Aluno: João Pedro Moura Oliveira - 190030879

Lab02 – Construindo aplicações distribuídas usando *sockets*
PSPD – Programação para Sistemas Paralelos e Distribuídos
Turma A (Prof. Fernando W Cruz)

1) Introdução

Após realizado o estudo e a experimentação com as *Remote Procedure Calls* (RPC), passa-se a utilizar uma abordagem diferente para a criação e transmissão de dados de um sistema distribuído. Em vez de se utilizar a camada de transporte, como o RPC fazia, agora se faz o uso da camada de rede através dos *sockets*. Em decorrência disso, uma das principais diferenças entre as abordagens é a não existência do uso do *portmapper* para mapeamento e redirecionamento das conexões do sistema distribuído construído.

Além disso, ao contrário das aplicações baseadas em chamadas de procedimento remoto, os *sockets* não fazem o uso de uma linguagem de definição de interface (*Interface Definition Language*), ficando à disposição do programador como as informações serão transmitidas pela camada de rede. Outro ponto muito importante, é que fica a critério do desenvolvedor como será o procedimento de realização da conexão *client-worker* e como serão tratadas as respostas, que podem muitas vezes virem incompletas.

Para a realização da passagem das informações utilizando essa nova arquitetura, ao contrário do RPC que possui conversões automatizadas através do XDR (*eXternal Data Representation*), os soquetes devem ser programado defensivamente para evitarem erros na transferência dos valores, levando em conta que toda a informação deve ser comunicada em *bytes* na camada de rede. Isso ocasiona na necessidade do programador em desenvolver uma aplicação que realize as conversões necessárias para os diferentes sistemas.

Portanto, entendido as principais diferenças e necessidades entre as diferentes arquiteturas parte-se para a definição do problema. Similar ao laboratório anterior, nesse também é requisitado a passagem de um vetor de *floats* calculado através da equação simplificada abaixo.

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

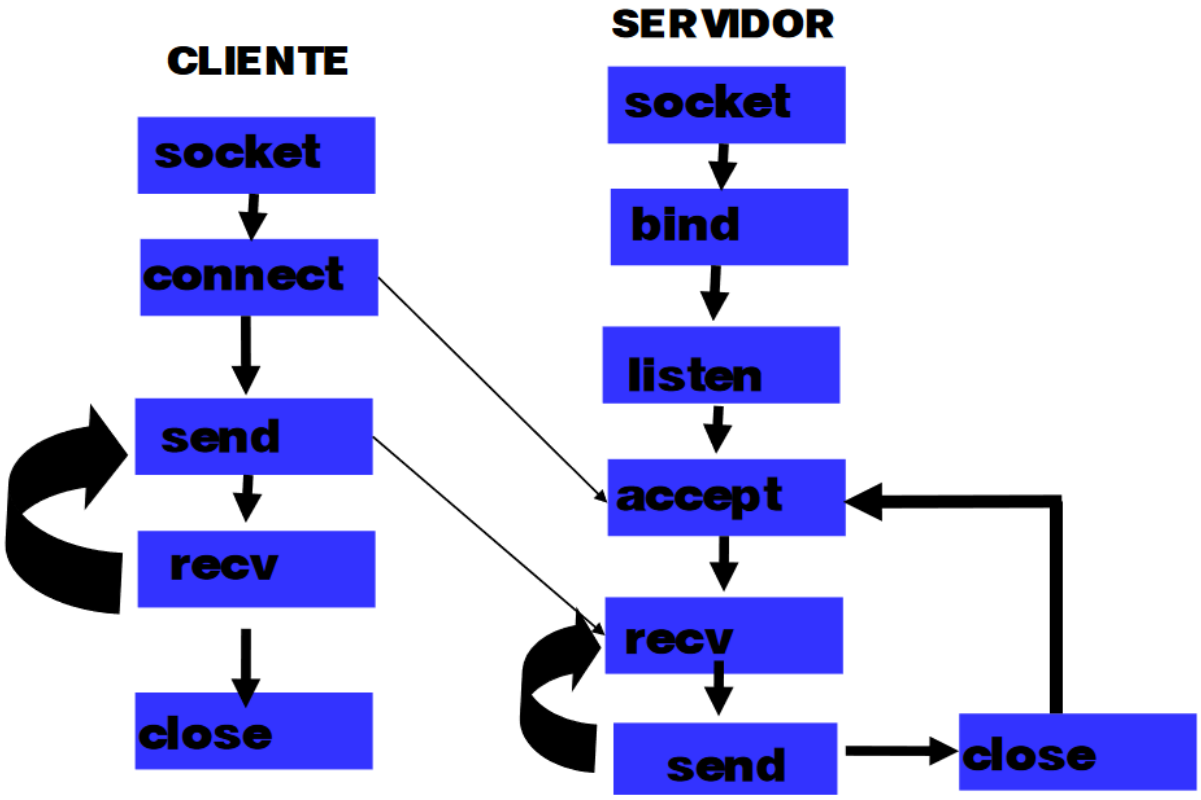
$$v[i] = (i - \frac{tamanho\_do\_vetor}{2})$$

