INSTITUTO SUPERIOR DE ENGENHARIA DE COIMBRA

DEPARTAMENTO DE INFORMÁTICA E DE SISTEMAS

ENGENHARIA INFORMÁTICA

SISTEMAS OPERATIVOS

TRABALHO PRÁTICO – META II

[2016012270] Miguel Pires

[2017019981] Maria Ferreira

ANO LETIVO 2019/2020

ÍNDICE

ÍNDICE DE FIGURAS

1. DESCRIÇÃO GERAL DO PROJETO

Este projeto está dividido em 3 programas independentes – o **cliente**, o **gestor** e o **verificador**. Cada programa desempenha uma função fulcral no funcionamento do projeto, de modo a atingir a sua finalidade total.

Nesta segunda fase, pretendeu-se organizar melhor as implementações efetuadas na fase anterior e reparar algumas das funcionalidades. Com a robustez e boas práticas de estruturação, a planificação da segunda meta deste projeto mostrou-se mais fácil e menos morosa.

Com o decorrer do tempo, foram-se destacando algumas particularidades que obrigaram a pequenas restruturações, de modo a tornar o código versátil, coeso e coerente.

2. ESTRUTURAS DE DADOS

2.1. HEADER FILE "MSGDIST_UTILS"

```c
typedef struct client {
    char username[UTIL_BUFFER_SIZE];
    char pipeName[UTIL_BUFFER_SIZE];
    int id;
} MSGDIST_CLIENT;

typedef struct server {
    char file[UTIL_BUFFER_SIZE];
    int maxUsers;
    int maxMsg;
    int maxNot;
    int timeout;
    int filter; // 1 - ON, 0 - OFF
} MSGDIST_SERVER;

typedef struct msg {
    char topic[UTIL_BUFFER_SIZE];
    char title[UTIL_BUFFER_SIZE];
    char body[UTIL_BODY_SIZE];
    int duration;
    int flag;
    MSGDIST_CLIENT client;
} MSG;

typedef struct pkt {
    MSGDIST_CLIENT client;
    int flag; // se -1 é porque está vazio
    int maxUsers;
} PACKAGE;

typedef struct topics {
    char topic[UTIL_BUFFER_SIZE];
    PACKAGE *clients;
    int msg; //numero de subs
    int flag; //topico ativo
} SUBS_TOPICS;
```

```c
#define UTIL_BUFFER_SIZE    20
#define UTIL_BODY_SIZE      1000
#define UTIL_EMPTY          ""              // "args" vazio

#define VAR_MAX_USERS       "MAXUSERS"
#define VAR_MAX_MSG         "MAXMSG"
#define VAR_MAX_NOT         "MAXNOT"
#define VAR_DFL_FILENAME    "WORDSNOT"
#define VAR_TIMEOUT         "TIMEOUT"

#define UTIL_TIMEOUT        10              // segundos
#define UTIL_MAX_USERS      10
#define UTIL_MAX_MSG        50
#define UTIL_MAX_NOT        3
#define UTIL_FILENAME       "palavras.txt"

#define SERVER_FILTER       "filter"
#define SERVER_USERS        "users"
#define SERVER_TOPICS       "topics"
#define SERVER_MSG          "msg"
#define SERVER_TOPIC        "topic"
#define SERVER_DEL          "del"
#define SERVER_KICK         "kick"
#define SERVER_SHUTDOWN     "shutdown"
#define SERVER_PRUNE        "prune"
#define SERVER_HELP         "help"
#define SERVER_CHECK        "check"
#define SERVER_INFO         "info"

#define CLIENT_WRITE_MENSAGE        SERVER_MSG
#define CLIENT_TOPICS               SERVER_TOPICS
#define CLIENT_TITLES               "titles"
#define CLIENT_SUBS_TOPIC           "subscribe"
#define CLIENT_HELP                 SERVER_HELP
#define CLIENT_SHUTDOWN             SERVER_SHUTDOWN
#define CLIENT_SHOW_MESSAGE         "showmsg"
```

Figura 1 - header file "msgdist_utils"

O ***msgdist_utils.h*** é responsável por alocar estruturas de dados que serão usadas por ambos programas intervenientes – o gestor e o cliente. Dentro deste estão definidos valores *default* para vários campos do sistema, tais como o nome do ficheiro de palavras proibidas, o número máximo de palavras proibidas por mensagem, o máximo de mensagens totais, entre outras. O ficheiro contém também variáveis definidas de modo a tornar o código mais sucinto e direto na sua leitura.

