CS402 Final Project Report of **James Watson**

Supervisor: **Weiping Zhu**

Scheduling Soft Real-Time Tasks in Clusters

Submitted to the School of Computer Science and Electrical Engineering

November 2000

Abstract

1 Introduction

2 Simulation Description

In order to test the soft real-time performance of the various scheduling policies in a real-world environment, a simulation of the distributed model presented in [1] was created and run with various parameters.

2.1 Technology

The simulation was written and run on a Sun Sparc, compiled with GCC 2.95.2. The light-weight communication library, 'com2', (written by David Leonard of the School of Computer Science and Electrical Engineering), was used as the code's communication framework.

2.2 Model Description

The simulation was composed of several communicating processes: the Task Generator, the Dispatcher, the Queue Manager and various numbers of Servers. This section gives a detailed description of the behaviour of each.

2.2.1: Task Generator

The Task Generator program is executed from the command line with the following parameters:

input-file
total-jobs
exec-rate
disp-rate
sl-min
sl-max

where:

- *input-file* is the web log used for job arrival and execution time,
- *total-jobs* is the total number of jobs to be sent from this web log,
- *exec-rate* is the execution time coefficient (the value with which execution times were multiplied to allow control over workload),
- *disp-rate* is the dispatch time coefficient (the value with which dispatch times from the web log were multiplied to allow control over the time between dispatched tasks,
- *sl-min* is the minimum value of the uniform distribution of possible slack time values,
- and *sl-max* is the maximum value of the uniform distribution of possible slack time values.

The Task Generator first looks up the network address of the Dispatcher module, sending it jobs in the form of:

job_id
execution_time
deadline

The time of job dispatch calculated by an “virtual clock”. When the virtual clock’s value was greater than or equal to the next calculated dispatch time (taken from the web log and multiplied by the disp-rate),

2.2.2: Dispatcher
2.2.3: Queue Manager
2.2.4: Servers

3 Experiments

4 Results

5 Discussion

6 Conclusion

7 References

[1] Zhu, W. & Fleisch, B.D. (2000). Performance evaluation of soft real-time scheduling for multicomputer cluster. In Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000).

8 Appendix A – Source Code

The source code of the distributed simulation is included below. Section A.1 the source code of the Task Generator module, Section A.2 the Dispatcher module, Section A.3 the Queque Manager and Section A.4 the details of the Servers.

The basic design of the simulation was thus: the Task Generator read job dispatch and execution times from a web log, calculated the job's deadline, and sent the deadline and execution time to the Dispatcher at the correct arrival time. The Dispatcher was then responsible for sending jobs to the various servers in the server pool (corresponding to the scheduling policy specified on the command line). The Queque Manager stored the number of tasks in each server's queue. This information was used by the dispatcher when the 'Join Shortest Queque' policy was specified. After Servers were sent job information from the dispatcher, they delayed for the specified execution time and then used the system clock to determine of the deadline had been met or not.

