ECE 254 : Operating Systems and Systems Programming

Lab 4 Report

Group 01
By: Rushan Yogaratnam & Ameen Patel
2B Computer Engineering
University of Waterloo
Spring 2014

Introduction:

In this lab a solution to the producer consumer problem was developed using the facilities of the POSIX API, in particular the message queue, fork and exec functionalities. A timing simulation was conducted to determine the relationships between the number of elements the consumer produced and the size of the message queue on the overall data transmission and system initialization time. We examine the results of both these in the following report.

Important Timing Considerations:

As per the lab manual we are required to have the consumer print out "X is consumed" where X is the element that is received from the message queue. A key issue is that the use of printf causes an IO wait and causes the timing results to be skewed. Therefore we have provided timing results for the program with and without the print statement. As you can see the trends are much are observable if the call to printf() is omitted. *In addition it should be noted that because of the interleaving and process switching the timing results vary during each trial and is therefore unpredictable at times.*

```
int i;
for (i = 1; i <= messages_to_consume; ++i) {
        int message;

        if (mq_receive(queue_descriptor, (char*) &message, sizeof(int),
                        0) == -1) {
                printf("failed to receive message %s \n", strerror(errno));
                return 1;

        } else {
                //Note that this printf causes an IO wait and slows down
                //the timing results.
                printf("%i is consumed\n", message);

        }
```

Figure 1.1: A code sample from the consumer where the messages are received, notice that the use of printf creates an IO wait scenario. A set of time results have also been provided where the print f is commented out.

The following timing data was collected with the printf statement present
All measurements are in seconds

Average System Initialization Time					
N\B	1	2	4	8	10
20	0.000236	0.000238	0.000237	0.000246	0.000236
40	0.000241	0.00024	0.00023	0.000234	0.00024
80	0.000237	0.000235	0.000241	0.000241	0.000241
160	0.000241	0.000242	0.000236	0.000223	0.000228
320	0.000234	0.000236	0.000238	0.00024	0.000241

Standard Deviation of System Initialization Time					
N\B	1	2	4	8	10
20	0.00007	0.000095	0.000072	0.000107	0.000075
40	0.000099	0.000083	0.000096	0.000097	0.000083
80	0.000089	0.000069	0.000094	0.000101	0.000097
160	0.000091	0.000092	0.000089	0.000118	0.000106
320	0.000076	0.000073	0.000071	0.0001	0.000106

Average Data Transmission Time					
N\B	1	2	4	8	10
20	0.001523	0.001472	0.001494	0.001445	0.001466
40	0.001621	0.001563	0.001499	0.001496	0.001517
80	0.001789	0.001651	0.00164	0.001599	0.001597
160	0.002223	0.001861	0.001802	0.00171	0.001746
320	0.003032	0.002283	0.002149	0.002136	0.001961

Standard Deviation of Data Transmission Time					
N\B	1	2	4	8	10
20	0.000188	0.000224	0.00021	0.000247	0.000212
40	0.000204	0.00022	0.000303	0.000267	0.000205
80	0.000235	0.00026	0.000717	0.000209	0.000212
160	0.000262	0.000327	0.000314	0.000389	0.00034
320	0.000335	0.00043	0.000362	0.000346	0.000233

The following timing data was collected without the printf in the consumer

Average System Initialization Time					
N\B	1	2	4	8	10
20	0.000236	0.000238	0.000237	0.000246	0.000239
40	0.000243	0.00024	0.000233	0.000238	0.000242
80	0.000236	0.000236	0.000242	0.000241	0.000241
160	0.000241	0.000244	0.000237	0.000228	0.000232
320	0.000236	0.000237	0.000239	0.000238	0.00024

Standard Deviation of System Initialization Time					
N\B	1	2	4	8	10
20	0.000066	0.00009	0.000071	0.000108	0.000088
40	0.000111	0.000081	0.000098	0.000104	0.000093
80	0.000085	0.000078	0.000096	0.0001	0.000091
160	0.000086	0.000099	0.00009	0.000112	0.000102
320	0.000084	0.000073	0.000079	0.00009	0.000104

