp(struct resources *res) /* exchange using TCP sockets info required to connect QPs */ local_con_data.qp_num = htonl(res->qp->qp_num); local_con_data.lid = htons(res->port_attr.lid); fprintf(stdout, "\nLocal LID = 0x%x\n", res->port_attr.lid); if (sock_sync_data(res->sock, !config.server_name, sizeof(struct cm_con_data_t), &local_con_data, &tmp_con_data) < 0) { fprintf(stderr, "failed to exchange connection data between sides\n"); return 1; } remote_con_data.qp_num = ntohl(tmp_con_data.qp_num); remote_con_data.lid = ntohs(tmp_con_data.lid);

hello_world_rc_send.c	hello_world_ud_send.c
fprintf(stdout, "Remote QP number = 0x%x\n", remote_con_data.qp_num); fprintf(stdout, "Remote LID = 0x%x\n", remote_con_data.lid);	fprintf(stdout, "Remote QP number = 0x%x\n", remote_con_data.qp_num); fprintf(stdout, "Remote LID = 0x%x\n", remote_con_data.lid); /* save the remote side attributes, we will need it for the post SR */ res->remote_props = remote_con_data;
/* modify the QP to RTR */ rc = modify_qp_to_rtr(res->qp, remote_con_data.qp_num, remote_con_data.lid); if (rc) { fprintf(stderr, "failed to modify QP state from RESET to RTS\n"); return rc; } /* only the daemon post SR, so only he should be in RTS (the client can be moved to RTS as well) */ if (config.server_name) fprintf(stdout, "QP state was change to RTR\n"); else {	/* modify the QP to RTR */ rc = modify_qp_to_rtr(res->qp); if (rc) { fprintf(stderr, "failed to modify QP state from RESET to RTS\n"); return rc; } /* only the daemon post SR, so only he should be in RTS (the client can be moved to RTS as well) */ if (config.server_name) fprintf(stdout, "QP state was change to RTR\n"); else { struct ibv_ah_attr ah_attr;
rc = modify_qp_to_rts(res->qp); if (rc) { fprintf(stderr, "failed to modify QP state from RESET to RTS\n"); return rc; } fprintf(stdout, "QP state was change to RTS\n"); } /* sync to make sure that both sides are in states that they can connect to prevent packet loose */ if (sock_sync_ready(res->sock, !config.server_name)) { fprintf(stderr, "sync after QPs are were moved to RTS\n"); return 1; } return 0; }	rc = modify_qp_to_rts(res->qp); if (rc) { fprintf(stderr, "failed to modify QP state from RESET to RTS\n"); return rc; } fprintf(stdout, "QP state was change to RTS\n"); /* create an Address Handle to be able to send the message to the remote side */ memset(&ah_attr, 0, sizeof(ah_attr)); ah_attr.is_global = 0; ah_attr.dlid = remote_con_data.lid; ah_attr.sl = 0; ah_attr.src_path_bits = 0; ah_attr.port_num = config.ib_port; res->ah = ibv_create_ah(res->pd, &ah_attr); if (!res->ah) { fprintf(stderr, "failed to create AH\n"); return 1; } fprintf(stdout, "AH was created\n"); } /* sync to make sure that both sides are in states that they can connect to prevent packet loose */ if (sock_sync_ready(res->sock, !config.server_name)) { fprintf(stderr, "sync after QPs are were moved to RTS\n"); return 1; } return 0; }

Function: resources_destroy
here we destroy the address handle created earlier

hello_world_rc_send.c	hello_world_ud_send.c
/**	/**
* Function: resources_destroy	* Function: resources_destroy
**/	**/
static int resources_destroy(	static int resources_destroy(
struct resources *res)	struct resources *res)
if (res->cq) {	if (res->cq) {
if (ibv_destroy_cq(res->cq)) {	if (ibv_destroy_cq(res->cq)) {
fprintf(stderr, "failed to destroy CQ\n");	fprintf(stderr, "failed to destroy CQ\n");
test_result = 1;	test_result = 1;
}	}
}	}
	if (res->ah) {
	if (ibv_destroy_ah(res->ah)) {
	fprintf(stderr, "failed to destroy AH\n");
	test_result = 1;
	}
}	}
if (res->pd) {	if (res->pd) {
if (ibv_dealloc_pd(res->pd)) {	if (ibv_dealloc_pd(res->pd)) {
fprintf(stderr, "failed to deallocate PD\n");	fprintf(stderr, "failed to deallocate PD\n");
test_result = 1;	test_result = 1;
}	}
}	}

Function: main()
In the receiver side the incoming data is always placed after the grh (whether the grh is present or not).

hello_world_rc_send.c	hello_world_ud_send.c
/********** ********** * Function: main ********** **********/ int main(int argc, char *argv[]) /* in both sides we expect to get a completion */ if (poll_completion(&res)) { fprintf(stderr, "poll completion failed\n"); goto cleanup; } /* after polling the completion we have the message in the client buffer too */ if (config.server_name) printf("Message is '%s'\n", res.buf); if (sock_sync_ready(res.sock, !config.server_name)) { fprintf(stderr, "sync before end of test\n"); goto cleanup; } test_result = 0;	/********** ********** * Function: main ********** **********/ int main(int argc, char *argv[]) /* in both sides we expect to get a completion */ if (poll_completion(&res)) { fprintf(stderr, "poll completion failed\n"); goto cleanup; } /* after polling the completion we have the message in the client buffer too, the data will be placed after the space that was reserved for the GRH */ if (config.server_name) printf("Message is '%s'\n", res.buf + GRH_SIZE); if (sock_sync_ready(res.sock, !config.server_name)) { fprintf(stderr, "sync before end of test\n"); goto cleanup; } test_result = 0;

A.6 Sock

This file is a provides a socket abstraction functions to the test.

A.7 Make File (How to Compile)

```
CC = gcc
OFED_PATH = /usr/local/ofed
DEFAULT_CFLAGS = -I${OFED_PATH}/include
DEFAULT_LDFLAGS = -L${OFED_PATH}/lib64 -L${OFED_PATH}/lib

CFLAGS += $(DEFAULT_CFLAGS) -g -O2 -Wall -Werror
LDFLAGS += $(DEFAULT_LDFLAGS) -libverbs
OBJECTS = hello_world_rc_send.o hello_world_uc_send.o hello_world_ud_send.o sock.o
OBJECTS +=hello_world_rc_send_event.o hello_world_rc_write.o hello_world_rc_read.o
TARGETS = hello_world_rc_send hello_world_uc_send hello_world_ud_send hello_world_rc_send_event
TARGETS += hello_world_rc_write hello_world_rc_read

all: $(TARGETS)

hello_world_rc_send: hello_world_rc_send.o sock.o
$(CC) $^ -o $@ $(LDFLAGS)

hello_world_uc_send: hello_world_uc_send.o sock.o
$(CC) $^ -o $@ $(LDFLAGS)

hello_world_ud_send: hello_world_ud_send.o sock.o
$(CC) $^ -o $@ $(LDFLAGS)

hello_world_rc_send_event: hello_world_rc_send_event.o sock.o
$(CC) $^ -o $@ $(LDFLAGS)
```

```
hello_world_rc_write: hello_world_rc_write.o sock.o
$(CC) $^ -o $@ $(LDFLAGS)

hello_world_rc_read: hello_world_rc_read.o
sock.o $(CC) $^ -o $@ $(LDFLAGS)

hello_world_rc_send.o: hello_world_rc_send.c
sock.h $(CC) -c $(CFLAGS) $<

hello_world_uc_send.o: hello_world_uc_send.c
sock.h $(CC) -c $(CFLAGS) $<

hello_world_ud_send.o: hello_world_ud_send.c
sock.h $(CC) -c $(CFLAGS) $<

hello_world_rc_send_event.o: hello_world_rc_send_event.c
sock.h $(CC) -c $(CFLAGS) $<

hello_world_rc_write.o: hello_world_rc_write.c
sock.h $(CC) -c $(CFLAGS) $<

hello_world_rc_read.o: hello_world_rc_read.c sock.h
$(CC) -c $(CFLAGS) $<

sock.o: sock.c sock.h
$(CC) -c $(CFLAGS) $<

clean:
rm -f $(OBJECTS) $(TARGETS)
```

A.8 From device to QP

In order to establish a reliable connection (RC) a number of actions must be performed to have one QP configured at every host. The procedure for opening an RC is summarized in Table 5.

This is done by first locating the device to use and opening it.

To search for the available devices use ***ibv_get_device_list*** that returns a list of devices. Query the devices in the list by calling functions like ***ibv_get_device_name***, ***ibv_get_device_guid***. To open a specific device call ***ibv_open_device***. This function returns a pointer to the device context. Remember to free the device list using ***ibv_free_device_list***. It is safe to call ***ibv_free_device_list*** after calling ***ibv_open_device***, the opened device will not be freed.

After opening the device create a protection domain (PD) by calling ***ibv_alloc_pd***. This function receives the device context received earlier. Upon success, the function returns the newly allocated PD. Now allocate the memory to be used for the RQ/SQ and register it. The allocation needs to be done on a valid memory buffer which can be obtained by calling malloc/post_memalign or any static array/variable and the memory regis-tration is done using ***ibv_reg_mr***. The registration process requires passing the PD, the memory address (in user space), size, and the required permissions. After the registration, the memory will be nonswapable and will be pinned in the same physical memory needed for HW to perform direct memory access.

After creating the PD, allocating the memory, and registering it, it is now possible to create the completion queue (CQ). This queue will hold entries of completed work requests (WR). The CQ is created using

ibv_create_cq. The function receives the following parameters: the device context, the queue length (number of entries), and pointer to completion channel if it needs to work with completion events. To work asyn-chronously create a completion channel before creating the CQ using ***ibv_create_comp_channel***.

Table 5 - From Device to Queue Pair

Step #	Verb	Description	Return Values
1	***Section 3.2.1 on page 32***	Search for the available devices	Returns a list of devices
2	***Section 3.2.4 on page 34***	Queries the device	Device name
3	***Section 3.2.3 on page 33***	Queries the device	Device Guid
4	***Section 3.3.1 on page 34***	Open a specific device	Returns a pointer to the device context
5	***Section 3.2.2 on page 33***	Free the device	Does not return any value
6	***Section 3.5.1 on page 39***	Create a protection domain	Successful: returns a pointer to the allocated PD when the operation is successfully competed Unsuccessful: returns NULL if the request fails.
7	***Section 3.13.1 on page 55***	Registers a memory region (MR)	Successful: returns a pointer to the registered MR Unsuccessful: returns NULL if the request fails.
8	***Section 3.11.1 on page 52***	Create a Completion Queue	Successful: returns a pointer to the CQ Unsuccessful: returns NULL if the request fails
9	***Section 3.10.1 on page 51***	Create a completion event channel for the InfiniBand device context context	Successful: returns a pointer to the created completion event channel Unsuccessful: returns NULL if the request fails.

A.8.1 Creating the QP

The procedure for creating a QP is summarized in Table 6.

To create the QP call ***ibv_create_qp*** using the protection domain (PD) created earlier, and a structure of ***ibv_qp_init_attr***. The type most important attributes that need to be updated in this structure are:

- ***send_cq, recv_cq*** – Pointers to the send and receive CQ. Use the same CQ for both send and receive, or two CQs, one per queue.
- ***max_send_sge, max_recv_sge*** (1) – parameters that define the maximum number of scatter gather elements per work requests. A simple application will use a value of 1.
- ***max_recv_wr, max_send_wr*** (1) – parameters that define the maximum number or pending work requests. This number should typically be smaller then the number of entries in the CQ in order to prevent the CQ from getting full.
- ***qp_type*** (IBV_QPT_RC) – the connection type ***ibv_qp_type*** (QP Transport Service Type RC/ UC/ UD)
- ***sq_sig_all*** (1) – defines whether a CQE will be generated on completion of all WQEs. If 0 the decision will be made according to the value of ***ibv_send_wr.send_flags*** in post send.
- ***max_inline_data*** (1) – defines the buffer area that might be required of the driver to perform scatter gather inline operations. See Section A.1.9, “Inline,” on page 119 for more details. A typ-ical value will be 1 (maybe 0?).

Table 6 - Creating the Queue Pair

	Verb or Argument	Use	Returns
1	***Section 3.8.1 on page 45***	Create a Queue Pair	Successful: returns a pointer to the created QP Unsuccessful: returns NULL if the request fails. Check the QP number (qp_num) in the returned QP.
2	***Section 3.9.1 on page 48***	modifies the attributes of QP qp with the attributes in attr according to the mask attr_mask.	Successful: returns 0 on success returns the value of error on failure (which indicates the failure reason).
3	***Section 3.11.1 on page 52***	Create a Queue Pair	Successful:returns a pointer to the created QP Unsuccessful:returns NULL if the request fails. Check the QP number (qp_num) in the returned QP.

After creating the QP move it to init state. To do this call ***ibv_modify_qp***. This function modifies the attributes of the QP according to the attributes specified in struct of type ***ibv_qp_attr*** and a mask that identifies the changed fields of the attribute struct (only fields of the attribute struct that are marked by the mask will be processed). The attributes which need to be modified using this verb depends on the QP transport type.

The following attributes are for Reliable Connected QP:

RESET->INIT

- ***qp_state*** – switched to IBV_QPS_INIT (bit IBV_QP_STATE in mask)
- ***port_num*** – the local port number for receiving / transmitting (bit IBV_QP_PORT in mask). This is the port that the data will be sent through.
- ***Pkey_index*** – (0) The index in the pkey table that this QP will be associated with. (bit IBV_QP_PKEY_INDEX in mask)
- ***qp_access_flags*** – memory access permissions that the QP will support as receiver. A value of 0 is sufficient and means that the QP will not accept any RDMA /Atomic operations. When work-ing with RDMA, this needs to be modified (bit IBV_QP_ACCESS_FLAGS in mask).

From init state advance to ready to receive state (RTR). To perform this again call ***ibv_modify_qp*** this time with other attributes. It is important to remember at this stage that in an RC the source address of a receive transaction and the destination address of a transmit transaction are the same and are predefined for the entire data transfer. This address is made up of the gid/lid, the port number, and the QP number. This address will be configured (among other fields) as we move to RTR.

- ***qp_state*** – switched to IBV_QPS_RTR (bit IBV_QP_STATE in mask)
- ***ah_attr*** – the source/destination address. This includes the following sub fields dlid, port_num (local port number to reach the other host). bit IBV_QP_AV in mask.
- ***dest_qp_num*** – the destination QP number (bit IBV_QP_DEST_QPN in mask)
- ***min_rnr_timer*** (12) – timeout to wait in case of a Receiver Not Ready (RNR) NACK is received. A value of 1 roughly corresponds to 10 uSec,
  a value 0f 10 roughly corresponds to 300 uSec and
  a value of 31 roughly corresponds to 490 mSec (bit IBV_QP_MIN_RNR_TIMER in mask).
- ***path_mtu*** - The maximal packet size that can traverse all the way to the destination. Note that this field is an enum and not a value. (bit IBV_QP_PATH_MTU in mask)
- ***max_dest_rd_atomic*** – (1) ???(bit IBV_QP_MAX_DEST_RD_ATOMIC in mask)
- ***rq_psn*** –packet sequence number The sequence number of the first sent packet (bit IBV_QP_RQ_PSN in mask)

From RTR switch to ready to send state (RTS). To perform this again call ***ibv_modify_qp*** this time with other attributes.

- ***qp_state*** - switched to IBV_QPS_RTS (bit IBV_QP_STATE in mask)
- ***timeout*** (12) –defines the time out period from transmission to the arrival of an ACK/NACK in μsec.

$$4.096 \times 2^{\text{TIMEOUT}}$$

A value of 1 roughly corresponds to 10 uSec,
a value 0f 10 roughly corresponds to 300 uSec and
a value of 31 roughly corresponds to 490 mSec (bit IBV_QP_TIMEOUT in mask)

- ***retry_cnt*** – the maximum number of transmission retries (bit IBV_QP_RETRY_CNT in mask).
- ***rnr_retry*** (5) – Number of retries in cases where a Receiver Not Ready (RNR) NACK is received (bit IBV_QP_RNR_RETRY in mask).
- ***sq_psn*** – send queue packet sequence number. The sequence number of the first sent packet. The sequence number will be incremented automatically from then on (bit IBV_QP_SQ_PSN in mask).
- ***max_rd_atomic*** (1) - ??? (bit IBV_QP_MAX_QP_RD_ATOMIC in mask).

A.9 Opening an Unreliable Datagram (UD)

The entire flow up to the point of creating the QP is similar to the RC flow.

The flow diverges in the call to ***ibv_create_qp*** where the QP type is changed from IBV_QPT_RC to are IBV_QPT_UD. The rest of the creation attributes similar to RC.

The first call to ***ibv_modify_qp*** is also similar to RC except that the qkey parameter needs to be configured. On the receiver side, this key will be compared to the qkey in received packets. The updated attributes at this stage are:

RESET->INIT

- ***qp_state*** – switched to IBV_QPS_INIT (bit IBV_QP_STATE in mask)
- ***port_num*** – This is the local port number of the port that the data will be sent / received through. (bit IBV_QP_PORT in mask).
- ***Pkey_index*** – (0) The index in the pkey table that this QP will be associated with (bit IBV_QP_PKEY_INDEX in mask)
- ***qpkey*** – value to be matched against incoming packets (bit IBV_QP_QKEY in mask)

Unlike RC, when moving to RTR and RTS in UD it is not necessary to know the source / destination address (the destination address will be needed for the ***ibv_post_send***). All that is necessary is to switch the state. The updated attributes are:

INIT->RTR:

- ***qp_state*** – switched to IBV_QPS_RTR (bit IBV_QP_STATE in mask)

RTR->RTS

- ***qp_state*** – switched to IBV_QPS_RTS (bit IBV_QP_STATE in mask)

- ***sq_psn*** – send queue packet sequence number. The sequence number of the first sent packet. The sequence number will be incremented automatically from then on (bit IBV_QP_SQ_PSN in mask).