A.1: Task Generator

```
/* written for the distributed simulation to evaluate
   real time performance of dispatching policies in a
   distributed architecture.

   James Watson, 2000

   Some code adapted from work by David Leonard


   task-generator.c - this program generates tasks for the
   simulation at a rate specified as its first parameter.
   Registers itself as 'task-generator', finds 'dispatcher',
   and sends the task to dispatcher.  Also sends the real-time
   deadline (in terms of system clock) with the task

 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include "cs334.h"

/* function which sends the task to the dispatcher */
void send_to_dispatcher(int, int, int, int, int);

struct sockaddr_in dispatch_addr;

FILE *ifp;  /* global so that all functions can access */

int main(int argc, char **argv)
{
    int endpoint, total_jobs;
    int exec_coeff, disp_coeff, sl_min, sl_max;

            /* check the parameters */
            if (argc != 7)
            {
                    printf("\nUsage:  %s in-file total-jobs ", argv[0]);
```

```
                printf("exec-rate disp-rate sl-min sl-max\n\n");
                printf("Where exec-rate is execution time coefficient,\n");
                printf("disp-rate is the dispatch time coefficient,\n");
                printf("sl-min is min boundary for slack distribution,\n");
                printf("and sl-max is maximum boundary value\n\n\n");
                exit(1);
        }

        /* open up the input file read-only*/
        if ((ifp = fopen(argv[1], "r")) == NULL)
        {
            printf("\nError reading the input file specified\n\n");
            exit(1);
        }

        /* get rest of the parameters */
        total_jobs = atoi(argv[2]);
        exec_coeff = atoi(argv[3]);
        disp_coeff = atoi(argv[4]);
        sl_min = atoi(argv[5]);
        sl_max = atoi(argv[6]);

        if (total_jobs <= 0)
        {
            printf("\nPlease use a more sensible number of jobs\n\n");
            exit(1);
        }

        printf("\nThe total number of jobs to send is:  %d\n", total_jobs);

        /* Create an endpoint */
        if ((endpoint = udp_socket()) == -1)
        {
            perror("udp_socket");
            exit(1);
        }

        /* Find the address of the dispatcher */
        printf("\n(task-generator)\tlooking up dispatcher\n");
        if (ns_lookup("dispatcher", &dispatch_addr) == -1)
        {
                        perror("ns_lookup");
                        exit(1);
        }

        /* send tasks at specified intervals to dispatcher */
        send_to_dispatcher(total_jobs, exec_coeff, disp_coeff, sl_min, sl_max);


        /* Close the endpoint down for a graceful exit */
        close(endpoint);

        /* close the input file */
        fclose(ifp);

        return 0;
}


void send_to_dispatcher(int num_jobs, int exec_rate, int disp_rate,
                        int sl_min, int sl_max)
{
        int temp_s, i, job_counter;
        int dispatch_time, execution_time;  /* variables taken from file */
        time_t system_time;
        time_t job_deadline;
        int virtual_clock = 0;
```

```
int read_this_job = 1;   /* 0 means true, 1 means false */
int slack_time, rand_num;

/* make a struct to be sent to the dispatcher */