Average Data Transmission Time					
N\B	1	2	4	8	10
20	0.001523	0.001473	0.001487	0.00145	0.001472
40	0.00160	0.001563	0.001497	0.0015	0.001512
80	0.001808	0.001644	0.001625	0.001591	0.001599
160	0.00224	0.001849	0.001786	0.001717	0.001741
320	0.003010	0.002234	0.002114	0.002098	0.001960

Standard Deviation of Data Transmission Time					
N\B	1	2	4	8	10
20	0.000191	0.000218	0.000208	0.000239	0.00022
40	0.000203	0.000227	0.000294	0.000256	0.000214
80	0.000234	0.000251	0.000629	0.000209	0.000216
160	0.000262	0.000314	0.000305	0.000359	0.000323
320	0.000331	0.000429	0.000342	0.00033	0.000228

Notice that the values are slightly smaller due to the lack of the printf () call in the consumer.

Graphical Analysis:

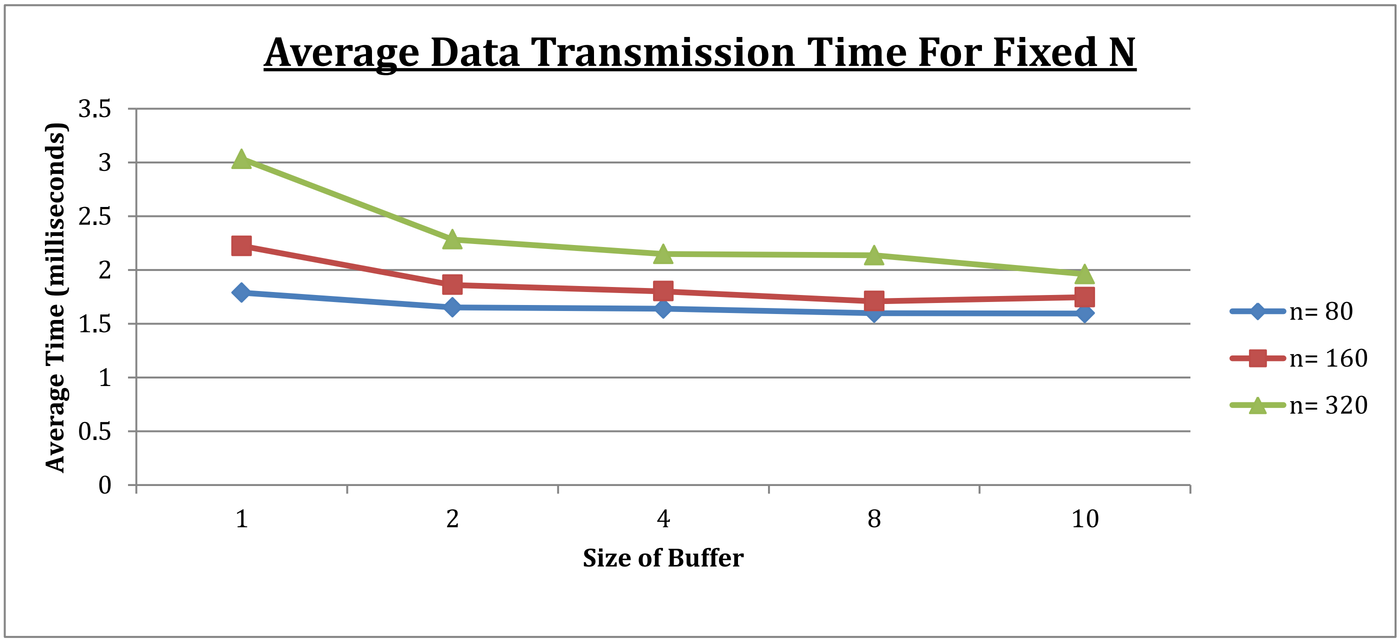


Figure 1.2: Plot of average data transmission time for a fixed value of integers produced (N) while varying the size of the buffer (B). As seen by the graph for large values of N the increase in buffer size causes the average time to decrease steadily. The slight variance in the relationship can be attributed to the differences in process interleaving.