A.10 Opening RDMA Write over RC/UC

Most of the setup sequence to establish a working QP is similar to the one performed for non RDMA RC with a few exceptions.

When using RDMA the receiver of the RDMA Write must configure the memory attributes of the memory region to enable remote memory access. This is done by setting in the ***ibv_access_flags*** parameter in the call to ***ibv_reg_mr*** the flag of IBV_ACCESS_REMOTE_WRITE (IBV_ACCESS_LOCAL_WRITE must be enabled too).

The memory access permissions are validated at the memory region level (above) and also at the QP level. As a result during QP creation (more specifically when switching the QP from reset to init state) it is crucial to set the receiver's QP access permissions to enable RDMA. This is done by setting the ***ibv_qp_attr.qp_access_flags*** to a value similar to that of the memory region (IBV_ACCESS_REMOTE_WRITE).

When working with RDMA write without immediate data (transaction opcode of IBV_WR_RDMA_WRITE) it is not necessary to post receive buffers on the receiver side using ibv_post_receive. Further more when working without immediate data no CQ event will be created after the RDMA transaction completes. As a result, polling the CQ is not a relevant way to know if a transaction occurred. When working with RDMA with immediate data (transaction opcode of IBV_WR_RDMA_WRITE_WITH_IMM) it is necessary to post receive buffers to accept the immediate data and a CQ event will be generated for every completed transaction.

A.11 Opening RDMA Read over RC

Most of the setup sequence to establish a working QP is similar to the one performed for non RDMA RC with a few exceptions.

When using RDMA the receiver of the RDMA Read must configure the memory attributes of the memory region to enable remote memory access. This is done by setting in the ibv_access_flags parameter in the call to ibv_reg_mem ibv_reg_mr the flag of IBV_ACCESS_REMOTE_READ.

The memory access permissions are validated at the memory region level (above) and also at the QP level. As a result during QP creation (more specifically when switching the QP from reset to init state) it is crucial to set the receiver's QP access permissions to enable RDMA. This is done by setting the ibv_qp_attr.qp_access_flags to a value similar to that of the memory region (IBV_ACCESS_REMOTE_WRITE).

The attribute ibv_qp_attr.max_dest_rd_atomic means how many outstanding RDMA Read/atomic operations this QP can process in parallel as the receiver, this value being set in INIT->RTR.

The attribute ibv_qp_attr.max_rd_atomic means how many outstanding RDMA Read/atomic operations this QP can process in parallel as the sender, this value being set in RTR->RTS.

A.12 Closing a Connection

In order to close the connection in an orderly fashion the following actions should be taken (the reverse order from the open sequence):

1. ibv_destroy_qp
2. ibv_destroy_cq
3. ibv_dereg_mr
4. free memory buffer allocated by user
5. ibv_dealloc_pd
6. ibv_destroy_comp_channel
7. ibv_close_device

If you init the all pointers to NULL before trying to open the various handles and if you check for non NULL pointers you can call the closing procedure from anywhere in your code as part of an escape sequence. The code will look something like this:

```
void close_ib_device(echo_server_db *ib_data)
{
   if(ib_data->ib_qp) ibv_destroy_qp(ib_data->ib_qp); if(ib_data-
   >ib_cq) ibv_destroy_cq(ib_data->ib_cq); if(ib_data-
   >ib_mem_region) ibv_dereg_mr(ib_data->ib_mem_region);
   if(ib_data->mem_ptr) free(ib_data->mem_ptr); if(ib_data-
   >ib_pd) ibv_dealloc_pd(ib_data->ib_pd);
   if(ib_data->ib_channel) ibv_destroy_comp_channel(ib_data-
    >ib_channel); /* optional */
   if(ib_data->ib_context) ibv_close_device(ib_data->ib_context);
}
```

Closing the connection is similar in RC and UD.

It is advised to check the return value of resource destruction verbs.

A.13 Receive flow

A.13.1 Reliable Connection

In order for the receive action to work it is necessary to post receive requests. This is done by calling ***ibv_post_recv***. Every posted request will contain (among other information) the receive buffer memory address, the length of the receive buffer, and the buffers lkey (received when registering the memory). The memory passed to the ***ibv_post_recv*** must be global or dynamically allocated (no local variables) and must have been previously registered using ***ibv_reg_mr***. A simple function that posts a receive action will look something like:

```
int post_receive_entry(struct ibv_qp *ib_qp, uint32_t lkey, char *mem_region,
        uint32_t length)
{
   struct ibv_sge mem_list;
   struct ibv_recv_wr wr;
   struct ibv_recv_wr *bad_wr;
   memset(&wr, 0, sizeof(struct ibv_recv_wr));
```

```
    wr.sg_list    = &mem_list;
    wr.wr_id       =(uint64_t)mem_region; /* store mem address for recv in wr_id
    wr.num_sge    */ = 1;
    mem_list.lkey  = lkey;
    mem_list.addr = (uintptr_t)mem_region;
    mem_list.length = length;
    return(ibv_post_recv(ib_data->ib_qp, &wr, &bad_wr));
}
```

When a packet is received its content will be put in the buffer passed by the ***ibv_post_recv***. If you want your application to be aware of the received packet you can take one of two actions.

- For an asynchronous notification open a completion channel using ***ibv_create_comp_channel***.
- For a synchronous notification use ***ibv_poll_cq*** to poll the completions queue. Every receive WR that was posted and fulfilled will be added to the completions queue. The function ***ibv_poll_cq*** returns a struct of type ***ibv_wc*** that contains the following information:
  - the event status (success or fail code) that must be checked by the user
  - the opcode (send, recv,…)
  - the transaction data length and the 64 bit wr_id
  - The wr_id can contain any user data (like memory address of the buffer) and can be used to manage the applications memory buffers. The value of ***wr_id*** will be the same value that was set either in the call to ***ibv_post_recv*** or in the ***ibv_post_send***.

The function also returns the number of completions (for a number >= 0) or a failure indication if less then 0.

A simple RCV code that assumes all the data in the CQ is due only to packet reception will look something like:

```
int receive_data(echo_server_db  *ib_data)
{
    struct ibv_wc       rcv_wc;
    int poll_ ret, rx_length, buff_length = MAX_SUPPORTED_STRING;
    uint32_t lkey = ib_data->ib_mem_region->lkey;
    poll_ret = ibv_poll_cq(ib_data->ib_cq, 1, &rcv_wc);
    if(poll_ret > 0) {/* message received */
        if(rcv_wc.status != IBV_WC_SUCCESS) fprintf(stderr,"Recieve
            failed with error %d\n",rcv_wc.status);
        rx_length = rcv_wc.byte_len;
        /* during the post_receive_entry the buffer address was inserted into the wr_id field */
        fprintf(stderr, "%s ",(char *)rcv_wc.wr_id);
        /* return the used entry to the receive queue so as not to run out of receives */
        /* Before doing this it is necessary to make sure there will not be any more access this memory */
        /* because memory is again available for receive */
        if(post_receive_entry(ib_data->ib_qp, lkey, (char *)rcv_wc.wr_id,
        buff_length)) fprintf(stderr, "receive_data: failed in post_receive_entry\n");
}
if(poll_ret < 0) {
        fprintf(stderr, "receieve_data: Failed in RCV in ibv_poll_cq\n");
```

```
        return(poll_ret);

}
return 0;
}
```

A.13.2 Unreliable Datagram

The receive flow of UD is similar to that of RC with one exception. The packet received contains the GRH header. This means that the received message size will be the TX length + sizeof (struct ***ibv_grh***). Similarly the data will be shifted by sizeof (struct ***ibv_grh***) bytes within the receive buffer.

Note: : In UD, the QP can get incoming messages from any other QP in the subnet (in cases where the Pkey and the Qkey of the sender QP matches the local QP attributes).

A.13.3 RDMA Write over RC

When working with RDMA without immediate data it is not necessary to post receive WQE. Also assume that no CQ events will be generated upon RDMA write completions. When working with RDMA with immediate data it is necessary to post receive events and handle the CQ events that will be generated after every transaction completion.

In both cases it is necessary to pass the virtual address (64 bit pointer address) of the buffer that was allocated for the RDMA operation, the R_KEY and length of the buffer to the originator of the RDMA write so it can use them. The R_KEY can be read from the ***ibv_mr.rkey*** field of the memory region that corresponds to the memory where the buffer is located.

A.14 Transmission flow

A.14.1 Reliable Connection

Once a connection is established use ***ibv_post_send*** to send a single packet. The action requires a QP, an ***lkey***, and registered memory that contains the data to be passed to the ***ibv_post_send***. Another useful field is the 64 bit ***wr_id*** that will be passed to the CQ upon transmission completion and can be used by the applica-tion in order to manage the memory buffers. It is important to remember that ***ibv_post_send*** will produce a CQ element (upon send or failure) that must be handled (at least cleared). In the case that the CQ is full the ***ibv_post_send*** will fail. A simple TX example that assumes all elements in the CQ result from TX opera-tions (no RCV) is given below.

```
int transmit_data(echo_server_db *ib_data, uint32_t lkey, char *mem_region, int
        length)
{
    int ret;
    struct ibv_wc tx_wc; struct
    ibv_send_wr *bad_wr; struct
    ibv_sge list;
    struct ibv_send_wr wr = {
       .wr_id  = (uint64_t)mem_region,
       .sg_list      = &list,
       .num_sge      = 1,
       .opcode     = IBV_WR_SEND,
```

```
        .send_flags = IBV_SEND_SIGNALED,
    };
    list.lkey = ib_data->ib_mem_region->lkey;
    list.addr = (uintptr_t)mem_region;
    list.length = length;

    if(ret = ibv_post_send(ib_data->ib_qp, &wr,&bad_wr))
    {
        fprintf(stderr,"transmit_data: post send failed\n");
        return(ret);
    }
    /* empty the completion queue even on the transmit */
    if(ret = ibv_poll_cq(ib_data->ib_cq, 1, &tx_wc) < 0) {
        fprintf(stderr,"transmit_data: ibv_poll_cq failed\n");
        return(ret);
    }
    return(0);
}
```

A.14.2 Unreliable Datagram

The UD transmit operation extends RC transmit by requiring the destination address for each and every ***ibv_post_send*** operation. This is accomplished by updating the ***wr.ud*** fields of the ***ibv_send_wr*** structure passed to the post function. The following code can be added to the RC transmit code above to support UD transmission. Data that was supplied as part of QP establishment in RC is marked in red, new data in green.

```
    struct ibv_send_wr wr;
    struct ibv_ah *ah;
    struct ibv_ah_attr ah_attr  = {
        .dlid = ib_data->net_data.dest_lid,
        .port_num = ib_data->net_data.local_port_num
    };

    ah = ibv_create_ah(ib_data-
    >ib_pd,&ah_attr) if(!ah) {
        fprintf(stderr,"transmit_data: failed to create address header\n");
    }
    wr.wr.ud.remote_qpn  = ib_data->net_data.dest_qpn;
    wr.wr.ud.remote_qkey = TRANSACTION_QKEY;
    wr.wr.ud.ah          = ah;
```

Note: The ***remote_qkey*** parameter must match that qkey configured by the receiver in the QP. A mismatch of the keys will cause the packet to be dropped.

A.14.3 RDMA Write over RC

The RDMA write over RC extends the regular RC send operation with the main differences being:

- The send opcode defined in ***ibv_send_wr.opcode*** needs to be ***IBV_WR_RDMA_WRITE*** or ***IBV_WR_RDMA_WRITE_WITH_IMM*** (instead of ***IBV_WR_SEND***).
- The sender needs to configure the receiver's destination address in the ***ibv_send_wr.wr.rdma.remote_addr*** fields and the receiver's R_KEY value for this address in the ***ibv_send_wr.wr.rdma.rkey*** field.

A simple transmit function (without the error handling) will look something like:

```
int transmit_data(echo_server_db  *ib_data)
{
   struct                         ibv_wc tx_wc;
   int                           string_length , ret, transmit_count = 0;
   struct                        ibv_send_wr *bad_wr;
   struct                        ibv_sge list;
   struct                        ibv_send_wr wr;

   scanf("%s", ib_data->local_data.mem_ptr);
   string_length = strlen(tmp_Buffer);
   list.lkey                     = ib_data->ib_mem_region->lkey;
   list.addr                     = (uintptr_t)ib_data->local_data.mem_ptr;
   list.length                    = string_length + 1;
   memset(&wr, 0, sizeof(struct ibv_send_wr));
   wr.sg_list                    = &list;
   wr.num_sge                    = 1;
   wr.opcode                     = IBV_WR_RDMA_WRITE;
   /* IBV_WR_RDMA_WRITE_WITH_IMM*/
   wr.send_flags                  = IBV_SEND_SIGNALED;/* IBV_SEND_SIGNALED */
   wr.wr.rdma.rkey                = ib_data->remote_data.rkey;
   wr.wr_id                      = 0x1234;
   wr.wr.rdma.remote_addr = (uintptr_t)ib_data->remote_data.mem_ptr;

   if(ret       =       ibv_post_send(ib_data->ib_qp,       &wr,&bad_wr))       {
                              fprintf(stderr,"transmit_data: ibv_post_send
   failed\n");
                              return(ret);
   }
   /* empty the completion queue, even on the transmit there is a race here. We
   might be reading the completion before it was generated, but its just an example
   */ if(ret = ibv_poll_cq(ib_data->ib_cq, 1, &tx_wc) < 0) {
                              fprintf(stderr,"transmit_data: ibv_poll_cq
   failed\n");
                              return(ret);
   }
}
```

A.15 Other Attributes:

A.15.1 Multiple Scatter Gather Elements (SGE)

It is possible to perform a scatter gather operation. This operation can work both on the receiver side and on the transmitter side. On the transmitter side the action collects data from multiple buffers and sends them as a single stream. On the receiver side the same operation receives a single stream and breaks it down to numerous buffers. To perform a scatter gather operation it is necessary to define the ***max_send_sge*** (for transmitter) or ***max_recv_sge*** for receiver. It is also necessary to supply an array of ***ibv_sge*** in the ***ibv_post_send*** or ***ibv_post_recv*** operations with the array size in ***num_sge***. The maximum number of SGE is defined by HW and is currently a few tens of SGEs.

A.15.2 Inline

When performing transmit with scatter gather it is possible to tell the driver to perform the scatter gather instead of the HCA. In this case the driver will copy the scattered data to a single continuous memory buffer and transmit it from there. Working with the inline option might improve performance if the scatter list is composed of small buffers. To enable inline it is necessary to define the maximum inline buffer in QP cre-ation by setting the value in the ***ibv_qp_init_attr.max_inline_data*** field. It is also necessary to enable the inline operation in every send operation where it is requested by setting the ***ibv_send_wr.send_flags*** to ***IBV_SEND_INLINE***.

Note: Working with an inline of size 24 B improves small transactions through put (smaller then 24B) considerably.

Note: Working with an inline of size up to 400 B improves latency of single packet exchange (smaller then 24B) considerably but reduces load throughput.

Note: Working with WQE lists can improve throughput considerably with almost no penalty. the list size can be small 50 WQE.

