Program Report

KV5002 Computer Networks, Security, and Operating Systems

ID: W20021023

# 1. The Lunar Lander Controller

## A. Threads and Semaphores

Threads allow a program to split its processes into different parts, which are executed concurrently and separately. The program uses five threads – one for the display, one for the user keyboard input, one for lander communication, one for the dashboard and one for logging data. The program uses threads, since it is not meant to be executed consecutively, for example while it waits for the user to press a key, it must still process and display the game, communicate with the server, and log data.

Since the threads of the program must share data with each other (for example the input key, received in the keyboard input managing thread, must be available for the lander managing thread to be processed and communicated with the server, or for the display thread to be shown on the screen). The communication is done through global (accessible from every thread) variables, such as “landercommand” or “landerstate”. Since the program does not have a (full) control of when is each thread being executed, this may lead to interference – two threads modifying the same variable at the same time, thus causing unwanted behaviour. This is also known as a race condition.

To prevent this, the code uses so called semaphores, a mechanism first proposed by Edsger Dijkstra in 1962. Each thread may have a critical section(s), which, once reached, must be executed consecutively, thus blocking other threads to enter their critical section(s) until it is finished. This is managed by special variables, called semaphores (initialized in main), with two functions – “sem\_wait” – queue the thread and once it is its turn start executing the critical section, and “sem\_post” – exit a critical section and unlock it to other threads. These sections are used in keyboard managing, display managing, and lander managing (implicitly through “parsecondition” and “parsestate” functions) threads. The execution of threads and the semaphore queue are controlled by the operating system.

Semaphores solve the problem of synchronization between threads. Compared to other options, they also avoid “busy wait” (repeatedly checking for a condition to be met), since the processes are queued, and thus avoid a larger overhead. The disadvantage of semaphores (and of using a queue system) is that low priority processes may block high priority processes in the execution order, this is called a priority inversion. A different approach for the program might be for example through the Peterson’s Algorithm for Mutual Lock (1981), however this mechanism does not avoid busy wait and may bring larger overhead costs compared to a semaphore-based solution.

User input is managed in the “display” function which runs as its own thread. It waits for the user to press a key (arrow key up, down, left, or right). Once an input is received, it enters its critical section by calling sem\_wait, where it parses the input and modifies the global variables (specifically the lander’s thrust and rotation) based on the input. It calls sem\_post afterwards, exiting its critical section.

## B. UDP and TCP

UDP and TCP are data transfer protocols used to communicate data between different devices.

The major differences in code between UDP and TCP are the socket type used (SOCK\_STREAM for TCP and SOCK\_DGRAM for UDP), the way messages are sent and received (“send” and “recv” functions for TCP and “sendto” and “recvfrom” functions for UDP – UDP also requires the address and address length of the server/client for each separate message, while TCP does not). In UDP, the server must actively listen for clients through the “listen” function; connection is then established with the “connect” function (client) and the “accept” function (server). The modified code can be found in Appendix 3 and Appendix 4.

Advantages of UDP are its smaller header size (8 versus 20 bytes) and that there is no connection between the server and the client, which would have to be created and maintained. However, data sent through UDP is prone to corruption – it does not attempt to recover the corrupted data, instead, it is simply discarded. It also does not compensate for lost data and cannot guarantee in-order delivery. There is also no congestion control in UDP.

TCP compensates for all the downsides of UDP. It secures a reliable communication channel between the client and the server (using a three-way handshake), however at the cost of larger data packets and the maintenance of the connection.

UDP is used in cases where receiving all data is not essential and smaller overhead is more important (i.e., video streaming), TCP is used in cases where all the data sent in a correct order is required and a larger overhead is not detrimental (i.e., sending e-mail).

Using TCP for the Lunar Lander controller could improve responsiveness of the program, since no game updates would be lost in transmission, however, it would increase the overhead for the data being transmitted.

## C. Data Logging

Data logging is a process of collecting and storing data over a period of time. The program logs data each 5 seconds into a csv (comma-separated values) file named “log.csv”. Data logged is formatted in a JSON (JavaScript Object Notation) format (example of several records logged into the file are shown in Appendix 5).

The program logs several different values. Each data point is a “time:state” value pair, where “time” represents the time of when the data point was logged and “state” the state of the game at that time. Logging time will help to put the logs into the order in which operations were performed and thus create an overall picture of the execution of operations from start to finish.

The “state” part of a data point consists of user input (key pressed by the user) and the state of lander, which is further divided into “command” (“thrust” and “rotation”), “state” (“x”, “y”, “O”, “dx”, “dy”, “dO”) and “condition” (“fuel”, “altitude”, “contact”). “Key” can be of values “up”, “down”, “left” and “right”; all other data is of value float, besides “contact”, which is a boolean (true of false). Logging all this data allows for a recreation of the exact state the lander was in at any point during the execution of the program, which in combination with the logged time allows for a full reenactment of the game from beginning to end.

The data is logged every 5 seconds. The interval in which data should be logged depends on the particular system it is logging the data for and the use case for the logged data. A balance must be found between a small interval, which captures the dynamics of the environment in detail, and a large interval, which lowers the impact on performance and the amount of storage needed for the logs. The data to be logged must be also correctly picked, such that important data is recorded, while redundant information omitted. The size of available storage for the logs and the method of logging may also play a significant role in what data should be logged and how often it should be logged. In some situations, data is not logged periodically over time, but instead tied to a completion of particular events (for example when the game is started or when the player presses a key).

For the Lunar Lander Controller, the data could be potentially logged in smaller intervals (than every 5 seconds), since one game takes a rather short amount of time to complete. This makes it produce only a few records, which do not give a proper, full overview of the game. Logging in even larger intervals could potentially make it produce only a single record per game (or per several games), making logging ineffective or even completely meaningless. The amount and information logged would also depend on the actual application of the data.

The size of the log file grows linearly over time – one record per 5 seconds. How much of space does a new record take compared to the whole log file can be calculated as . One log has a size varying between 232 and 271 characters with one additional character (comma) as a delimiter, thus, the size of the file will grow on average (mean) by 252.5 characters per 5 seconds. One character in C takes one byte of memory, thus, the file will expand on average by 2,020 bits per 5 seconds (3,030 bytes per minute; 181.8 kilobytes per hour). Graphs comparing different rates of the log file growth can be found in Appendix 6.

# 2. Operating Systems Theory and Concepts

## A. Advantages of a Command Line in Linux OS

The main advantage of using a command line is its greater speed of performing many (in particular complex) actions, some of which cannot be even achieved through a graphical user interface (GUI). On top of that, additional functionality can be added through scripting. The user can for example quickly move through the computer’s filesystem, compile and run code, edit (text-based) documents or even create short applications to automate tasks.

Another advantage of the command line is that it is lightweight - it uses less processing power and memory compared to a GUI. This makes it perfect in situations in which such things are limited, such as server-side programming, or in less-powerful machines.

The third advantage is that a command line can be used even when GUI is not present or not necessary (for example, in hardware-related or server-side applications).

## B. Processes and Threads

Process (sometimes referred to as “task”) is an instance of a program scheduled for execution by the operating system. It includes all the necessary information needed for running that particular program, such as the process ID, the program’s code and its environment variables, virtual address space and its size. A process can be in several different states depending on the execution circumstance it is currently in – new, ready, running, blocked (waiting for interrupt or other process), terminated. A process can create child processes; they can either share all, some, or no resources. The parent may wait for its child processes to finish execution, or they may execute simultaneously. A process is isolated from other processes – it does not share memory with other processes. The creation, termination and execution of processes are usually handled by the operating system itself.