typedef struct{
    int job_id;
    int exec_time;
    time_t deadline;
} to_send;

/* make an array of jobs to send */
to_send jobs_to_send[num_jobs];

/* give an ID to each job */
for (i = 0; i < num_jobs; i++)
{
    jobs_to_send[i].job_id = i;
}

/* Create a temporary socket */
if ((temp_s = udp_socket()) == -1)
{
                        perror("udp_socket");
                        exit(1);
}

/* send the jobs */
job_counter = 0;
while (job_counter < num_jobs)
{
    if (read_this_job == 1)
    {
            /* get dispatch time from modified web log */
            fscanf(ifp, "%d%d", &dispatch_time, &execution_time);
            read_this_job = 0;
    }

    /* modify the values according to parameters -
      input files are very small (ie 7ms, 100ms / 10s, 11s)
            so integer multiplication is acceptable */
    dispatch_time = dispatch_time * disp_rate;
    execution_time = execution_time * exec_rate;

    if (virtual_clock >= dispatch_time)
    {
            jobs_to_send[job_counter].exec_time = execution_time;

            /* calculate slack time for job deadline -
              uses uniform distribution between sl_min & sl_max */
            rand_num = rand();

            /* due to the conversion of floats to ints (ie
              9.9999 converts to 9), the chance of getting
              an int of 1 with rand()/RAND_MAX is smaller
              than any other value (from 0-9).  So, we make the
              maximum 1 greater and don't allow a random number
              equal to RAND_MAX to be used, to keep a uniform
              distribution */
            /* the modulus operator cannot be used here because if the
              value of (max-min) does not divide equally into
              RAND_MAX, some numbers will have a greater (but not
              much) chance of occurring, and we need a uniform
              distribution */
            while (rand_num == RAND_MAX)
            {
                rand_num = rand();
            }
```

```
                slack_time = sl_min + ((float)rand_num / (float)RAND_MAX) *
                    ((sl_max + 1) - sl_min);

                /* get system time to calculate deadline */
                time(&system_time);

                /* calculate and set what the deadline will be
                   (execution time(ms) + system_time) */
                job_deadline = system_time + (execution_time / 1000);
                job_deadline = job_deadline + slack_time;
                jobs_to_send[job_counter].deadline = job_deadline;

                /* make asynchronous call */
                printf("(task-gen)\tsending job:%d,exec:%d,at time:%ld\n",
                        job_counter, execution_time, system_time);
                if (sync_sendto(temp_s, &jobs_to_send[job_counter],
                                sizeof jobs_to_send[job_counter],
                                &dispatch_addr) == -1)
                {
                    { perror("sync_sendto"); exit(1); }
                }

                job_counter++;
                read_this_job = 1;
            }

            /* update the virtual time counter with a second's delay */
            sleep(1);
            virtual_clock++;
        }

        /* Get rid of temp socket */
        close(temp_s);

}
```