A.16 Data Structures:

In order to have an RC connection, your application must hold a valid copy of each of the following data structures:

- struct ibv_device *ib_device;
- struct ibv_context *ib_context;
- struct ibv_comp_channel *ib_channel; /* optional - completion channel */
- struct ibv_pd *ib_pd;
- unsigned int mem_lngth;
- void *mem_ptr;/* pointer to allocated memory */
- struct ibv_mr *ib_mem_region;
- struct ibv_ah *ib_ah;
- struct ibv_cq *ib_cq;

struct ibv_qp *ib_qp;

A.17 Echo_Server.c

```
#include <stdio.h>
#include <string.h>
#include <memory.h>
#include <verbs.h>

#define PSN                     0x4321
#define COMPLETE_QUEUE_LENGTH100
#define MAX_SUPPORTED_STRING256

#define ECHO_SERVER_ID          0x1234

/* data needed to establish the connection */
typedef struct
{
   unsigned int             local_dev_num;
   unsigned int             local_port_num;
   unsigned int             dest_lid;
   unsigned int             dest_port;
   unsigned long            dest_qpn;
   unsigned int             is_recv;
} network_data_t;

/* all the IB structures needed for the connection */
typedef struct
{
   struct ibv_device                    *ib_device;
   struct ibv_context                   *ib_context;
   struct ibv_comp_channel *ib_channel;                 /* completion event channel */
   struct ibv_pd                          *ib_pd;       /* protection domain */
   unsigned int                           mem_lngth;
   void                                                 *mem_ptr;
   struct ibv_mr                          *ib_mem_region;
   struct ibv_ah                          *ib_ah;
   struct ibv_cq                          *ib_cq;
   struct ibv_qp                          *ib_qp;
   network_data_t                         net_data;
} echo_server_db;


int close_ib_device(echo_server_db *ib_data)
{
   fprintf(stderr, "close_ib_device: calling ibv_destroy_qp\n");
   if(ib_data->ib_qp)                   ibv_destroy_qp(ib_data->ib_qp);
   fprintf(stderr, "close_ib_device: calling ibv_destroy_cq\n");
   if(ib_data->ib_cq)                   ibv_destroy_cq(ib_data->ib_cq);
   fprintf(stderr, "close_ib_device: calling ibv_dereg_mr\n");
   if(ib_data->ib_mem_region) ibv_dereg_mr(ib_data->ib_mem_region);
   fprintf(stderr, "close_ib_device: calling free\n");
   if(ib_data->mem_ptr)      free(ib_data->mem_ptr);
   fprintf(stderr, "close_ib_device: calling ibv_dealloc_pd\n");
   if(ib_data->ib_pd) ibv_dealloc_pd(ib_data->ib_pd);
   fprintf(stderr, "close_ib_device: calling ibv_destroy_comp_channel\n");
   if(ib_data->ib_channel) ibv_destroy_comp_channel(ib_data->ib_channel);
   fprintf(stderr, "close_ib_device: calling ibv_close_device\n"); if(ib_data-
   >ib_context) ibv_close_device(ib_data->ib_context);
```

```
    fprintf(stderr, "finished closing device\n");

    return 0;
}

/* post a single recieve entry.*/
inline int post_receive_entry(struct ibv_qp *ib_qp, const uint32_t lkey,
                                    char *mem_region, const uint32_t length)
{
    struct ibv_sge          mem_list;
    struct ibv_recv_wr      wr;
    struct ibv_recv_wr      *bad_wr;

    memset(&wr, 0, sizeof(struct ibv_recv_wr));
    wr.sg_list              &mem_list;
    wr.num_sge              = 1;
    wr.wr_id                = (uint64_t)mem_region;/* save address for receive in wr_id */

    mem_list.lkey           = lkey;
    mem_list.addr           = (uintptr_t)mem_region;
    mem_list.length         = length;

    return(ibv_post_recv(ib_qp, &wr, &bad_wr));
}

/* Initial posting of recive requests. More requests will be added */
/* as these requests will be consumed.                             */
int post_receive(echo_server_db  *ib_data)
{
    int                             i;
    char                    *mem_area;
    uint32_t                lkey = ib_data->ib_mem_region->lkey;

    for (i = 0; i < COMPLETE_QUEUE_LENGTH; ++i)
    {
                            /* use double buffer */
                            mem_area = (char *)(((unsigned long)ib_data->mem_ptr) + ((i & 1)*
    MAX_SUPPORTED_STRING));

                            if(post_receive_entry(ib_data->ib_qp, lkey,
    mem_area, MAX_SUPPORTED_STRING))
                                    break;
    }

    if(i < COMPLETE_QUEUE_LENGTH-1) {
        fprintf(stderr,"receieve_data: failed to allocate receive queue using ibv_post_recv
        i=%d\n",i); return(-1);
    }

    fprintf(stderr,"receieve_data: Allocated receive WE using ibv_post_recv\n");

    return 0;
}

/* recive data and print the data received to the stderr */
```

```
/* the function assumes that all data in CQ was due to RCV and not to
*/ /* TX or errors. */
int receive data(echo server db  *ib data)
{
   struct ibv_wc      rcv_wc;
   int                                    poll_ret, rx_length, recieve_count = 0;
   uint32_t                               lkey = ib_data->ib_mem_region->lkey;

   while(recieve_count <= 60) {
       poll_ret = ibv_poll_cq(ib_data->ib_cq, 1, &rcv_wc);
       if(poll_ret > 0) {
                       if(rcv_wc.status != IBV_WC_SUCCESS)
                             fprintf(stderr,"Recieve of message failed with error %d\n",rcv_wc.status);

                       rx_length = rcv_wc.byte_len; fprintf(stderr,
                       "%s ",(char *)rcv_wc.wr_id);

                       /* return the used entry to the receive queue so we don't run out of receives */
                       if(post_receive_entry(ib_data->ib_qp, lkey, (char *)rcv_wc.wr_id,
       MAX_SUPPORTED_STRING)) {
                                fprintf(stderr, "receive_data: failed in post_receive_entry\n");
                       }
                       recieve_count++;
       }
       if(poll_ret < 0) fprintf(stderr, "receieve_data: Failed in RCV in ibv_poll_cq\n");
   }
   return 0;
}

/* transmit data received from the stdin and send it to the server */
/* the function assumes that all entries in the CQ are due to transmit
*/ /* and not to recieve. */
int transmit_data(echo_server_db  *ib_data)
{
   int                         transmit_count = 0;
   int                         cq_count = 0;
   char                        *tmp_Buffer;
   struct ibv_wc tx_wc;
   int                         string_length;
   int                         ret;
   struct                      ibv_send_wr *bad_wr;
   struct ibv_sge list;
   struct ibv_send_wr wr = {
       .wr_id                = ECHO_SERVER_ID,
       .sg_list      = &list,
       .num_sge      = 1,
       .opcode       = IBV_WR_SEND,
       .send_flags = IBV_SEND_SIGNALED,
   };

   list.lkey                    = ib_data->ib_mem_region->lkey;

   while(transmit_count < 60) {
       /* use double buffer on receive */
       tmp_Buffer = (char *)(((unsigned long)ib_data->mem_ptr) +
```

```
                        (transmit_count & 1) * MAX_SUPPORTED_STRING);

        /* get data from user */
        memset(tmp_Buffer, 0, MAX_SUPPORTED_STRING);
        scanf("%s",tmp_Buffer);
        string_length = strlen(tmp_Buffer);

        list.addr = (uintptr_t)tmp_Buffer;
        list.length = string_length + 1;

        if(ret = ibv_post_send(ib_data->ib_qp, &wr,&bad_wr)) {
                        fprintf(stderr,"transmit_data: ibv_post_send failed\n");
                        return(ret);
        }

        /* increment and wrap around
        */ transmit_count ++;

        /* we must empty the completion queue even on the transmit
        */ do {
                        if(ret = ibv_poll_cq(ib_data->ib_cq, 1, &tx_wc) < 0)
                        { fprintf(stderr,"transmit_data: ibv_poll_cq failed\n");
                        return(ret);
                        }
                        cq_count += ret;
        } while (cq_count < transmit_count);
    }
}



inline void get_data_from_user(echo_server_db  *ib_data)
{
    fprintf(stderr, "Type 1 for server\n");
       scanf("%d",&ib_data->net_data.is_recv);
    printf("Type remote LID\n");
       scanf("%d",&ib_data->net_data.dest_lid);
    printf("Type remote PORT\n");
       scanf("%d",&ib_data->net_data.dest_port);
    printf("Type remote QP number\n");
       scanf("%d",&ib_data->net_data.dest_qpn);
}

int create_qp(echo_server_db  *ib_data)
{
    int ret, func_ret = -1;
    struct ibv_qp_init_attr                 qp_init_attr;
    struct ibv_qp_attr                        qp_attr;
    enum  ibv_qp_attr_mask                  qp_attr_mask;
    struct ibv_ah_attr                        dest_attr;
    unsigned long long                        guid;
    struct ibv_port_attr                    port_attr;
    uint16_t                                  lid;
```

```
    qp_init_attr.qp_context                 = NULL;// ??
    qp_init_attr.send_cq                    = ib_data->ib_cq;
    qp_init_attr.recv_cq                    = ib_data->ib_cq;
    qp_init_attr.srq                        = NULL;
    qp_init_attr.cap.max_inline_data=     /* max inline = 1 */
    1; qp_init_attr.cap.max_recv_sge        = 1;/* do not use scatter gather on RX */
    qp_init_attr.cap.max_send_sge           = 1;/* do not use scatter gather on TX */
    qp_init_attr.cap.max_recv_wr            = COMPLETE_QUEUE_LENGTH - 1;// smaller then CQ length
    qp_init_attr.cap.max_send_wr            = COMPLETE_QUEUE_LENGTH - 1;// smaller then CQ length
    qp_init_attr.qp_type                    = IBV_QPT_RC;// connection type
    qp_init_attr.sq_sig_all                 = 1;/* generate CQ for every WQE*/
    //qp_init_attr.xrc_domain               = NULL;// ??

// ibv_create_qp - Create a queue pair.
ib_data->ib_qp = ibv_create_qp(ib_data->ib_pd,
&qp_init_attr); if(!ib_data->ib_qp) {
                                fprintf(stderr, "create_qp: failed in
ibv_create_qp\n");
                                goto func_return;
}

/* Switch QP from reset to init state. Also config no ATOMIC or RDMA
*/ /* and configure to work with physical port 1 */
memset((char *)&qp_attr,0, sizeof(struct ibv_qp_attr));
qp attr.qp state                = IBV QPS INIT;
qp_attr.port_num                = ib_data->net_data.local_port_num;// port number
qp_attr.pkey_index              = 0;
qp_attr.qp_access_flags = 0;  /* */
qp_attr_mask =                  IBV_QP_STATE          |
                                IBV_QP_PORT           |
                                IBV_QP_PKEY_INDEX     |
                                IBV_QP_ACCESS_FLAGS;
if(ret = ibv_modify_qp(ib_data->ib_qp, &qp_attr, qp_attr_mask))
   {  fprintf(stderr, "create_qp: failed to init QP with ibv_modify_qp error %d\n", ret);
   goto func_return;
}

// ---- query the port and QP for debug purposes ------
ibv_query_port(ib_data->ib_context, ib_data->net_data.local_port_num,
&port_attr); lid = port_attr.lid;
qp_attr_mask =                  IBV_QP_PORT          |
                                IBV_QP_STATE         |
                                IBV_QP_AV            |
                                IBV_QP_PATH_MTU      |
                                IBV_QP_DEST_QPN      |
                                IBV_QP_RQ_PSN;
memset(&qp_attr,0, sizeof(struct ibv_qp_attr)); ibv_query_qp(ib_data-
>ib_qp, &qp_attr,qp_attr_mask,&qp_init_attr);
fprintf(stderr, "QP query: State %d, port %d, MTU %d, QPN %d lid %d
   \n",  qp_attr.qp_state, qp_attr.port_num, qp_attr.
   path_mtu, ib_data->ib_qp->qp_num, (uint16_t)lid);

/* get the destination lid, port, and qpn from the
user*/ get_data_from_user(ib_data);
```

```
    if(ib_data->net_data.is_recv)
        { if(post_receive(ib_data))
                        fprintf(stderr, "create_qp: failed to allocate receive queue\n");
}

/* ---- Switch QP to recieve mode ------ */
/* config addresses and other stuff */
memset(&qp_attr,0, sizeof(struct ibv_qp_attr));
qp attr.qp state              = IBV QPS RTR;
qp_attr.ah_attr.dlid          = ib_data->net_data.dest_lid;
qp_attr.ah_attr.port_num= ib_data->net_data.local_port_num;
qp_attr.max_dest_rd_atomic    = 1;// ***
qp_attr.min_rnr_timer         = 12;// ***
qp_attr.path_mtu              = IBV_MTU_512;
qp_attr.dest_qp_num           = ib_data->net_data.dest_qpn;
qp_attr.rq_psn                = PSN;

qp_attr_mask =                IBV_QP_AV   |
   IBV_QP_STATE                  |
   IBV_QP_AV                     |
   IBV_QP_PATH_MTU               |
   IBV_QP_DEST_QPN               |
   IBV_QP_RQ_PSN                 |
   IBV_QP_MAX_DEST_RD_ATOMIC |
   IBV_QP_MIN_RNR_TIMER;

if(ret = ibv_modify_qp(ib_data->ib_qp, &qp_attr, qp_attr_mask))
{
   fprintf(stderr, "create_qp: failed to move QP to RTR with ibv_modify_qp with error
   %d\n",ret); goto func_return;
}

/* transmit side only */
if(! ib_data->net_data.is_recv)
{
   //memset(&qp_attr,0, sizeof(struct ibv_qp_attr));
   qp_attr.qp_state              = IBV_QPS_RTS;
   qp_attr.timeout               = 5;
   qp_attr.retry_cnt             = 5;
   qp_attr.rnr_retry             = 5;
   qp_attr.sq_psn                = PSN;
   qp_attr.max_rd_atomic         = 1;

   qp_attr_mask =          IBV_QP_STATE |
       IBV_QP_TIMEOUT|
       IBV_QP_RETRY_CNT|
       IBV_QP_RNR_RETRY|
       IBV_QP_SQ_PSN|
       IBV_QP_MAX_QP_RD_ATOMIC;

   if(ret = ibv_modify_qp(ib_data->ib_qp, &qp_attr, qp_attr_mask))
   {
       fprintf(stderr, "create_qp: failed to move QP to RTS with ibv_modify_qp with error %d\n",
       ret); goto func_return;
   }
```

```
    }

    func_ret = 0;

    func_return:
        return(func_ret);
}

int open_ib_structs(echo_server_db  *ib_data)
{
    int func_ret = -
    1; int ret;
    fprintf(stderr, "open_ib_structs: attempting to configure IB channel\n");

    /* Create a completion event channel */
    /*ib_data->ib_channel = ibv_create_comp_channel(ib_data-
    >ib_context); if(! ib_data->ib_channel) {
        fprintf(stderr, "open_ib_structs: failed in ibv_create_comp_channel\n");
        goto func_return;
    }
*/
    fprintf(stderr, "open_ib_structs: attempting to create and configure PD, pointer
    to CTX %ld\n", (unsigned long)ib_data->ib_context);
    /* Allocate a protection domain */ ib_data-
    >ib_pd = ibv_alloc_pd(ib_data->ib_context);
    if(!ib_data->ib_pd) {
        fprintf(stderr, "open_ib_structs: failed in
        ibv_alloc_pd\n"); goto func_return;
    }

    /* allocate memory page for future transactions
    */ ib_data->mem_lngth = getpagesize();
    fprintf(stderr, "open_ib_structs: attempting to allocate memory of size
    %d\n", ib_data->mem_lngth);
    ret = posix_memalign(&ib_data->mem_ptr, getpagesize(), ib_data-
    >mem_lngth); if(ret) {
        fprintf(stderr, "open_ib_structs: failed in
        posix_memalign\n"); goto func_return;
    }

    fprintf(stderr, "open_ib_structs: attempting to create and configure MR\n");

    /* ibv_reg_mr - Register a memory region */
    ib_data->ib_mem_region = ibv_reg_mr(ib_data->ib_pd, ib_data-
           >mem_ptr, ib_data->mem_lngth, IBV_ACCESS_LOCAL_WRITE);
    if(!ib_data->ib_mem_region) {
        fprintf(stderr, "open_ib_structs: failed in
        ibv_reg_mr\n"); goto func_return;
    }

    ib_data->ib_cq = ibv_create_cq(ib_data->ib_context, COMPLETE_QUEUE_LENGTH,
          NULL, ib_data->ib_channel, 0);
    if(!ib_data->ib_cq) {
```

```
        fprintf(stderr, "open_ib_structs: failed in ibv_create_cq\n");
        goto func_return;
    }

    fprintf(stderr, "open_ib_structs: attempting to create and configure QP\n");

    /* create the rcv and transmit QPs
    */ if(create_qp(ib_data)) {
        fprintf(stderr, "open_ib_structs: failed in create_qp\n");
        goto func_return;
    }

    func_ret = 0;

func_return:
    return(func_ret);
}

int get_open_ib_device(int dev_number, echo_server_db *ib_data)
{
    struct                          ibv_device **dev_list;
    struct                          ibv_device *dev_2_use = NULL;
    unsigned int                    dev_count;
    int                             i,ret_value = -1;
    const char                      *dev_name;

    ib_data->ib_device  = NULL;
    ib_data->ib_context = NULL;

    dev_list = ibv_get_device_list(&dev_count);
    if(!dev_list) {
                                    fprintf(stderr, "No devices
                                    found\n"); return(ret_value);
    }

    /* search for requested device in
    list*/ for(i=0; i<dev_count; i++) {
        dev_name = ibv_get_device_name(dev_list[i]);
        if(dev_name) {
            fprintf(stderr, "\nfound device %s", dev_name);
        }
        if(i == dev_number) { /* device found*/ ib_data-
            >ib_device = dev_list[i];
            fprintf(stderr, "  <- Device to be used in test");

            ib_data->ib_context = ibv_open_device(ib_data->ib_device);

            if(ib_data->ib_context)
            {
              fprintf(stderr, "\nOpened
              device"); ret_value = 0;
            }
            else
            {
              ib_data->ib_device = NULL;
```

```
            fprintf(stderr, "\nFailed to open device");
          }
        }
    }
    fprintf(stderr, "\n");

    if(!ib_data->ib_device) {
        fprintf(stderr, "No device allocated for test (requested dev num %d from %d
            devices)\n", dev_number, dev_count);
    }

    /* In both cases, where we find the device and open it, and where we do not find
    the device, we should release the list.*/
    /* will not be released due to ibv_open_device
    */ ibv_free_device_list(dev_list);
    return(ret_value);
}



int main()
{
    struct                          ibv_device *dev_2_use = NULL;
    unsigned int                    dev_count;
    int                             ret_value = -1;
    echo_server_db  ib_data;
    FILE                            *config_fd;

    memset(&ib_data, 0, sizeof(echo_server_db));

    config_fd = fopen("./echo_server_init.txt","rt");
    if(config_fd){
        fscanf(config_fd, "%d %d",
          &ib_data.net_data.local_dev_num,
          &ib_data.net_data.local_port_num);
        fclose(config_fd);
    };

    ret_value = get_open_ib_device(ib_data.net_data.local_dev_num,
    &ib_data); if(ret_value) {
        fprintf(stderr, "main: failed in
        get_open_ib_device\n"); goto exit_handler;
    }

    ret_value = open_ib_structs(&ib_data);
    if(ret_value) {
        fprintf(stderr, "main: failed in open_ib_structs\n");
        goto exit_handler;
    }

    /* handle receive and transmit */
    if(ib_data.net_data.is_recv) receive_data(&ib_data);
    else transmit_data(&ib_data);

  exit_handler:

    close_ib_device(&ib_data);
```

```
return(ret_value);
}
```