3. FUNCIONALIDADES

Aqui estão apresentadas as funcionalidades realizadas nesta meta. As funcionalidades estão organizadas por tópicos.

3.1. LEITURA DE VARIÁVEIS DE AMBIENTE

```c
// MAXNOT

int getVarMaxNot() {
    char * env_var = getenv(VAR_MAX_NOT);

    if (env_var == NULL || atoi(env_var) <= 0)
        return UTIL_MAX_NOT;

    return atoi(env_var);
}

// TIMEOUT

int getVarTimeout() {
    char * env_var = getenv(VAR_TIMEOUT);

    if (env_var == NULL || atoi(env_var) <= 0)
        return UTIL_TIMEOUT;

    return atoi(env_var);
}

// FILE

char * getVarFile() {
    char * env_var = getenv(VAR_DFL_FILENAME);

    if (env_var == NULL)
        return UTIL_FILENAME;

    return env_var;
}
```

```c
MSGDIST_SERVER serverInitialize() {
    MSGDIST_SERVER server;

    server.maxUsers = getVarMaxUsers();
    server.maxMsg = getVarMaxMsg();
    server.maxNot = getVarMaxNot();
    server.timeout = getVarTimeout();
    server.filter = 1;
    strncpy(server.file, getVarFile(), UTIL_BUFFER_SIZE);

    return server;
}
```

Figura 2 - Leitura de variáveis de ambiente

A leitura das variáveis de ambiente (se estas forem definidas) e suposta atribuição nos campos da estrutura do servidor, é feita, exclusivamente, no gestor, ou seja, no programa ***msgdist_server.c***. Se a variável não estiver no ambiente da *Shell*, então o valor tomado é da variável que está definida no *header file*. Este valor será atribuído a uma estrutura do tipo ***MSGDIST_SERVER***.

3.2. LEITURA DE COMANDOS (GESTOR)

```c
void readRoutineCommands(THREAD_PACKAGE * threadPackage) {
    char cmd[UTIL_BUFFER_SIZE];
    char * argument;

    do {
        fflush(stdout);
        fflush(stdin);

        printf("MSGDIST@SERVER >> ");
        scanf(" %19[^\n]s", cmd);

        // Separação do pedido recebido em comando e argumento
        strtok_r(cmd, " ", &argument);

        threadPackage->server = cmdRoutine(threadPackage->server, threadPackage->clientsPackage, cmd, argument,
    } while (1);
}
```

Figura 3 - leitura dos comandos do gestor

Ciclo infinito que espera a introdução dos comandos e respetivos argumentos, se estes existirem no contexto do próprio comando. O conteúdo escrito pelo cliente é, então, separado em comando e argumento para melhor análise na função ***cmdRoutine***.

3.3. ROTINA DE COMANDOS (GESTOR)

```c
MSGDIST_SERVER cmdRoutine(MSGDIST_SERVER sv, PACKAGE * clients, char * cmd, char * arg, MSG *allmsg, SUBS_TOPICS *topics) {

    if (strcmp(cmd, SERVER_FILTER) == 0) {
        if (strcmp(arg, UTIL_EMPTY) == 0) {
            printf("MSGDIST: (ERROR) This command needs an argument. Try 'help'\n");
        } else {
            sv = cmdRoutineFilter(arg, sv);
        }
    } else if (strcmp(cmd, SERVER_USERS) == 0) {
        cmdRoutineUsers(clients, sv);
    } else if (strcmp(cmd, SERVER_TOPICS) == 0) {
        cmdRoutineTopics(allmsg);
    } else if (strcmp(cmd, SERVER_MSG) == 0) {
        cmdRoutineMsg(allmsg);
    } else if (strcmp(cmd, SERVER_TOPIC) == 0) {
        cmdRoutineTopic(allmsg);
    } else if (strcmp(cmd, SERVER_DEL) == 0) {
        cmdRoutineDel(allmsg, topics);
    } else if (strcmp(cmd, SERVER_KICK) == 0) {
        if (strcmp(arg, UTIL_EMPTY) == 0) {
            printf("MSGDIST: (ERROR) This command needs an argument. Try 'help'\n");
        } else {
            cmdRoutineKick(arg, clients);
        }
    } else if (strcmp(cmd, SERVER_PRUNE) == 0) {
        cmdRoutinePrune(allmsg, topics);
    } else if (strcmp(cmd, SERVER_SHUTDOWN) == 0) {
        cmdRoutineShutdown(clients);
    } else if (strcmp(cmd, SERVER_HELP) == 0) {
        cmdRoutineHelp();
```

Figura 4 – rotina de comandos da consola do gestor

Nesta rotina, é recebido o comando (e argumento, se existir), sendo verificado mediante as opções que comandos que o gestor dispõe.

