Aluno: João Pedro Moura Oliveira - 190030879

Lab01 – Construindo aplicações distribuídas usando RPC
PSPD – Programação para Sistemas Paralelos e Distribuídos
Turma A (Prof. Fernando W Cruz)

1) Introdução

O RPC ou *Remote Procedure Call's* é um mecanismo de troca de mensagens muito utilizado em sistemas distribuídos. As chamadas de sistemas distribuídos se assemelham bastante àquelas de sistemas locais, com a diferença que nesses as funções ficam hospedadas em hosts/sistemas distintos.

Portanto, para ocorrer essa comunicação o RPC permite ao desenvolvedor - através de um arquivo de definição de interfaces - definir e encapsular todo o código de forma simples e eficiente, os conhecidos *Stubs*. Além disso, o RPC associado à biblioteca XDR (*eXternal Data Representation*) permite padronizar e converter os diversos tipos de dados, permitindo a comunicação entre os diferentes tipos de máquinas.

Entendendo as capacidades e o funcionamento do RPC parte-se então para o problema, construir uma aplicação distribuída que calcula o menor e o maior valor de um vetor gerado pela seguinte equação simplificada:

$$v[i] = \sqrt{(i - \frac{tamanho\_do\_vetor}{2})^2}$$

$$v[i] = (i - \frac{tamanho\_do\_vetor}{2})$$

Para a resolução, propõe-se então a seguinte definição de interface que atende às necessidades do problema (Figura 1). Nessa interface, é perceptível a utilização de um vetor de *float's* que permite a escalabilidade para todas as ocasiões do problema. Por fim, é possível ver que o caminho escolhido para resolução desse problema foi utilizando-se duas funções, sendo uma para calcular o menor valor do vetor e uma para encontrar o maior.

FIGURA 1 - Arquivo de definição de interface (IDL).

```
typedef float vetor<>;
program PROG {
    version VERSAO {
        float MENOR(vetor v) = 1;
        float MAIOR(vetor v) = 2;
    } = 10;
} = 100;
```

Fonte: Autor

2) Solução com 1 *worker* sem *threads*

A primeira resolução para esse problema encontrada, foi utilizando-se apenas um *worker* para encontrar o menor e o maior valor dentro do vetor passado. O programa desenvolvido conseguiu trabalhar corretamente e eficientemente com vetores de tamanho até um pouco mais de 1000 posições, equivalente aos 8*Kbytes* suportados pela conexão UDP conforme documentado nas *manuals pages* da função *clnt_create()* do RPC [1].

Para superar essa dificuldade, passou-se a utilizar a conexão TCP que não possui essa limitação. Entretanto, apesar de conseguir carregar vetores maiores, o tamanho dos mesmos passaram a ser fatores limitantes para a velocidade do cálculo da resposta. Isso impediu o cálculo para estruturas com mais de 1.000.000.000 posições, visto que a demora causava a finalização da conexão sendo enviado um sinal de *kill* para a aplicação RPC.

3) Solução com 2 ou mais *worker* sem *threads*

Para solucionar o problema da velocidade e da interrupção da conexão, realizou-se a confecção de uma solução que aceita 2 ou mais *workers*. Essa solução, permitiu ultrapassar o limite de 1.000.000.000 posições porém não melhorou o tempo necessário para o cálculo da resposta. Isso se deve ao fato da solução não utilizar *threads* para a consulta nos *workers* de forma paralela.

O mecanismo adotado para coordenar e organizar as respostas geradas pelos *workers* foi bem simples, visto que por ser uma aplicação sem concorrência não existia a necessidade de se checar qual seria o menor e o maior valor de acordo com as respostas recebidas, pois o menor sempre estaria na solução do primeiro *worker*, enquanto a maior sempre estaria na do último.

4) Solução com 2 ou mais *worker* com *threads*

Para resolver os problemas pendentes das arquiteturas anteriores, passa-se a utilizar portanto *threads* em conjunto com os *workers*. Apesar de ser mais eficiente, essa solução é bem mais complexa e exige bem mais dos processadores para a realização das requisições e ordenação das respostas.

O código anterior teve que ser modificado e melhorado para comportar a utilização das *threads* dentro do *Stub* do cliente. A primeira dessas alterações se faz em relação à utilização de uma *struct* (Figura 2) essencial para coordenação das *threads* e das respostas recebidas pelos *servers*.