A.18 Echo_Server.h

```
#include <verbs.h>

#define PSN                     0x4321
#define COMPLETE_QUEUE_LENGTH100
#define MAX_SUPPORTED_STRING256
#define ECHO_SERVER_ID          0x1234

/* data needed to establish the connection */
/*typedef struct
{
   unsigned int             local_dev_num;
   unsigned int             local_port_num;
   unsigned int             local_lid;
   unsigned int             local_port;
   unsigned long            local_qpn;
   uint64_t                 local_mem_ptr;
   uint32_t                 local_rkey;

   unsigned int             dest_lid;
   unsigned int             dest_port;
   unsigned long            dest_qpn;
   uint64_t                 dest_mem_ptr;
   uint32_t                 dest_rkey;
   unsigned int             is_recv;
} network_data_t;
*/



typedef struct
{
   unsigned int             dev_num    ;
   unsigned int             lid;
   unsigned int             port_num    ;
   unsigned long            qpn;
   uint64_t                 mem_ptr;
   uint32_t                 rkey;
} port_data_t;

/* all the IB structures needed for the connection */
typedef struct
{
   struct ibv_device                *ib_device;
   struct ibv_context               *ib_context;
   struct ibv_comp_channel           *ib_channel;/* completion event channel */
   struct ibv_pd                    *ib_pd;/* protection domain */
   unsigned int                     mem_lngth;
   void                             *mem_ptr;
   struct ibv_mr                    *ib_mem_region;
   struct ibv_ah                    *ib_ah;
   struct ibv_cq                    *ib_cq;
   struct ibv_qp                    *ib_qp;
```

```
    port_data_t                 local_data;
    port_data_t                 remote_data;
    int             is_recv;

} echo_server_db;

int side_a(const port_data_t *local, port_data_t *remote,
                            const char *remote_name, const int portno);

int side_b(const port_data_t *local, port_data_t *remote,
                            const char *remote_name, const int portno);
```

A.19 Echo_Server_UD.c

```
#include <stdio.h>
#include <string.h>
#include <memory.h>
#include <verbs.h>

#define PSN                     0x4321
#define COMPLETE_QUEUE_LENGTH100
#define MAX_SUPPORTED_STRING256

#define TRANSACTION_QKEY      0x1234

/* data needed to establish the connection */
typedef struct
{
   unsigned int             local_dev_num;
   unsigned int             local_port_num;
   unsigned int             dest_lid;
   unsigned int             dest_port;
   unsigned long            dest_qpn;
   unsigned int             is_recv;
} network_data_t;

/* all the IB structures needed for the connection */
typedef struct
{
   struct ibv_device                *ib_device;
   struct ibv_context               *ib_context;
   struct ibv_comp_channel          *ib_channel;/* completion event channel */
   struct ibv_pd                    *ib_pd;/* protection domain */
   unsigned int                     mem_lngth;
   void                             *mem_ptr;
   struct ibv_mr                    *ib_mem_region;
   struct ibv_ah                    *ib_ah;
   struct ibv_cq                    *ib_cq;
   struct ibv_qp                    *ib_qp;
   network_data_t                   net_data;
} echo_server_db;


int close_ib_device(echo_server_db *ib_data)
```

```
{
    fprintf(stderr, "close_ib_device: calling ibv_destroy_qp\n");
    if(ib_data->ib_qp)                  ibv_destroy_qp(ib_data->ib_qp);
    fprintf(stderr, "close_ib_device: calling ibv_destroy_cq\n");
    if(ib_data->ib_cq)                  ibv_destroy_cq(ib_data->ib_cq);
    fprintf(stderr, "close_ib_device: calling ibv_dereg_mr\n");
    if(ib_data->ib_mem_region) ibv_dereg_mr(ib_data->ib_mem_region);
    fprintf(stderr, "close_ib_device: calling free\n");
    if(ib_data->mem_ptr)      free(ib_data->mem_ptr);
    fprintf(stderr, "close_ib_device: calling ibv_dealloc_pd\n");
    if(ib_data->ib_pd)     ibv_dealloc_pd(ib_data->ib_pd);
    fprintf(stderr, "close_ib_device: calling ibv_destroy_comp_channel\n");
    if(ib_data->ib_channel) ibv_destroy_comp_channel(ib_data->ib_channel);
    fprintf(stderr, "close_ib_device: calling ibv_close_device\n");
    if(ib_data->ib_context) ibv_close_device(ib_data->ib_context);
    fprintf(stderr, "finished closing device\n");

    return 0;
}

/* post a single recieve entry.*/
inline int post_receive_entry(struct ibv_qp *ib_qp, const uint32_t lkey,
                              char *mem_region, const uint32_t length)
{
    struct ibv_sge              mem_list;
    struct ibv_recv_wr          wr;
    struct ibv_recv_wr          *bad_wr;

    memset(&wr, 0,          sizeof(struct ibv_recv_wr));
    wr.sg_list              = &mem_list;
    wr.num_sge              = 1;
    wr.wr_id                = (uint64_t)mem_region;/* save address for receive in wr_id */

    mem_list.lkey           = lkey;
    mem_list.addr           = (uintptr_t)mem_region;
    mem_list.length         = length;

    return(ibv_post_recv(ib_qp, &wr, &bad_wr));
}

/* Initial posting of recive requests. More requests will be added */
/* as these requests will be consumed.                             */
int post_receive(echo_server_db  *ib_data)
{
    int                         i;
    char                    *mem_area;
    uint32_t                lkey = ib_data->ib_mem_region->lkey;

    for (i = 0; i < COMPLETE_QUEUE_LENGTH; ++i)
    {
        /* use cyclic buffer of length 7 on receive */
        mem_area = (char *)(((unsigned long)ib_data->mem_ptr) + ((i & 0x7)*
        MAX_SUPPORTED_STRING));
```

```
            if(post_receive_entry(ib_data->ib_qp, lkey, mem_area,
                                    MAX_SUPPORTED_STRING)) break;
    }

    if(i < COMPLETE_QUEUE_LENGTH-1) {
        fprintf(stderr,"receieve_data: failed to allocate receive queue using ibv_post_recv i=%d\n",i);
        return(-1);
    }

    fprintf(stderr,"receieve_data: Allocated receive WE using ibv_post_recv\n");

    return 0;
}

/* recive data and print the data received to the stderr */
/* the function assumes that all data in CQ was due to RCV and not to
*/ /* TX or errors. */
int receive_data(echo_server_db  *ib_data)
{
    struct ibv_wc       rcv_wc;
    int                         poll_ret, rx_length, recieve_count = 0;
    uint32_t                    lkey = ib_data->ib_mem_region->lkey;

    while(recieve_count <= 60) {
        poll_ret = ibv_poll_cq(ib_data->ib_cq, 1, &rcv_wc);
        if(poll_ret > 0) {
            if(rcv_wc.status != IBV_WC_SUCCESS)
                                    fprintf(stderr,"Recieve of message failed with error %d\n",rcv_wc.status);

            rx_length = rcv_wc.byte_len;
            /* fprintf(stderr,"length %d",rx_length); */
            fprintf(stderr, "%s ",((char *)rcv_wc.wr_id)+40);

            /* return the used entry to the receive queue so we don't run out of receives */
            if(post_receive_entry(ib_data->ib_qp, lkey, (char *)rcv_wc.wr_id, MAX_SUPPORTED_STRING)) {
                                fprintf(stderr, "receive_data: failed in post_receive_entry\n");
            }
            recieve_count++;
        }
        if(poll_ret < 0) fprintf(stderr, "receieve_data: Failed in RCV in ibv_poll_cq\n");
    }
    return 0;
}

/* transmit data received from the stdin and send it to the server */
/* the function assumes that all entries in the CQ are due to transmit
*/ /* and not to recieve. */
int transmit_data(echo_server_db  *ib_data)
{
    volatile int l;
    int                         transmit_count = 0;
    char                        *tmp_Buffer;
    struct  ibv_wc tx_wc;
```

```
int                             string_length;
int                             ret = -1;
struct                          ibv_send_wr *bad_wr;
struct ibv_sge list;
struct ibv_ah  *ah;
struct ibv_send_wr wr;
struct ibv_ah_attr ah_attr = {
   .dlid = ib_data->net_data.dest_lid,
   .port_num = ib_data->net_data.local_port_num
};

ah = ibv_create_ah(ib_data-
>ib_pd,&ah_attr); if(!ah) {
   fprintf(stderr,"transmit_data: failed to create address header\n");
   goto func_return;
}

memset(&wr, 0, sizeof(struct ibv_send_wr));
wr.sg_list    = &list;
wr.num_sge    = 1;
wr.opcode     = IBV_WR_SEND;
wr.send_flags = IBV_SEND_SIGNALED;
wr.wr.ud.remote_qpn  = ib_data->net_data.dest_qpn;
wr.wr.ud.remote_qkey = TRANSACTION_QKEY;
wr.wr.ud.ah          = ah;

list.lkey                       = ib_data->ib_mem_region->lkey;

while(transmit_count < 60) {

   /* small delay to prevent overruns in cyclic buffer
   */ for(l=0; l<1000; l++) ;

   /* use cyclic buffer of length 7 on receive */
   tmp_Buffer = (char *)(((unsigned long)ib_data->mem_ptr) +
                         (transmit_count & 0x7) * MAX_SUPPORTED_STRING);

   /* get data from user */
   memset(tmp_Buffer, 0, MAX_SUPPORTED_STRING);
   scanf("%s",tmp_Buffer);
   string_length = strlen(tmp_Buffer);

   list.addr                    = (uintptr_t)tmp_Buffer;
   list.length = string_length + 1;
   wr.wr_id                     = (uint64_t)tmp_Buffer;

   if(ret = ibv_post_send(ib_data->ib_qp, &wr,&bad_wr)) {
      fprintf(stderr,"transmit_data:          ibv_post_send
      failed\n"); goto func_return;
   }
   transmit_count ++;

   if(ret = ibv_poll_cq(ib_data->ib_cq, 1, &tx_wc) < 0) {
      fprintf(stderr,"transmit_data:          ibv_poll_cq
      failed\n"); return(ret);
   }
```

```
    }

    func_return:
        if(ah) ibv_destroy_ah(ah);
        return(ret);
}



    inline void get_data_from_user(echo_server_db  *ib_data)
    {
        fprintf(stderr, "Type 1 for server\n");
          scanf("%d",&ib_data->net_data.is_recv);

        if(! ib_data->net_data.is_recv) { printf("Type
            remote LID\n"); scanf("%d",&ib_data-
            >net_data.dest_lid); //printf("Type
            remote PORT\n");
            //scanf("%d",&ib_data->net_data.dest_port);
            printf("Type remote QP number\n");
            scanf("%d",&ib_data->net_data.dest_qpn);
    }
}

int create_qp(echo_server_db  *ib_data)
{
    int ret, func ret = -1;
    struct ibv_qp_init_attr       qp_init_attr;
    struct ibv_qp_attr                 qp_attr;
    enum   ibv_qp_attr_mask       qp_attr_mask;
    struct ibv_ah_attr                 dest_attr;
    unsigned long long                 guid;
    struct ibv_port_attr          port_attr;

    uint16_t                           lid;

    memset(&qp_init_attr,0,sizeof(struct ibv_qp_init_attr));
    qp_init_attr.send_cq          = ib_data->ib_cq;
    qp_init_attr.recv_cq          = ib_data->ib_cq;
    qp_init_attr.cap.max_inline_data= 1;/* max inline = 1 */
    qp_init_attr.cap.max_recv_sge = 1;/* do not use scatter gather on RX */
    qp_init_attr.cap.max_send_sge = 1;/* do not use scatter gather on TX */
    qp_init_attr.cap.max_recv_wr  = COMPLETE_QUEUE_LENGTH - 1;// smaller then CQ length
    qp_init_attr.cap.max_send_wr  = COMPLETE_QUEUE_LENGTH - 1;// smaller then CQ length
    qp_init_attr.qp_type          = IBV_QPT_UD;// connection type
    qp_init_attr.sq_sig_all       = 1;/* generate CQ for every WQE*/

    //qp_init_attr.xrc_domain     = NULL;// ??

    // ibv_create_qp - Create a queue pair.
    ib_data->ib_qp = ibv_create_qp(ib_data->ib_pd,
    &qp_init_attr); if(!ib_data->ib_qp) {
        fprintf(stderr, "create_qp: failed in
        ibv_create_qp\n"); goto func_return;
    }

    /* Switch QP from reset to init state. Also config no ATOMIC or RDMA

    */ /* and configured to work with physical port 1 */
```

```
memset((char *)&qp_attr,0, sizeof(struct ibv_qp_attr));
qp_attr.qp_state              = IBV_QPS_INIT;
qp_attr.port_num              = ib_data->net_data.local_port_num;// port number
qp_attr.pkey_index            = 0;
qp_attr.qkey                  = TRANSACTION_QKEY;
qp_attr_mask =                IBV_QP_STATE          |
                              IBV_QP_PORT           |
                              IBV_QP_PKEY_INDEX       |
                              IBV_QP_QKEY;
if(ret = ibv_modify_qp(ib_data->ib_qp, &qp_attr, qp_attr_mask)) {
   fprintf(stderr, "create_qp: failed to init QP with ibv_modify_qp error %d\n", ret);
   goto func_return;
}
// ---- query the port and QP for debug purposes ------
ibv_query_port(ib_data->ib_context, ib_data->net_data.local_port_num,
&port_attr); lid = port_attr.lid;
qp_attr_mask =                IBV_QP_PORT          |
                              IBV_QP_STATE             |
                              IBV_QP_AV            |
                              IBV_QP_PATH_MTU      |
                              IBV_QP_DEST_QPN      |
                              IBV_QP_RQ_PSN;
memset(&qp_attr,0, sizeof(struct ibv_qp_attr)); ibv_query_qp(ib_data-
>ib_qp, &qp_attr,qp_attr_mask,&qp_init_attr);
fprintf(stderr, "QP query: State %d, port %d, MTU %d, QPN %d lid %d
   \n",  qp_attr.qp_state, qp_attr.port_num, qp_attr.
   path_mtu, ib_data->ib_qp->qp_num, (uint16_t)lid);
/* get the destination lid, port, and qpn from the
user*/ get_data_from_user(ib_data);
if(ib_data->net_data.is_recv)
   {  if(post_receive(ib_data))
        fprintf(stderr, "create_qp: failed to allocate receive queue\n");
}
in the host                   /* config addresses and other stuff */
memset(&qp_attr,0, sizeof(struct ibv_qp_attr));
qp_attr.qp_state              = IBV_QPS_RTR;
qp_attr_mask =                IBV_QP_STATE ;
if(ret = ibv_modify_qp(ib_data->ib_qp, &qp_attr, qp_attr_mask))
{
   fprintf(stderr, "create_qp: failed to move QP to RTR with ibv_modify_qp with error
   %d\n",ret); goto func_return;
}
/* transmit side only */
if(! ib_data->net_data.is_recv)
{
   //memset(&qp_attr,0, sizeof(struct ibv_qp_attr));
   qp_attr.qp_state                 = IBV_QPS_RTS;
   qp_attr.sq_psn                     = PSN;
   qp_attr_mask =          IBV_QP_STATE            |
```

```
                                IBV_QP_SQ_PSN;

        if(ret = ibv_modify_qp(ib_data->ib_qp, &qp_attr, qp_attr_mask))
        {
            fprintf(stderr, "create_qp: failed to move QP to RTS with ibv_modify_qp with error %d\n",
            ret); goto func_return;
        }
    }

    func_ret = 0;

    func_return:
        return(func_ret);
}

int open_ib_structs(echo_server_db  *ib_data)
{
    int func_ret = -
    1; int ret;
    fprintf(stderr, "open_ib_structs: attempting to configure IB channel\n");

    /* Create a completion event channel */
    /*ib_data->ib_channel = ibv_create_comp_channel(ib_data-
    >ib_context); if(! ib_data->ib_channel) {
        fprintf(stderr, "open_ib_structs: failed in ibv_create_comp_channel\n");
        goto func_return;
    }
    */
    fprintf(stderr, "open_ib_structs: attempting to create and configure PD, pointer
    to CTX %ld\n", (unsigned long)ib_data->ib_context);
    /* Allocate a protection domain */
    ib_data->ib_pd = ibv_alloc_pd(ib_data-
    >ib_context); if(!ib_data->ib_pd) {
        fprintf(stderr, "open_ib_structs: failed in
        ibv_alloc_pd\n"); goto func_return;
    }

    /* allocate memory page for future transactions
    */ ib_data->mem_lngth = getpagesize();
    fprintf(stderr, "open_ib_structs: attempting to allocate memory of size
    %d\n", ib_data->mem_lngth);
    ret = posix_memalign(&ib_data->mem_ptr, getpagesize(), ib_data-
    >mem_lngth); if(ret) {
        fprintf(stderr, "open_ib_structs: failed in
        posix_memalign\n"); goto func_return;
    }

    fprintf(stderr, "open_ib_structs: attempting to create and configure MR\n");

    /* ibv_reg_mr - Register a memory region */
    ib_data->ib_mem_region = ibv_reg_mr(ib_data->ib_pd, ib_data->mem_ptr,
                                ib_data->mem_lngth, IBV_ACCESS_LOCAL_WRITE);
    if(!ib_data->ib_mem_region) {
        fprintf(stderr, "open_ib_structs: failed in ibv_reg_mr\n");
```

```
        goto func_return;
    }

    ib_data->ib_cq = ibv_create_cq(ib_data->ib_context, COMPLETE_QUEUE_LENGTH,
                                   NULL, ib_data->ib_channel, 0);
    if(!ib_data->ib_cq) {
        fprintf(stderr, "open_ib_structs: failed in ibv_create_cq\n");
        goto func_return;
    }

    fprintf(stderr, "open_ib_structs: attempting to create and configure QP\n");

    /* create the rcv and transmit QPs
    */ if(create_qp(ib_data)) {
        fprintf(stderr, "open_ib_structs: failed in create_qp\n");
        goto func_return;
    }

    func_ret = 0;

func_return:
    return(func_ret);
}

int get_open_ib_device(int dev_number, echo_server_db *ib_data)
{
    struct                      ibv_device **dev_list;
    struct                      ibv_device *dev_2_use = NULL;
    unsigned int                dev_count;
    int                         i,ret_value = -1;
    const char                  *dev_name;

    ib_data->ib_device  = NULL;
    ib_data->ib_context = NULL;

    dev_list = ibv_get_device_list(&dev_count);
    if(!dev_list) {
        fprintf(stderr, "No devices
        found\n"); return(ret_value);
    }

    /* search for requested device in
    list*/ for(i=0; i<dev_count; i++) {
        dev_name = ibv_get_device_name(dev_list[i]);
        if(dev_name) {
            fprintf(stderr, "\nfound device %s", dev_name);
        }
        if(i == dev_number) { /* device found*/ ib_data-
            >ib_device = dev_list[i];
            fprintf(stderr, "  <- Device to be used in test");

            ib_data->ib_context = ibv_open_device(ib_data->ib_device);

            if(ib_data->ib_context)
            {
```

```
            fprintf(stderr, "\nOpened
            device"); ret_value = 0;
          }
          else
          {
            ib_data->ib_device = NULL;
            fprintf(stderr, "\nFailed to open device");
          }
        }
    }
    fprintf(stderr, "\n");

    if(!ib_data->ib_device) {
                                fprintf(stderr, "No device allocated for
    test (requested dev num %d from %d devices)\n",
                                dev_number, dev_count);
    }

    /* In both cases, where we find the device and open it, and where we do not find
    the device, we should release the list. */
    /* will not be released due to ibv_open_device
    */ ibv_free_device_list(dev_list);
    return(ret_value);
}



int main()
{
    struct                      ibv_device *dev_2_use = NULL;
    unsigned int                dev_count;
    int                         ret_value = -1;
    echo_server_db  ib_data;
    FILE                        *config_fd;

    memset(&ib_data, 0, sizeof(echo_server_db));


    config_fd = fopen("./echo_server_init.txt","rt");
    if(config_fd){
        fscanf(config_fd, "%d %d",
              &ib_data.net_data.local_dev_num,
              &ib_data.net_data.local_port_num);
        fclose(config_fd);
    };

    ret_value = get_open_ib_device(ib_data.net_data.local_dev_num,
    &ib_data); if(ret_value) {
        fprintf(stderr, "main: failed in
        get_open_ib_device\n"); goto exit_handler;
    }

    ret_value = open_ib_structs(&ib_data);
    if(ret_value) {
        fprintf(stderr, "main: failed in open_ib_structs\n");
        goto exit_handler;
    }
```

```
   /* handle receive and transmit */
   if(ib_data.net_data.is_recv) receive_data(&ib_data);
   else transmit_data(&ib_data);

  exit_handler:
   close_ib_device(&ib_data);
   return(ret_value);
   }
```