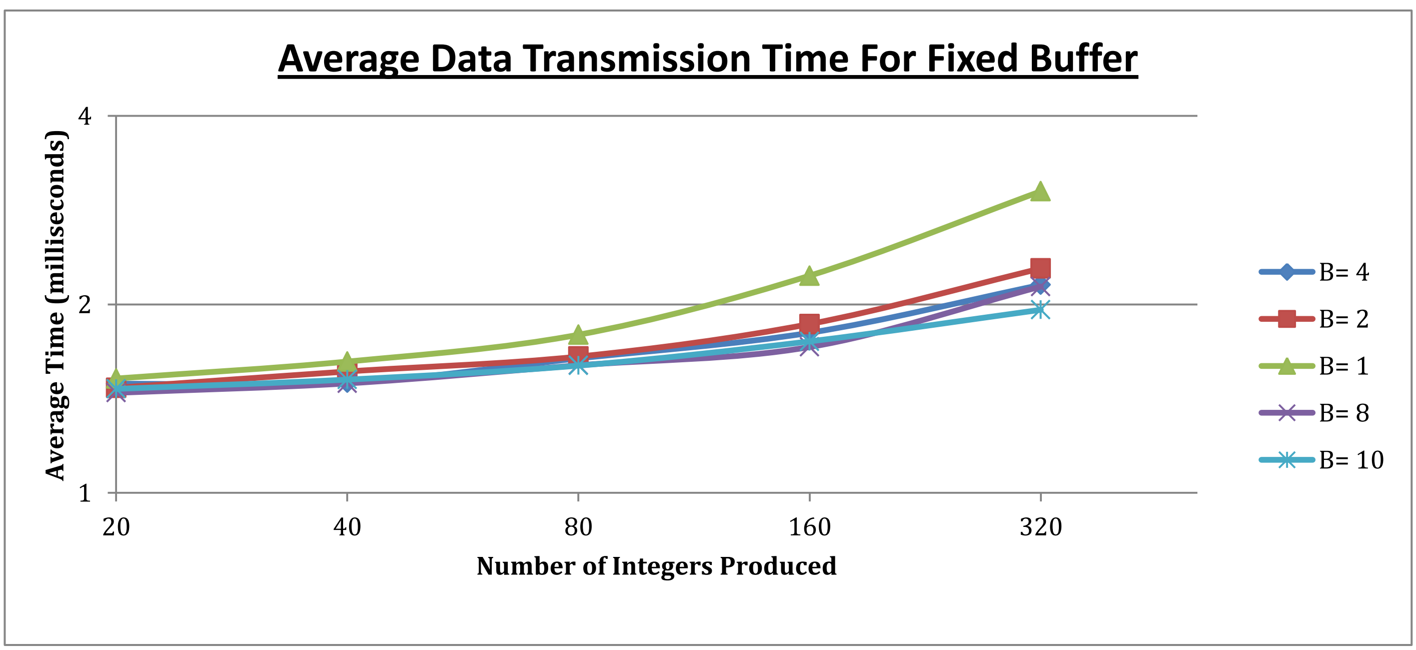


Figure 1.3: Log Log plot of average data transmission time for a fixed buffer size and varying integers produced. As seen by the graph for large values of N the increased buffer size results in a small transmission time.

