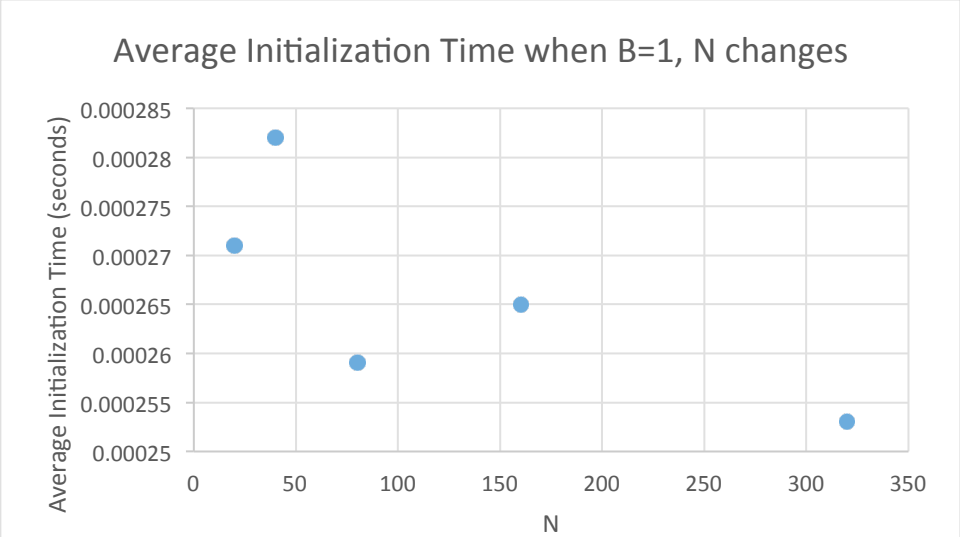
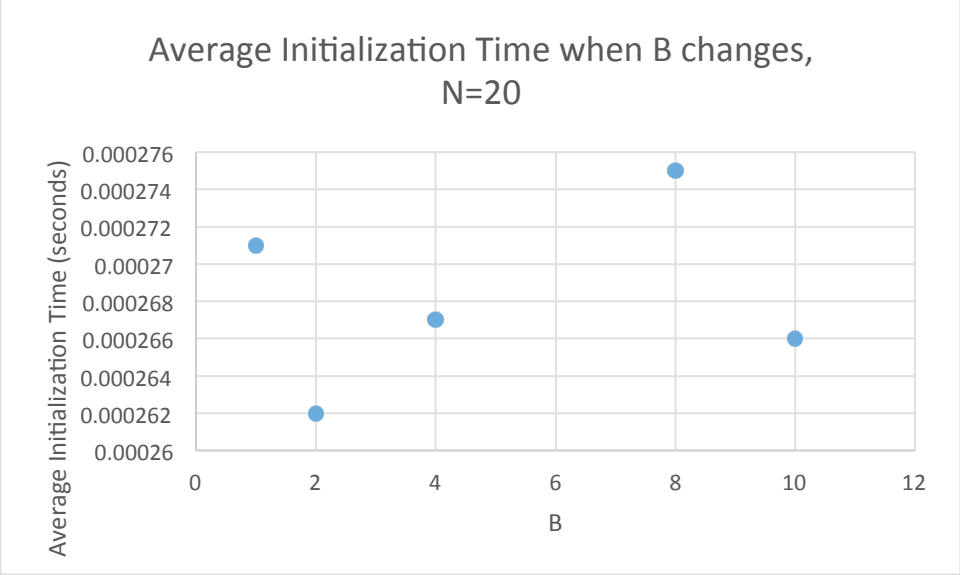
		Average Initialization Time (seconds)				
		B				
		1	2	4	8	10
N	20	0.000271	0.000262	0.000267	0.000275	0.000266
	40	0.000282	0.000276	0.000283	0.000267	0.000257
	80	0.000259	0.000255	0.000261	0.000256	0.000264
	160	0.000265	0.000272	0.00026	0.000248	0.000259
	320	0.000253	0.000258	0.000246	0.000253	0.000238