A.20 Data Passing c

```
   #include <sys/types.h>
   #include <sys/socket.h>
   #include <netinet/in.h>
   #include <netdb.h>
   #include <stdio.h>
   #include <memory.h>
   #include "echo_server.h"
   #define BUFFER_SIZE 100

inline print_port_data(const port_data_t *to_print)
{
   fprintf(stderr,"dev num=%d, lid=%d, port num=%d, QPN=0x%x, mem=%lx,
         rkey=0x%x\n", to_print->dev_num, to_print->lid , to_print->port_num,
         to_print->qpn , to_print->mem_ptr, to_print->rkey );

}


/* should copy field by field using ntohll, htol (infiniband/arch.h) */

int side_a(const port_data_t *local, port_data_t
       *remote, const char *remote_name, const int portno)
{
   int sockfd,n;
      unsigned int packet_size, delay_on=0;

   struct  sockaddr_in  serv_addr;
   struct   hostent   *remote_addr;
   char buffer[BUFFER_SIZE];
      struct timeval start, stop;
      int server_addr_length = sizeof(struct sockaddr_in);

      packet_size = sizeof(port_data_t);

      fprintf(stderr, ">>> Local: "); print_port_data((port_data_t *)local);

      // open a socket
   sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0) {
```

```
            fprintf(stderr,"ERROR opening
            socket\n"); return(-1);
    }

    remote_addr = gethostbyname(remote_name);
    if (remote_addr == NULL) {
       fprintf(stderr,"ERROR, no such
       host\n"); return(-1);
    }

    /* configure comm header for socket */
    memset((char *) &serv_addr, 0,
    sizeof(serv_addr)); serv_addr.sin_family =
    AF_INET; serv_addr.sin_port = htons(portno);
       serv_addr.sin_addr.s_addr = INADDR_ANY;
       if (bind(sockfd,(struct sockaddr
           *)&serv_addr,server_addr_length)<0) error("binding");

       n = recvfrom(sockfd,buffer, BUFFER_SIZE,0,(struct sockaddr *)&serv_addr
       ,&server_addr_length); memcpy(remote,buffer,packet_size);

       fprintf(stderr, ">>> rcvd: "); print_port_data((port_data_t *)remote);

       sleep(1);

    serv_addr.sin_port = htons(portno+1);
       memcpy((char *)&serv_addr.sin_addr.s_addr,(char *)remote_addr->h_addr, remote_addr-
       >h_length); memcpy(buffer, local, packet_size);
       do {
           n=sendto(sockfd,buffer,packet_size,0,(struct sockaddr
       *)&serv_addr,sizeof(serv_addr)); } while (!n);

       close(sockfd);
    return 0;
}


int side_b(const port_data_t *local, port_data_t
        *remote,  const char *remote_name, const int portno)
{
    int sockfd_tx, sockfd_rx,n;
       unsigned int packet_size, delay_on=0;

    struct  sockaddr_in  serv_addr;
    struct   hostent   *remote_addr;
    char buffer[BUFFER_SIZE];
       struct timeval start, stop;
       int server_addr_length = sizeof(struct sockaddr_in);

       packet_size = sizeof(port_data_t);

       fprintf(stderr, ">>> Local: "); print_port_data((port_data_t *)local);

       // open a socket
```

```
    sockfd_tx = socket(AF_INET, SOCK_DGRAM,
    0); if (sockfd_tx < 0) {
          fprintf(stderr,"ERROR opening
          socket\n"); return(-1);
    }
    sockfd_rx = socket(AF_INET, SOCK_DGRAM,
    0); if (sockfd_rx < 0) {
          fprintf(stderr,"ERROR opening
          socket\n"); return(-1);
    }

    remote_addr = gethostbyname(remote_name);
    if (remote_addr == NULL) {
          fprintf(stderr,"ERROR, no such
          host\n"); return(-1);
    }

    /* configure comm header for socket */
    memset((char *) &serv_addr, 0,
    sizeof(serv_addr)); serv_addr.sin_family =
    AF_INET; serv_addr.sin_port = htons(portno);
    memcpy((char *)&serv_addr.sin_addr.s_addr,(char *)remote_addr->h_addr,
    remote_addr->h_length);

        memcpy(buffer, local,
        packet_size); do {
            n=sendto(sockfd_tx,buffer,packet_size,0,(struct sockaddr
        *)&serv_addr,sizeof(serv_addr)); } while (!n);

        /* configure comm header for socket */
    serv_addr.sin_port = htons(portno+1);
        serv_addr.sin_addr.s_addr = INADDR_ANY;
        if (bind(sockfd_rx,(struct sockaddr
            *)&serv_addr,server_addr_length)<0) error("binding");

        n = recvfrom(sockfd_rx, buffer, BUFFER_SIZE,0,(struct sockaddr *)&serv_addr
        ,&server_addr_length); memcpy(remote,buffer,packet_size);

        close(sockfd_tx);
        close(sockfd_rx);

        fprintf(stderr, ">>> rcvd: "); print_port_data(remote);

    return 0;
}
```