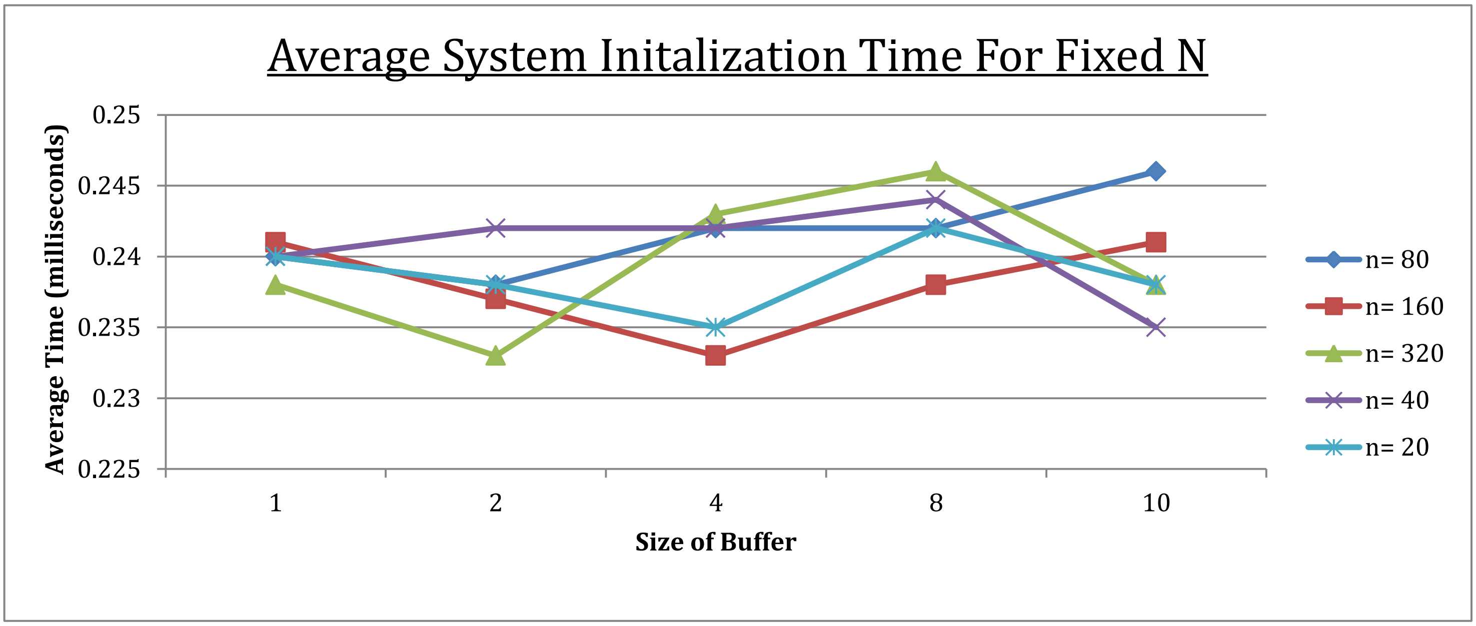


Figure 1.4: The plot of the average initialization time for a fixed N shows no apparent pattern. It should be noted that all the values are approximately 0.2 milliseconds. From this we can conclude that a variation on the buffer size and on the number of integers produced has no effect on the system initialization time. *There is no correlation between N, B and the initialization time.*

Average Data Transmission Time Histograms:

For generating the histograms we have used milliseconds to allow a finer granularity of measurement as opposed to seconds. These histograms represent timing data when N = 320 and B = 10.