Para além dos que foram pedidos no enunciado, foi adicionado também os comandos ***help***, que imprime no ecrã uma pequena explicação de ajuda de cada comando, o comando ***info***, que dita a informação do sistema, como por exemplo o número máximo de mensagens, o nome do ficheiro de palavras proibidas, entre outras. Nesta fase, foi removido o comando ***check***, comando anteriormente implementado com a finalidade de testar o funcionamento do programa ***verificador***. O ciclo é interrompido, bem como todo o programa do gestor, com o comando ***shutdown***. Este último comando é, também, responsável por avisar os clientes que se encontra encerrar, o que leva, posteriormente, ao encerramento de cada um.

3.4. LIGAÇÃO VERIFICADOR-GESTOR

```c
int verifier(char * words, char * file) {
    int p[2], pr[2], res, state;
    char message[UTIL_BODY_SIZE];
    char result[30];

    strncpy(message, words, UTIL_BODY_SIZE);

    pipe(p); //pipe de escrita
    pipe(pr); //pipe de leitura

    res = fork();
    // caminho do filho
    if (res == 0) {

        //PIPE ANONIMO DE ESCRITA
        close(0); //liberta a posicao de stdin
        dup(p[0]); //duplica a extremidade de leitura do pipe para a pos 0
        close(p[0]); //fecha a extremidade pq já foi duplicada
        close(p[1]); //fecha a extremidade de escrita do pipe pq não vai ser utilizada


        //PIPE ANONIMO DE LEITURA

        close(1); //liberta a posicao de stdout
        dup(pr[1]); //duplica a extremidade de escrita do pipe para a pos 1
        close(pr[1]); //fecha a extremidade pq já foi duplicada
        close(pr[0]); //fecha a extremidade de leitura do pipe pq não vai ser utilizada


        execl("verificador", "verificador", file, NULL);
        printf("MSGDIST: (ERROR) Failure in opening Verifier. Check name of files\n");
        exit(1);
    }

    // caminho do pai
    close(p[0]); //fecha a extremidade de leitura do pipe de escrita
    close(pr[1]); //fecha a extremidade de escrita do pipe de leitura
```

Figura 5 - verificação de palavras através da ligação ao programa "verificador.c"

Esta função é responsável pela verificação de palavras proibidas numa mensagem, assegurando uma ligação do gestor ao verificador. Aqui, são usados dois *pipes* anónimos – um que escreve e um que lê – onde cada palavra é enviada. Aquando a expressão **##MSGEND##**, é parada a verificação e o verificador retorna o número de palavras proibidas que encontrou.

Nesta meta, sempre que um cliente cria uma mensagem e tenta enviá-la ao gestor, este é responsável pela validação de todos os campos da mesma. Se não for válida, o gestor emite uma mensagem ao cliente com essa resposta.

3.5. VERIFICAÇÃO DO *USERNAME*

```
strncpy(name, clientReceived.username, UTIL_BUFFER_SIZE);


for (i = 0; i < maxUsers; i++) {
    if (threadPKG->clientsPackage[i].flag == -1 && flag == 0) {
        for (j = 0; j < maxUsers; j++) {
            // Se o username já existir nos clientes
            if (strcmp(threadPKG->clientsPackage[j].client.username, clientReceived.username) == 0) {
                strncpy(clientReceived.username, name, UTIL_BUFFER_SIZE);
                snprintf(identifier, UTIL_BUFFER_SIZE, "%d", ++cnt);
                strncat(clientReceived.username, identifier, UTIL_BUFFER_SIZE);
            }
        }
        strncpy(threadPKG->clientsPackage[i].client.username, clientReceived.username, UTIL_BUFFER_SIZE);
        threadPKG->clientsPackage[i].client.id = clientReceived.id;
        strncpy(threadPKG->clientsPackage[i].client.pipeName, clientReceived.pipeName, UTIL_BUFFER_SIZE);

        threadPKG->clientsPackage[i].flag = 1;
        flag = 1;
    }
    if (threadPKG->clientsPackage[i].flag != -1 && flag == 0) {
        threadPKG->clientsPackage[i].flag = 1;
    }
}
```