A.2: Dispatcher

```
/* written for the distributed simulation to evaluate
   real time performance of dispatching policies in a
   distributed architecture.

   James Watson, 2000

   Some code adapted from work by David Leonard


   dispatcher.c - this process dispatches jobs to the pool
   of servers according to the policy specified as its 2nd parameter,
   to the number of servers specified in its 1st

 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include "cs334.h"
#include <signal.h>
#include <time.h>

#define SERVER_NAME_LENGTH 20
#define FILENAMELENGTH 20
#define SERVER_NUM_LENGTH 4   /* wouldn't expect more than 10000 servers 8) */
#define COMMANDLENGTH 80

/* function prototypes */
void rr_dispatcher(void);
void jsq_dispatcher(void);
void balance_dispatcher(void);
void sigint(int);
void unreg_self(void);
void handler(int, void *, size_t, int, struct sockaddr_in *);
void get_lock(char *);
void free_lock(char *);

int usleep(unsigned int);

int num_servers;

int main(int argc, char **argv)
{
        const char *round_robin = "RR";
        const char *join_shortest_queue = "JSQ";
        const char *balance = "Balance";

        /* check the parameters */
        if (argc != 3)
        {
            printf("\nUsage:  %s num-servers dispatching-policy\n\n",
                    argv[0]);
            exit(1);
        }

        num_servers = atoi(argv[1]);
        if (num_servers <= 0)
        {
            printf("\nNeed to specify a more useful number of servers\n\n");
            exit(1);
        }
```

```
        /* do an strcmp between argv and the allowable policies */
        if (strcmp(argv[2], round_robin) == 0)
        {
            rr_dispatcher();
        }

        else if (strcmp(argv[2], join_shortest_queue) == 0)
        {
            jsq_dispatcher();
        }

        else if (strcmp(argv[2], balance) == 0)
        {
            balance_dispatcher();
        }

        else
        {
                printf("\nSpecified policy not valid.  Valid policies are:\n");
                printf("\tRR   (round robin)\n");
                printf("\tJSQ  (join shortest queue)\n");
                printf("\tBalance (random)\n\n");
                exit(1);
        }

        return 0;
}


/* dispatcher that uses the round robin policy to send jobs
   to the server pool
 */
void rr_dispatcher(void)
{
    char next_server[SERVER_NAME_LENGTH];
    int the_endpoint, len, server_counter, job_counter;
    struct sockaddr_in disp_addr, taskgen_addr;
    typedef struct{
            int job_id;
            int exec_time;
            time_t deadline;
    } from_taskgen;
    from_taskgen buffer;

    /* register dispatcher */
    if ((the_endpoint = udp_socket()) == -1)
    {
            perror("udp_socket");
            exit(1);
    }

    if (udp_sockname(the_endpoint, &disp_addr) == -1)
    {
            perror("udp_sockname");
            exit(1);
    }


    /* Register self's address with the name 'dispatcher' */
    if (ns_register("dispatcher", &disp_addr) == -1)
    {
            perror("ns_register");
            exit(1);
    }

    /* Automatically de-register self on exit or control-C.
            code modified from David Leonard */
```

```
    atexit(unreg_self);
    signal(SIGINT, sigint);

    /* Go into infinite loop ready to receive messages from task-generator */
    job_counter = 0;
    server_counter = 1;
    for (;;)
    {
            int temp_s;
            struct sockaddr_in nserver_addr;

            len = sync_recvfrom(the_endpoint, &buffer, sizeof buffer,
                                    &taskgen_addr);
            if (len == -1)
            { perror("sync_recvfrom"); exit(1);}
            job_counter++;
            server_counter = server_counter % num_servers + 1;
            printf("(Dispatcher)\treceived job:%d,id:%d,exec:%d,deadline:%ld\n",
                    job_counter,buffer.job_id,buffer.exec_time,buffer.deadline);

            /* determine who the next server is */
            sprintf(next_server, "server-%d", server_counter);

            /* send the job to the next server */
            printf("(Dispatcher)\tsending job %d to %s\n",
                    job_counter, next_server);