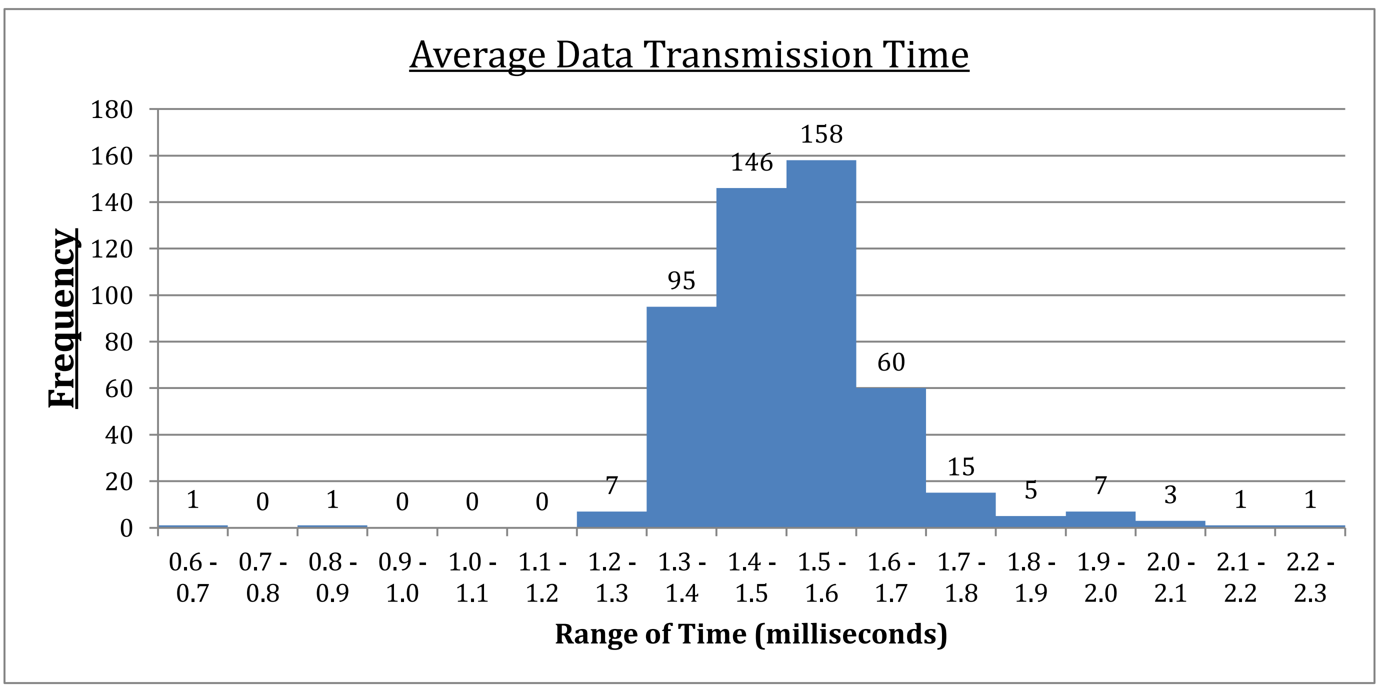
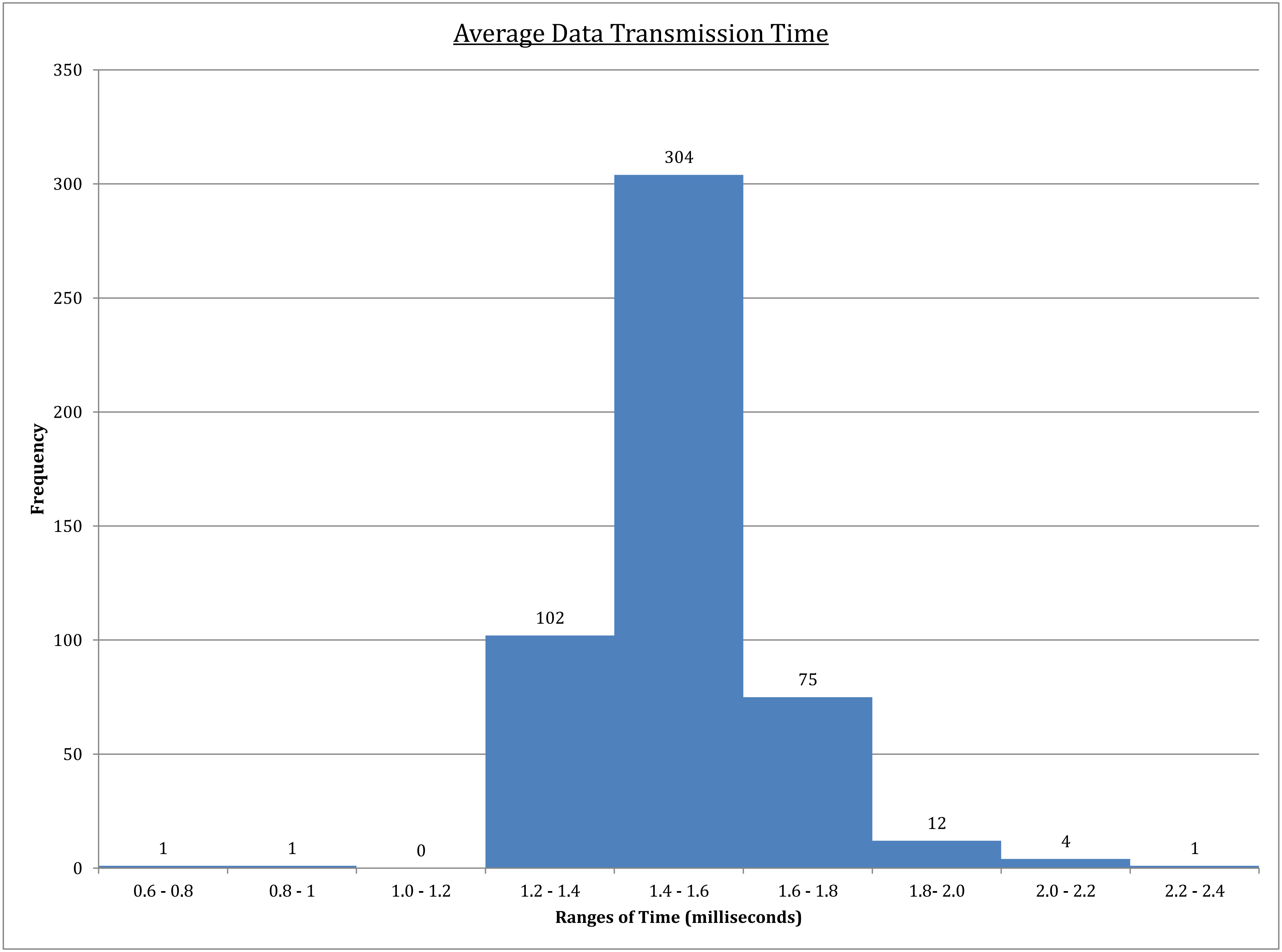