Outro ponto muito importante na definição da arquitetura é a escolha de como será feita a comunicação da camada de rede, sendo as escolhas possíveis: UDP e TCP. A ideia principal desse laboratório seria o desenvolvimento utilizando ambas, porém devido ao tempo e problemas decorrentes da implementação do

protocolo UDP, não foi possível a realização das duas. Portanto, utilizou-se de uma base teórica para embasar a escolha do formato de comunicação TCP nos *sockets*.

A escolha por essa abordagem, se deve ao fato da necessidade de uma conexão *client* e *worker* confiável em contradição à conexão eficiente provida pelo protocolo UDP. A confiança nesse protocolo é decorrente do ciclo de vida TCP, onde é utilizado o mecanismo de *Handshake* (figura 1) para o estabelecimento da conexão (orientada). Outro ponto muito interessante para a utilização desse protocolo, está na capacidade *full duplex*, onde é possível a transmissão simultânea em ambas as direções utilizada para amenizar os problemas de eficiência (1).

FIGURA 1 - *Three-way Handshake*



Fonte: Willian F. C. 2022 (2).

Entendido as vantagens do TCP, inicia-se a discussão do motivo de sua escolha. O principal argumento se baseia na necessidade da transmissão de um vetor grande mantendo-se sua ordenação original, visto que, a posição dos seus índices é um fato necessário para a eficiência da resposta do servidor. Além disso, por permitir uma comunicação *full duplex* a utilização de múltiplos *workers* se encontra favorecida por essa característica.

2) Solução com 1 *worker* e 1 *client*

Para o primeiro problema, foi pensado uma solução utilizando-se apenas um *worker* e um *client* comunicando-se pelo protocolo TCP. O código criado para esse problema é bem intuitivo, visto que, o lado cliente segue o seguinte fluxo: configuração do *socket*, criação, conexão (*connect*), envio (*send*) e recebimento (*recv*) das informações e finalização da conexão (*close*).

O envio do vetor pela camada de rede, portanto, foi feito em N (tamanho do vetor) + 1 requisições (figura 2 e 3), onde a primeira delas indica o tamanho do vetor que será recebido enquanto as seguintes são os valores em si que estão armazenados no vetor do lado do cliente. Esse fluxo apesar de ser o mais intuitivo não é o mais eficiente, visto que, o envio de múltiplos pacotes na camada de rede acaba sendo bem mais custoso em relação ao tempo gasto ao se enviar um pacote único.

FIGURA 2 e 3 - *Loop* de geração do vetor e *loop* de envio dos pacotes

```c
void
generate_vetor (vetor *argp, int tam, int inicio, int final)
{
        int tamanho_v = final - inicio;
        argp->vetor_len = tamanho_v;
        argp->vetor_val = malloc (sizeof (float) * tamanho_v);
        printf("[VETOR] => Criando vetor com Inicio = %d, final = %d e tamanho = %d\n", inicio, final, tamanho_v);

        for (int i = 0; i < tamanho_v; i++)
                argp->vetor_val[i] = ((inicio + i) - tam / 2);
}
```

```c
// Envia dados para os sockets ja conectados
int tam = atoi (argv[3]);
generate_vetor (&bufout, tam, 0, tam);

int tmp = htonl(bufout.vetor_len);
send (sd, &tmp, sizeof (tmp), 0);

for (int i = 0; i < bufout.vetor_len; i++)
    send (sd, &bufout.vetor_val[i], sizeof (float), 0);
```

Fonte: Autor.

3) Solução com múltiplos *workers* e 1 *client*

Para solucionar o problema principal do envio de pacotes na rede, passou-se a realizar um único envio contendo toda a estrutura do vetor de *floats* (figura 4) que estava armazenado no lado do cliente. Isso permitiu a melhoria da velocidade de resposta do servidor em relação ao modelo anteriormente proposto.