Thread is a segment of a process. All threads belonging to a process share common system resources and each thread has a subset of its process’s address space. A process may have one or multiple threads. Threads of a process may share some resources, however since they may run in parallel, such resources must be carefully managed to ensure a proper execution of the program. The creation, termination and execution of threads is usually handled by their corresponding process.

Scenario 1: Use of multiple threads (multiple windows of a web browser)

Scenario 2: Use of multiple processes (text editor and a web browser)

## C. Context Switching in an Operating System

Since most machines use only a single CPU, but still need to run several processes at the same time, the operating system must perform a context switch, such that the CPU may be used by multiple processes. A context switch is also used between threads.

Context switch simply means changing from one currently executed process to another. During a context switch, the old process’s state must be saved to the Process Control Block (PCB) (this will ensure that once resumed, the paused process will start executing from the same point at which it ended) and its state updated to blocked, new process is then selected by the scheduler for execution and loaded from the PCB into the memory, its state is changed to running and it starts being executed. [Shown in Figure 1 below]

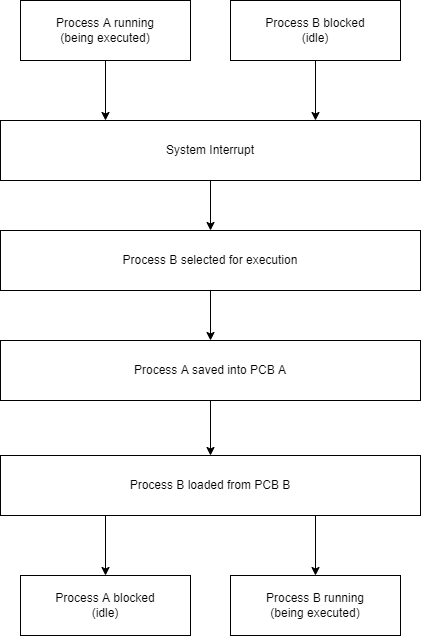
A context switch may occur under different circumstances, these may be for example a system interrupt (the current program must wait for input/output or a server response), or a timer interrupt (at regular intervals, the operating system automatically switches between one process to another). A context switch is considered to be an expensive operation, since it uses the system’s resources, while not performing any actual task. There are several overhead costs – saving, restoring and scheduling processes and the loss of cached resources.

Figure 1, This figure shows a diagram of a context switch between two processes

# 3. Security

## A. Secure Socket Layer Protection

Processes in an operating system are usually independent of each other – one process does not affect other processes, however in some cases the processes must have a way of communicating. This may be done either through a shared portion of memory which can be accessed and modified by multiple processes or through passing messages between the processes. Inter-process communication is commonly used for server-client communication – the client requests a response and the server answers it. Since such communication is usually carried over the internet and may include sensitive data, the communication is secured by using an SSL protocol.

The browser will connect the user to a server through https – secure hypertext transfer protocol, which uses cryptographic methods such as prime factorization and private/public keys to encrypt the data that is sent between the client and the server over the internet. This means that even if the data was compromised, the attacker would not be able to decrypt it and use it.

SSL uses a method called “three-way handshake” to ensure secure connection. The process is further described below.

## B. Symmetric (Private-Key) and Asymmetric (Public-Key) Cryptography

When some data is encrypted, it is impossible to make sense of it without decrypting it – a specific decryption “key” is needed.

In symmetric cryptography, the same key is used for both encryption and decryption of data. If Alice wants to securely send Bob a file, she first encrypts its contents with a chosen key, sends the encrypted file to Bob, and he then decrypts it with the same key. The main advantage of symmetric cryptography over asymmetric cryptography is speed, however, since both parties need to have access to the same key, it needs to be securely transmitted and safely stored.

In asymmetric cryptography, two keys – public and private – are used instead. Public keys are shared and can be known by anybody, while private keys are only known to a single party; only the combination of both keys allows one to decrypt the encrypted data. If Alice wants to securely send Bob a file, she encrypts its contents with his public key (can be known by anyone), sends the encrypted file to him and he then decrypts it with his private key, which is only known to him. Asymmetric cryptography is slower than symmetric cryptography, however no private keys need to be shared and if one is stolen, since both parties use separate private keys, only one party’s security is compromised.

In SSL, both methods are used for secure communication. The process of establishing a secure connection is called “three-way handshake”. The client (browser) sends a request to the server containing its available cipher suites and SLL versions, the server replies with its certificate, signed by a trusted authority (i.e., Google). After the client verifies this certificate (using asymmetric cryptography), the client and the server then generate a private master key which they use for encrypting and decrypting data. Asymmetric cryptography is used as a secure channel for exchanging this master key and symmetric cryptography using this key is then used for faster communication.

## C. Distributed Denial of service (DDoS) Attack

In distributed denial of service attack (DDoS), the target (server) is flooded with fraudulent incoming traffic from multiple sources (distributed), thus depleting the server’s resources (CPU, memory, bandwidth), which makes it unable to operate properly for legitimate users (denial of service). Sometimes a large number of computers are infected with a virus and then unknowingly used in a DDoS attack – this is usually referred to as “botnet”.

SSL is particularly susceptible to DDoS attack since it is computationally expensive and must be performed for each client connecting to the server (to establish secure connection). After a secure connection is established, the attacker client can immediately request a renegotiation of the contract again and again, or, if renegotiating is disabled by the server, the client immediately closes the connection and opens a new one, triggering the SSL protocol again. This way, the server becomes stuck establishing meaningless connections with fraudulent clients and cannot serve its legitimate users. [Shown if Figure 2 below]

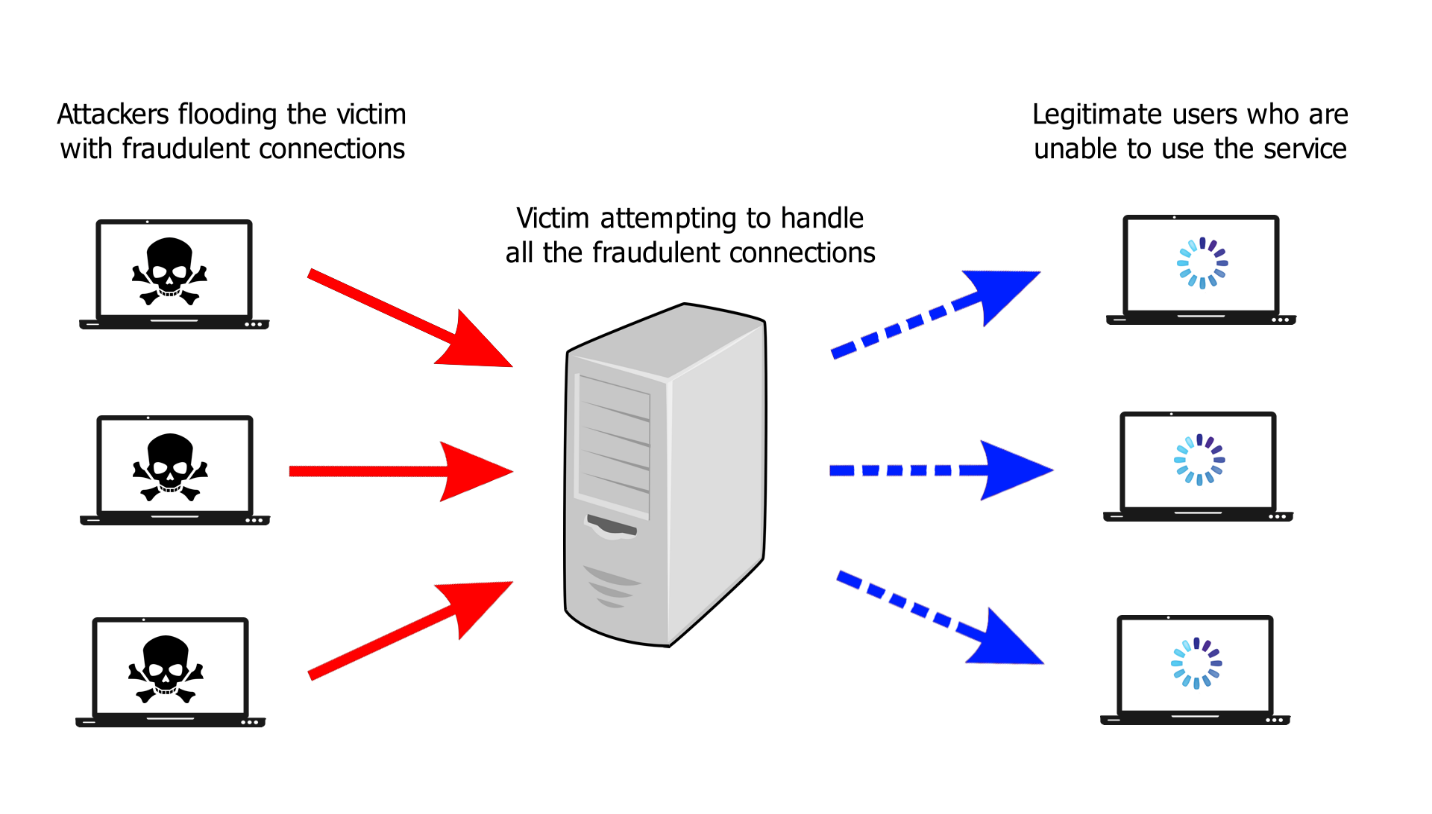
DDoS is known to be a hard to prevent attack, since the server would have to analyze each individual connection and decide which are legitimate and which are fraudulent, which would take a lot of resources. On top of that, if fraudulent requests are coming from a large number of sources, the sever cannot keep up with detecting and closing all such connections. The server may protect itself by monitoring incoming traffic or by employing specific protection applications or services.