In gerneral, the initialization time is not very affected by N or B. As we can see from the graphs, there is no correlation between average initialization time and N or B. In cases, we can see that the average initialization time, regardless of N or B, is pretty much constant. This makes sense because even if we

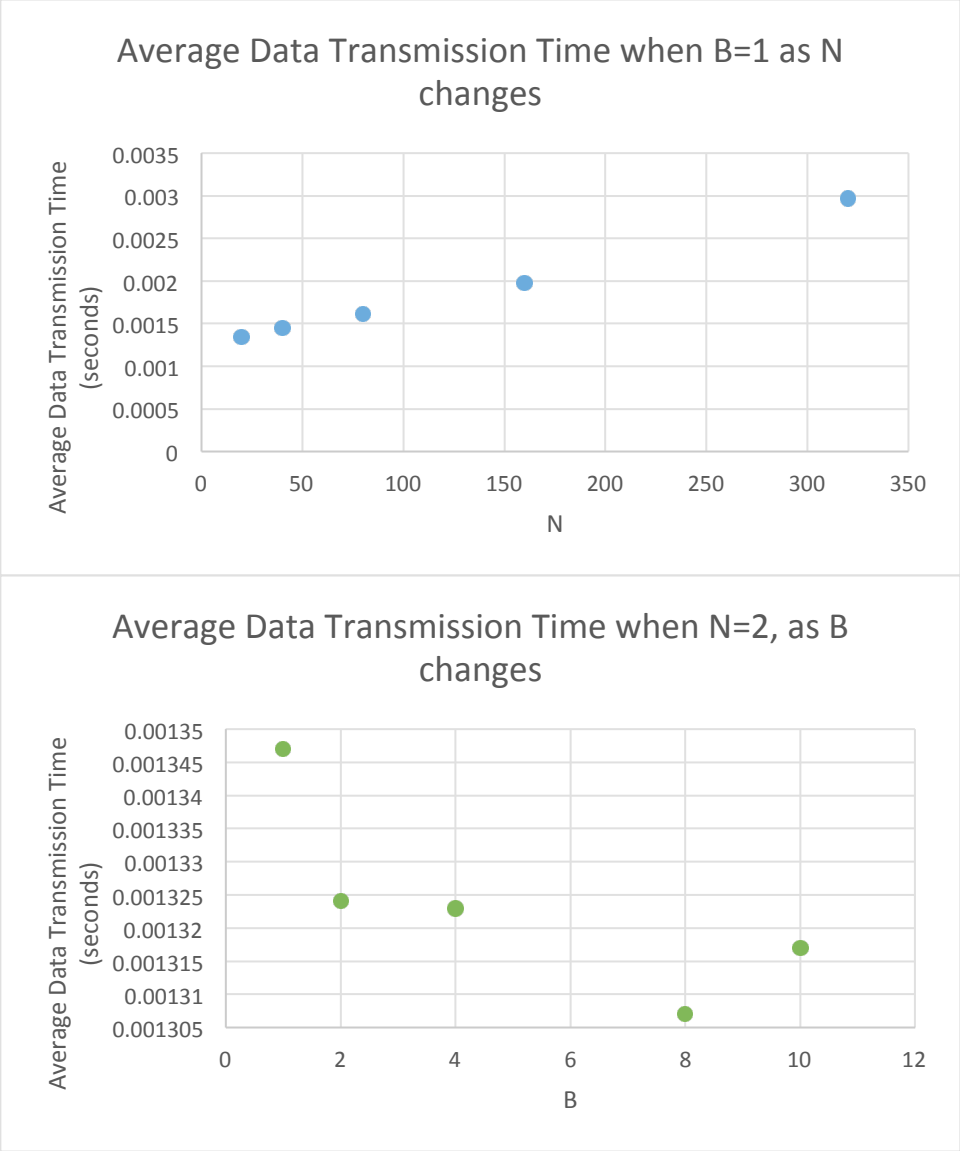
increase the size of our queue or the amount of data that is going to be transmitted, the time to setup the system (which is the time to create a separate child process) should not change.