            /* make a connection to the next server in the pool */
            if ((temp_s = udp_socket()) == -1)
            {
                perror("udp_socket");
                exit(1);
            }

            if (ns_lookup(next_server, &nserver_addr) == -1)
            {
                perror("ns_lookup");
                exit(1);
            }

            /* send the job */
            if (sync_sendto(temp_s, &buffer, sizeof buffer, &nserver_addr) == -1)
            {
                { perror("sync_sendto"); exit(1); }
            }

            /* close the connection to the server */
            close(temp_s);
    }

}


/* Dispatcher that uses the join shortest queue policy to send
   jobs to the server pool.  Uses the queque_manager to keep
   track of how many tasks are quequed for each server
*/
void jsq_dispatcher(void)
{
    char next_server[SERVER_NAME_LENGTH];
    char server_num_string[SERVER_NUM_LENGTH];
    int the_endpoint, len, server_index, job_counter, man_s;
    struct sockaddr_in disp_addr, taskgen_addr, man_addr;
    typedef struct{
            int job_id;
            int exec_time;
            time_t deadline;
```

```
} from_taskgen;
from_taskgen buffer;

typedef struct{
        int server_num;
        char the_action;
} to_queque_manager;
to_queque_manager man_buffer;

/* register dispatcher */
if ((the_endpoint = udp_socket()) == -1)
{
        perror("udp_socket");
        exit(1);
}

if (udp_sockname(the_endpoint, &disp_addr) == -1)
{
        perror("udp_sockname");
        exit(1);
}


/* Register self's address with the name 'dispatcher' */
if (ns_register("dispatcher", &disp_addr) == -1)
{
        perror("ns_register");
        exit(1);
}

/* Automatically de-register self on exit or control-C.
        code modified from David Leonard */
atexit(unreg_self);
signal(SIGINT, sigint);


/* open a connection to the queque_manager */
if ((man_s = udp_socket()) == -1)
{
        perror("udp_socket");
        exit(1);
}

if (ns_lookup("queque_manager", &man_addr) == -1)
{
        perror("ns_lookup");
        exit(1);
}

/* Go into infinite loop ready to receive messages from task-generator */
job_counter = 0;
for (;;)
{
        int temp_s, i, lowest, current_tasks;
        struct sockaddr_in nserver_addr;
        int queque_of[num_servers];
        char current_queque_name[FILENAMELENGTH];
        FILE *current_queque;
        int minimum_queque[num_servers];   /* to allow random allocation */
        int num_min_queques;               /* in case of ties */
        int rand_num;

        len = sync_recvfrom(the_endpoint, &buffer, sizeof buffer,
                                &taskgen_addr);
        if (len == -1)
        { perror("sync_recvfrom"); exit(1);}
        job_counter++;
```

```
printf("(Dispatcher)\treceived job:%d,id:%d,exec:%d,deadline:%ld\n",
            job_counter,buffer.job_id,buffer.exec_time,buffer.deadline);


/* get the tasks in each server's queque */
for (i = 0; i < num_servers; i++)
{
    strcpy(current_queque_name, "");
    strcat(current_queque_name, "queques/server-");
    sprintf(server_num_string, "%d", (i+1));
    strcat(current_queque_name, server_num_string);
    strcat(current_queque_name, ".queque.tmp");

    if ((current_queque = fopen(current_queque_name, "r")) == NULL)
    {
            printf("(Dispatcher)\tFile IO error: %s\n",
                    current_queque_name);
            exit(1);
    }

    fscanf(current_queque, "%d", &current_tasks);
    fclose(current_queque);

    queque_of[i] = current_tasks;
}

lowest = queque_of[0];

/* work out what is the least tasks quequed */
for (i = 0; i < num_servers; i++)
{
    if (queque_of[i] < lowest)
    {
            lowest = queque_of[i];
    }
}

/* solve ties by randomly allocating job to one of the servers
   with minimum tasks in queque */
num_min_queques = 0;
for (i = 0; i < num_servers; i++)
{
    if (queque_of[i] == lowest)
    {
            minimum_queque[num_min_queques] = i;
            num_min_queques++;
    }
}

/* randomly pick the server the tasks is to go to */
rand_num = rand();
rand_num = rand_num % num_min_queques;

server_index = minimum_queque[rand_num];