Figure , This figure shows the nature of a DDoS attack

# References

Dijkstra, E. W. (1968) “Cooperating Sequential Processes”, in Hansen, P.B. (eds) *The Origin of Concurrent Programming*. Springer, New York, NY. <https://doi.org/10.1007/978-1-4757-3472-0_2>

Peterson, G. L. (1981) “Myths About the Mutual Exclusion Problem”, *Information Processing Letters* 12(3), 115–116

# Appendices

Appendix 1 (File: controller.c) 11

Appendix 2 (File: libnet.c) 23

Appendix 3 (TCP Version of the File: controller.c) 27

Appendix 4 (TCP Version of the File: libnet.c) 40

Appendix 5 (Example log.csv File Contents) 45

Appendix 6 (Log File Expansion Rate) 47

## Appendix 1 (File: controller.c)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <signal.h>

#include <pthread.h>

#include <semaphore.h>

#include <errno.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include "libnet.h"

#include "console.h"

#include <ctype.h>

#include <curses.h>

#include <time.h>

/\* -------------------- Semaphores and Global Variables --------------------

    Used for communication between the threads

\*/

struct command

{

    float thrust;

    float rotn;

} landercommand;

sem\_t cmdlock;

struct state

{

    float x, y, O;

    float dx, dy, dO;

} landerstate;

sem\_t statelock;

enum condstate

{

    Flying,

    Down,

    Crashed

};

struct condition

{

    float fuel;

    float altitude;

    int contact;

} landercond;

sem\_t condlock;

/\* -------------------- Keyboard Input --------------------

    Runs in own thread, interprets user input

    Generates command structures for transmission to lander

\*/

int last;

void \*keyboard(void \*data)

{

    landercommand.thrust = 0;

    landercommand.rotn = 0;

    while (true)

    {

        int key;

        while ((key = key\_pressed()) == ERR)

        {

            ;

        } /\* wait for a key-press \*/

        last = key;

        sem\_wait(&cmdlock); /\*Enter critical section\*/

        switch (key)

        {

        case KEY\_UP:

            landercommand.thrust += 2;

            if (landercommand.thrust > 100)

                landercommand.thrust = 100;

            break;

        case KEY\_DOWN:

            landercommand.thrust -= 2;

            if (landercommand.thrust < 0)

                landercommand.thrust = 0;

            break;

        case KEY\_RIGHT:

            landercommand.rotn += 0.1;

            if (landercommand.rotn >= 1)

                landercommand.rotn = 1;

            break;

        case KEY\_LEFT:

            landercommand.rotn -= 0.1;

            if (landercommand.rotn <= -1.0)

                landercommand.rotn = -1;

            break;

        }

        sem\_post(&cmdlock); /\*Exit critical section\*/

    }

}

/\* -------------------- Display Management --------------------

    Updates the display with lander diagnostic information

\*/

void \*display(void \*data)

{

    while (true)

    {

        sem\_wait(&condlock);

        switch (landercond.contact)

        {

        case Flying:

            lcd\_set\_colour(3, 0);

            lcd\_write\_at(1, 30, "Flying ");

            break;

        case Down:

            lcd\_set\_colour(2, .0);

            lcd\_write\_at(1, 30, "Down   ");

            break;

        case Crashed:

            lcd\_set\_colour(1, 0);

            lcd\_write\_at(1, 30, "Crashed");

            break;

        }

        lcd\_set\_colour(7, 0);

        lcd\_write\_at(0, 0, "fuel %f", landercond.fuel);

        lcd\_write\_at(1, 0, "alt  %f", landercond.altitude);

        sem\_post(&condlock);

        sem\_wait(&statelock);

        lcd\_write\_at(3, 0, "x %-6.1f  x' %-8.6f", landerstate.x, landerstate.dx);

        lcd\_write\_at(4, 0, "y %-6.1f  y' %-8.6f", landerstate.y, landerstate.dy);

        lcd\_write\_at(5, 0, "O %-6.3f  O' %-8.6f", landerstate.O, landerstate.dO);

        sem\_post(&statelock);

        sem\_wait(&cmdlock);

        lcd\_write\_at(3, 30, "thrust %6.1f", landercommand.thrust);

        lcd\_write\_at(4, 30, "rotn %6.1f", landercommand.rotn);

        sem\_post(&cmdlock);

        switch (last)

        {

        case KEY\_UP:

            lcd\_write\_at(0, 40, "up   ");

            break;

        case KEY\_DOWN:

            lcd\_write\_at(0, 40, "down ");

            break;

        case KEY\_LEFT:

            lcd\_write\_at(0, 40, "left ");

            break;

        case KEY\_RIGHT:

            lcd\_write\_at(0, 40, "right");

            break;

        default:

            lcd\_write\_at(0, 40, "%c   ", last);

        }

        usleep(500000);

    }

}

// --- Parse condition reply message --

void parsecondition(char \*m)

{

    char \*line;

    char \*rest;

    for (/\* split into lines lecture 07-2 slide 21 \*/

         line = strtok\_r(m, "\r\n", &rest);

         line != NULL;

         line = strtok\_r(NULL, "\r\n", &rest))

    {

        char \*key, \*value;

        key = strtok(line, ":");

        value = strtok(NULL, ":");

        sem\_wait(&condlock); /\* Enter critical section \*/

        if (strcmp(key, "fuel") == 0)

            sscanf(value, "%f%%", &(landercond.fuel));

        if (strcmp(key, "altitude") == 0)

            sscanf(value, "%f", &(landercond.altitude));

        if (strcmp(key, "contact") == 0)

        {

            if (strcmp(value, "flying") == 0)

                landercond.contact = Flying;

            if (strcmp(value, "down") == 0)

                landercond.contact = Down;

            if (strcmp(value, "crashed") == 0)

                landercond.contact = Crashed;