A segunda mudança essencial, está no fluxo principal de chamada dos *clients* e coordenação e ordenação dos resultados. A diferença se encontra principalmente na utilização de três *loops* (Figura 3), sendo o primeiro responsável pela *linkagem* do *Stub* do cliente como os *Stubs* dos servidores, em adição da criação das *threads* para o cálculo em paralelo. O segundo *loop* é essencial para garantir que todas as respostas foram recebidas, ou seja, é responsável por dar *join* em todas as *threads* criadas. E por fim, o terceiro *loop* passa por todas as respostas a fim de encontrar o maior e o menor valor dentre as respostas recebidas.

Figura 2 e 3 - *Struct* principal do programa e três *loops* principais.

```c
typedef struct a {
        float result_1;
        float result_2;
} resps;


typedef struct b {
        CLIENT *clnt;
        int tam;
        int inicio;
        int final;
        struct a resposta;
        pthread_t tid;
} param_thread;
```

```c
for (int i = 0; i < qtdClnts; i++) {
        CLIENT *clnt;
        clnt = create_server (argv[i + 2], "tcp");
        int inicio = i * tam_por_server;

        resps[i].clnt = clnt, resps[i].tam = tam;
        resps[i].inicio = inicio;

        if (i + 1 < qtdClnts) resps[i].final = inicio + tam_por_server;
        else resps[i].final = inicio + tam_por_server + (tam % qtdClnts);
        pthread_create (&(resps[i].tid), NULL, prepare_serve, &resps[i]);
}

printf("[CLIENTE]: Esperando finalizacao de %d Thread(s)...\n", qtdClnts);
for (int i = 0; i < qtdClnts; i++)
        pthread_join(resps[i].tid, NULL);

printf("[CLIENTE]: Calculando o menor e o maior valor dos vetores...\n");
menor = resps[0].resposta.result_1;
maior = resps[0].resposta.result_2;
for (int i = 1; i < qtdClnts; i++) {
        if (resps[i].resposta.result_1 < menor) menor = resps[i].resposta.result_1;
        else if (resps[i].resposta.result_2 > maior) maior = resps[i].resposta.result_2;
}
```

Fonte: Autor

Apesar dessa solução atender às principais necessidades do problema, ela é extremamente exaustiva para os processadores. Isso se deve ao fato de que uma *thread* é criada para cada *worker*, gerando uma dependência desnecessária que pode ser corrigida com a ciclagem das *threads* pelos *workers* que estiverem inativos.

Outro ponto importante de se ressaltar, é que toda essa solução foi feita utilizando-se dockers em uma única máquina, gerando um *stress* bem maior - para uma arquitetura com um *server* local e dois *servers* em dockers alcançava-se facilmente 100% em 6 dos 8 núcleos do processador - caso a aplicação fosse dividida em diversas máquinas.

5) Tabela comparativa de desempenho

Finalizados os códigos, foram testadas cada solução utilizando-se um vetor de 1.000.000.000 posições em uma máquina com 8GBytes de memória RAM e um processador com 8 núcleos de processamento. Os resultados obtidos foram os seguintes:

Programa	Quantidade de *Servers*	Tempo gasto (s)	CPU utilizado (%)
letra_a/lab1_a_client	1	24,235 total (killed)	69
letra_b/lab1_b_client	2 (1 docker)	19,791 total	65
letra_b/lab1_b_client	3 (2 dockers)	20,248 total	63
extra_com_threads/lab1_b_client	3 (2 dockers)	11,566 total	197
extra_com_threads/lab1_b_client	4 (3 dockers)	11,390 total	229

5) Opinião geral e nota

Em relação à opinião geral desse laboratório, eu achei bem interessante pois consegui aprender bastante sobre RPC, principalmente como o rpcgen gera os arquivos e funções, além de lembrar conceitos de *threads* e suas utilizações. As principais dificuldades e limitações encontradas nesse projeto foram principalmente em relação ao docker, na sua criação, subida e utilização.

Muito tempo foi gasto para conseguir realizar a criação de um docker que comportasse e funcionasse com o rpcbind. As principais limitações também fazem referência ao docker, visto que ao utilizar o docker com *threads* alcançava-se facilmente 100% de uso em no mínimo dois núcleos do processador, impedindo que o experimento fosse testado com mais máquinas.

Por fim, acredito que minha nota nesse laboratório deveria ser 10 pois muito foi aprendido e muito também foi desenvolvido, sendo necessário quase que uma semana inteira para a completude do experimento.

6) Referências

[1] rpc(3) - Linux man page. Disponível em: <https://linux.die.net/man/3/rpc>. Acesso em: 10 de fev. de 2022.