Standard Deviation of System Initialization Time

		B				
		1	2	4	8	10
N	20	0.000162	0.000125	0.000142	0.00016	0.000142
	40	0.000187	0.000167	0.000174	0.000129	0.00012
	80	0.000123	0.000115	0.000123	0.000096	0.000136
	160	0.000136	0.000157	0.000094	0.000087	0.000135
	320	0.0001	0.000129	0.000085	0.000098	0.000063

Average Data Transmission Time (seconds)

		B				
		1	2	4	8	10
N	20	0.001347	0.001324	0.001323	0.001307	0.001317
	40	0.001449	0.001416	0.001414	0.001386	0.001383
	80	0.001612	0.001571	0.00154	0.001507	0.001514
	160	0.001973	0.001897	0.001839	0.001876	0.001873
	320	0.002968	0.002746	0.002585	0.002717	0.003683



The average data transmission time is not very affected by B. As we change B, the average data transmission time stays pretty constant, and only responds very slightly. As seen by the graph, there seems to be no correlation between transmission time and B. This makes sense because even as we increase the size of our queue, the time it takes to transmit all the data should not be affected. Even if the size of our queue increases, then we still have to transmit the same amount of data, so the overall time to transmit the data should remain the same.

There average data transmission time seems to be proportional to N. It is clear that as we increase N, the data transmission time increases as well. This is further supported by the graph, which shows a linear relationship between the transmission time and N. This makes sense because as we increase the number of elements in our queue that we add, it makes sense the average time to transfer all the elements should increase as well, because it takes longer to transmit more data.

Standard deviation of the data transmission time

		B				
		1	2	4	8	10
N	20	0.000244	0.000221	0.000239	0.000234	0.000265
	40	0.000261	0.000212	0.000246	0.000208	0.000243
	80	0.00023	0.000207	0.000211	0.000202	0.000241
	160	0.000261	0.000267	0.000256	0.000308	0.000319
	320	0.000582	0.000524	0.00051	0.00064	0.000562

Appendix A

Receiver.c code

```
/**
 * @file:   code/lab4/mqueue/receiver.c
 * @brief:  receive a message from POSIX queue. Typle Ctrl-C to quit
 * @date:   2014/06/25
 * NOTES:
 * To execute: <executable> <arg1>, where arg1 is a posix queue name.
 * EXAMPLE: ./receiver.out /mailbox1_userid
 * A valid posix queue name must start with a "/".
 * Execute command "man mq_overview" for details.
 */

#include <stdbool.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <signal.h>
#include "common.h"
#include "point.h"

#define _XOPEN_SOURCE 600

bool g_continue = true;

void sig_handler(int sig)
```

```
{
        g_continue = false;
}

int main(int argc, char *argv[])
{
        mqd_t qdes;
        char  *qname = NULL;
        mode_t mode = S_IRUSR | S_IWUSR;
        struct mq_attr attr;
        struct timeval tv1;
        double t3;
        double t4;
        /* checking to make sure only two arguments come in */
        if ( argc !=2 ) {
                printf("Usage: %s <qname>\n", argv[0]);
                printf("The qname must start with a \"/\". \n");
                printf("An example qname: /mailbox1_userid\n");
                exit(1);
        }
        /* setting a default name for our mailbox*/
        qname = "/mailbox1_userid";

        attr.mq_maxmsg  = QUEUE_SIZE;
        attr.mq_msgsize = sizeof(struct point);
        attr.mq_flags   = 0;        /* a blocking queue  */
        int counter = 0;
        qdes  = mq_open(qname, O_RDONLY, mode, &attr);
        if (qdes == -1 ) {
```

```
        perror("mq_open()");
        exit(1);
}

signal(SIGINT, sig_handler);        /* install Ctl-C signal handler */

srand(time(0));
int j;
/* we will only consume up until the number of items in the queue*/
for (j = 0; j < atoi(argv[0]); j++){
        struct point pt;
        struct timespec ts = {time(0) + 5, 0};

        /* only block for a limited time if the queue is empty */
        if (mq_timedreceive(qdes, (char *) &pt, \
           sizeof(struct point), 0, &ts) == -1) {
                perror("mq_timedreceive() failed");
                printf("Type Ctrl-C and wait for 5 seconds to terminate.\n");
        } else {
                printf("random num %d\n", \
                 get_x_coord(pt));

        }

}
/* we must close the connection to the queue still*/
if (mq_close(qdes) == -1) {
        perror("mq_close() failed");
        exit(2);
```

```
        }


        return 0;
}
```