        }

        sem\_post(&condlock); /\* Exit critical section \*/

    }

}

// --- Parse state message ---

void parsestate(char \*m)

{

    char \*line;

    char \*rest;

    for (/\* split into lines lecture 07-2 slide 21 \*/

         line = strtok\_r(m, "\r\n", &rest);

         line != NULL; line = strtok\_r(NULL, "\r\n", &rest))

    {

        char \*key, \*value;

        key = strtok(line, ":");

        value = strtok(NULL, ":");

        sem\_wait(&statelock); /\*Enter critical section \*/

        if (strcmp(key, "x") == 0)

            sscanf(value, "%f", &(landerstate.x));

        if (strcmp(key, "y") == 0)

            sscanf(value, "%f", &(landerstate.y));

        if (strcmp(key, "O") == 0)

            sscanf(value, "%f", &(landerstate.O));

        if (strcmp(key, "x'") == 0)

            sscanf(value, "%f", &(landerstate.dx));

        if (strcmp(key, "y'") == 0)

            sscanf(value, "%f", &(landerstate.dy));

        if (strcmp(key, "O'") == 0)

            sscanf(value, "%f", &(landerstate.dO));

        sem\_post(&statelock); /\*Exit critical section \*/

    }

}

/\* -------------------- Lander communication --------------------

    Communicates with the lander model

    Sends commands and queries state

    Parses and decodes messages

    Arguments:

        data -> port number

\*/

void \*lander(void \*data)

{

    size\_t msgsize = 1000;

    char msgbuf[msgsize];

    int l;

    struct addrinfo \*landr;

    const char conditionq[] = "condition:?\n";

    const char stateq[] = "state:?\n";

    // Get address and open a socket

    if (!getaddr("127.0.1.1", (char \*)data, &landr))

        fprintf(stderr, "Can't get lander address\n");

    ;

    l = mksocket();

    while (true)

    {

        int m;

        usleep(50000); /\* 20Hz = 0.05s = 50ms = 50000us \*/

        /\* poll for condition \*/

        sendto(l, conditionq, strlen(conditionq), 0, landr->ai\_addr,

               landr->ai\_addrlen);

        m = recvfrom(l, msgbuf, msgsize, 0, NULL, NULL);

        msgbuf[m] = '\0';

        /\* parse condition \*/

        parsecondition(msgbuf);

        /\* poll for state \*/

        sendto(l, stateq, strlen(stateq), 0, landr->ai\_addr,

               landr->ai\_addrlen);

        m = recvfrom(l, msgbuf, msgsize, 0, NULL, NULL);

        msgbuf[m] = '\0';

        parsestate(msgbuf);

        /\* format command to send to lander \*/

        sprintf(msgbuf,

                "command:!\n"

                "main-engine: %f\n"

                "rcs-roll: %f\n",

                landercommand.thrust, landercommand.rotn);

        sendto(l, msgbuf, strlen(msgbuf), 0, landr->ai\_addr, landr->ai\_addrlen);

        m = recvfrom(l, msgbuf, msgsize, 0, NULL, NULL);

        msgbuf[m] = '\0';

        usleep(100000);

    }

}

/\* -------------------- Dashboard communication --------------------

    Formats and sends data messages to the dashboard

\*/

void \*dashboard(void \*data)

{

    size\_t bufsize = 1024;

    char buffer[bufsize];

    int d;

    struct addrinfo \*daddr;

    // Get address and open a socket

    if (!getaddr("127.0.1.1", (char \*)data, &daddr))

        fprintf(stderr, "Canott get dashboard address");

    d = mksocket();

    while (true)

    {

        int buffer\_error = sprintf(buffer, "fuel:%f\naltitude:%f\n", landercond.fuel, landercond.altitude);

        if (buffer\_error == -1)

            fprintf(stderr, "Error creating buffer array");

        // Send buffer with the message to the dashboard through socket

        sendto(d, buffer, strlen(buffer), 0, daddr->ai\_addr, daddr->ai\_addrlen);

        usleep(500000);

    }

}

/\* -------------------- Data Logging --------------------

    Periodically (every 5 seconds) logs data to a file

\*/

void \*datalogging(void \*data)

{

    FILE \*fileptr;

    // Open the data file

    fileptr = fopen("log.csv", "w");

    if (fileptr == NULL)

    {

        fprintf(stderr, "Log file could not be opened or created");

        exit(1);

    }

    /\* - Logged variables - \*/

    const int message\_size = 300;

    const int round\_numbers = 4;

    char \*delimiter = ",";

    // Time

    time\_t raw\_time;

    struct tm \*time\_info;

    char \*current\_time = "";

    // Key pressed

    char \*key\_pressed = "";

    // Lander command

    char lander\_thrust[10];

    char lander\_rotation[10];

    // Lander state

    char lander\_state\_x[10];

    char lander\_state\_y[10];

    char lander\_state\_O[10];

    char lander\_state\_dx[10];

    char lander\_state\_dy[10];

    char lander\_state\_dO[10];

    // Lander condition

    char lander\_condition\_fuel[10];

    char lander\_condition\_altitude[10];

    char \*lander\_condition\_contact;

    while (true)

    {

        // Get current time

        time(&raw\_time);

        time\_info = localtime(&raw\_time);

        current\_time = asctime(time\_info);

        current\_time[strcspn(current\_time, "\n")] = 0;

        // Get currently pressed key

        switch (last)

        {

        case KEY\_UP:

            key\_pressed = "up";

            break;

        case KEY\_DOWN:

            key\_pressed = "down";

            break;

        case KEY\_LEFT:

            key\_pressed = "left";

            break;

        case KEY\_RIGHT:

            key\_pressed = "right";

            break;

        default:

            key\_pressed = "none";

        }

        // Get current lander command

        gcvt(landercommand.thrust, round\_numbers, lander\_thrust);

        gcvt(landercommand.rotn, round\_numbers, lander\_rotation);

        // Get current lander state

        gcvt(landerstate.x, round\_numbers, lander\_state\_x);

        gcvt(landerstate.y, round\_numbers, lander\_state\_y);

        gcvt(landerstate.O, round\_numbers, lander\_state\_O);

        gcvt(landerstate.dx, round\_numbers, lander\_state\_dx);

        gcvt(landerstate.dy, round\_numbers, lander\_state\_dy);

        gcvt(landerstate.dO, round\_numbers, lander\_state\_dO);

        // Get current lander condition

        gcvt(landercond.fuel, round\_numbers, lander\_condition\_fuel);

        gcvt(landercond.altitude, round\_numbers, lander\_condition\_altitude);

        if (landercond.contact)

            lander\_condition\_contact = "true";

        else

            lander\_condition\_contact = "false";

        // Build the string that will be logged

        char log\_text[message\_size];

        int string\_building\_error = sprintf(log\_text, "{\"%s\":[{\"key\":\"%s\",\"lander\":[{\"command\":[{\"thrust \":\"%s\", \"rotation\":\"%s\"}], \"state\":[{\"x\":\"%s\", \"y\":\"%s\", \"O\":\"%s\", \"dx\":\"%s\", \"dy\":\"%s\", \"dO\":\"%s\"}], \"condition\":[{\"fuel\":\"%s\", \"altitude\":\"%s\", \"contact\":%s}]}]}]}",

                                            current\_time,

                                            key\_pressed,

                                            lander\_thrust,

                                            lander\_rotation,

                                            lander\_state\_x,

                                            lander\_state\_y,

                                            lander\_state\_O,

                                            lander\_state\_dx,

                                            lander\_state\_dy,

                                            lander\_state\_dO,

                                            lander\_condition\_fuel,

                                            lander\_condition\_altitude,

                                            lander\_condition\_contact);

        if (string\_building\_error == -1)

            fprintf(stderr, "Failed to build data logging string");