A.21 Echo Server.c.c

```
    #include <stdio.h>
    #include <string.h>
    #include <memory.h>
    #include <verbs.h>
    #define PSN                     0x4321
    #define COMPLETE_QUEUE_LENGTH100
```

```
#define MAX_SUPPORTED_STRING256

#define ECHO_SERVER_ID        0x1234

/* data needed to establish the connection */
typedef struct
{
   unsigned int             local_dev_num;
   unsigned int             local_port_num;
   unsigned int             dest_lid;
   unsigned int             dest_port;
   unsigned long            dest_qpn;
   unsigned int             is_recv;
} network_data_t;

/* all the IB structures needed for the connection */
typedef struct
{
   struct ibv_device                *ib_device;
   struct ibv_context               *ib_context;
   struct ibv_comp_channel *ib_channel;          /* completion event channel */
   struct ibv_pd                    *ib_pd;              /* protection domain */
   unsigned int                     mem_lngth;
   void                             *mem_ptr;
   struct ibv_mr                    *ib_mem_region;
   struct ibv_ah                    *ib_ah;
   struct ibv_cq                    *ib_cq;
   struct ibv_qp                    *ib_qp;
   network_data_t                   net_data;
} echo_server_db;


int close_ib_device(echo_server_db *ib_data)
{
   fprintf(stderr, "close_ib_device: calling ibv_destroy_qp\n");
   if(ib_data->ib_qp) ibv_destroy_qp(ib_data->ib_qp); fprintf(stderr,
   "close_ib_device: calling ibv_destroy_cq\n"); if(ib_data->ib_cq)
   ibv_destroy_cq(ib_data->ib_cq); fprintf(stderr, "close_ib_device:
   calling ibv_dereg_mr\n"); if(ib_data->ib_mem_region)
   ibv_dereg_mr(ib_data->ib_mem_region); fprintf(stderr,
   "close_ib_device: calling free\n"); if(ib_data->mem_ptr) free(ib_data-
   >mem_ptr);
   fprintf(stderr, "close_ib_device: calling ibv_dealloc_pd\n");
   if(ib_data->ib_pd) ibv_dealloc_pd(ib_data->ib_pd);
   fprintf(stderr, "close_ib_device: calling ibv_destroy_comp_channel\n");
   if(ib_data->ib_channel) ibv_destroy_comp_channel(ib_data->ib_channel);
   fprintf(stderr, "close_ib_device: calling ibv_close_device\n"); if(ib_data-
   >ib_context) ibv_close_device(ib_data->ib_context); fprintf(stderr,
   "finished closing device\n");

   return 0;
}

/* post a single recieve entry.*/
inline int post_receive_entry(struct ibv_qp *ib_qp, const uint32_t
                                    lkey,  char *mem_region, const uint32_t length)
{
```

```
    struct ibv_sge              mem_list;
    struct ibv_recv_wr          wr;
    struct ibv_recv_wr          *bad_wr;

    memset(&wr, 0, sizeof(struct ibv_recv_wr));
    wr.sg_list          = &mem_list;
    wr.num_sge          = 1;
    wr.wr_id            = (uint64_t)mem_region;/* save address for receive in wr_id */

    mem_list.lkey       = lkey;
    mem_list.addr       = (uintptr_t)mem_region;
    mem_list.length     = length;

    return(ibv_post_recv(ib_qp, &wr, &bad_wr));
}

/* Initial posting of recive requests. More requests will be added */
/* as these requests will be consumed.                             */
int post_receive(echo_server_db  *ib_data)
{
    int                     i;
    char                *mem_area;
    uint32_t            lkey = ib_data->ib_mem_region->lkey;

    for (i = 0; i < COMPLETE_QUEUE_LENGTH; ++i)
    {
        /* use double buffer */
        mem_area = (char *)(((unsigned long)ib_data->mem_ptr) + ((i & 1)* MAX_SUPPORTED_STRING));

        if(post_receive_entry(ib_data->ib_qp, lkey, mem_area,
                        MAX_SUPPORTED_STRING)) break;
    }

    if(i < COMPLETE_QUEUE_LENGTH-1) {
        fprintf(stderr,"receieve_data: failed to allocate receive queue using ibv_post_recv
        i=%d\n",i); return(-1);
    }

    fprintf(stderr,"receieve_data: Allocated receive WE using ibv_post_recv\n");

    return 0;
}

/* Recive data and print the data received to the stderr */
/* The function assumes that all data in CQ was due to RCV and not to */
/* TX or errors. */
int receive_data(echo_server_db  *ib_data)
{
    struct ibv_wc           rcv_wc;
    int                             poll_ret, rx_length, recieve_count = 0;
    uint32_t                        lkey = ib_data->ib_mem_region->lkey;

    while(recieve_count <= 60) {
        poll_ret = ibv_poll_cq(ib_data->ib_cq, 1, &rcv_wc);
        if(poll_ret > 0) {
                if(rcv_wc.status != IBV_WC_SUCCESS)
```

```
                        fprintf(stderr,"Recieve of message failed with error %d\n",rcv_wc.status);

                    rx_length = rcv_wc.byte_len; fprintf(stderr,
                    "%s ",(char *)rcv_wc.wr_id);

                    /* return the used entry to the receive queue so we don't run out of receives */
                    if(post_receive_entry(ib_data->ib_qp, lkey, (char *)rcv_wc.wr_id,
                    MAX_SUPPORTED_STRING)) {
                        fprintf(stderr, "receive_data: failed
                     in post_receive_entry\n");
                    }
                    recieve_count++;
            }
            if(poll_ret < 0) fprintf(stderr, "receieve_data: Failed in RCV in ibv_poll_cq\n");
        }
        return 0;
    }

    /* Transmit data received from the stdin and send it to the server. */
    /* The function assumes that all entries in the CQ are due to transmit
    */ /* and not to recieve. */
    int transmit_data(echo_server_db  *ib_data)
    {
        int             transmit_count = 0;
        int             cq_count = 0;
        char            *tmp_Buffer;
        struct ibv_wc tx_wc;
        int             string_length;
        int             ret;
        struct          ibv_send_wr *bad_wr;
        struct ibv_sge list;
        struct ibv_send_wr wr = {
            .wr_id          = ECHO_SERVER_ID,
            .sg_list        = &list,
            .num_sge        = 1,
            .opcode         = IBV_WR_SEND,
            .send_flags     = IBV_SEND_SIGNALED,
        };

        list.lkey = ib_data->ib_mem_region->lkey;

        while(transmit_count < 60) {
            /* use double buffer on receive */
            tmp_Buffer = (char *)(((unsigned long)ib_data->mem_ptr) +
                                  (transmit_count & 1) * MAX_SUPPORTED_STRING);

            /* get data from user */
            memset(tmp_Buffer, 0, MAX_SUPPORTED_STRING);
            scanf("%s",tmp_Buffer);
            string_length = strlen(tmp_Buffer);

            list.addr = (uintptr_t)tmp_Buffer;
            list.length = string_length + 1;

            if(ret = ibv_post_send(ib_data->ib_qp, &wr,&bad_wr)) {
```

```
                        fprintf(stderr,"transmit_data: ibv_post_send
                        failed\n"); return(ret);
        }

        /* increment and wrap around
        */ transmit_count ++;

        /* we must empty the completion queue even on the transmit
        */ do {
                        if(ret = ibv_poll_cq(ib_data->ib_cq, 1, &tx_wc) < 0) {
                                fprintf(stderr,"transmit_data: ibv_poll_cq
                                failed\n"); return(ret);
                        }
                        cq_count += ret;
        } while (cq_count < transmit_count);
    }
}



inline void get_data_from_user(echo_server_db  *ib_data)
{
    fprintf(stderr, "Type 1 for server\n");
      scanf("%d",&ib_data->net_data.is_recv);
    printf("Type remote LID\n"); scanf("%d",&ib_data-
      >net_data.dest_lid);
    printf("Type remote PORT\n"); scanf("%d",&ib_data-
      >net_data.dest_port);
    printf("Type remote QP number\n");
      scanf("%d",&ib_data->net_data.dest_qpn);
}
int create_qp(echo_server_db  *ib_data)
{
    int ret, func_ret = -1;
    struct ibv_qp_init_attr   qp_init_attr;
    struct ibv_qp_attr                  qp_attr;
    enum  ibv_qp_attr_maskqp_attr_mask;
    struct ibv_ah_attr                  dest_attr;
    unsigned long long                                  guid;
    struct ibv_port_attr                port_attr;
    uint16_t                                            lid;

    qp_init_attr.qp_context             = NULL;         // ??
    qp_init_attr.send_cq                = ib_data->ib_cq;
    qp_init_attr.recv_cq                = ib_data->ib_cq;
    qp_init_attr.srq                    = NULL;
    qp_init_attr.cap.max_inline_data= 1;    /* max inline = 1 */
    qp_init_attr.cap.max_recv_sge       = 1;            /* do not use scatter gather on RX */
    qp_init_attr.cap.max_send_sge       = 1;            /* do not use scatter gather on TX */
    qp_init_attr.cap.max_recv_wr        = COMPLETE_QUEUE_LENGTH - 1;// smaller then CQ length
    qp_init_attr.cap.max_send_wr        = COMPLETE_QUEUE_LENGTH - 1;// smaller then CQ length
    qp_init_attr.qp_type                = IBV_QPT_RC;       // connection type
    qp_init_attr.sq_sig_all             = 1; /* generate CQ for every WQE*/
    //qp_init_attr.xrc_domain           = NULL;         // ??
```

```
// ibv_create_qp - Create a queue pair.
ib_data->ib_qp = ibv_create_qp(ib_data->ib_pd,
&qp_init_attr); if(!ib_data->ib_qp) {
      fprintf(stderr, "create_qp: failed in
      ibv_create_qp\n"); goto func_return;
}

/* Switch QP from reset to init state. Also config no ATOMIC or RDMA
*/ /* and configured to work with physical port 1 */
memset((char *)&qp_attr,0, sizeof(struct ibv_qp_attr));
qp attr.qp state              = IBV QPS INIT;
qp_attr.port_num              = ib_data->net_data.local_port_num;// port number
qp_attr.pkey_index            =  0;
qp_attr.qp_access_flags = 0;  /*  */
qp_attr_mask =      IBV_QP_STATE  |
                  IBV_QP_PORT |
                  IBV_QP_PKEY_INDEX |
                  IBV_QP_ACCESS_FLAGS;
if(ret = ibv_modify_qp(ib_data->ib_qp, &qp_attr, qp_attr_mask)) {
      fprintf(stderr, "create_qp: failed to init QP with ibv_modify_qp error %d\n", ret);
      goto func_return;
}

// ---- query the port and QP for debug purposes ------
ibv_query_port(ib_data->ib_context, ib_data->net_data.local_port_num,
&port_attr); lid = port_attr.lid;
qp_attr_mask =      IBV_QP_PORT|
                  IBV_QP_STATE      |
                  IBV_QP_AV |
                  IBV_QP_PATH_MTU |
                  IBV_QP_DEST_QPN |
                  IBV_QP_RQ_PSN;
memset(&qp_attr,0, sizeof(struct ibv_qp_attr)); ibv_query_qp(ib_data-
>ib_qp, &qp_attr,qp_attr_mask,&qp_init_attr);
fprintf(stderr, "QP query: State %d, port %d, MTU %d, QPN %d lid %d
      \n", qp_attr.qp_state, qp_attr.port_num, qp_attr.
      path_mtu, ib_data->ib_qp->qp_num, (uint16_t)lid);

/* get the destination lid, port, and qpn from the
user*/ get_data_from_user(ib_data);
if(ib_data->net_data.is_recv)
      {  if(post_receive(ib_data))
                  fprintf(stderr, "create_qp: failed to allocate receive queue\n");
}

in the host                   /* config addresses and other stuff */
memset(&qp_attr,0, sizeof(struct ibv_qp_attr));
qp_attr.qp_state              = IBV_QPS_RTR;
qp_attr.ah_attr.dlid          = ib_data->net_data.dest_lid;
qp_attr.ah_attr.port_num= ib_data->net_data.local_port_num;
qp_attr.max_dest_rd_atomic    = 1;// ***
qp_attr.min_rnr_timer         = 12;// ***
```

```
    qp_attr.path_mtu              = IBV_MTU_512;
    qp_attr.dest_qp_num           = ib_data->net_data.dest_qpn;
    qp_attr.rq_psn                = PSN;

    qp_attr_mask =      IBV_QP_AV    |
          IBV_QP_STATE                    |
          IBV_QP_AV                       |
          IBV_QP_PATH_MTU                 |
          IBV_QP_DEST_QPN                 |
          IBV_QP_RQ_PSN                   |
          IBV_QP_MAX_DEST_RD_ATOMIC  |
          IBV_QP_MIN_RNR_TIMER;

    if(ret = ibv_modify_qp(ib_data->ib_qp, &qp_attr, qp_attr_mask))
    {
          fprintf(stderr, "create_qp: failed to move QP to RTR with ibv_modify_qp with error
          %d\n",ret); goto func_return;
    }

    /* transmit side only */
    if(! ib_data->net_data.is_recv)
    {
       //memset(&qp_attr,0, sizeof(struct ibv_qp_attr));
          qp_attr.qp_state            = IBV_QPS_RTS;
          qp_attr.timeout             = 5;
          qp_attr.retry_cnt           = 5;
          qp_attr.rnr_retry           = 5;
          qp_attr.sq_psn              = PSN;
          qp_attr.max_rd_atomic   = 1;

          qp_attr_mask = IBV_QP_STATE        |
              IBV_QP_TIMEOUT|
              IBV_QP_RETRY_CNT|
              IBV_QP_RNR_RETRY|
              IBV_QP_SQ_PSN        |
              IBV_QP_MAX_QP_RD_ATOMIC;

          if(ret = ibv_modify_qp(ib_data->ib_qp, &qp_attr, qp_attr_mask))
              {
              fprintf(stderr, "create_qp: failed to move QP to RTS with ibv_modify_qp with error %d\n",
              ret); goto func_return;
              }
    }

    func_ret = 0;

    func_return:
          return(func_ret);
}

    int open_ib_structs(echo_server_db  *ib_data)
    {
    int func_ret = -
    1; int ret;
    fprintf(stderr, "open_ib_structs: attempting to configure IB channel\n");
```

```
    /* Create a completion event channel */
    /*ib_data->ib_channel = ibv_create_comp_channel(ib_data-
    >ib_context); if(! ib_data->ib_channel) {
          fprintf(stderr, "open_ib_structs: failed in
          ibv_create_comp_channel\n"); goto func_return;
    }
    */
    fprintf(stderr, "open_ib_structs: attempting to create and configure PD, pointer
    to CTX %ld\n", (unsigned long)ib_data->ib_context);

    /* Allocate a protection domain */
    ib_data->ib_pd = ibv_alloc_pd(ib_data-
    >ib_context); if(!ib_data->ib_pd) {
          fprintf(stderr, "open_ib_structs: failed in
          ibv_alloc_pd\n"); goto func_return;
    }

    /* allocate memory page for future transactions
    */ ib_data->mem_lngth = getpagesize();
    fprintf(stderr, "open_ib_structs: attempting to allocate memory of size
    %d\n", ib_data->mem_lngth);
    ret = posix_memalign(&ib_data->mem_ptr, getpagesize(), ib_data-
    >mem_lngth); if(ret) {
          fprintf(stderr, "open_ib_structs: failed in
          posix_memalign\n"); goto func_return;
    }

    fprintf(stderr, "open_ib_structs: attempting to create and configure MR\n");

    /* ibv_reg_mr - Register a memory region */
    ib_data->ib_mem_region = ibv_reg_mr(ib_data->ib_pd, ib_data-
          >mem_ptr,  ib_data->mem_lngth, IBV_ACCESS_LOCAL_WRITE);
    if(!ib_data->ib_mem_region) {
          fprintf(stderr, "open_ib_structs: failed in
          ibv_reg_mr\n"); goto func_return;
    }

    ib_data->ib_cq = ibv_create_cq(ib_data->ib_context, COMPLETE_QUEUE_LENGTH,
                                          NULL, ib_data->ib_channel, 0);
    if(!ib_data->ib_cq) {
          fprintf(stderr, "open_ib_structs: failed in
          ibv_create_cq\n"); goto func_return;
    }

    fprintf(stderr, "open_ib_structs: attempting to create and configure QP\n");

    /* create the rcv and transmit QPs
    */ if(create_qp(ib_data)) {
          fprintf(stderr, "open_ib_structs: failed in
          create_qp\n"); goto func_return;
    }

    func_ret = 0;

func_return:

    return(func_ret);
```

```
}

int get_open_ib_device(int dev_number, echo_server_db *ib_data)
{
    struct                          ibv_device **dev_list;
    struct                          ibv_device *dev_2_use = NULL;
    unsigned int            dev_count;
    int                             i,ret_value = -1;
    const char              *dev_name;

    ib_data->ib_device = NULL;
    ib_data->ib_context = NULL;
    dev_list = ibv_get_device_list(&dev_count);
    if(!dev_list) {
          fprintf(stderr, "No devices
          found\n"); return(ret_value);
    }

    /* search for requested device in
    list*/ for(i=0; i<dev_count; i++) {
       dev_name = ibv_get_device_name(dev_list[i]);
       if(dev_name) {
          fprintf(stderr, "\nfound device %s", dev_name);
       }
       if(i == dev_number) { /* device found*/ ib_data-
          >ib_device = dev_list[i];
          fprintf(stderr, "  <- Device to be used in test");

          ib_data->ib_context = ibv_open_device(ib_data->ib_device);

          if(ib_data->ib_context)
          {
                          fprintf(stderr, "\nOpened
                          device"); ret_value = 0;
          }
          else
          {
                          ib_data->ib_device = NULL;
                          fprintf(stderr, "\nFailed to open device");
          }
       }
    }
    fprintf(stderr, "\n");

    if(!ib_data->ib_device) {
       fprintf(stderr, "No device allocated for test (requested dev num %d from %d
          devices)\n", dev_number, dev_count);
    }

    /* In both cases, where we find the device and open it, and where we do not find
    the device, we should release the list.*/
    /* will not be released due to ibv_open_device
    */ ibv_free_device_list(dev_list);
    return(ret_value);
```

```
}


int main()
{
    struct                          ibv_device *dev_2_use = NULL;
    unsigned int           dev_count;
    int                             ret_value = -1;
    echo_server_db  ib_data;
    FILE                            *config_fd;

    memset(&ib_data, 0, sizeof(echo_server_db));

    config_fd = fopen("./echo_server_init.txt","rt");
    if(config_fd){
        fscanf(config_fd, "%d %d",
              &ib_data.net_data.local_dev_num,
              &ib_data.net_data.local_port_num);
        fclose(config_fd);
    };

    ret_value = get_open_ib_device(ib_data.net_data.local_dev_num,
    &ib_data); if(ret_value) {
        fprintf(stderr, "main: failed in
        get_open_ib_device\n"); goto exit_handler;
    }

    ret_value = open_ib_structs(&ib_data);
    if(ret_value) {
        fprintf(stderr, "main: failed in open_ib_structs\n");
        goto exit_handler;
    }

    /* handle receive and transmit */
    if(ib_data.net_data.is_recv) receive_data(&ib_data);
    else transmit_data(&ib_data);

      exit_handler:
    close_ib_device(&ib_data);
    return(ret_value);
}
```