server_index++;  /* cause servers are indexed from 1 */


/* send the job to the right server */
sprintf(next_server, "server-%d", server_index);
printf("(Dispatcher)\tsending job %d to %s\n",
            job_counter, next_server);

/* add to the stored queque tally */
man_buffer.server_num = server_index;
man_buffer.the_action = 'a';
if (sync_sendto(man_s, &man_buffer, sizeof man_buffer,
                        &man_addr) == -1)
```

```
            {
                { perror("sync_sendto"); exit(1); }
            }


            /* make a connection to the server */
            if ((temp_s = udp_socket()) == -1)
            {
                perror("udp_socket");
                exit(1);
            }

            if (ns_lookup(next_server, &nserver_addr) == -1)
            {
                perror("ns_lookup");
                exit(1);
            }

            /* send the job */
            if (sync_sendto(temp_s, &buffer, sizeof buffer, &nserver_addr) == -1)
            {
                { perror("sync_sendto"); exit(1); }
            }

            /* close the connection to the server */
            close(temp_s);
    }

    /* close the connection to the queque_manager */
    close(man_s);

}


/* dispatcher that uses the balance policy to allocate jobs to servers
   in the server pool (this policy is an equal probability random allocator) */
void balance_dispatcher(void)
{
    char next_server[SERVER_NAME_LENGTH];
    int the_endpoint, len, server_index, job_counter;
    struct sockaddr_in disp_addr, taskgen_addr;
    typedef struct{
            int job_id;
            int exec_time;
            time_t deadline;
    } from_taskgen;
    from_taskgen buffer;

    /* register dispatcher */
    if ((the_endpoint = udp_socket()) == -1)
    {
            perror("udp_socket");
            exit(1);
    }

    if (udp_sockname(the_endpoint, &disp_addr) == -1)
    {
            perror("udp_sockname");
            exit(1);
    }

    /* Register self's address with the name 'dispatcher' */
    if (ns_register("dispatcher", &disp_addr) == -1)
    {
            perror("ns_register");
            exit(1);
    }
```

```
    /* Automatically de-register self on exit or control-C.
            code modified from David Leonard */
    atexit(unreg_self);
    signal(SIGINT, sigint);

    /* Go into infinite loop ready to receive messages from task-generator */
    job_counter = 0;
    for (;;)
    {
            int temp_s, rand_num;
            struct sockaddr_in nserver_addr;

            len = sync_recvfrom(the_endpoint, &buffer, sizeof buffer,
                                    &taskgen_addr);
            if (len == -1)
            { perror("sync_recvfrom"); exit(1);}
            job_counter++;
            printf("(Dispatcher)\treceived job:%d,id:%d,exec:%d,deadline:%ld\n",
                    job_counter,buffer.job_id,buffer.exec_time,buffer.deadline);

            /* determine who the next server is:
               get random number between 1 and num_servers (inclusive) */
            rand_num = rand();
            rand_num = rand_num % num_servers;
            rand_num++;

            /* send the job to the random server */
            server_index = rand_num;
            sprintf(next_server, "server-%d", server_index);
            printf("(Dispatcher)\tsending job %d to %s\n",
                    job_counter, next_server);

            /* make a connection to the next server in the pool */
            if ((temp_s = udp_socket()) == -1)
            {
                perror("udp_socket");
                exit(1);
            }

            if (ns_lookup(next_server, &nserver_addr) == -1)
            {
                perror("ns_lookup");
                exit(1);
            }

            /* send the job */
            if (sync_sendto(temp_s, &buffer, sizeof buffer, &nserver_addr) == -1)
            {
                { perror("sync_sendto"); exit(1); }
            }

            /* close the connection to the server */
            close(temp_s);
    }
}

/* On control-C, we exit cleanly. */
void sigint(sig)
int sig;
{
    exit(1);
}

/* Unregister the dispatcher. */
void unreg_self()
{
```

```
    ns_unregister("dispatcher");
}
```