        // Write into the data file

        fprintf(fileptr, "%s%s", log\_text, delimiter);

        fflush(fileptr); // Use fflush due to buffered IO

        usleep(5000000); // Sleep 5 seconds -> log data each 5 seconds

    }

    // Close the data file

    fclose(fileptr);

}

/\* -------------------- MAIN --------------------

Arguments:

    argv[1] -> lander

    argv[2] -> dashboard

\*/

int main(int argc, char \*argv[])

{

    pthread\_t keyboard\_thread;     // Keyboard

    pthread\_t display\_thread;      // Display

    pthread\_t lander\_thread;       // Lander

    pthread\_t dashboard\_thread;    // Dashboard

    pthread\_t data\_logging\_thread; // Data logging

    int thread\_error;

    // Initialize semaphores

    sem\_init(&condlock, 0, 1);

    sem\_init(&statelock, 0, 1);

    sem\_init(&cmdlock, 0, 1);

    // Initialize the console display

    console\_init();

    // --- Create threads ---

    // Display thread

    if ((thread\_error = pthread\_create(&display\_thread, NULL, display, NULL)))

        fprintf(stderr, "Failed creating display thread: %s\n", strerror(thread\_error));

    // Keyboard thread

    if ((thread\_error = pthread\_create(&keyboard\_thread, NULL, keyboard, NULL)))

        fprintf(stderr, "Failed creating keyboard thread: %s\n", strerror(thread\_error));

    // Lander thread

    if ((thread\_error = pthread\_create(&lander\_thread, NULL, lander, argv[1])))

        fprintf(stderr, "Failed creating lander thread: %s\n", strerror(thread\_error));

    // Dashboard thread

    if ((thread\_error = pthread\_create(&dashboard\_thread, NULL, dashboard, argv[2])))

        fprintf(stderr, "Failed creating dashboard thread: %s\n", strerror(thread\_error));

    // Data logging thread

    if ((thread\_error = pthread\_create(&data\_logging\_thread, NULL, datalogging, NULL)))

        fprintf(stderr, "Failed creating data logging thread: %s\n", strerror(thread\_error));

    pthread\_join(display\_thread, NULL);

}

## Appendix 2 (File: libnet.c)

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdbool.h>

#include <errno.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

#include <netinet/in.h>

#include <arpa/inet.h>

// Gets address info

// Returns false if address info not found

int getaddr(const char \*node, const char \*service,

            struct addrinfo \*\*address)

{

    struct addrinfo hints =

        {

            .ai\_flags = 0,

            .ai\_family = AF\_INET,

            .ai\_socktype = SOCK\_DGRAM};

    if (node)

        hints.ai\_flags = AI\_ADDRCONFIG;

    else

        hints.ai\_flags = AI\_PASSIVE;

    int addr\_error = getaddrinfo(node, service, &hints, address);

    if (addr\_error)

    {

        fprintf(stderr, "Error trying to get address %s\n", gai\_strerror(addr\_error));

        return false;

    }

    return true;

}

// Creates an endpoint for communication using a socket

int mksocket(void)

{

    int socketfd = socket(AF\_INET, SOCK\_DGRAM, 0);

    if (socketfd == -1)

    {

        fprintf(stderr, "Error creating socket: %s\n", strerror(errno));

        return 0;

    }

    return socketfd;

}

// Binds socket to and address

int bindsocket(int sfd, const struct sockaddr \*addr, socklen\_t addrlen)

{

    int err = bind(sfd, addr, addrlen);

    if (err == -1)

    {

        fprintf(stderr, "Error binding socket: %s\n", strerror(errno));

        return false;

    }

    return true;

}

// Converts data from addr to URI host:port notation

char uri[80];

char \*addrtouri(struct sockaddr \*addr)

{

    struct sockaddr\_in \*a = (struct sockaddr\_in \*)addr;

    sprintf(uri, "%s:%d", inet\_ntoa(a->sin\_addr), ntohs(a->sin\_port));

    return uri;

}

typedef size\_t (\*handler\_t)(

    char \*,

    size\_t,

    char \*, size\_t,

    struct sockaddr\_in \*);

// Handles the server

int server(int srvrsock, handler\_t handlemsg)

{

    const size\_t buffsize = 4096; /\* 4k \*/

    char message[buffsize], reply[buffsize];

    size\_t msgsize, replysize;

    struct sockaddr clientaddr;

    socklen\_t addrlen = sizeof(clientaddr);

    while (true)

    {

        // Receive message through socket

        msgsize = recvfrom(

            srvrsock,    /\* server socket listening on \*/

            message,     /\* buffer to put message \*/

            buffsize,    /\* size of receiving buffer \*/

            0,           /\* flags \*/

            &clientaddr, /\* fill in with address of client \*/

            &addrlen);   /\* number of bytes filled in \*/

        replysize = handlemsg(

            message,  /\* incoming message \*/

            msgsize,  /\* incoming message size \*/

            reply,    /\* buffer for reply \*/

            buffsize, /\* size of outgoing buffer \*/

            (struct sockaddr\_in \*)&clientaddr);

        if (replysize)

        {

            // Send message through socket to an address

            sendto(

                srvrsock,    /\* server socket to use \*/

                reply,       /\* outgoing message to send \*/

                replysize,   /\* size of message \*/

                0,           /\* flags \*/

                &clientaddr, /\* address to send to \*/

                addrlen);    /\* size of address structure \*/

        }

    }

}

void finished(int signal)

{

    exit(0);

}

int cleanupsock;

// Closes the socket

void cleanup(void)

{

    close(cleanupsock);

}

## Appendix 3 (TCP Version of the File: controller.c)

/\* Sample solution for the LunarLander assignment

 \* KV5002

 \*

 \* Dr Alun Moon

 \*/

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <signal.h>

#include <pthread.h>

#include <semaphore.h>

#include <errno.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include "libnet.h"

#include "console.h"

#include <ctype.h>

#include <curses.h>

#include <time.h>

/\* -------------------- Semaphores and Global Variables --------------------

    Used for communication between the threads

\*/

struct command

{

    float thrust;

    float rotn;

} landercommand;

sem\_t cmdlock;

struct state

{

    float x, y, O;

    float dx, dy, dO;

} landerstate;

sem\_t statelock;

enum condstate

{

    Flying,

    Down,

    Crashed

};

struct condition

{

    float fuel;

    float altitude;

    int contact;

} landercond;

sem\_t condlock;

/\* -------------------- Keyboard Input --------------------

    Runs in own thread, interprets user input

    Generates command structures for transmission to lander

\*/

int last;

void \*keyboard(void \*data)

{

    landercommand.thrust = 0;

    landercommand.rotn = 0;

    while (true)

    {

        int key;

        while ((key = key\_pressed()) == ERR)

        {

            ;

        } /\* wait for a key-press \*/

        last = key;

        sem\_wait(&cmdlock); /\*Enter critical section\*/

        switch (key)

        {

        case KEY\_UP:

            landercommand.thrust += 2;

            if (landercommand.thrust > 100)

                landercommand.thrust = 100;

            break;

        case KEY\_DOWN:

            landercommand.thrust -= 2;

            if (landercommand.thrust < 0)

                landercommand.thrust = 0;

            break;

        case KEY\_RIGHT:

            landercommand.rotn += 0.1;

            if (landercommand.rotn >= 1)

                landercommand.rotn = 1;

            break;

        case KEY\_LEFT:

            landercommand.rotn -= 0.1;

            if (landercommand.rotn <= -1.0)

                landercommand.rotn = -1;

            break;