Figure 1.1: Average Data Transmission time using bin size of 0.1 ms for finer granularity.



<u>Figure 1.2:</u> Average Data Transmission time with a bin size of 0.2 ms.

Discussion:

For small values of N the change in buffer size does not have an immediately observable impact on the average data transmission time. For small values of N the change in data transmission time is relatively small, this is because the producer and consumer are still taking turns inserting and taking an element from the queue. This is because as soon as the queue becomes full the producer blocks and then the consumer must consume. For larger queue sizes (larger values of B) the queue takes longer to become full this means the producer will not be blocked as frequently from adding to the queue.

In general the following relationships hold:

- For a fixed value of B as N increases the data transmission time increase. This may not be 100% visible as the variance in interleaving and the IO wait of printf skews the results but it can still be seen. This is because when the queue becomes full the producer blocks until the consumer consumes and when the queue is empty the consumer blocks until the queue has an item on it. This causes many blocks and context switches which causes a lot of CPU overhead and excess time. For example the table above shows this.

N\B	1
20	0.001523s
40	0.001621s
80	0.001789s
160	0.002223s
320	0.003032s

Data Transmission Time Increases

- For a fixed value of N as B increases the average data transmission time decreases. This again is more apparent for large values of N such as N =320 and is because more items can fit and the queue and so the producer will block less because the queue will take longer to become full. More data can be transmitted at once and there are less blocks and context switches as a result. This is shown for the case of N = 320 and when B is varied as shown below.

N\B	1	2	4	8	10
320	0.003010	0.002234	0.002114	0.002098	0.001960

Data Transmission Time Decreases

- The average system initialization time remains constant regardless of changes in N or B as the system initialization time is simply the time taken to fork the consumer process and is therefore independent of N and B. Therefore the system initialization time is simply a fixed constant relative to the machine it is being run on.

