Laboratório 01

Construindo aplicações distribuídas usando RPC e gRPC

Disciplina: Programação para Sistemas Paralelos e Distribuídos (PSPD) - 2022-2

Carlos Eduardo de Sousa Fiuza – 19/0056843

Denniel William Roriz Lima – 17/0161871

INTRODUÇÃO

O problema que este laboratório busca resolver é o uso de chamadas de procedimento remotas para computação da frequência e quantidade de palavras que um arquivo gerado randomicamente possui, utilizando um modelo cliente-servidor que se utilizam do protocolo RPC e gRPC para tal. Duas soluções foram propostas, sendo a B1 utilizando a linguagem C e a biblioteca rpc para construção de 1 cliente e 1 servidor distribuídos. Já a solução B3 construiu de forma distribuída e paralela 2 workers (um cliente e um servidor) utilizando a biblioteca gRPC em linguagem python.

RPC

O RPC (Remote Call Procedure) define um protocolo para permitir a execução remota de procedimento em outro computador conectado em rede, onde a preocupação deste protocolo é apenas relacionada a especificação e interpretação da mensagem, devendo sua implementação depender do protocolo de transporte escolhido.

Para seu funcionamento é utilizado o modelo cliente-servidor, onde o programa solicitador é o cliente e o programa provedor de serviço é o servidor. Assim como acontece com uma chamada de procedimento local, o RPC é síncrono, onde o solicitante espera (suspenso) o retorno do procedimento.

Para a definição da API é utilizada o IDF (arquivo de definição de interface), provendo uma ponte de ligação entre as máquinas.

Ao compilar um programa que possua o protocolo RPC é gerado um arquivo stub que atua como representante do código de procedimento remoto. Quando um programa é executado e a chamada de procedimento é feita, o stub recebe a solicitação e encaminha para o programa cliente em tempo de execução (client runtime program), este possui conhecimento de como endereçar o computador remoto e enviar a

mensagem. O servidor também possui um stub e um programa servidor em tempo de execução (server runtime program), retornando uma mensagem da mesma forma que o cliente solicita.

gRPC

O gRPC é um framework de alta performance para chamadas de procedimento remotas, criado para ser mais performático que o padrão ReST (por utilizar HTTP/2 e IDL (Protocol Buffers)) e para evoluir o RPC. Protocol Buffers provém uma forma de serialização de dados estruturados de maneira compatível com versões anteriores e posteriores. Um serviço é definido especificando os métodos que podem ser chamados definindo seus parâmetros e tipos de retorno. Do lado do servidor é implementado esse serviço e colocado em execução um servidor gRPC, já do lado do cliente um stub possui os mesmos métodos que o servidor.

DIFERENÇAS

No protocolo RPC o cliente manda uma mensagem e o servidor responde com uma mensagem, sendo que o servidor assim que percebe que o stub do cliente invocou o método RPC pode retornar os metadados iniciais (metadados do cliente) imediatamente ou pode popular uma resposta e enviar para o cliente com o status e metadados. Já no gRPC é possível que lotes de mensagens podem ser enviados tanto pelo servidor quando pelo cliente, e não apenas uma única mensagem por vez.

No gRPC também não é necessário se preocupar com o portmapper (mapeamento de portas) dos programas no servidor.

No gRPC também é possível escolher entre o sincronismo e assincronismo entre a troca de mensagens entre o servidor e cliente, diferente do uso do RPC em que é síncrono.

EXPERIMENTO 1 – RPC entre duas máquinas

Problema: Construir, em c, uma aplicação que leia um arquivo de entrada com palavras de forma a obter a frequência em que cada palavra aparece e quantas palavras estão presentes no arquivo. A aplicação deve ser feita de forma distribuída.

Execução:

O primeiro passo a ser pensado na construção da aplicação foi em seu arquivo de definição, para a resolução do problema o ponto chave foi em qual seria o dado e a forma que o nosso server devolveria. O principal problema na definição foi devido ao fato de inicialmente não sabermos quantas palavras diferentes o arquivo tem, o que complicaria a estrutura da linguagem c que não possui tanta flexibilidade com facilidade como outras linguagens, sendo assim, a tipo a ser pensado foi em uma string que posteriormente seria tratada, dessa forma, o arquivo IDF teria sido escrito da seguinte forma:

```
program PROG {
  version VERSAO {
    string WORDS_FREQ(string) = 1;
    string WORDS_LENGTH(string) = 2;
  } = 100;
} = 12121212;
```

Cada tarefa foi dividida em uma função que o rpcgen irá disponibilizar.