        }

        sem\_post(&cmdlock); /\*Exit critical section\*/

    }

}

/\* -------------------- Display Management --------------------

    Updates the display with lander diagnostic information

\*/

void \*display(void \*data)

{

    while (true)

    {

        sem\_wait(&condlock);

        switch (landercond.contact)

        {

        case Flying:

            lcd\_set\_colour(3, 0);

            lcd\_write\_at(1, 30, "Flying ");

            break;

        case Down:

            lcd\_set\_colour(2, .0);

            lcd\_write\_at(1, 30, "Down   ");

            break;

        case Crashed:

            lcd\_set\_colour(1, 0);

            lcd\_write\_at(1, 30, "Crashed");

            break;

        }

        lcd\_set\_colour(7, 0);

        lcd\_write\_at(0, 0, "fuel %f", landercond.fuel);

        lcd\_write\_at(1, 0, "alt  %f", landercond.altitude);

        sem\_post(&condlock);

        sem\_wait(&statelock);

        lcd\_write\_at(3, 0, "x %-6.1f  x' %-8.6f", landerstate.x, landerstate.dx);

        lcd\_write\_at(4, 0, "y %-6.1f  y' %-8.6f", landerstate.y, landerstate.dy);

        lcd\_write\_at(5, 0, "O %-6.3f  O' %-8.6f", landerstate.O, landerstate.dO);

        sem\_post(&statelock);

        sem\_wait(&cmdlock);

        lcd\_write\_at(3, 30, "thrust %6.1f", landercommand.thrust);

        lcd\_write\_at(4, 30, "rotn %6.1f", landercommand.rotn);

        sem\_post(&cmdlock);

        switch (last)

        {

        case KEY\_UP:

            lcd\_write\_at(0, 40, "up   ");

            break;

        case KEY\_DOWN:

            lcd\_write\_at(0, 40, "down ");

            break;

        case KEY\_LEFT:

            lcd\_write\_at(0, 40, "left ");

            break;

        case KEY\_RIGHT:

            lcd\_write\_at(0, 40, "right");

            break;

        default:

            lcd\_write\_at(0, 40, "%c   ", last);

        }

        usleep(500000);

    }

}

// --- Parse condition reply message --

void parsecondition(char \*m)

{

    char \*line;

    char \*rest;

    for (/\* split into lines lecture 07-2 slide 21 \*/

         line = strtok\_r(m, "\r\n", &rest);

         line != NULL;

         line = strtok\_r(NULL, "\r\n", &rest))

    {

        char \*key, \*value;

        key = strtok(line, ":");

        value = strtok(NULL, ":");

        sem\_wait(&condlock); /\* Enter critical section \*/

        if (strcmp(key, "fuel") == 0)

            sscanf(value, "%f%%", &(landercond.fuel));

        if (strcmp(key, "altitude") == 0)

            sscanf(value, "%f", &(landercond.altitude));

        if (strcmp(key, "contact") == 0)

        {

            if (strcmp(value, "flying") == 0)

                landercond.contact = Flying;

            if (strcmp(value, "down") == 0)

                landercond.contact = Down;

            if (strcmp(value, "crashed") == 0)

                landercond.contact = Crashed;

        }

        sem\_post(&condlock); /\* Exit critical section \*/

    }

}

// --- Parse state message ---

void parsestate(char \*m)

{

    char \*line;

    char \*rest;

    for (/\* split into lines lecture 07-2 slide 21 \*/

         line = strtok\_r(m, "\r\n", &rest);

         line != NULL; line = strtok\_r(NULL, "\r\n", &rest))

    {

        char \*key, \*value;

        key = strtok(line, ":");

        value = strtok(NULL, ":");

        sem\_wait(&statelock); /\*Enter critical section \*/

        if (strcmp(key, "x") == 0)

            sscanf(value, "%f", &(landerstate.x));

        if (strcmp(key, "y") == 0)

            sscanf(value, "%f", &(landerstate.y));

        if (strcmp(key, "O") == 0)

            sscanf(value, "%f", &(landerstate.O));

        if (strcmp(key, "x'") == 0)

            sscanf(value, "%f", &(landerstate.dx));

        if (strcmp(key, "y'") == 0)

            sscanf(value, "%f", &(landerstate.dy));

        if (strcmp(key, "O'") == 0)

            sscanf(value, "%f", &(landerstate.dO));

        sem\_post(&statelock); /\*Exit critical section \*/

    }

}

/\* -------------------- Lander communication --------------------

    Communicates with the lander model

    Sends commands and queries state

    Parses and decodes messages

    Arguments:

        data -> port number

\*/

void \*lander(void \*data)

{

    size\_t msgsize = 1000;

    char msgbuf[msgsize];

    int l;

    struct addrinfo \*landr;

    const char conditionq[] = "condition:?\n";

    const char stateq[] = "state:?\n";

    // Get address and open a socket

    if (!getaddr("127.0.1.1", (char \*)data, &landr))

        fprintf(stderr, "Can't get lander address\n");

    ;

    l = mksocket();

    // Connect to the server with TCP

    if (connect(l, landr->ai\_addr, landr->ai\_addrlen) == -1)

    {

        fprinf(stderr, "Could not connect to the server\n");

        exit(1);

    }

    while (true)

    {

        int m;

        usleep(50000); /\* 20Hz = 0.05s = 50ms = 50000us \*/

        /\* poll for condition \*/

        int sent = send(l, conditionq, strlen(conditionq), 0);

        if (m == -1)

            fprinf(stderr, "Error sending message: %s\n", strerror(errno));

        m = recv(l, msgbuf, msgsize, 0);

        if (m == -1)

            fprinf(stderr, "Error receiving message: %s\n", strerror(errno));

        msgbuf[m] = '\0';

        /\* parse condition \*/

        parsecondition(msgbuf);

        /\* poll for state \*/

        sent = send(l, stateq, strlen(stateq), 0);

        if (m == -1)

            fprinf(stderr, "Error sending message: %s\n", strerror(errno));

        m = recv(l, msgbuf, msgsize, 0);

        if (m == -1)

            fprinf(stderr, "Error receiving message: %s\n", strerror(errno));

        msgbuf[m] = '\0';

        parsestate(msgbuf);

        /\* format command to send to lander \*/

        sprintf(msgbuf,

                "command:!\n"

                "main-engine: %f\n"

                "rcs-roll: %f\n",

                landercommand.thrust, landercommand.rotn);

        sendto(l, msgbuf, strlen(msgbuf), 0);

        if (m == -1)

            fprinf(stderr, "Error sending message: %s\n", strerror(errno));

        m = recv(l, msgbuf, msgsize, 0);

        if (m == -1)

            fprinf(stderr, "Error receiving message: %s\n", strerror(errno));

        msgbuf[m] = '\0';

        usleep(100000);

    }

}

/\* -------------------- Dashboard communication --------------------

    Formats and sends data messages to the dashboard

\*/

void \*dashboard(void \*data)

{

    size\_t bufsize = 1024;

    char buffer[bufsize];

    int d;

    struct addrinfo \*daddr;

    // Get address and open a socket

    if (!getaddr("127.0.1.1", (char \*)data, &daddr))

        fprintf(stderr, "Canott get dashboard address");

    d = mksocket();

    // Connect to the server with TCP

    if (connect(d, dadddr->ai\_addr, dadddr->ai\_addrlen) == -1)

    {

        fprinf(stderr, "Could not connect to the server\n");

        exit(1);

    }

    while (true)

    {

        int buffer\_error = sprintf(buffer, "fuel:%f\naltitude:%f\n", landercond.fuel, landercond.altitude);

        if (buffer\_error == -1)

            fprintf(stderr, "Error creating buffer array");