FIGURA 4 - Novo formato de envio de pacotes

```c
int tmp = htonl(final - inicio);
send (sd, &tmp, sizeof (tmp), 0);

// Envia dados para os sockets ja conectados
vetor bufout;
generate_vetor (&bufout, tam, inicio, final);
send (sd, bufout.vetor_val, sizeof (float) * bufout.vetor_len, 0);
```

Fonte: Autor.

Além disso, para melhorar a eficiência do programa, projetou-se uma arquitetura em que são utilizados múltiplos *sockets* ou *workers* para o cálculo desse problema. Entretanto, apesar dessa divisão o programa continua sequencial, já que a implementação de *sockets* não garante que todos os pacotes chegarão ao mesmo tempo no *worker*. Por esse motivo, o servidor deve esperar que todos os pacotes cheguem (figura 5) para então poder enviar a resposta para o *client*, e como o mesmo utiliza a função *recv*() que é bloqueante, todo o fluxo do programa principal no cliente fica esperando as respostas dos *workers* (figura 6).

FIGURA 5 e 6 - Recebimento de respostas nos *workers* e nos *clients*

```
bufin.vetor_len = ntohl(tmp);
printf("[%s:%u] => Recebido tamanho do vetor: %d\n", inet_ntoa (infoCli.sin_addr), ntohs (infoCli.sin_port), bufin.vetor_len);

bufin.vetor_val = malloc (sizeof (float) * bufin.vetor_len);
printf("[%s:%u] => Recebido vetor: ", inet_ntoa (infoCli.sin_addr), ntohs (infoCli.sin_port));
recv (sd, bufin.vetor_val, sizeof (float) * bufin.vetor_len, MSG_WAITALL);
```

```
// Calcula o menor e o maior valor retornado
float menorTmp, maiorTmp;
recv (sd, &menorTmp, sizeof (float), 0);
recv (sd, &maiorTmp, sizeof (float), 0);
// printf("%f %f\n", menorTmp, maiorTmp);

if (i == 0) menor = menorTmp;
if (i == qtdClnts - 1) maior = maiorTmp;

close (sd);
free(bufout.vetor_val);
```

Fonte: Autor.

Apesar dessa limitação, o recebimento e coordenação de respostas dos servidores no cliente é simplificada visto que todo o processo é sequencial. Dessa maneira, e pela característica de geração do vetor possibilitar isso, é garantido que sempre o menor valor estará presente na resposta do primeiro *worker*, enquanto a maior resposta estará disponível na resposta do último. Portanto, basta fazer uma simples checagem que será obtido os valores esperados (figura 6).

4) Tabela comparativa de desempenho

Após realizado o desenvolvimento da aplicação com múltiplos *workers*, iniciou-se a última etapa desse laboratório, que é a realização do comparativo de desempenho. Como já citado anteriormente, devido às limitações dos *sockets* e a característica sequencial da aplicação, não houve um ganho significativo de eficiência de acordo com a quantidade de servidores. Vale ressaltar que, todo esse cenário de testes foi realizado em uma única máquina, que possui uma CPU com 8 núcleos utilizando uma entrada padrão de 1.000.000.000 valores.

Quantidade de *workers*	Tempo gasto (s)	CPU utilizada (%)
2	4,978 total	84
4	4,904 total	86
6	4,847 total	86
8	4,898 total	87
10	4,939 total	86

5) Opinião geral e nota

Ao concluir esse laboratório, portanto, foi possível realizar uma comparação bem interessante com relação ao RPC. A primeira delas pode ser vista no gráfico de desempenho, que demonstra que a utilização de *sockets* foi bem mais eficiente que as chamadas de procedimento remoto. Entretanto, esse desempenho trouxe como principal problema a necessidade do programador converter, gerenciar e validar a transferência de informações, processo esse feito anteriormente pelas bibliotecas do RPC.

Apesar de todos os problemas encontrados nesse laboratório, assim como o anterior, me proporcionou o aprendizado em diversos conteúdos relacionados à aplicações distribuídas. Entretanto, não pude extrair a mesma quantidade de informações como nas chamadas de procedimentos remotos, e isso se deve à limitação de tempo encontrada no decorrer da semana. Por fim, a nota que me atribuiria para esse laboratório está entre 7 e 8.

6) Referências

[1] **Protocolo de controle de transmissão**. Wikipedia. Disponível em: <https://en.wikipedia.org/wiki/Transmission_Control_Protocol>. Acesso em: 20 de fev. de 2022.

[2] **Diálogo Cliente-Servidor com TCP**. Fundamentos de Redes de Computadores. Prof. Fernando William Cruz.