Relatório T.1

Maria Eduarda de Melo Hang (17202304)

Abril 2021

1 Configuração do sistema

O usuário pode configurar os seguintes parâmetros no sistema:

- Tamanho da memória;
- Endereço IP dos servidores;
- Número base da porta do servidores;
- Número do servidor;
- Quantidade de servidores;
- Periodicidade do *logger*.

A Figura 1 mostra o arquivo **config.txt**, cujo caminho é **config/config.txt**, onde é possível configurar os parâmetros. O **memory_size** será o valor para o tamanho da memória, ou seja, quantos caracteres cada memória de um servidor deve ter. O **server_base_ip** será o endereço IP para todos os servidores enquanto que **server_base_port** será o valor de base para as portas dos servidores, ou seja, se tivermos três servidores os endereços serão 192.168.0.7 : 9734, 192.168.0.7 : 9735 e 192.168.0.7 : 9736 considerando os valores que estão na Figura 1. O parâmetro **server_range_index** determina qual faixa da memória da qual o servidor é responsável, por exemplo, se for colocado o valor 0, o servidor será responsável pelos 5 primeiros caracteres da memória (Memória[0:4]), considerando novamente os valores da Figura 1. Caso fosse 1, seria responsável pelos endereços de 5 a 9. Por fim, os parâmetros **number_of_servers** e **log_time** são a quantidade de servidores e a periodicidade do *logger* respectivamente.

2 Detalhes de implementação e decisões de projeto

Optei por deixar a estrutura do sistema mais rigorosa para facilitar a implementação, uma vez que limito a quantidade de servidores que podem estar ativos pelo arquivo de configurações, não deixando o sistema mais dinâmico.

```
memory_size=5
server_base_ip=192.168.0.7
server_base_port=9734
server_range_index=0
number_of_servers=2
log_time=20
```

Figura 1: Configuração do sistema

Além disso, com essa estrutura é possível que a própria aplicação do cliente determine quais servidores precisarão ser acessados para obter os valores da memória, evitando assim a necessidade de um *proxy* para redirecionar os clientes.

2.1 Servidor de memória compartilhada

O servidor é responsável por receber as requisições dos clientes, verificar se é uma escrita ou leitura e modificar ou acessar a memória conforme a requisição. Essa aplicação não deve tratar nenhum erro do usuário, uma vez que isso é feito pela aplicação do cliente. As memórias foram protegidas das condições de corrida através da adição de um ***pthread_write_read_lock*** em cada célula, ou seja, vários processos podem ler a célula ao mesmo tempo (como um semáforo), enquanto que apenas um processo pode escrever nela (como um *lock*). Além disso, se algum processo estiver lendo, o outro não pode escrever ao mesmo tempo e vice-versa. As Figuras 2 e 3 mostram como é feita a leitura e a escrita na memória de um servidor respectivamente.

```c
char * read_memory(shared_memory_element_t * memory, char * result, int start, int end) {
    int j = 0;
    for (int i = start; i <= end; i++) {
        pthread_rwlock_rdlock(&(memory[i].lock));
        result[j] = memory[i].value;
        pthread_rwlock_unlock(&(memory[i]).lock);
        j++;
    }
    return result;
}
```

Figura 2: Leitura na memória

```c
void write_memory(shared_memory_element_t * memory, char value, int index) {
    pthread_rwlock_wrlock(&(memory[index].lock));
    memory[index].value = value;
    pthread_rwlock_unlock(&(memory[index].lock));
}
```

Figura 3: Escrita na memória

Para cada conexão estabelecida, o servidor criar uma *thread* nova e atribuí

a função ***handler*** para tratar a requisição do cliente. A Figura 4 mostra a parte do código responsável por essa atribuição enquanto que a Figura 5 mostra o método ***handler***. Essa função basicamente irá verificar se é uma leitura ou escrita, se for uma leitura, irá ler da memória e escreverá no *socket* o resultado, caso contrário apenas escreverá na memória.