        // Send buffer with the message to the dashboard through socket

        send(d, buffer, strlen(buffer), 0);

        if (m == -1)

            fprinf(stderr, "Error sending message: %s\n", strerror(errno));

        usleep(500000);

    }

}

/\* -------------------- Data Logging --------------------

    Periodically (every 5 seconds) logs data to a file

\*/

void \*datalogging(void \*data)

{

    FILE \*fileptr;

    // Open the data file

    fileptr = fopen("log.csv", "w");

    if (fileptr == NULL)

    {

        fprintf(stderr, "Log file could not be opened or created");

        exit(1);

    }

    /\* - Logged variables - \*/

    const int message\_size = 300;

    const int round\_numbers = 4;

    char \*delimiter = ",";

    // Time

    time\_t raw\_time;

    struct tm \*time\_info;

    char \*current\_time = "";

    // Key pressed

    char \*key\_pressed = "";

    // Lander command

    char lander\_thrust[10];

    char lander\_rotation[10];

    // Lander state

    char lander\_state\_x[10];

    char lander\_state\_y[10];

    char lander\_state\_O[10];

    char lander\_state\_dx[10];

    char lander\_state\_dy[10];

    char lander\_state\_dO[10];

    // Lander condition

    char lander\_condition\_fuel[10];

    char lander\_condition\_altitude[10];

    char \*lander\_condition\_contact;

    while (true)

    {

        // Get current time

        time(&raw\_time);

        time\_info = localtime(&raw\_time);

        current\_time = asctime(time\_info);

        current\_time[strcspn(current\_time, "\n")] = 0;

        // Get currently pressed key

        switch (last)

        {

        case KEY\_UP:

            key\_pressed = "up";

            break;

        case KEY\_DOWN:

            key\_pressed = "down";

            break;

        case KEY\_LEFT:

            key\_pressed = "left";

            break;

        case KEY\_RIGHT:

            key\_pressed = "right";

            break;

        default:

            key\_pressed = "none";

        }

        // Get current lander command

        gcvt(landercommand.thrust, round\_numbers, lander\_thrust);

        gcvt(landercommand.rotn, round\_numbers, lander\_rotation);

        // Get current lander state

        gcvt(landerstate.x, round\_numbers, lander\_state\_x);

        gcvt(landerstate.y, round\_numbers, lander\_state\_y);

        gcvt(landerstate.O, round\_numbers, lander\_state\_O);

        gcvt(landerstate.dx, round\_numbers, lander\_state\_dx);

        gcvt(landerstate.dy, round\_numbers, lander\_state\_dy);

        gcvt(landerstate.dO, round\_numbers, lander\_state\_dO);

        // Get current lander condition

        gcvt(landercond.fuel, round\_numbers, lander\_condition\_fuel);

        gcvt(landercond.altitude, round\_numbers, lander\_condition\_altitude);

        if (landercond.contact)

            lander\_condition\_contact = "true";

        else

            lander\_condition\_contact = "false";

        // Build the string that will be logged

        char log\_text[message\_size];

        int string\_building\_error = sprintf(log\_text, "{\"%s\":[{\"key\":\"%s\",\"lander\":[{\"command\":[{\"thrust \":\"%s\", \"rotation\":\"%s\"}], \"state\":[{\"x\":\"%s\", \"y\":\"%s\", \"O\":\"%s\", \"dx\":\"%s\", \"dy\":\"%s\", \"dO\":\"%s\"}], \"condition\":[{\"fuel\":\"%s\", \"altitude\":\"%s\", \"contact\":%s}]}]}]}",

                                            current\_time,

                                            key\_pressed,

                                            lander\_thrust,

                                            lander\_rotation,

                                            lander\_state\_x,

                                            lander\_state\_y,

                                            lander\_state\_O,

                                            lander\_state\_dx,

                                            lander\_state\_dy,

                                            lander\_state\_dO,

                                            lander\_condition\_fuel,

                                            lander\_condition\_altitude,

                                            lander\_condition\_contact);

        if (string\_building\_error == -1)

            fprintf(stderr, "Failed to build data logging string");

        // Write into the data file

        fprintf(fileptr, "%s%s", log\_text, delimiter);

        fflush(fileptr); // Use fflush due to buffered IO

        usleep(5000000); // Sleep 5 seconds -> log data each 5 seconds

    }

    // Close the data file

    fclose(fileptr);

}

/\* -------------------- MAIN --------------------

Arguments:

    argv[1] -> lander

    argv[2] -> dashboard

\*/

int main(int argc, char \*argv[])

{

    pthread\_t keyboard\_thread;     // Keyboard

    pthread\_t display\_thread;      // Display

    pthread\_t lander\_thread;       // Lander

    pthread\_t dashboard\_thread;    // Dashboard

    pthread\_t data\_logging\_thread; // Data logging

    int thread\_error;

    // Initialize semaphores

    sem\_init(&condlock, 0, 1);

    sem\_init(&statelock, 0, 1);

    sem\_init(&cmdlock, 0, 1);

    // Initialize the console display

    console\_init();

    // --- Create threads ---

    // Display thread

    if ((thread\_error = pthread\_create(&display\_thread, NULL, display, NULL)))

        fprintf(stderr, "Failed creating display thread: %s\n", strerror(thread\_error));

    // Keyboard thread

    if ((thread\_error = pthread\_create(&keyboard\_thread, NULL, keyboard, NULL)))

        fprintf(stderr, "Failed creating keyboard thread: %s\n", strerror(thread\_error));

    // Lander thread

    if ((thread\_error = pthread\_create(&lander\_thread, NULL, lander, argv[1])))

        fprintf(stderr, "Failed creating lander thread: %s\n", strerror(thread\_error));

    // Dashboard thread

    if ((thread\_error = pthread\_create(&dashboard\_thread, NULL, dashboard, argv[2])))

        fprintf(stderr, "Failed creating dashboard thread: %s\n", strerror(thread\_error));

    // Data logging thread

    if ((thread\_error = pthread\_create(&data\_logging\_thread, NULL, datalogging, NULL)))

        fprintf(stderr, "Failed creating data logging thread: %s\n", strerror(thread\_error));

    pthread\_join(display\_thread, NULL);

}

## Appendix 4 (TCP Version of the File: libnet.c)

/\* Network and UDP handling library

 \* KV5002

 \*

 \* Dr Alun Moon

 \*/

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <stdbool.h>

#include <errno.h>

#include <string.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netdb.h>

#include <netinet/in.h>

#include <arpa/inet.h>

// Gets address info

// Returns false if address info not found

int getaddr(const char \*node, const char \*service,

            struct addrinfo \*\*address)

{

    struct addrinfo hints =

        {

            .ai\_flags = 0,

            .ai\_family = AF\_INET,

            .ai\_socktype = SOCK\_STREAM};

    if (node)

        hints.ai\_flags = AI\_ADDRCONFIG;

    else

        hints.ai\_flags = AI\_PASSIVE;

    int addr\_error = getaddrinfo(node, service, &hints, address);

    if (addr\_error)

    {

        fprintf(stderr, "Error trying to get address %s\n", gai\_strerror(addr\_error));

        return false;

    }

    return true;

}

// Creates an endpoint for communication using a socket

int mksocket(void)

{

    int socketfd = socket(AF\_INET, SOCK\_STREAM, 0);

    if (socketfd == -1)

    {

        fprintf(stderr, "Error creating socket: %s\n", strerror(errno));

        return 0;

    }

    return socketfd;

}

// Binds socket to and address

int bindsocket(int sfd, const struct sockaddr \*addr, socklen\_t addrlen)

{

    int err = bind(sfd, addr, addrlen);

    if (err == -1)