Figura 6 – mecanismo de verificação de username, seguida de respetiva alocação no servidor

Esta funcionalidade já se encontra completamente funcional, isto é, aquando a inserção do ***username*** por parte do cliente, é enviado esse *username* para o gestor de modo a ser verificado. Se se encontrar um cliente registado no gestor com o mesmo *username*, então é colocado um número no *username* do cliente que pretende estabelecer ligação.

3.6. LEITURA DE COMANDOS (CLIENTE)

```c
void readRoutineCommands(MSGDIST_CLIENT * client) {
    char cmd[UTIL_BUFFER_SIZE];
    char * argument;

    do {
        fflush(stdout);
        fflush(stdin);

        printf("MSGDIST@CLIENT >> ");
        scanf(" %19[^\n]s", cmd);

        // Separação do pedido recebido em comando e argumento
        strtok_r(cmd, " ", &argument);

        cmdRoutine(client, cmd, argument);
    } while (1);
}
```

Figura 7 - leitura de comandos no cliente

Tal como foi implementado no gestor, foi também obrigatório criar uma rotina de comandos de modo a satisfazer as necessidades do cliente. Funciona da mesma maneira como no gestor.

3.7. ROTINA DE COMANDOS (CLIENTE)

```c
void cmdRoutine(MSGDIST_CLIENT * client, char * cmd, char * arg) {

    if (strcmp(cmd, CLIENT_WRITE_MENSAGE) == 0) {
        writeMessage(client);
    } else if (strcmp(cmd, CLIENT_TOPICS) == 0) {
        cmdRoutineShowTopics(client);
    } else if (strcmp(cmd, CLIENT_TITLES) == 0) {
        if (strcmp(arg, UTIL_EMPTY) == 0) {
            printf("MSGDIST: (ERROR) This command needs an argument. Try 'help'\n");
        } else {
            cmdRoutineTitles(arg, client);
        }
    } else if (strcmp(cmd, CLIENT_SUBS_TOPIC) == 0) {
        cmdRoutineSubscribe(client);
    } else if (strcmp(cmd, CLIENT_HELP) == 0) {
        cmdRoutineHelp();
    } else if (strcmp(cmd, CLIENT_SHOW_MESSAGE) == 0) {
        cmdRoutineShowMessage(client);
    } else if (strcmp(cmd, CLIENT_SHUTDOWN) == 0) {
        cmdRoutineShutdown(client->username);
    } else {
        printf("MSGDIST: (ERROR) Command not found.\n");
        putchar('\n');
    }
}
```

Figura 8 - rotina de comandos do gestor

Nesta rotina, tal como acontece no gestor, é recebido o comando (e argumento, se existir), sendo verificado mediante as opções que comandos que o gestor dispõe.

Na secção do cliente, este pode listar os títulos dos tópicos que indicar, listar todos os tópicos e subscrever a um tópico, se este existir. Pode também usar os comandos de ajuda.

3.8. COMUNICAÇÃO BASEADA EM NAMED PIPES

```
//mkfifo
mkfifo(SERVER_PIPE, 0600);
fd = open(SERVER_PIPE, O_RDONLY);

if (fd == -1) {
    printf("MSGDIST: (ERROR) Server pipe.\n");
    exit(EXIT_FAILURE);
}

buffer = 0;

receivedMessage.flag = 0;

buffer = read(fd, &receivedMessage, sizeof (INFO_CLIENT));


if (buffer == sizeof (INFO_CLIENT)) {
    if (receivedMessage.flag == 1) {
        pthread_mutex_lock(&mflag);
        threadPKG = verifyUsername(receivedMessage.client, threadPKG);
        pthread_mutex_unlock(&mflag);
    }

    if (receivedMessage.flag == 2) {
        pthread_mutex_lock(&mflag);
        threadPKG = verifyMessage(receivedMessage.msg, threadPKG);
        pthread_mutex_unlock(&mflag);
    }
    if (receivedMessage.flag == 3) {
        pthread_mutex_lock(&mflag);
        sendAllMessages(receivedMessage.client, threadPKG);
        pthread_mutex_unlock(&mflag);
    }
    if (receivedMessage.flag == 4) {
        pthread_mutex_lock(&mflag);
        removeClient(&receivedMessage.client, threadPKG);
        pthread_mutex_unlock(&mflag);
    }
```