As requisições que são aceitas pelo servidor são um pouco diferentes das que são feitas pelo usuário no lado do cliente, sendo elas:

- $r\#x\#y$: Leia da memória a partir da posição x até a posição y
- $w\#x\#y\#s$: Escreva na memória a frase s a partir da posição x até a y

```
pthread_t thread;
handler_args_t args;
while (1)
{
    client_sockfd = accept(server_sockfd, (struct sockaddr *)&client_address, &client_len);

    args.client_sockfd = client_sockfd;
    args.configs = configs;
    args.memory = memory;
    pthread_create(&thread, NULL, handler, &args);
}
```

Figura 4: Servidor delegando o tratamento da requisição a uma *thread*

```
void * handler(void * arg) {
    handler_args_t * args = (handler_args_t *) arg;
    char request[REQ_SIZE], result[REQ_SIZE] = "";
    read(args->client_sockfd, &request, REQ_SIZE);
    verify_client_request(result, request[0], request, args->configs, args->memory);
    write(args->client_sockfd, &result, sizeof(result));
    close(args->client_sockfd);
}
```

Figura 5: Método *handler*

3 Cliente

O cliente é responsável por receber as requisições do usuário, impedir alguns erros e descobrir para quais servidores as requisições precisam ser enviadas. O usuário pode realizar os seguintes comandos:

- $r\ x\ n$: Leia n caracteres partir da posição x;
- $w\ x\ t\ n$: Escreve a frase t, de tamanho n, a partir da posição x;
- q: Saia da aplicação;

- *h*: Mostre os comandos da aplicação.

A partir da requisição do usuário, a aplicação precisará descobrir quais servidores precisam ser solicitados para atender à necessidade do usuário. Por exemplo, se temos uma memória de 5 caracteres por servidor, temos 2 servidores e o usuário solicita a leitura de 6 caracteres a partir da posição 0, o processo cliente precisará separar a requisição em duas, uma para o primeiro servidor pedindo a leitura de todos os caracteres e outra para o segundo lendo apenas o primeiro caractere. Caso o usuário tente ler mais caracteres do que existe na memória compartilhada ou colocar a posição inicial com um valor maior que o último índice da memória compartilhada, receberá um erro. O mesmo vale para a escrita.

Após descobrir quais os servidores que serão solicitados, o cliente irá criar algumas *threads* para realizarem as requisições para cada servidor. No caso de ser uma leitura, a função que será escolhida é a ***reader***, caso contrário será a ***writer***. Ambas realizam a conexão e enviam a requisição, conforme explicado na seção 2.1. Apenas os *readers* esperam a resposta do servidor no *socket* para depois retornar o valor e mostrar ao usuário. As Figuras 6 e 7 mostram as funções de ***reader*** e ***writer*** respectivamente. O gif mostra como é realizada a troca de mensagens entre a aplicação do cliente e do servidor de forma mais completa.

3.1 *Logger*

O *logger* é um processo rodando junto com o servidor da memória compartilhada e é responsável por guardar o estado atual da memória com um período pré-definido. Sobre a implementação, foi desenvolvida uma função *logger* que será executada por uma *thread*. Essa função fica em um laço infinito e utiliza o método ***sleep(t)*** do *pthreads* para fazer a *thread* esperar o tempo definido pelo usuário, como mostrado na Figura 8. O *logger* responsável por pegar as memórias é o com número do servidor mais alto (**server_range_index**), essa ideia foi retirada do algoritmo *bully* de sistemas distribuídos. Caso seja o responsável, o processo criará *threads* com a função ***reader*** para solicitar as memórias e esperará pelo retorno delas. Assim que todas retornarem, ele concatenará os resultados com o da própria memória e escrever em um arquivo **.txt** no diretório **logs**. As Figuras 9 e 10 mostram a parte da criação dos ***readers*** e a concatenação dos resultados e escrita no arquivo de *log* respectivamente. Caso contrário, o processo de *logger* não fará nada. O método de escrita do arquivo de *log* é simples, primeiro o programa pega um *timestamp* para adicionar ao nome do arquivo, de forma a terem nomes diferentes, e escreve no arquivo o parâmetro recebido.

4 Limitações

O serviço desenvolvido não é confiável o suficiente para ser colocado em produção, uma vez que existem vários pontos de melhora. Esses pontos serão

```c
void *reader(void *arg)
{
    reader_args_t *args = (reader_args_t *)arg;

    // Configuring to connect with the server
    int sockfd, len, result;
    struct sockaddr_in address;
    unsigned int mem_size = args->mem_size;
    char answer[mem_size], read_request[mem_size];

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = inet_addr(args->server_base_ip);
    address.sin_port = args->server_base_port + args->server_number;
    len = sizeof(address);

    result = connect(sockfd, (struct sockaddr *)&address, len);

    if (result == -1)
    {
        perror("Was not possible to read the memory :(\n");
        return 0;
    }

    sprintf(read_request, "r#%d#%d", args->start, args->end);
    write(sockfd, &read_request, REQ_SIZE);
    read(sockfd, &answer, mem_size);
    pthread_exit((void *)answer);
}
```

Figura 6: Método de leitura utilizado pelas *threads* do cliente

discutidos nas subseções seguintes.