    {

        fprintf(stderr, "Error binding socket: %s\n", strerror(errno));

        return false;

    }

    return true;

}

// Converts data from addr to URI host:port notation

char uri[80];

char \*addrtouri(struct sockaddr \*addr)

{

    struct sockaddr\_in \*a = (struct sockaddr\_in \*)addr;

    sprintf(uri, "%s:%d", inet\_ntoa(a->sin\_addr), ntohs(a->sin\_port));

    return uri;

}

typedef size\_t (\*handler\_t)(

    char \*,

    size\_t,

    char \*, size\_t,

    struct sockaddr\_in \*);

// Handles the server

int server(int srvrsock, handler\_t handlemsg)

{

    const size\_t buffsize = 4096; /\* 4k \*/

    char message[buffsize], reply[buffsize];

    size\_t msgsize, replysize;

    struct sockaddr clientaddr;

    socklen\_t addrlen = sizeof(clientaddr);

    struct addrinfo hints;

    struct addrinfo \*results;

    int addr\_error;

    hints.ai\_family = AF\_INET;

    hints.ai\_protocol = 0;

    hints.ai\_socktype = SOCK\_STREAM;

    hints.ai\_flags = AI\_PASSIVE;

    char port\_number\_string[10];

    // Get addrinfo

    addr\_error = getaddrinfo(NULL, port\_number\_string, &hints, &results);

    if (addr\_error == -1)

    {

        fprintf(stderr, "Error trying to get address %s\n", gai\_strerror(addr\_error));

        return 1;

    }

    // Create a socket

    int listen\_sock;

    if ((listen\_sock = socket(AF\_INET, SOCK\_STREAM, 0)) < 0)

    {

        fprintf(stderr, "Could not create socket: %s\n", strerror(errno));

        return 1;

    }

    // Bind a socket

    if ((bind(listen\_sock, results->ai\_addr, results->ai\_addrlen)))

    {

        fprinf(stderr, "Could not bind socket\n");

        return 1;

    }

    // Listen to clients

    int wait\_size = 16; // Maximum number of clients before dropping

    if (listen(listen\_sock, wait\_size) < 0)

    {

        fprintf(stderr, "Could not open socket for listening\n");

        return 1;

    }

    while (true)

    {

        // Accept a communication channel

        int sock;

        if ((sock = accept(listen\_sock, (struct sockaddr \*)&clientaddr, &addrlen)) < 0)

        {

            fprintf(stderr, "Could not open socket to accept data\n");

            return 1;

        }

        // Receive message through socket

        msgsize = recv(

            srvrsock, /\* server socket listening on \*/

            message,  /\* buffer to put message \*/

            buffsize, /\* size of receiving buffer \*/

            0);       /\* flags \*/

        if (msgsize == -1)

            fprinf(stderr, "Error receiving message: %s\n", strerror(errno));

        replysize = handlemsg(

            message,  /\* incoming message \*/

            msgsize,  /\* incoming message size \*/

            reply,    /\* buffer for reply \*/

            buffsize, /\* size of outgoing buffer \*/

            (struct sockaddr\_in \*)&clientaddr);

        if (replysize)

        {

            // Send message through socket

            m = send(

                srvrsock,  /\* server socket to use \*/

                reply,     /\* outgoing message to send \*/

                replysize, /\* size of message \*/

                0);        /\* flags \*/

            if (m == -1)

                fprinf(stderr, "Error sending message: %s\n", strerror(errno));

        }

    }

    cleanup();

}

void finished(int signal)

{

    exit(0);

}

int cleanupsock;

// Closes the socket

void cleanup(void)

{

    close(cleanupsock);

}

## Appendix 5 (Example log.csv File Contents)

{"Wed May 11 00:06:01 2022":[{"key":"none","lander":[{"command":[{"thrust ":"0", "rotation":"0"}], "state":[{"x":"0", "y":"0", "O":"0", "dx":"0", "dy":"0", "dO":"0"}], "condition":[{"fuel":"0", "altitude":"0", "contact":false}]}]}]},{"Wed May 11 00:06:06 2022":[{"key":"none","lander":[{"command":[{"thrust ":"0", "rotation":"0"}], "state":[{"x":"357.5", "y":"587", "O":"0", "dx":"0", "dy":"0", "dO":"0"}], "condition":[{"fuel":"100", "altitude":"-0.002025", "contact":true}]}]}]},{"Wed May 11 00:06:11 2022":[{"key":"right","lander":[{"command":[{"thrust ":"0", "rotation":"1"}], "state":[{"x":"146.5", "y":"114", "O":"146.7", "dx":"10", "dy":"17.53", "dO":"0"}], "condition":[{"fuel":"100", "altitude":"486", "contact":false}]}]}]},{"Wed May 11 00:06:16 2022":[{"key":"right","lander":[{"command":[{"thrust ":"10", "rotation":"0.5"}], "state":[{"x":"195.4", "y":"219.3", "O":"300.6", "dx":"9.818", "dy":"25.31", "dO":"0"}], "condition":[{"fuel":"99.92", "altitude":"364.7", "contact":false}]}]}]},{"Wed May 11 00:06:21 2022":[{"key":"left","lander":[{"command":[{"thrust ":"16", "rotation":"0.9"}], "state":[{"x":"243.3", "y":"362.8", "O":"355.5", "dx":"9.119", "dy":"32", "dO":"0"}], "condition":[{"fuel":"99.32", "altitude":"225.2", "contact":false}]}]}]},{"Wed May 11 00:06:26 2022":[{"key":"right","lander":[{"command":[{"thrust ":"16", "rotation":"0.1"}], "state":[{"x":"290.2", "y":"537.4", "O":"345", "dx":"9.297", "dy":"37.8", "dO":"0"}], "condition":[{"fuel":"98.52", "altitude":"45.63", "contact":false}]}]}]},

**Data point with indentation (JSON format):**

{

    "Wed May 11 00:06:16 2022": [

        {

            "key": "right",

            "lander": [

                {

                    "command": [

                        {

                            "thrust ": "10",

                            "rotation": "0.5"

                        }

                    ],

                    "state": [

                        {

                            "x": "195.4",

                            "y": "219.3",

                            "O": "300.6",

                            "dx": "9.818",

                            "dy": "25.31",

                            "dO": "0"

                        }

                    ],

                    "condition": [

                        {

                            "fuel": "99.92",

                            "altitude": "364.7",

                            "contact": false

                        }

                    ]

                }

            ]

        }

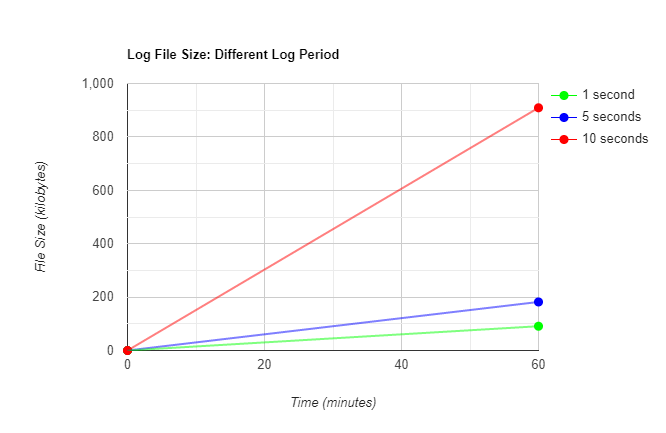
    ]

}

## Appendix 6 (Log File Expansion Rate)

Chart, line chart

Description automatically generatedThe first graph shows how the differing lengths of the record message affect the size of the file over time. Average = 252.5 characters; Shortest = 233 characters; Longest = 272 characters.

The second graph shows how would different periods of logging the data affect the file size over time.