A.22 Echo Server RDMA.c

```
#include <stdio.h>
#include <string.h>
#include <memory.h>
#include <infiniband/arch.h>
#include "echo_server.h"
#define USE_RCQ 1

int close_ib_device(echo_server_db *ib_data)
{
```

```
        fprintf(stderr, "close_ib_device: calling ibv_destroy_qp\n");
        if(ib_data->ib_qp)                      ibv_destroy_qp(ib_data->ib_qp);
        fprintf(stderr, "close_ib_device: calling ibv_destroy_cq\n");
        if(ib_data->ib_cq)                      ibv_destroy_cq(ib_data->ib_cq);
        fprintf(stderr, "close_ib_device: calling ibv_dereg_mr\n");
        if(ib_data->ib_mem_region)              ibv_dereg_mr(ib_data->ib_mem_region);
        fprintf(stderr, "close_ib_device: calling free\n");
        if(ib_data->mem_ptr)                    free(ib_data->mem_ptr);
        fprintf(stderr, "close_ib_device: calling ibv_dealloc_pd\n");
        if(ib_data->ib_pd)                      ibv_dealloc_pd(ib_data->ib_pd);
        fprintf(stderr, "close_ib_device: calling ibv_destroy_comp_channel\n");
        if(ib_data->ib_channel)                 ibv_destroy_comp_channel(ib_data->ib_channel);
        fprintf(stderr, "close_ib_device: calling ibv_close_device\n");
        if(ib_data->ib_context)                 ibv_close_device(ib_data->ib_context);
        fprintf(stderr, "finished closing device\n");

        return 0;
}


/* post a single recieve
entry.*/ #if USE_RCQ
inline int post_receive_entry(struct ibv_qp *ib_qp, const uint32_t
           lkey, char *mem_region, const uint32_t length)
{
        struct ibv_sge          mem_list;
        struct ibv_recv_wr      wr;
        struct ibv_recv_wr      *bad_wr;

        memset(&wr, 0, sizeof(wr));
        wr.sg_list              = &mem_list;
        wr.num_sge              = 1;
        wr.wr_id                = (uintptr_t)mem_region;      /* save address for receive in wr_id */

        memset(&mem_list, 0, sizeof             (mem_list));
        mem_list.lkey           = lkey;
        mem_list.addr           = (uintptr_t)     mem_region;
        mem_list.length         = length;

        return(ibv_post_recv(ib_qp, &wr, &bad_wr));
}

/* Initial posting of recive requests. More requests will be added */
/* as these requests will be consumed.                              */
int post_receive(echo_server_db  *ib_data)
{
        int                             i;
        char                    *mem_area;
        uint32_t                lkey = ib_data->ib_mem_region->lkey;

        for (i = 0; i < COMPLETE_QUEUE_LENGTH; ++i)
```

```
        {
            mem_area = (char *)(((unsigned long)ib_data->mem_ptr + getpagesize()) + (i *
            MAX_SUPPORTED_STRING));


    if(post_receive_entry(ib_data->ib_qp, lkey, mem_area, MAX_SUPPORTED_STRING))
                                { printf("Failed to post RR\n");
                                break;
            }
        }

        if(i < COMPLETE_QUEUE_LENGTH-1) {
            fprintf(stderr,"receieve_data: failed to allocate receive queue using ibv_post_recv
            i=%d\n",i); return(-1);
        }

        fprintf(stderr,"receieve_data: Allocated receive WE using ibv_post_recv\n");

        return 0;
}

#endif

/* Recive data and print the data received to the stderr. */
/* The function assumes that all data in CQ was due to RCV and not to */
/* TX or errors. */
int receive_data(echo_server_db  *ib_data)
{
        struct ibv_wc      rcv_wc;
        int                poll_ret, rx_length, recieve_count = 0;
        uint32_t           lkey = ib_data->ib_mem_region->lkey;
        char               *input_buff;
        char               *print_buff;


        while(recieve_count < 30) {

            input_buff = (char *)((uintptr_t)ib_data->mem_ptr) +
                                        ((recieve_count & 1) * MAX_SUPPORTED_STRING);

            /* if the transmitter signals us, we can poll the CQ */
            poll_ret = ibv_poll_cq(ib_data->ib_cq, 1, &rcv_wc);
            if(poll_ret > 0) {
                    if(rcv_wc.status != IBV_WC_SUCCESS)
                            fprintf(stderr,"Recieve of message failed with error %d\n",rcv_wc.status);

                    rx_length = rcv_wc.byte_len;
                    fprintf(stderr, "%s ", input_buff);

                    recieve_count++;
            }
            else
            {
```

```
                    /* if the transmitter does not use signal, look for changes in the
                    data.*/ if(*input_buff != '\0') {
                        fprintf(stderr, "%s ", input_buff);
                        memset(input_buff, 0,
                        MAX_SUPPORTED_STRING); recieve_count++;
            }
        }

        if(poll_ret < 0) fprintf(stderr, "receieve_data: Failed in RCV in ibv_poll_cq\n");
    }

    fprintf(stderr,"\nFinished receiving %d messages\n", recieve_count);

    return 0;
}


/* Transmit data received from the stdin and send it to the server. */
/* The function assumes that all entries in the CQ are due to transmit
*/ /* and not to recieve. */
int transmit_data(echo_server_db  *ib_data)
{
    int             transmit_count = 0;
    char            *tmp_Buffer;
    uint64_t remote_buffer;
    struct  ibv_wc tx_wc;
    int             string_length;
    int             ret;
    struct  ibv_qp_init_attr      qp_init_attr;
    struct  ibv_qp_attr           qp_attr;
    enum    ibv_qp_attr_mask      qp_attr_mask;
    struct          ibv_send_wr *bad_wr;
    struct ibv_sge list;
    struct ibv_send_wr wr;

    list.lkey = ib_data->ib_mem_region->lkey;

    memset(&wr, 0, sizeof(struct ibv_send_wr));

    wr.sg list                    = &list;
    wr.num_sge                    = 1;
    wr.opcode        = IBV_WR_RDMA_WRITE;    /* IBV_WR_RDMA_WRITE_WITH_IMM */
    wr.send_flags    = 0;                    /* don't create a local signal. */
    wr.wr.rdma.rkey = ib_data->remote_data.rkey;

    while(transmit_count < 30) {

        tmp_Buffer = (char *)(((uintptr_t)(ib_data->mem_ptr)) +
            ((transmit_count & 1) * MAX_SUPPORTED_STRING));
        memset(tmp_Buffer, 0, MAX_SUPPORTED_STRING);

        wr.wr_id = (uintptr_t)tmp_Buffer;
        wr.wr.rdma.remote_addr = ((uintptr_t)(ib_data->remote_data.mem_ptr))
            + ((transmit_count & 1) * MAX_SUPPORTED_STRING);

        scanf("%s", tmp_Buffer);
```

```
        string_length = strlen(tmp_Buffer);

        list.addr = (uintptr_t)tmp_Buffer;
        list.length = string_length + 1;

        if(ret = ibv_post_send(ib_data->ib_qp, &wr,&bad_wr)) {
            fprintf(stderr,"transmit_data:              ibv_post_send
            failed\n"); return(ret);
        }

        qp_attr_mask = IBV_QP_STATE ;
        memset(&qp_attr,0, sizeof(struct ibv_qp_attr));
        if (ret = ibv_query_qp(ib_data->ib_qp, &qp_attr,qp_attr_mask,&qp_init_attr))
            { fprintf(stderr, "failed to query QP state\n");
            return(ret);
        }

        if(qp_attr.qp_state != IBV_QPS_RTS) fprintf(stderr, "TX QP switched to state %d\n", qp_attr.qp_state);

        transmit_count ++;
    }
    fprintf(stderr,"\nFinished transmitting %d messages\n", transmit_count);

}

int create_qp(echo_server_db    *ib_data)
{
    int ret, func_ret = -1;
    struct  ibv_qp_init_attr       qp_init_attr;
    struct ibv_qp_attr             qp_attr;
    enum ibv_qp_attr_mask          qp_attr_mask;
    struct ibv_ah_attr             dest_attr;
    unsigned long long             guid;
    struct ibv_port_attr           port_attr;
    uint16_t                       lid;

    memset(&qp_init_attr, 0, sizeof(struct ibv_qp_init_attr));
    qp_init_attr.send_cq          = ib_data->ib_cq;
    qp_init_attr.recv_cq          = ib_data->ib_cq;
    qp_init_attr.cap.max_inline_data= 1;/* don't use inline data */
    qp_init_attr.cap.max_send_sge = 1;/* don't use scatter gather on transmit */
    qp_init_attr.cap.max_send_wr  = COMPLETE_QUEUE_LENGTH - 1;// smaller then CQ length
#ifdef USE_RCQ
    qp_init_attr.cap.max_recv_sge = 1; /* we will not post receives */
    qp_init_attr.cap.max_recv_wr  = COMPLETE_QUEUE_LENGTH - 1; /* we will not post
    receives */
#else
    qp_init_attr.cap.max_recv_sge = 0; /* we will not post receives */
    qp_init_attr.cap.max_recv_wr  = 0; /* we will not post receives */
#endif
    qp_init_attr.qp_type          = IBV_QPT_RC;// connection type
    qp_init_attr.sq_sig_all       = 0;/* don't create a local signal. */
    //qp_init_attr.xrc_domain     = NULL;// ??
```

```
    // ibv_create_qp - Create a queue pair.
    ib_data->ib_qp = ibv_create_qp(ib_data->ib_pd,
    &qp_init_attr); if(!ib_data->ib_qp) {
       fprintf(stderr, "create_qp: failed in
       ibv_create_qp\n"); goto func_return;
    }

    /* Switch QP from reset to init state. Also config no ATOMIC or RDMA
    */ /* and configure to work with physical port 1 */
    memset((char *)&qp_attr,0, sizeof(struct ibv_qp_attr));
    qp attr.qp state              = IBV QPS INIT;
    qp_attr.port_num              = ib_data->local_data.port_num;// port number
    qp_attr.pkey_index            = 0;
    qp_attr.qp_access_flags = IBV_ACCESS_REMOTE_WRITE | IBV_ACCESS_REMOTE_READ;


    qp_attr_mask =                IBV_QP_STATE              |
                                  IBV_QP_PORT                |
                                  IBV_QP_PKEY_INDEX          |
                                  IBV_QP_ACCESS_FLAGS;
    if(ret = ibv_modify_qp(ib_data->ib_qp, &qp_attr, qp_attr_mask))
       {  fprintf(stderr, "create_qp: failed to init QP with ibv_modify_qp error %d\n", ret);
       goto func_return;
    }

    // ---- query the port and QP for debug purposes ------
    ibv_query_port(ib_data->ib_context, ib_data->local_data.port_num,
    &port_attr); lid = port_attr.lid;

    qp_attr_mask =                IBV_QP_PORT               |
                                  IBV_QP_STATE              |
                                  IBV_QP_AV                 |
                                  IBV_QP_PATH_MTU           |
                                  IBV_QP_DEST_QPN           |
                                  IBV_QP_RQ_PSN;
    memset(&qp_attr,0, sizeof(struct ibv_qp_attr)); ibv_query_qp(ib_data-
    >ib_qp, &qp_attr,qp_attr_mask,&qp_init_attr);
    ib_data->local_data.lid = lid; ib_data-
    >local_data.port_num = qp_attr.port_num; ib_data-
    >local_data.qpn = ib_data->ib_qp->qp_num; ib_data-
    >local_data.mem_ptr = (uint64_t)ib_data->mem_ptr;
    ib_data->local_data.rkey = (uint32_t)ib_data->ib_mem_region->rkey;


    /* get the destination lid, port, and qpn from the
    user*/ if(ib_data->is_recv) {
       side_a(&ib_data->local_data,&ib_data->remote_data, "10.4.3.113", 2017);
    }
    else {
       side_b(&ib_data->local_data,&ib_data->remote_data, "10.4.3.112", 2017);
    }

#ifdef USE_RCQ
    if(ib_data->is_recv) {
          if(post_receive(ib_data))
                        fprintf(stderr, "create_qp: failed to allocate receive queue\n");
```

```
    }

#endif

    /in the host                  /* config addresses and other stuff */
    memset(&qp_attr,0, sizeof(struct ibv_qp_attr));
    qp_attr.qp_state              = IBV_QPS_RTR;
    qp_attr.ah_attr.dlid          = ib_data->remote_data.lid;
    qp_attr.ah_attr.port_num= ib_data->local_data.port_num;
    qp_attr.max_dest_rd_atomic    = 1;// ***
    qp_attr.min_rnr_timer         = 12;// ***
    qp_attr.path_mtu              = IBV_MTU_512;
    qp_attr.dest_qp_num           = ib_data->remote_data.qpn;
    qp_attr.rq_psn                = PSN;

    qp_attr_mask =                IBV_QP_AV     |
          IBV_QP_STATE|
          IBV_QP_AV  |
          IBV_QP_PATH_MTU|
          IBV_QP_DEST_QPN|
          IBV_QP_RQ_PSN|
          IBV_QP_MAX_DEST_RD_ATOMIC|
          IBV_QP_MIN_RNR_TIMER;

    if(ret = ibv_modify_qp(ib_data->ib_qp, &qp_attr, qp_attr_mask))
    {
          fprintf(stderr, "create_qp: failed to move QP to RTR with ibv_modify_qp with error
          %d\n",ret); goto func_return;
    }

    /* transmit side only */
    if(! ib_data->is_recv)
    {
       //memset(&qp_attr,0, sizeof(struct ibv_qp_attr));
       qp_attr.qp_state          = IBV_QPS_RTS;
       qp_attr.timeout                   = 5;
       qp_attr.retry_cnt         = 5;
       qp_attr.rnr_retry         = 5;
       qp_attr.sq_psn                    = PSN;
       qp_attr.max_rd_atomic  = 1;

       qp_attr_mask = IBV_QP_STATE |
          IBV_QP_TIMEOUT|
          IBV_QP_RETRY_CNT|
          IBV_QP_RNR_RETRY|
          IBV_QP_SQ_PSN |
          IBV_QP_MAX_QP_RD_ATOMIC;

       if(ret = ibv_modify_qp(ib_data->ib_qp, &qp_attr, qp_attr_mask))
       {
          fprintf(stderr, "create_qp: failed to move QP to RTS with ibv_modify_qp with error %d\n",
          ret); goto func_return;
       }
    }

    func_ret = 0;
```

```
    func_return:
          return(func_ret);
}

    int open_ib_structs(echo_server_db  *ib_data)
    {
    int func_ret = -
    1; int ret;
    int mem_access_mask = 0;

    fprintf(stderr, "open_ib_structs: attempting to configure IB channel\n");

    /* Create a completion event channel */
    /*ib_data->ib_channel = ibv_create_comp_channel(ib_data-
    >ib_context); if(! ib_data->ib_channel) {
          fprintf(stderr, "open_ib_structs: failed in
          ibv_create_comp_channel\n"); goto func_return;
    }
    */
    fprintf(stderr, "open_ib_structs: attempting to create and configure PD, pointer
    to CTX %ld\n", (unsigned long)ib_data->ib_context);
    /* Allocate a protection domain */
    ib_data->ib_pd = ibv_alloc_pd(ib_data-
    >ib_context); if(!ib_data->ib_pd) {
          fprintf(stderr, "open_ib_structs: failed in
          ibv_alloc_pd\n"); goto func_return;
    }

    /* allocate memory page for future transactions
    */ ib_data->mem_lngth = getpagesize();
    fprintf(stderr, "open_ib_structs: attempting to allocate memory of size
    %d\n", ib_data->mem_lngth);
    ret = posix_memalign(&ib_data->mem_ptr, getpagesize(), ib_data-
    >mem_lngth); if(ret) {
          fprintf(stderr, "open_ib_structs: failed in
          posix_memalign\n"); goto func_return;
    }

    memset(ib_data->mem_ptr, 0, ib_data->mem_lngth);

    fprintf(stderr, "open_ib_structs: attempting to create and configure
    MR\n"); mem_access_mask = IBV_ACCESS_LOCAL_WRITE |
                                    IBV_ACCESS_REMOTE_WRITE|
                                   IBV_ACCESS_REMOTE_READ;
    /* ibv_reg_mr - Register a memory region */
    ib_data->ib_mem_region = ibv_reg_mr(ib_data->ib_pd, ib_data->mem_ptr,
                                     ib_data->mem_lngth, mem_access_mask);
    if(!ib_data->ib_mem_region) {
          fprintf(stderr, "open_ib_structs: failed in
          ibv_reg_mr\n"); goto func_return;
    }

    ib_data->ib_cq = ibv_create_cq(ib_data->ib_context,
                                       COMPLETE_QUEUE_LENGTH, NULL, NULL, 0);
    if(!ib_data->ib_cq) {
```

```
            fprintf(stderr, "open_ib_structs: failed in
            ibv_create_cq\n"); goto func_return;
    }

    fprintf(stderr, "open_ib_structs: attempting to create and configure QP\n");

    /* create the rcv and transmit QPs
    */ if(create_qp(ib_data)) {
            fprintf(stderr, "open_ib_structs: failed in
            create_qp\n"); goto func_return;
    }

    func_ret = 0;

func_return:
    return(func_ret);
}

int get_open_ib_device(int dev_number, echo_server_db *ib_data)
{
    struct                          ibv_device **dev_list;
    struct                          ibv_device *dev_2_use = NULL;
    unsigned int            dev_count;
    int                             i,ret_value = -1;
    const char              *dev_name;

    ib_data->ib_device = NULL;
    ib_data->ib_context = NULL;
    dev_list = ibv_get_device_list(&dev_count);
    if(!dev_list) {
            fprintf(stderr, "No devices
            found\n"); return(ret_value);
    }

    /* search for requested device in
    list*/ for(i=0; i<dev_count; i++) {
        dev_name = ibv_get_device_name(dev_list[i]);
        if(dev_name) {
            fprintf(stderr, "\nfound device %s", dev_name);
        }
        if(i == dev_number) { /* device found*/ ib_data-
            >ib_device = dev_list[i];
            fprintf(stderr, "  <- Device to be used in test");

            ib_data->ib_context = ibv_open_device(ib_data->ib_device);

            if(ib_data->ib_context)
            {
                            fprintf(stderr, "\nOpened
                            device"); ret_value = 0;
            }
            else
            {
                            ib_data->ib_device = NULL;
```

```
                        fprintf(stderr, "\nFailed to open device");
                }
            }
        }
        fprintf(stderr, "\n");

        if(!ib_data->ib_device) {
                fprintf(stderr, "No device allocated for test (requested dev num %d from %d
                        devices)\n", dev_number, dev_count);
        }

        /* In both cases, where we find the device and open it, and where we do not find
        the device, we should release the list.*/
        /* will not be released due to ibv_open_device
        */ ibv_free_device_list(dev_list);
        return(ret_value);
}


int main()
{
        struct                          ibv_device *dev_2_use = NULL;
        unsigned int            dev_count;
        int                             ret_value = -1;
        echo_server_db  ib_data;
        FILE                            *config_fd;

        memset(&ib_data, 0, sizeof(echo_server_db));


        config_fd = fopen("./echo_server_init.txt","rt");
        if(config_fd){
            fscanf(config_fd, "%d %d",
                  &ib_data.local_data.dev_num,
                  &ib_data.local_data.port_num);
            fclose(config_fd);
        };

        fprintf(stderr, "Type 1 for server\n");
             scanf("%d",&ib_data.is_recv);
        ret_value = get_open_ib_device(ib_data.local_data.dev_num,
        &ib_data); if(ret_value) {
            fprintf(stderr, "main: failed in
            get_open_ib_device\n"); goto exit_handler;
        }

        ret_value = open_ib_structs(&ib_data);
        if(ret_value) {
            fprintf(stderr, "main: failed in open_ib_structs\n");
            goto exit_handler;
        }

        /* handle receive and transmit */
        if(ib_data.is_recv) receive_data(&ib_data);
        else transmit_data(&ib_data);
```

```
      exit_handler:
    close_ib_device(&ib_data);
    return(ret_value);
}
```

A.23 Echo Server UD.c.c

```
#include <stdio.h>
#include <string.h>
#include <memory.h>
#include <verbs.h>

#define PSN 0x4321
#define COMPLETE_QUEUE_LENGTH100
#define  MAX_SUPPORTED_STRING256
#define TRANSACTION_QKEY0x1234

/* data needed to establish the connection
*/ typedef struct
{
        unsigned intl           ocal_dev_num;
        unsigned intl           ocal_port_num;
        unsigned int            dest_lid;
        unsigned int            dest_port;
        unsigned long           dest_qpn;
        unsigned int            is_recv;
} network_data_t;

/* all the IB structures  needed for the connection */
typedef struct
{
        struct ibv_device       *ib_device;
        struct ibv_context      *ib_context;
        struct ibv_comp_channel *ib_channel;/            * completion event channel */
        struct ibv_pd           *ib_pd;                  * protection domain */
        unsigned int            mem_lngth;
        void                                *mem_ptr;
        struct ibv_mr           *ib_mem_region;
        struct ibv_ah           *ib_ah;
        struct ibv_cq           *ib_cq;
        struct ibv_qp           *ib_qp;
        network_data_t          net_data;
} echo_server_db;


int close_ib_device(echo_server_db *ib_data)
{
```

```
        fprintf(stderr, "close_ib_device: calling ibv_destroy_qp\n");
        if(ib_data->ib_qp) ibv_destroy_qp(ib_data->ib_qp); fprintf(stderr,
        "close_ib_device: calling ibv_destroy_cq\n"); if(ib_data->ib_cq)
        ibv_destroy_cq(ib_data->ib_cq); fprintf(stderr, "close_ib_device:
        calling ibv_dereg_mr\n"); if(ib_data->ib_mem_region)
        ibv_dereg_mr(ib_data->ib_mem_region); fprintf(stderr,
        "close_ib_device: calling free\n"); if(ib_data->mem_ptr) free(ib_data-
        >mem_ptr);
        fprintf(stderr, "close_ib_device: calling ibv_dealloc_pd\n");
        if(ib_data->ib_pd) ibv_dealloc_pd(ib_data->ib_pd);
        fprintf(stderr, "close_ib_device: calling ibv_destroy_comp_channel\n");
        if(ib_data->ib_channel) ibv_destroy_comp_channel(ib_data->ib_channel);
        fprintf(stderr, "close_ib_device: calling ibv_close_device\n"); if(ib_data-
        >ib_context) ibv_close_device(ib_data->ib_context); fprintf(stderr,
        "finished closing device\n");

        return 0;
}
```