It can be stated that from the timing analysis we have found that the values of N and B play no role in the system initialization time, but play an important role in the data transmission time. Since send and receive are blocking calls a small queue size (value of B) would mean that the queue would get full quickly and get empty quickly. This means that both the consumer and producer will be blocked frequently and this will increase the time significantly. So if you increase the queue size/value of B we have seen that the average data transmission time is reduced. One of the troubles with measurement in this lab is that the process interleaving is unpredictable and so sometimes the values are slightly higher than we might expect. Also no 2 runs of the program are 100% identical due to the variability. In future labs we hope to perform better timing analysis.

Source code listing:

```c
/*
 * common.h
 *
 * ECE254 Group 01
 * By :  Rushan Yogaratnam and Ameen Patel
 * University of Waterloo Computer Engineering
 * Spring 2014
 *
 * The purpose of this header file is to
 * declare common constants and functions
 * that are used between both the producer and the consumer.
 */

#ifndef COMMON_H_
#define COMMON_H_

//global queue_name
extern const char* queue_name;

#endif
```

```c
/*
 * common.c
 *
 * ECE254 Group 01
 * By :  Rushan Yogaratnam and Ameen Patel
 * University of Waterloo Computer Engineering
 * Spring 2014
 *
 * Implementation of common.h
 *
 */

#include <mqueue.h>

//constant queue_name for both producer and consumer.
const char* queue_name = "/mailbox_ece254_ryo";
```

```c
/*
 * Producer.c
 * ECE254 Group 01
 * By :  Rushan Yogaratnam and Ameen Patel
 * University of Waterloo Computer Engineering
 * Spring 2014
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
#include <sys/time.h>
#include <time.h>
#include <string.h>

//Include helper functions and global constants
#include "producer.h"
#include "common.h"

int main(int argc, char **argv) {

    int queue_size;
    int production_count;

    //check if the arguments are valid if so set them
    if (process_arguments(argc, argv, &queue_size, &production_count))
{
        printf("Invalid arguments\n");
        return 1;
    }

    //create queue attributes
    struct mq_attr queue_attributes;
    queue_attributes.mq_maxmsg = queue_size;
    queue_attributes.mq_msgsize = sizeof(int);
    queue_attributes.mq_flags = 0;

    mqd_t queue_descriptor;
    mode_t permissions = S_IRUSR | S_IWUSR;

    //attempt to open the queue
    queue_descriptor = mq_open(queue_name, O_RDWR | O_CREAT,
permissions,
            &queue_attributes);

    //check if the queue couldn't be opened
    if (queue_descriptor == -1) {
        printf("error creating the queue %s\n", strerror(errno));
        return 1;
```

```
    }

    double time_before_fork;

    //attempt to spawn consumer and set the time before forking
    //proceed only if the forking is successful
    if (spawn_consumer("consumer", argv, queue_descriptor,
&time_before_fork)
            != -1) {
        double time_before_first_int = get_time_in_seconds();

        produce_and_send_elements(production_count, queue_descriptor);

        //wait for the consumer to consume all the elements.
        wait_on_child(time_before_fork, time_before_first_int);
    }

    //Close and mark the queue for deletion
    if (mq_close(queue_descriptor) == -1) {
        perror("mq_close failed");
        exit(2);
    }
    //mark queue for deletion.
    if (mq_unlink(queue_name) != 0) {
        perror("mq_unlink failed");
        exit(3);
    }

    return 0;

}

int process_arguments(int argc, char* argv[], int * queue_size,
        int * production_count) {

    if (argc < 3) {
        return 1;
    } else {
        *production_count = atoi(argv[1]);
        *queue_size = atoi(argv[2]);

        return ((*production_count <= 0 || *queue_size <= 0)) ? 1 : 0;
    }

}

int spawn_consumer(char* program, char **arg_list, mqd_t
queue_descriptor,
        double * fork_time) {

    arg_list[0] = program;
    pid_t child_pid;

    //get the time before fork.
    *fork_time = get_time_in_seconds();
