Universidade Federal de Ouro Preto
Departamento de computação

Relatório de Redes

Alunos:
Caio Costa
Gabriel Caetano
Lucas Urzedo
Matheus Almeida

Abril 2021
Ouro Preto - Minas Gerais - Brasil

Universidade Federal de Ouro Preto
Departamento de Computação
Programa

Relatório

Relatório de Redes, referente ao trabalho passado na disciplina de Redes de Computadores.

Alunos:
Caio Costa
Gabriel Caetano
Lucas Urzedo
Matheus Almeida

Abril 2021
Ouro Preto - Minas Gerais - Brasil

Conteúdo

1 Descrição do Problema **1**

2 Solução Implementada **2**
2.1 Solução TCP 2
2.1.1 Cliente 2
2.1.2 Servidor 4
2.2 Solução UDP 6
2.2.1 Cliente 6
2.2.2 Servidor 6

3 Cenários de Testes **8**

4 Análise dos Resultados **9**

Lista de Figuras

1 Essa imagem apresenta os testes do protocolo TCP sendo aplicados. 9
2 Essa imagem apresenta os testes do protocolo UDP sendo aplicados. 9

1 Descrição do Problema

O trabalho em questão trata-se de um contador de caracteres, onde o programa desenvolvido deverá ler um arquivo com *strings* aleatórias e criar um arquivo de saída com o número de vogais, consoantes e números de cada *string*. Caso exista caracteres especiais, será retornado no arquivo de saída o texto: "erro".

Para que o programa seja executado, é preciso ter um servidor que gerencie os pedidos de contagem de cada cliente. Também é necessário que os clientes forneçam os arquivos a serem processados e também estejam prontos para processar os pedidos do servidor.

Então, o que basicamente acontece no programa é: o servidor é aberto e espera algum cliente enviar para ele alguma mensagem contendo as *strings* que precisam ser processadas. Assim que um cliente envia essa mensagem, o servidor passa a procurar outros clientes, que não seja o mesmo que esteja fazendo a requisição, para processar e contar quantas vogais, consoantes e números há naquela *string*. Assim que aquele cliente responder com os valores

encontrados o servidor devolve para o cliente que fez a requisição os dados. Após todas as *strings* serem finalizadas, os clientes são fechados e também o servidor e todos os arquivos com os resultados já vão ter sido gerados.

2 Solução Implementada

O programa foi desenvolvido em *JavaScript* devido ao fato de a maioria dos integrantes terem maior conhecimento com essa linguagem de programação e, além disso, ser uma linguagem que torna possível implementar os protocolos TCP e UDP de forma mais simples.

2.1 Solução TCP

Para a implementação do protocolo TCP no código em *JavaScript* foi importada a biblioteca *net* e para a leitura de arquivos foi utilizada a biblioteca *FileSystem (fs)*.

De forma que fique mais simples para a compreensão de como foi solucionado o problema é necessário que seja explicado o cliente e servidor de forma separada.

2.1.1 Cliente

O cliente é quem executa os pedidos do servidor, nele que é feito a contagem de número de vogais, consoantes, números e indica se há erro ou não. O cliente também é responsável por enviar ao servidor as *strings* que precisam ser processadas por outros clientes.

Para fazer a contagem das *strings* foi utilizado o *Regex* para facilitar nessa contagem. O código a seguir mostra como foram feita as contagens *strings*.

```
1     let regC = /[b-df-hj-np-tv-z]/gi;
2     let regV = /[aeiou]/gi;
3     let regN = /\d/gi;
4
5     let consoants = data.input.match(regC)?.length || 0;
6     let vowels = data.input.match(regV)?.length || 0;
7     let numbers = data.input.match(regN)?.length || 0;
8
9     if (consoants == 0 && vowels == 0 && numbers == 0) {
10      client.write(JSON.stringify({ output: 'erro\n', client:
            data.client }));
11    } else {
12      let output = `C=${consoants};V=${vowels};N=${numbers}\n
            `;
```

```
13        client.write(JSON.stringify({ output, client: data.
              client }));
14    }
```

Listing 1: Regex

As variáveis *regC*, *regV*, *regN*, guardam as expressões que serão utilizadas no *Regex* como pode se observar nas linhas 6, 7 e 8. Logo após, faz-se uma verificação de que, se nao for encontrado nenhuma consoante, vogal e números, devolve-se para o servidor a informação de erro, caso contrário, é enviada uma mensagem para o servidor com o *output* descrito na linha 12.