End of Differences Note/* post a single recieve entry. */

```
inline int post_receive_entry(struct ibv_qp *ib_qp, const uint32_t
           lkey, char *mem_region, const uint32_t length)
{
        struct ibv_sge          mem_list;
        struct ibv_recv_wr      wr;
        struct ibv_recv_wr      *bad_wr;

        memset(&wr, 0, sizeof(struct ibv_recv_wr));
        wr.sg_list              = &mem_list;
        wr.num_sge              = 1;
        wr.wr_id                = (uint64_t)mem_region;/* save address for receive in wr_id */

        mem_list.lkey           = lkey;
        mem_list.addr           = (uintptr_t)mem_region;
        mem_list.length         = length;

        return(ibv_post_recv(ib_qp, &wr, &bad_wr));
}

/* Initial posting of recive requests. More requests will be added */
/* as these requests will be consumed.                             */
int post_receive(echo_server_db  *ib_data)
{
   int                  i;
   char             *mem_area;
   uint32_t         lkey = ib_data->ib_mem_region->lkey;

   for (i = 0; i < COMPLETE_QUEUE_LENGTH; ++i)
   {
      /* use cyclic buffer of length 7 on receive */
```

```
        mem_area = (char *)(((unsigned long)ib_data->mem_ptr) + ((i & 0x7)* MAX_SUPPORTED_STRING));


        if(post_receive_entry(ib_data->ib_qp, lkey, mem_area,
            MAX_SUPPORTED_STRING)) break;
    }

    if(i < COMPLETE_QUEUE_LENGTH-1) {
        fprintf(stderr,"receieve_data: failed to allocate receive queue using ibv_post_recv i=%d\n",i);
        return(-1);
    }

    fprintf(stderr,"receieve_data: Allocated receive WE using ibv_post_recv\n");

    return 0;
}

/* Recive data and print the data received to the stderr. */
/* The function assumes that all data in CQ was due to RCV and not to */
/* TX or errors. */
int receive_data(echo_server_db  *ib_data)
{
    struct ibv_wc       rcv_wc;
    int                         poll_ret, rx_length, recieve_count = 0;
    uint32_t                    lkey = ib_data->ib_mem_region->lkey;

    while(recieve_count <= 60) {
        poll_ret = ibv_poll_cq(ib_data->ib_cq, 1, &rcv_wc);
        if(poll_ret > 0) {
            if(rcv_wc.status != IBV_WC_SUCCESS)
                        fprintf(stderr,"Recieve of message failed with error %d\n",rcv_wc.status);

            rx_length = rcv_wc.byte_len;
            /* fprintf(stderr,"length %d",rx_length); */
            fprintf(stderr, "%s ",((char *)rcv_wc.wr_id)+40);

            /* return the used entry to the receive queue so we don't run out of receives */
            if(post_receive_entry(ib_data->ib_qp, lkey, (char *)rcv_wc.wr_id, MAX_SUPPORTED_STRING)) {
                        fprintf(stderr, "receive_data: failed in post_receive_entry\n");
            }
            recieve_count++;
        }
        if(poll_ret < 0) fprintf(stderr, "receieve_data: Failed in RCV in ibv_poll_cq\n");
    }
    return 0;
}

/* Transmit data received from the stdin and send it to the server. */
/* The function assumes that all entries in the CQ are due to transmit
*/ /* and not to recieve. */
int transmit_data(echo_server_db  *ib_data)
{
    volatile int l;
```

```
int transmit_count =
0; char*tmp_Buffer;
struct ibv_wc tx_wc;
int string_length;
int ret = -1;
struct ibv_send_wr *bad_wr;
struct ibv_sge list;
struct ibv_ah *ah;
struct ibv_send_wr wr;
struct ibv_ah_attr ah_attr = {
   .dlid = ib_data->net_data.dest_lid,
   .port_num = ib_data->net_data.local_port_num
};

ah = ibv_create_ah(ib_data-
>ib_pd,&ah_attr); if(!ah) {
   fprintf(stderr,"transmit_data: failed to create address header\n");
   goto func_return;
}

memset(&wr, 0, sizeof(struct ibv_send_wr));
wr.sg_list              = &list;
wr.num_sge              = 1;
wr.opcode               = IBV_WR_SEND;
wr.send_flags           = IBV_SEND_SIGNALED;
wr.wr.ud.remote_qpn     = ib_data->net_data.dest_qpn;
wr.wr.ud.remote_qkey     = TRANSACTION_QKEY;
wr.wr.ud.ah             = ah;

list.lkey = ib_data->ib_mem_region->lkey;

while(transmit_count < 60) {

   /* small delay to prevent overruns in cyclic buffer
   */ for(l=0; l<1000; l++) ;

   /* use cyclic buffer of length 7 on receive */
   tmp_Buffer = (char *)(((unsigned long)ib_data->mem_ptr) +
                                        (transmit_count & 0x7) * MAX_SUPPORTED_STRING);

   /* get data from user */
   memset(tmp_Buffer, 0, MAX_SUPPORTED_STRING);
   scanf("%s",tmp_Buffer);
   string_length           = strlen(tmp_Buffer);

   list.addr               = (uintptr_t)tmp_Buffer;
   list.length             = string_length + 1;
   wr.wr_id                = (uint64_t)tmp_Buffer;

   if(ret = ibv_post_send(ib_data->ib_qp, &wr,&bad_wr)) {
      fprintf(stderr,"transmit_data:          ibv_post_send
      failed\n"); goto func_return;
   }
   transmit_count ++;

   if(ret = ibv_poll_cq(ib_data->ib_cq, 1, &tx_wc) < 0) {
```

```
            fprintf(stderr,"transmit_data: ibv_poll_cq
            failed\n"); return(ret);
        }
    }

    func_return:
        if(ah) ibv_destroy_ah(ah);
        return(ret);
}



inline void get_data_from_user(echo_server_db  *ib_data)
{
        fprintf(stderr, "Type 1 for server\n");
    scanf("%d",&ib_data->net_data.is_recv);
    if(! ib_data->net_data.is_recv) {
        printf("Type remote LID\n");
        scanf("%d",&ib_data->net_data.dest_lid);
        //printf("Type remote PORT\n");
        //scanf("%d",&ib_data->net_data.dest_port);
        printf("Type remote QP number\n");
        scanf("%d",&ib_data->net_data.dest_qpn);
    }
}

int create_qp(echo_server_db  *ib_data)
{
    int ret, func_ret = -1;
    struct  ibv_qp_init_attr       qp_init_attr;
    struct ibv_qp_attr             qp_attr;
    enum ibv_qp_attr_mask          qp_attr_mask;
    struct ibv_ah_attr             dest_attr;
    unsigned long long             guid;
    struct ibv_port_attr           port_attr;
    uint16_t                       lid;
    memset(&qp_init_attr,0,sizeof(struct ibv_qp_init_attr));
    qp_init_attr.send_cq          = ib_data->ib_cq;
    qp_init_attr.recv_cq          = ib_data->ib_cq;
    qp_init_attr.cap.max_inline_data= 1;/* max inline = 1 */
    qp_init_attr.cap.max_recv_sge = 1;/* do not use scatter gather on RX */
    qp_init_attr.cap.max_send_sge = 1;/* do not use scatter gather on TX */
    qp_init_attr.cap.max_recv_wr  = COMPLETE_QUEUE_LENGTH - 1;// smaller then CQ length
    qp_init_attr.cap.max_send_wr  = COMPLETE_QUEUE_LENGTH - 1;// smaller then CQ length
    qp_init_attr.qp_type          = IBV_QPT_UD;// connection type
    qp_init_attr.sq_sig_all       = 1;/* generate CQ for every WQE*/
    //qp_init_attr.xrc_domain     = NULL;// ??

    // ibv_create_qp - Create a queue pair.
    ib_data->ib_qp = ibv_create_qp(ib_data->ib_pd,
    &qp_init_attr); if(!ib_data->ib_qp) {
        fprintf(stderr, "create_qp: failed in
        ibv_create_qp\n"); goto func_return;
```

```
}

/* Switch QP from reset to init state. Also config no ATOMIC or RDMA
*/ /* and configured to work with physical port 1 */
memset((char *)&qp_attr,0, sizeof(struct ibv_qp_attr));
qp attr.qp state              = IBV QPS INIT;
qp_attr.port_num              = ib_data->net_data.local_port_num;// port number
qp_attr.pkey_index            = 0;
qp_attr.qkey                  = TRANSACTION_QKEY;
qp_attr_mask =         IBV_QP_STATE                  |
                  IBV_QP_PORT                        |
                  IBV_QP_PKEY_INDEX                  |
                  IBV_QP_QKEY;
if(ret = ibv_modify_qp(ib_data->ib_qp, &qp_attr, qp_attr_mask))
   {  fprintf(stderr, "create_qp: failed to init QP with ibv_modify_qp error %d\n", ret);
   goto func_return;
}

// ---- query the port and QP for debug purposes ------
ibv_query_port(ib_data->ib_context, ib_data->net_data.local_port_num,
&port_attr); lid = port_attr.lid;
qp_attr_mask =         IBV_QP_PORT|
                  IBV_QP_STATE                       |
                  IBV_QP_AV                          |
                  IBV_QP_PATH_MTU                    |
                  IBV_QP_DEST_QPN                    |
                  IBV_QP_RQ_PSN;
memset(&qp_attr,0, sizeof(struct ibv_qp_attr));
ibv_query_qp(ib_data->ib_qp, &qp_attr,qp_attr_mask,&qp_init_attr);

fprintf(stderr, "QP query: State %d, port %d, MTU %d, QPN %d lid %d
   \n",  qp_attr.qp_state, qp_attr.port_num, qp_attr.
   path_mtu, ib_data->ib_qp->qp_num, (uint16_t)lid);

/* get the destination lid, port, and qpn from the
user*/ get_data_from_user(ib_data);
if(ib_data->net_data.is_recv)
   {  if(post_receive(ib_data))
       fprintf(stderr, "create_qp: failed to allocate receive queue\n");
}

in the host/* config addresses and other stuff */
memset(&qp_attr,0, sizeof(struct ibv_qp_attr));
qp_attr.qp_state              = IBV_QPS_RTR;
qp_attr_mask =      IBV_QP_STATE ;

if(ret = ibv_modify_qp(ib_data->ib_qp, &qp_attr, qp_attr_mask))
{
   fprintf(stderr, "create_qp: failed to move QP to RTR with ibv_modify_qp with error
   %d\n",ret); goto func_return;
}

/* transmit side only */
if(! ib_data->net_data.is_recv)
```

```
    {
        //memset(&qp_attr,0, sizeof(struct ibv_qp_attr));
        qp_attr.qp_state            = IBV_QPS_RTS;
        qp_attr.sq_psn                          = PSN;
        qp_attr_mask = IBV_QP_STATE |
            IBV_QP_SQ_PSN;

        if(ret = ibv_modify_qp(ib_data->ib_qp, &qp_attr, qp_attr_mask))
        {
            fprintf(stderr, "create_qp: failed to move QP to RTS with ibv_modify_qp with error %d\n",
            ret); goto func_return;
        }
    }

    func_ret = 0;

    func_return:
        return(func_ret);
}

int open_ib_structs(echo_server_db  *ib_data)
{
    int func_ret = -
    1; int ret;
    fprintf(stderr, "open_ib_structs: attempting to configure IB channel\n");

    /* Create a completion event channel */
/*ib_data->ib_channel = ibv_create_comp_channel(ib_data->ib_context);
    if(! ib_data->ib_channel) {
        fprintf(stderr, "open_ib_structs: failed in ibv_create_comp_channel\n");
        goto func_return;
    }
*/
    fprintf(stderr, "open_ib_structs: attempting to create and configure PD, pointer
    to CTX %ld\n", (unsigned long)ib_data->ib_context);
    /* Allocate a protection domain */
    ib_data->ib_pd = ibv_alloc_pd(ib_data-
    >ib_context); if(!ib_data->ib_pd) {
        fprintf(stderr, "open_ib_structs: failed in
        ibv_alloc_pd\n"); goto func_return;
    }

    /* allocate memory page for future transactions
    */ ib_data->mem_lngth = getpagesize();
    fprintf(stderr, "open_ib_structs: attempting to allocate memory of size
    %d\n", ib_data->mem_lngth);
    ret = posix_memalign(&ib_data->mem_ptr, getpagesize(), ib_data-
    >mem_lngth); if(ret) {
        fprintf(stderr, "open_ib_structs: failed in
        posix_memalign\n"); goto func_return;
    }

    fprintf(stderr, "open_ib_structs: attempting to create and configure MR\n");
```

```
    /* ibv_reg_mr - Register a memory region */
    ib_data->ib_mem_region = ibv_reg_mr(ib_data->ib_pd, ib_data-
           >mem_ptr,  ib_data->mem_lngth, IBV_ACCESS_LOCAL_WRITE);
    if(!ib_data->ib_mem_region) {
       fprintf(stderr, "open_ib_structs: failed in
       ibv_reg_mr\n"); goto func_return;
    }

    ib_data->ib_cq = ibv_create_cq(ib_data->ib_context, COMPLETE_QUEUE_LENGTH,
             NULL, ib_data->ib_channel,
    0); if(!ib_data->ib_cq) {
           fprintf(stderr, "open_ib_structs: failed in
           ibv_create_cq\n"); goto func_return;
    }

    fprintf(stderr, "open_ib_structs: attempting to create and configure QP\n");

    /* create the rcv and transmit QPs
    */ if(create_qp(ib_data)) {
           fprintf(stderr, "open_ib_structs: failed in
           create_qp\n"); goto func_return;
    }

    func_ret = 0;

func_return:
    return(func_ret);
}

int get_open_ib_device(int dev_number, echo_server_db *ib_data)
{
    struct                          ibv_device **dev_list;
    struct                          ibv_device *dev_2_use = NULL;
    unsigned int     dev_count;
    int                             i,ret_value = -1;
    const char                      *dev_name;

    ib_data->ib_device = NULL;

    ib_data->ib_context = NULL;

    dev_list = ibv_get_device_list(&dev_count);
    if(!dev_list) {
       fprintf(stderr, "No devices
       found\n"); return(ret_value);
    }

    /* search for requested device in
    list*/ for(i=0; i<dev_count; i++) {
           dev_name = ibv_get_device_name(dev_list[i]);
           if(dev_name) {
                           fprintf(stderr, "\nfound device %s", dev_name);
           }
           if(i == dev_number) {/* device found*/ ib_data-
                           >ib_device = dev_list[i];
```

```
                        fprintf(stderr, "  <- Device to be used in test");

                        ib_data->ib_context = ibv_open_device(ib_data->ib_device);

                        if(ib_data->ib_context)
        {
                        fprintf(stderr, "\nOpened
                        device"); ret_value = 0;
        }
        else
        {
                        ib_data->ib_device = NULL;
                        fprintf(stderr, "\nFailed to open device");
        }
      }
    }
    fprintf(stderr, "\n");

    if(!ib_data->ib_device) {
        fprintf(stderr, "No device allocated for test (requested dev num %d from %d
            devices)\n", dev_number, dev_count);
    }

    /* In both cases, where we find the device and open it, and where we do not find
    the device, we should release the list. */
    /* will not be released due to ibv_open_device
    */ ibv_free_device_list(dev_list);
    return(ret_value);
}



int main()
{
    struct                          ibv_device *dev_2_use = NULL;
    unsigned int                    dev_count;
    int                             ret_value = -1;
    echo_server_db  ib_data;
    FILE                            *config_fd;

    memset(&ib_data, 0, sizeof(echo_server_db));


    config_fd = fopen("./echo_server_init.txt","rt");
    if(config_fd){
        fscanf(config_fd, "%d %d",
              &ib_data.net_data.local_dev_num,
              &ib_data.net_data.local_port_num);
        fclose(config_fd);
    };

    ret_value = get_open_ib_device(ib_data.net_data.local_dev_num,
    &ib_data); if(ret_value) {
        fprintf(stderr, "main: failed in
        get_open_ib_device\n"); goto exit_handler;
    }
```

```
    ret_value = open_ib_structs(&ib_data);
    if(ret_value) {
       fprintf(stderr, "main: failed in open_ib_structs\n");
       goto exit_handler;
    }

    /* handle receive and transmit */
    if(ib_data.net_data.is_recv) receive_data(&ib_data);
    else transmit_data(&ib_data);

  exit_handler:
    close_ib_device(&ib_data);
    return(ret_value);
}
```

A.24 Echo Server.h

```
    #include <verbs.h>

    #define PSN                     0x4321
    #define COMPLETE_QUEUE_LENGTH100
    #define MAX_SUPPORTED_STRING256
    #define ECHO_SERVER_ID          0x1234

    /* data needed to establish the connection
    */ /*typedef struct
    {
       unsigned int             local_dev_num;
       unsigned int             local_port_num;
       unsigned int             local_lid;
       unsigned int             local_port;
       unsigned long            local_qpn;
       uint64_t                         local_mem_ptr;
       uint32_t                         local_rkey;

       unsigned int             dest_lid;
       unsigned int             dest_port;
       unsigned long            dest_qpn;
       uint64_t                         dest_mem_ptr;
       uint32_t                         dest_rkey;
       unsigned int             is_recv;
    } network_data_t;
    */
    typedef struct
    {
       unsigned int             dev_num     ;
       unsigned int             lid;
       unsigned int             port_num    ;
       unsigned long            qpn;
       uint64_t                         mem_ptr;
       uint32_t                         rkey;
    } port_data_t;
```

```
/* all the IB structures needed for the connection
*/ typedef struct
{
   struct ibv_device                *ib_device;
   struct ibv_context               *ib_context;
   struct ibv_comp_channel *ib_channel;                    /* completion event channel
   struct ibv_pd                     *ib_pd;          */ /* protection domain */
   unsigned int                      mem_lngth;
   void                              *mem_ptr;
   struct ibv_mr                     *ib_mem_region;
   struct ibv_ah                     *ib_ah;
   struct ibv_cq                     *ib_cq;
   struct ibv_qp                     *ib_qp;
   port_data_t                       local_data;
   port_data_t                       remote_data;
   int            is_recv;

} echo_server_db;

int side_a(const port_data_t *local, port_data_t *remote, const
                    char *remote_name, const int portno);
int side_b(const port_data_t *local, port_data_t *remote, const
                    char *remote_name, const int portno);
```