A.3: Queue Manager

```
/* queque manager - (written by James Watson)

   Since there are concurrency issues with the dispatcher and servers
   editing the queque tallys simultaneously, need to have a process
   which has sole control of this data.

   Writes the number of quequed tasks to file which are read by the
   dispatcher (doesn't send messages back to dispatcher since these
   messages get confused with messages from task generator).

   Used by the JSQ policy of the dispatcher.

*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include "cs334.h"
#include <signal.h>

#define FILENAMELENGTH 80
#define COMMANDLENGTH 60
#define SERVERNUMSIZE 4

/* function prototypes */
void sigint(int);
void unreg_self(void);
void handler(int, void *, size_t, int, struct sockaddr_in *);
void accept_messages(int);

int usleep(unsigned int);

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

    /* check the parameters */
    if (argc != 2)
    {
            printf("\nUsage:  %s numservers\n\n", argv[0]);
            exit(1);
    }

    num_servers = atoi(argv[1]);
    if (num_servers <= 0)
    {
            printf("\nNeed to specify a more realistic number of servers\n\n");
            exit(1);
    }

    /* Automatically de-register self on exit or ctrl-c.
           Code modified from David Leonard */
    atexit(unreg_self);
    signal(SIGINT, sigint);

    /* go into an infinite loop to keep track of quequed tasks */
    accept_messages(num_servers);

    return 0;
}
```

```
/* function which accepts the messages from dispatcher and servers
   and keep track of the tasks quequed for each server */
void accept_messages(int num_of_servers)
{
    int the_endpoint, len, i;
    int this_queque_tally;
    struct sockaddr_in man_addr, sender_addr;
    FILE *current_queque;
    char current_queque_name[FILENAMELENGTH];
    char server_num_string[SERVERNUMSIZE];

    typedef struct{
            int server_num;
            char the_action;   /* 'r' = remove, 'a' = add */
    } from_sender;
    from_sender buffer;

    /* register the queque manager */
    if ((the_endpoint = udp_socket()) == -1)
    {
            perror("udp_socket");
            exit(1);
    }

    if (udp_sockname(the_endpoint, &man_addr) == -1)
    {
            perror("udp_sockname");
            exit(1);
    }

    /* register self's address with name 'queque_manager' */
    if (ns_register("queque_manager", &man_addr) == -1)
    {
            perror("ns_register");
            exit(1);
    }

    /* initialize the queque files */
    printf("(Queque_Manager)\tcreating queque files.....\n");
    for (i = 0; i < num_of_servers; i++)
    {
            strcpy(current_queque_name, "");
            strcat(current_queque_name, "queques/server-");
            sprintf(server_num_string, "%d", i+1);
            strcat(current_queque_name, server_num_string);
            strcat(current_queque_name, ".queque.tmp");

            if ((current_queque = fopen(current_queque_name, "w")) == NULL)
            {
                printf("(Queque_Manager)\tFile IO Error: %s\n\n",
                        current_queque_name);
                exit(1);
            }
            fprintf(current_queque, "%d\n", 0);
            fclose(current_queque);
    }

    /* go into infinite loop to accept and process messages */
    for (;;)
    {
            len = sync_recvfrom(the_endpoint, &buffer, sizeof buffer,
                                    &sender_addr);
            if (len == -1)
            { perror("sync_recvfrom"); exit(1);}

            printf("(Queque_Manager)\treceived %c for server %d\n",
                    buffer.the_action, buffer.server_num);
```

```
            if (buffer.the_action == 'r')   /* remove task from queque */
            {
                strcpy(current_queque_name, "");
                strcat(current_queque_name, "queques/server-");
                sprintf(server_num_string, "%d", buffer.server_num);
                strcat(current_queque_name, server_num_string);
                strcat(current_queque_name, ".queque.tmp");

                if ((current_queque = fopen(current_queque_name, "r+")) == NULL)
                {
                        printf("(Queque_Manager)\tFile IO Error: %s\n\n",
                                current_queque_name);
                        exit(1);
                }
                fscanf(current_queque, "%d", &this_queque_tally);
                this_queque_tally--;
                fseek(current_queque, 0, SEEK_SET);
                fprintf(current_queque, "%d\n", this_queque_tally);
                fclose(current_queque);
            }

            if (buffer.the_action == 'a')  /* add task to queque */
            {
                strcpy(current_queque_name, "");
                strcat(current_queque_name, "queques/server-");
                sprintf(server_num_string, "%d", buffer.server_num);
                strcat(current_queque_name, server_num_string);
                strcat(current_queque_name, ".queque.tmp");

                if ((current_queque = fopen(current_queque_name, "r+")) == NULL)
                {
                        printf("(Queque_Manager)\tFile IO Error: %s\n\n",
                                current_queque_name);
                        exit(1);
                }
                fscanf(current_queque, "%d", &this_queque_tally);
                this_queque_tally++;
                fseek(current_queque, 0, SEEK_SET);
                fprintf(current_queque, "%d\n", this_queque_tally);
                fclose(current_queque);
            }

    }
}


/* On control-C, we exit cleanly. */
void sigint(sig)
int sig;
{
    char command[COMMANDLENGTH];
    printf("(Queque_Manager)\tremoving temp files\n\n");
    strcpy(command, "rm queques/*.tmp");
    system(command);
    exit(1);
}


/* Unregister the queque_manager. */
void unreg_self()
{
    ns_unregister("queque_manager");
}
```