Além disso, o cliente faz a leitura dos arquivos que precisam ser processados. Isso se da de uma forma muito simples, faz-se a leitura do arquivo com somente os dados que sejam necessários, ou seja, é ignorado o que não seja o número de processos e as *strings* também. O código a seguir exemplifica melhor o que é feito.

```
1  function readFile() {;
2    let data = fs.readFileSync(`./tests/${process.argv[2]}`, {
         encoding: 'utf8' });
3    let dataSplit = data.split('\r\n');
4
5    if (dataSplit.length === 1) dataSplit = data.split('\n');
6
7    let operations = [];
8    dataSplit.forEach(str => {
9      if (!str.startsWith('//') && str != '')
10       operations.push(str);
11   });
12
13   if (Number(operations[0]) !== operations.length - 1) {
14     console.log('Invalid file!');
15     process.exit();
16   }
17
18   return operations;
19 }
```

Listing 2: Leitura de Arquivos

Primeiro é feita a leitura do arquivo como um todo com as quebras de linha $\backslash r \backslash n$ para caso o arquivo tenha sido escrito no Windows. Caso o arquivo tenha tamanho igual a 1, quer dizer que o arquivo nao foi leito de correta e por isso ele foi criado no Linux e deve-se fazer a leitura novamente, porem dessa vez só com $\backslash n$. Isso pode ser visualizado nas primeiras 5 linhas do código.

Logo em seguida, todas as *strings* que não são úteis nao para processamento são eliminadas. Isso pode ser observado nas linhas 8 à 11, onde

a condição *if* procura por textos que comecem com "//"ou seja um *string* vazia. A partir disso, verifica-se se o número de operações é diferente do tamanho do vetor de linhas, caso isso aconteça, o cliente finaliza o processo.

2.1.2 Servidor

O servidor é quem gerencia as requisições dos clientes e envia para eles a *strings* que precisa ser processada.

Assim que o servidor é inicializado, ele fica esperando que algum cliente o envie uma mensagem com as *strings* que precisam ser processadas, logo após isso ele verifica se a estrutura de dado *Map* esta vazia ou não, caso ela esteja vazia, o servidor inicializa ela com o cliente que está fazendo a requisição das *strings* e adiciona o número de operações a ser realizadas. Isso pode ser observado no código abaixo.

```
1  if (!clients.get(client)) {
2    clients.set(client, []);
3
4    if (data.operations) operations += Number(data.operations);
5    client.write(JSON.stringify({ output: 'connected', client
        }));
6  }
```

Listing 3: Comparação Map

Caso a estrutura de dados *Map* esteja inicializada, verifica-se se o *data* possui um *input*. Se ele tiver é gerado um valor aleatório para representar qual cliente deverá processar aquela *string* requisitada. Caso o valor gerado seja represente o mesmo cliente que esta fazendo a requisição, o processo é colocado em espera até que seja gerado um valor diferente daquele. Quando o valor for diferente, envia-se a *string* em questão para o cliente fazer o processamento. Isso é observado no código abaixo.

```
1  } else if (data.input) {
2    let randomClient = keys[Math.floor(Math.random() * keys.
        length)];
3
4    if (randomClient === client)
5      randomClient.write(JSON.stringify({
6        output: 'wait',
7        operations: data.input,
8        client: data.client
9      }));
10   else
11     randomClient.write(JSON.stringify(data));
```

```
12 }
```

Listing 4: Chamando Cliente Input

A próxima verificação feita pelo servidor é se ele possui algum *output*, ou seja, uma resposta do cliente. Caso tenha, o servidor procura no *Map* o cliente que fez a requisição de processamento da *string* e insere o resultado nessa estrutura de dado, para que dessa forma, sempre seja possível saber a qual cliente aquela *string* pertence. O código abaixo mostra de forma mais detalhada esse acontecimento.

```
1 } else if (data.output) {
2   let client = keys.find(c => c.remotePort === data.client.
        _peername.port);
3
4   clients.get(client).push(data.output);
5   client.write(JSON.stringify({ output: data.output, client:
        data.client }));
6   processed += 1;
7
8 }
```

Listing 5: Chamando Cliente Output

A verificação seguinte é de se o número de operações a serem feitas são iguais ao número de processos a serem feitos. Essa condição existe para que o servidor saiba quando ele pode finalizar a conexão com um determinado cliente. O simples código abaixo mostra isso.

```
1 } else if (processed === operations) {
2   client.write(JSON.stringify({ output: 'close' }));
3
4 }
```

Listing 6: Condição de Parada

Por fim, caso nenhum desses casos ocorra, isso quer dizer que algum dos clientes que estão fazendo alguma requisição ao servidor já conseguiu processar todas as suas *strings*. Então, para que o servidor não feche, ele envia para o cliente uma mensagem de espera que só irá ser finalizada quanto todas os clientes processarem todas as suas *strings*. O código abaixo demonstra isso.

```
1 } else {
2   client.write(JSON.stringify({ output: 'wait', client: data.
        client }));
3 }
```

Listing 7: Mensagem de Espera

2.2 Solução UDP

A solução UDP não difere muito da TCP pensando somente nos códgidos em *JavaScript*. Foi necessário importar uma outra biblioteca, sendo ela a *dgram*.

2.2.1 Cliente

Com a mudança de bibliotecas, fez-se necessário utilizar diferentes funções do *JavaScritp*. Um exemplo é a função de envio, no protocolo TCP era o comando *write* e passou a ser *send* no protocolo UDP.

```
} else if (data.output === 'wait') {
    await delay();
  client.send(JSON.stringify({
    input: data.operations,
    client: data.client
  }));
}
```

Listing 8: Cliente Protocolo UDP

Além da mudança de comenados, como pode ser visualizado no código acima, também foi utilizada a funcionalidade *delay* no programa para simular perdas de pacotes e atrasos de envio como foi pedido na especificação do trabalho.