    child_pid = fork();
```

```
    if (child_pid > 0) {
        return child_pid;
    } else if (child_pid < 0) {
        printf("error creating the child process %s\n",
strerror(errno));
        return -1;
    } else {
        execvp("./consume", arg_list);
        printf("error occurred in execvp %s\n", strerror(errno));
        abort();
    }

}

void produce_and_send_elements(int process_count, mqd_t
queue_descriptor) {

    srand(time(0));
    int i;

    for (i = 1; i <= process_count; ++i) {

        int message = ((rand() % 80));

        if (mq_send(queue_descriptor, (char*) &message, sizeof(int), 0)
== -1) {
            printf("error sending the message %s\n", strerror(errno));
        }
    }
}

int wait_on_child(double time_before_fork, double time_after_fork) {

    int child_status;
    wait(&child_status);

    if (WIFEXITED(child_status)) {

        double time_after_last_consumed = get_time_in_seconds();

        printf("Time to initialize system: %f seconds\n",
                time_after_fork - time_before_fork);
        printf("Time to transmit data: %f seconds\n",
                time_after_last_consumed - time_after_fork);
        return 0;

    } else {
        printf("child process exited abnormally \n");
        return 1;
    }

}
double get_time_in_seconds() {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (tv.tv_sec + tv.tv_usec / 1000000.0);
```

```c
}

/*
 * Producer.h
 * ECE254 Group 01
 * By :  Rushan Yogaratnam and Ameen Patel
 * University of Waterloo Computer Engineering
 * Spring 2014
 *
 *
 *      These are a bunch of helper functions for the
 *      producer.c file. The are just functions which
 *      make the code cleaner by providing abstraction.
 *
 */

#ifndef PRODUCER_H_
#define PRODUCER_H_


//This method provides a convenient way to get the current time in
seconds
double get_time_in_seconds();

//this method causes the consumer to wait on the child and
//once the child process is down waiting prints the initialization and
//data transmission times.
int wait_on_child(double time_before_fork, double time_after_fork);


//Provider error checking on the command line arguments,
// if they are invalid -1 is return. The last two parameters
//are pointers to queue size and process count, these are set
//if valid
int process_arguments(int, char**, int *, int *);

//spawns the child process and sets the time before forking,
//which is the last parameter.
int spawn_consumer(char*, char **,mqd_t ,double *);

//The main part of the producer code
//this creates elements and sends them to the msg queue.
void produce_and_send_elements(int, mqd_t);

#endif /* PRODUCER_H_ */
```

```c
/*
 * consumer.c
 * ECE254 Group 01
 * By :  Rushan Yogaratnam and Ameen Patel
 * University of Waterloo Computer Engineering
 * Spring 2014
 *
 */
#include <mqueue.h>
#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include "common.h"

int main(int argc, char **argv) {

    int queue_size = atoi(argv[2]);
    int messages_to_consume = atoi(argv[1]);

    struct mq_attr queue_attributes;
    queue_attributes.mq_maxmsg = queue_size;
    queue_attributes.mq_msgsize = sizeof(int);
    queue_attributes.mq_flags = 0;

    mqd_t queue_descriptor;
    mode_t permissions = S_IRUSR | S_IWUSR;

    //attempt to open the queue
    queue_descriptor = mq_open(queue_name, O_RDONLY, permissions,
            &queue_attributes);

    //error checking on the queue
    if (queue_descriptor == -1) {
        printf("there was an error opening the queue in the consumer");
        printf("the error is %s \n", strerror(errno));
        return 1;
    }

    int i;
    for (i = 1; i <= messages_to_consume; ++i) {
        int message;

        if (mq_receive(queue_descriptor, (char*) &message, sizeof(int),
                0) == -1) {
            printf("failed to receive message %s \n", strerror(errno));
            return 1;

        } else {
            //Note that this printf causes an IO wait and slows down
            //the timing results.
            printf("%i is consumed\n", message);
```

```
        }
    }

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

    return 0;

}
```