4.1 *Logger*

Um dos problemas encontrados se encontra no caso do *logger* responsável parar de funcionar, impedindo a funcionalidade de ser realizada. Uma forma de resolver esse problema seria utilizar um algoritmo de eleição, como de *bully*, já que se um servidor perceber que o *logger* responsável parar de funcionar, seria realizada uma eleição para escolher um novo responsável.

4.2 Servidor

A mesma falha citada na subseção anterior vale para os servidores, visto que se um servidor da memória compartilhada falhar, a faixa da qual ele é responsável não estará mais disponível. Uma solução seria replicar os servidores, todavia o problema de sincronização surgiria, já que seria necessário manter as

```c
void *writer(void *arg)
{
    writer_args_t * args = ( writer_args_t *)arg;

    // Configuring to connect with the server
    int sockfd, len, result;
    struct sockaddr_in address;
    unsigned int mem_size = args->mem_size;
    char answer[mem_size], write_request[mem_size];

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = inet_addr(args->server_base_ip);
    address.sin_port = args->server_base_port + args->server_number;
    len = sizeof(address);

    result = connect(sockfd, (struct sockaddr *)&address, len);

    if (result == -1)
    {
        perror("Was not possible to write the memory :(\n");
        exit(1);
    }

    sprintf(write_request, "w#%d#%d#%s", args->start, args->end, args->string);
    write(sockfd, &write_request, REQ_SIZE);
}
```

Figura 7: Método de escrita utilizado pelas *threads* do cliente

memórias replicadas iguais, aumentando drasticamente o *overhead* da aplicação. Outro problema de existir apenas um servidor para cada faixa de memória é a sobrecarga, deixando o servidor muito lento e os usuários insatisfeitos caso muitos clientes façam requisições.

4.3 Cliente

Caso o usuário solicite uma leitura, o terminal ficará ocupado impedindo novas requisições enquanto espera pelo retorno dos servidores. Se isso for combinado com a sobrecarga do servidor, o usuário pode ter que esperar muito para realizar outro comando. Além disso, nem todos os erros do usuário foram tratados, como verificar se o tamanho que ele enviou para uma escrita é realmente o tamanho da string enviada pelo terminal.

```c
void *logger(void *arg)
{
    logger_args_t *args = (logger_args_t *)arg;
    configs_t configs = args->configs;

    while(1) {
        sleep(configs.log_time); // sleep a while

        const int num_servers = configs.number_of_servers - 1;
        const int mem_size = configs.memory_size;
```

Figura 8: Começo do método do *logger*

```c
if (isLoggerLeader(configs.server_range_index, configs.number_of_servers))
{
    if (num_servers >= 1)
    {
        pthread_t readers[num_servers];
        reader_args_t r_args[num_servers];

        for (int i = 0; i < num_servers; i++)
        {
            r_args[i].server_number = i;
            r_args[i].start = 0;
            r_args[i].end = mem_size - 1;
            r_args[i].server_base_port = configs.server_base_port;
            r_args[i].mem_size = mem_size;
            strcpy(r_args[i].server_base_ip, configs.server_base_ip);

            pthread_create(&(readers[i]), NULL, reader, &(r_args[i]));
        }

        char *result = (char *)malloc(sizeof(char) * num_servers * mem_size);
```

Figura 9: Criação dos *readers* para pedirem as memórias dos outros servidores

```c
char *result = (char *)malloc(sizeof(char) * num_servers * mem_size);

for (int i = 0; i < num_servers; i++)
{
    char *temp;
    pthread_join(readers[i], (void **)&temp);
    strcat(result, temp);
}

char my_mem[mem_size];
read_memory(args->memory, my_mem, 0, mem_size - 1);

// Como ele eh o id mais alto sempre, vai ser o ultimo a concatenar
strcat(result, my_mem);
write_log(result);

free(result);
```

Figura 10: Concatenação dos resultados e escrita no arquivo de *log*