No cliente do protocolo UDP, quando ele finaliza sua conexão, uma mensagem é enviada para o servidor o avisando disso, sendo essa uma das diferenças do código do protocolo UDP para o TCP.

2.2.2 Servidor

No código do servidor, também não há grandes diferenças, algumas delas estão relacionadas aos comandos, como no código do cliente, e também ao *delay*. A maior diferença esta em como é salvo qual cliente é responsável por uma determinada requisição, isso é feito utilizando a estrutura de dados *Map* também, porém, não é salvo qual cliente fez a requisição e sim a porta que foi utilizada para fazer a requisição.

```
if (!clients.get(client.port)) {
    clients.set(client.port, []);

    if (data.operations) operations += Number(data.operations
       );
    await delay();
```

```
6       server.send(JSON.stringify({ output: 'connected', client
            }), client.port, client.address);
7
8   }
```

Listing 9: Servidor Protocolo UDP

O código acima, na linha 2, confirma que o que é salvo no *Map* é a porta de qual cliente esta fazendo a requisição.

3 Cenários de Testes

Para realizar os testes, foram criados três arquivos diferentes para que fossem gerados três clientes fazendo requisições para o servidor. Dois dos quatro arquivos possuíam um baixo número de *strings* a serem processados e um deles possui cerca de cem *strings*. Isso foi feito para mostrar o funcionamento do pedido de espera que o servidor faz para os clientes que já finalizaram todos os seus requisitos. O outro arquivo possui somente caracteres especiais para mostrar que o programa da o resultado necessário para essa condição.

Foi necessário não utilizar o *encoding* do *JavaScript* para fazer a leitura dos arquivos de teste, uma vez que um deles só possui caracteres que não estão descritos na tabela ASCII.

O nome dos arquivos de teste foram: *modelo*, *input1*, *input2* e *input3* Para rodá-los, inicialize o servidor em um terminal e inicialize outros três clientes, cada um com um arquivo diferente.

Para que sejam feitos os testes, basta digitar: node server.js. Para cada cliente digite: node client.js nome do arquivo.

4 Análise dos Resultados

Visualizando os arquivos criados dentro da pasta *out* do diretório do projeto, pode-se concluir que o programa desenvolvido consegue executar o que foi pedido na descrição do trabalho tanto com o protocolo TCP quanto o protocolo UDP.

As duas imagens a seguir mostram os testes feitos com o protocolo TCP e UDP respectivamente.

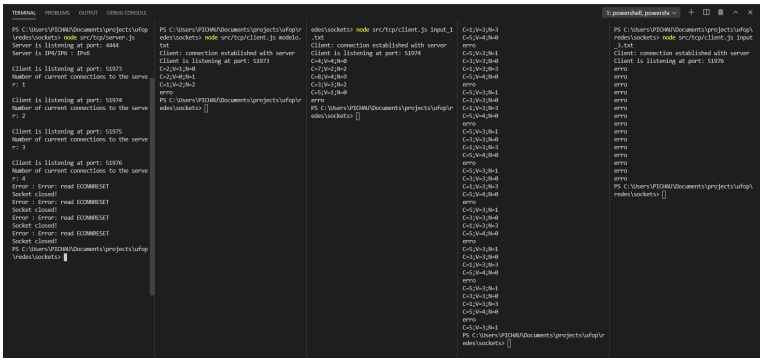

Figura 1: Essa imagem apresenta os testes do protocolo TCP sendo aplicados.

Fonte: Autores do trabalho

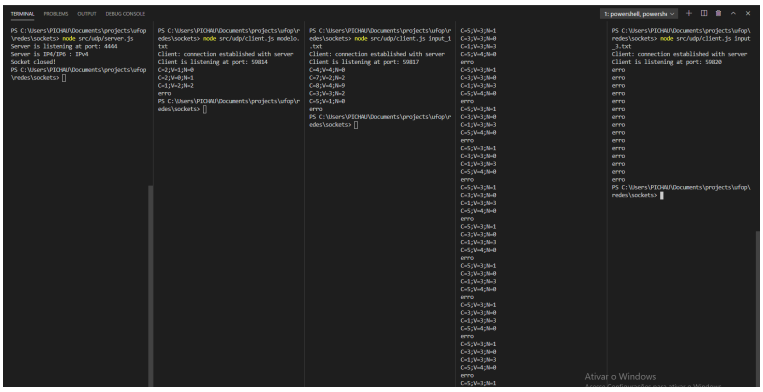

Figura 2: Essa imagem apresenta os testes do protocolo UDP sendo aplicados.

Fonte: Autores do trabalho

Como pode ser observado, o programa consegue fazer a contagem de qualquer arquivo que esteja no padrão exigido na descrição do trabalho, usando tando o protocolo TCP quanto o protocolo UDP.