Sender.c

```
/**
 * @file:   code/lab4/mqueue/sender.c
 * @brief:  send random points to a POSIX queue. Type 'q' to terminate.
 * @date:   2014/06/25
 * NOTES:
 * To execute: <executable> <arg1>, where arg1 is a posix queue name.
 * EXAMPLE: ./sender.out /mailbox1_userid
 * A valid posix queue name must start with a "/".
 * Execute command "man mq_overview" for details.
 * Check /dev/mqueue to clean up the queue if sender fails to do so.
 */

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <time.h>
#include "common.h"
#include "point.h"
#include <wait.h>

int spawn (char* program, char** arg_list)
```

```
{
  pid_t child_pid;

  /* Duplicate this process.  */
  child_pid = fork ();
  if (child_pid != 0)
    /* This is the parent process.  */
    return child_pid;
  else {
    /* Now execute PROGRAM, searching for it in the path.  */
    execvp (program, arg_list);
    /* The execvp function returns only if an error occurs.  */
    fprintf (stderr, "an error occurred in execvp\n");
    abort ();
  }
}

int main(int argc, char *argv[])
{
        int child_status;/* this variable is only used for wait*/
        mqd_t qdes;
        char  quit = '\0';
        int i;
        /* queue name must start with '/'. man mq_overview */
        //char  qname[] = "/mailbox1_yqhuang";
        char *qname = NULL;
        mode_t mode = S_IRUSR | S_IWUSR;
        struct mq_attr attr;
        int status;
```

```
        struct timeval tv;
double t1;
double t2;
double t3;
/* ensuring that the three arguments required are coming in*/
if ( argc !=3 ) {
        printf("Usage: %s <qname>\n", argv[0]);
        printf("The qname must start with a \"/\". \n");
        printf("You must specify your N value");
        printf("You must specify your B value");
        exit(1);
}
/* capacity = N*/
int capacity = atoi(argv[1]);
qname = "/mailbox1_userid";/*default mailbox name*/

attr.mq_maxmsg  = atoi(argv[2]);/*max queue size*/
attr.mq_msgsize = sizeof(struct point);
attr.mq_flags   = 0;                    /* a blocking queue  */

qdes  = mq_open(qname, O_RDWR | O_CREAT, mode, &attr);
if (qdes == -1 ) {
        perror("mq_open() failed");
        exit(1);
}
/* this is the time just before fork*/
gettimeofday(&tv, NULL);
t1 = tv.tv_sec + tv.tv_usec/1000000.0;
/* building our child process's argument list*/
```

```
        char* arg_list_reciever[] = {
        argv[1],
        argv[2],
        NULL

};

//receiver process
spawn("./receiver.out", arg_list_reciever);



/* seed for the random int*/
srand(time(0));
/* produce our N values*/
        struct point pt;
        for (i=0; i < capacity; i++){
                set_position(rand() % 80, rand() % 24, &pt);
                /* we will set time before first data transfer here*/
                if (i == 0){
                        gettimeofday(&tv, NULL);
                        t2 = tv.tv_sec + tv.tv_usec/1000000.0;
                }
                if (mq_send(qdes, (char *)&pt, sizeof(struct point), 0) == -1) {
                        perror("mq_send() failed");
                }


        }
        /* time after last data transfer*/
        gettimeofday(&tv, NULL);
```

```
        t3 = tv.tv_sec + tv.tv_usec/1000000.0;
        /* before unlinking queue we must wait for child to finish*/
        wait(&child_status);
        if (WIFEXITED (child_status)){

                if (mq_close(qdes) == -1){
                        perror("mq_close() failed");
                        exit(2);
                }

                                                if (mq_unlink(qname) != 0){
                        perror("mq_unlink() failed");
                        exit(3);
                }
                printf("Time to initialize system: %.6lf seconds\n", t2-t1);
                printf("Time to transmit data: %.6lf seconds\n",t3-t2);
        }
        else{
                printf ("the child process exited abnormally\n");
        }



    return 0;
}
```