A.4: Servers

```
/* written for the distributed simulation to evaluate
   real time performance of dispatching policies in a
   distributed architecture.

   James Watson, 2000

   Some code adapted from work by David Leonard


   server.c - registers itself under a name relating to the
   parameter specified (ie ./server 1 will make the process register as
   'server-1'.  This process gets each job and checks to see if it
   passed the deadline.  If it did, it increments a counter, if it
   didn't, increments another, so that the total processes that made
   the deadline and those that don't are calculated.


 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include "cs334.h"
#include <signal.h>
#include <time.h>

#define FILENAMELENGTH 60
#define COMMANDLENGTH 80

void sigint(int);
void unreg_self(void);
void handler(int, void *, size_t, int, struct sockaddr_in *);
void get_lock(void);
void free_lock(void);

int usleep(unsigned int);

char server_name[] = "server-";
char this_queque_name[FILENAMELENGTH];
char this_lock_name[FILENAMELENGTH];


int main(int argc, char **argv)
{
    int missed_deadlines = 0;  /* the counter of jobs that miss deadlines */
    int server_num, the_endpoint, len, man_s;
    struct sockaddr_in this_addr, dispatcher_addr, man_addr;
    typedef struct{
            int job_id;
            int exec_time;
            time_t deadline;
    } from_dispatcher;
    from_dispatcher buffer;

    typedef struct{
            int server_num;
            char the_action;
    } for_queque_manager;
    for_queque_manager man_buffer;

    /* check the parameters */
    if (argc != 2)
```

```
{
        printf("\nUsage:  %s server-num\n\n", argv[0]);
        exit(1);
}

server_num = atoi(argv[1]);

if (server_num <= 0)  /* can't have 0 - that's atoi() if no conversion */
{
        printf("\nPlease use a more sensible server number ( >0 )\n\n");
        exit(1);
}

/* make the server name */
strcat(server_name, argv[1]);

/* register the server */
if ((the_endpoint = udp_socket()) == -1)
{
        perror("udp_socket");
        exit(1);
}
if (udp_sockname(the_endpoint, &this_addr) == -1)
{
        perror("udp_sockname");
        exit(1);
}


/* Register self's address */
if (ns_register(server_name, &this_addr) == -1)
{
        perror("ns_register");
        exit(1);
}

/* Automatically de-register self on exit or control-C.
       code modified from David Leonard */
atexit(unreg_self);
signal(SIGINT, sigint);


/* open a connection to the queque_manager */
if ((man_s = udp_socket()) == -1)
{
        perror("udp_socket");
        exit(1);
}

if (ns_lookup("queque_manager", &man_addr) == -1)
{
        perror("ns_lookup");
        exit(1);
}

/* Go into infinite loop ready to receive messages from dispatcher */
for (;;)
{
        time_t system_time;

        /* receive message */
        len = sync_recvfrom(the_endpoint, &buffer, sizeof buffer,
                                &dispatcher_addr);
        if (len == -1)
        { perror("sync_recvfrom"); exit(1);}

        printf("(%s)\treceived job_id:%d,exec:%d and deadline %ld\n",
```

```
                    server_name,buffer.job_id,buffer.exec_time,buffer.deadline);

            /* delay for the execution time to simulate job working */
            printf("\tworking....");
            sleep(buffer.exec_time / 1000);
            printf("done.\n");

            /* work out if missed deadline or not */
            time(&system_time);

            if (system_time <= buffer.deadline)
            {
                printf("(%s)\tjob:%d MADE deadline -> missed deadlines: %d\n",
                        server_name, buffer.job_id, missed_deadlines);
            }
            else
            {
                missed_deadlines++;
                printf("(%s)\tjob:%d MISSED deadline -> missed deadlines: %d\n",
                        server_name, buffer.job_id, missed_deadlines);
            }

            /* tell queque_manager that another task has been done */
            man_buffer.server_num = server_num;
            man_buffer.the_action = 'r';

            /* send the job */
            if (sync_sendto(man_s, &man_buffer, sizeof man_buffer,
                                &man_addr) == -1)
            {
                { perror("sync_sendto"); exit(1); }
            }
        }
        close(man_s);

}


/* On control-C, we exit cleanly. */
void sigint(sig)
int sig;
{
    exit(1);
}

/* Unregister the server and remove temp files */
void unreg_self()
{
    ns_unregister(server_name);
}
```