Figura 9 - receção de mensagens por parte do gestor

De forma a assegurar uma comunicação entre o gestor e os vários clientes, foi indispensável desenvolver uma arquitetura de visasse resolver esta problemática. Foi, então, criado um *named pipe* para o gestor, com o nome ***SERVER***, e um *named pipe* para cada um dos clientes que pretendam estabelecer ligação com o gestor, tendo cada um o nome de ***CLI*** seguido do seu ***process id***.

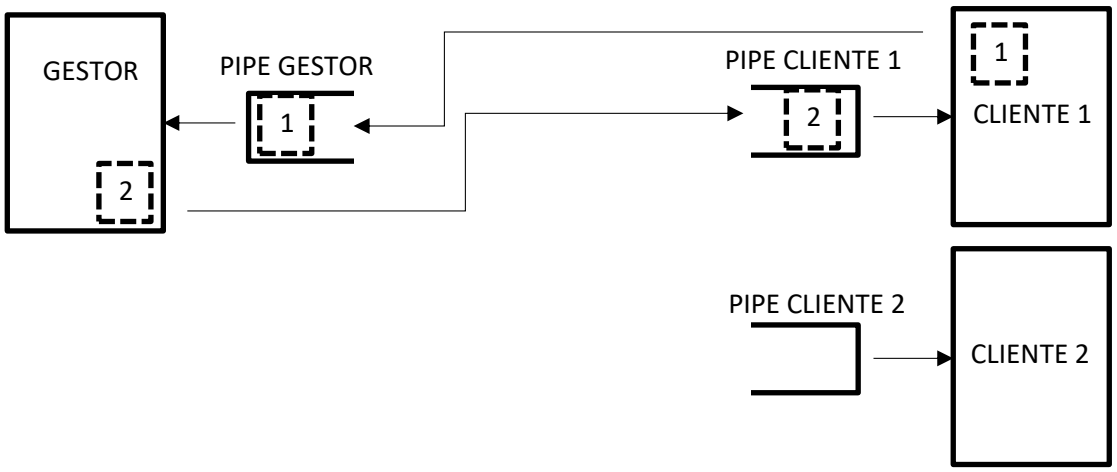

Figura 10 - arquitetura da comunicação

Imaginando o cenário de uma ligação de um cliente ao gestor: o cliente insere o seu *username* e enviado um ***package*** ao gestor com essa informação, usando o ***named pipe*** do gestor. Aquando receção dessa informação, o gestor faz a verificação e emite uma mensagem ao cliente, enviando-lhe uma *flag* de controlo a partir do seu ***named pipe***, que, depois, será compreendida dentro do cliente.

3.9. THREADS (SERVIDOR)

```
// Threads
tasks = (pthread_t*) malloc(sizeof (pthread_t) * 3); // 3 = numero de threads

if (pthread_create(&tasks[0], NULL, receptionOfData, threadPKG) != 0) {
    printf("MSGDIST: (ERROR) Creating thread.\n");
}
```

Figura 11 - criação e lançamento da thread

Como forma de assegurar a execução de tarefas em simultâneo com a leitura de comandos por parte do gestor, foi implementado uma ***thread*** que recebe todas as mensagens dos clientes. A ***thread*** tem também a tarefa de dividir essa informação a partir do seu conteúdo, isto é, se a mensagem for um *username*, este será enviado para a verificação do mesmo, enquanto que se informação que for recebida for uma mensagem criada pelo cliente, esta será enviada para a validação de mensagens.

3.10. CONSOLA CLIENTE BASEADA NA BIBLIOTECA NCURSES

```
mvprintw(0, 0, "TOPIC: \n");
mvprintw(1, 0, ">> \n");
mvprintw(2, 0, "TITLE: \n");
mvprintw(3, 0, ">> \n");
mvprintw(4, 0, "MESSAGE: \n");
mvprintw(5, 0, ">> \n");
move(posy, posx);
do {
    ch = getch();
    oposy = posy;
    oposx = posx;
    switch (ch) {
        case KEY_UP:
            if (posy == 3 || posy == 5) {
                posy -= 2;
                posx = 4;
            } else if (posy > 1) {
                posy--;
                posx = 4;
            }

            break;
        case KEY_DOWN:
            if (posy == 1 || posy == 3) {
                posy += 2;
                posx = 4;
            } else if (posy < (nrow - 1)) {
                posy++;
                posx = 4;
            }
            break;
        case KEY_LEFT:
            posx = (posx > 4) ? posx - 1 : posx;
            break;
        case KEY_RIGHT:
            posx = (posx < (ncol - 1)) ? posx + 1 : posx;
            break;
        case KEY_BACKSPACE:
            if (posx > 4) {
                posx = posx - 1;
                mvdelch(posy, posx);
```

Figura 12 - início da leitura de teclas pressionadas

Finalmente, foi criada, com recurso às funções da biblioteca *ncurses* uma consola de escrita de mensagens. Nesta consola, o cliente utiliza as teclas de navegação para navegar pela consola e preencher os campos necessários. Quando terminar, apenas precisa de carregar ENTER para finalizar o processo de criação da mensagem.

4. FICHEIROS ADICIONAIS

4.1. MAKEFILE

```
all: msgdist_server.c msgdist_client.c
    gcc msgdist_server.c msgdist_utils.h -o msgser
    gcc msgdist_client.c msgdist_utils.h -o msgcli
    gcc verificador.c -o verificador

verif: verificador.c
    gcc verificador.c -o verificador

server: msgdist_server.c
    gcc msgdist_server.c -o msgser

client: msgdist_client.c
    gcc msgdist_client.c -o msgcli

clear:
    rm msgcli msgser verificador
```

Figura 13 - ficheiro Makefile

O ficheiro ***Makefile*** é bastante explícito e segue as implementações enunciadas: ao executar **"make all"** na linha de comandos, compila todos os programas intervenientes do projeto. Para se compilar o verificador, o servidor ou o cliente, executa-se **"make verif"**, **"make server"** ou **"make cliente"**, respetivamente. De modo a limpar os executáveis, executa-se **"make clear"**. É importante referir que o ficheiro ***Makefile*** só funcionará se estiver presente na mesma diretoria que os outros ficheiros necessários e que fazem parte integrante no projeto.

4.2. SCRIPTS

Existem, ainda, dois ficheiros que foram criados para sentido de teste: ***initvar.sh*** e ***rmvar.sh***, onde o primeiro cria algumas variáveis de ambiente e o segundo remove eventuais variáveis criadas.

5. VERIFICAÇÃO, VALIDAÇÃO E OUTRAS QUESTÕES

5.1. FUNCIONALIDADES IMPLEMENTADAS

Devido a uma fraca flexibilidade horária e a pouca disponibilidade, todo o trabalho prático sofreu em termos de funcionalidades e problemas resolvidos. A comunicação entre o cliente e o servidor, apesar de estar codificada, não se encontra funcional.

5.2. TESTES

Devido às grandes mudanças no projeto, foram feitos testes específicos a cada funcionalidade criada, de modo a verificar a fiabilidade das funções requeridas de cada programa.

5.3. ORGANIZAÇÃO DO CÓDIGO

Esta meta continuou a reger-se pelas boas práticas de organização de código, de modo a simplificar a sua leitura, navegação e posteriores implementações.

5.4. ANOMALIAS NO CÓDIGO

Existe um reduzido número de anomalias que não foram corrigidas, ou por falta de conhecimento de tal erro, ou até mesmo por alteração do plano final de entrega, de modo a simplificar a estrutura do projeto num todo.

Entre elas, são de referir:

- Erro de memória ao criar uma segunda mensagem consecutiva pelo cliente;
- *Buffer* da escrita do *body* da mensagem não aloca 1000 caracteres, apesar de estar preparado para isso;
- Ao criar uma segunda mensagem, as impressões no ecrã mostram-se incorretas;
- O cliente não consegue realizar corretamente uma subscrição a um tópico existente;
- Não lista tópicos que contenham mensagens.

5.5. OBSERVAÇÕES E CONCLUSÕES FINAIS

Este projeto mostrou-se um desafio para cada um de nós. Foi realmente um projeto enriquecedor em termos de aprendizagem, e criou bastante empenho em ambos. Não só deu gosto de acabar com um programa maioritariamente funcional, como serviu de um estudo continuo sobre a matéria dada. É certo que, neste momento, encontramo-nos muito mais preparados para o exame final da unidade curricular.