Com isso, a partir desse arquivo foi utilizado o comando "rpcgen –a rpc.x" que gerou os stubs e todos os arquivos necessários para a implementação.

O servidor na estrutura demonstrada anteriormente receberá uma string e irá devolver uma, dessa forma, foi pensado em uma forma de tratar esses dados. Sendo assim, após uma análise prévia, chegamos à conclusão de utilizar uma lista encadeada que iria armazenar uma palavra e a frequência com que ela apareceu no texto. As duas structs que implementam a fila seguem abaixo:

```
typedef struct word_freq {
  char word[100];
  int freq;
  struct word_freq *next;
} word_freq_link;
```

```
typedef struct list_words {
  struct word_freq *words;
  int tam;
} Words_freq;
```

Essas listas possuem os mesmos métodos associados só mudando em onde são chamadas, foi pensado em adicionar um novo elemento na fila toda vez que não for encontrada a palavra comparada nela, e caso encontrar a palavra que está sendo

colocada para comparação o valor de "freq" será atualizado. Para conseguir quebrar o texto a cada caractere de espeço, para checar palavra por palavra, foi utilizado o seguinte código que abusa da função strtok:

```
char *numDiffWords(Words_freq *wf, char *text){
  char *sub_str = strtok(text, " ");

  while(sub_str != NULL){
    exists(wf, sub_str);
    sub_str = strtok(NULL, " ");
    wf->tam++;
  }
}
```

E a cada palavra ele chama a função exists que checa se ele existe na lista encadeada, caso ele não existir, ele adiciona um novo elemento a lista com essa palavra que não foi encontrada. Outro ponto interessante da função é o "wf->tam++" que é incrementado a cada vez que esse loop roda, ou seja, a cada palavra lida.

Após ter sido feito todo o tratamento e terminado de ler todas as palavras do texto o servidor irá gerar um novo texto para enviar para o cliente. Cada função gera um texto diferente.

Seguindo então para o cliente. O primeiro ponto sobre a aplicação do cliente é onde o servidor vai se instalar e sediar aquele serviço a partir da sua máquina local, então a máquina cliente espera, no seu argumento de execução, que o usuário passe o ip da máquina de onde se localiza o servidor, completando assim o nosso objetivo de implementar a aplicação distribuída.

O cliente, então, irá abrir e fazer a leitura completa do arquivo de forma a obter o tamanho total de buffer necessário para processá-lo. Após isso ele fazer um tratamento em cima de dados relevantes filtrando apenas por letras e evitando caracteres que podem dificultar o processamento, como mostra o código abaixo:

```
// percorre todo o arquivo
int index = 0;
for (int i = 0; i <= buffertTam; i++) {
  if ((buffer[i] >= 65 && buffer[i] <= 90) ||
      (buffer[i] >= 97 && buffer[i] <= 122) ||
      (buffer[i] == 32 && i < buffertTam)) {
    text[index++] = buffer[i];
  }
}
```

Após isso ele chama as duas funções que irá solicitar a execução das funções lá no servidor e imprimir sua resposta. Um exemplo de como ficou uma das funções de chamada é:

```
char *word_tam(CLIENT *clnt, char *request){
  char **response;

  response = words_length_100(&request, clnt);
  if (response == NULL){
    fprintf(stderr, "Problema na chamada RPC\n");
    exit(0);
  }

  return (*response);
}
```

EXPERIMENTO 3 – RPC entre três máquinas com gRPC (um cliente e dois servidores)

Problema: Construir, em c ou python, uma aplicação que leia um arquivo de entrada com palavras de forma a obter a frequência em que cada palavra aparece e quantas palavras estão presentes no arquivo. A aplicação deve ser feita de forma distribuída e paralela.

Execução:

O primeiro passo a ser feito foi a implementação do .proto que é o nosso arquivo IDF. A linguagem escolhida para esse experimento foi python devido a flexibilidade e facilidade da linguagem. O python possui o dado chamado dicionário que permite associar um valor a uma key, sendo assim o .proto foi implementado de forma a utilizar dessa estrutura para envio de dados de uma máquina para outra, diferentemente do experimento anterior. O .proto ficou da seguinte forma:

```
syntax = "proto3";

// The greeting service definition.
service WordCount {
  // Sends a greeting
  rpc SendText (WordCountRequest) returns (WordCountResponse) {}
}

// The request message containing the user's name.
message WordCountRequest {
  string text = 1;
}

// The response message containing the greetings
message WordCountResponse {
  int32 number_words = 1;       You, 3 days ago • Finish gRPC …
  map<string, int32> words_freq = 2;
}
```

Será mantida a forma de envio de string e dessa vez ele receberá dois retornos em uma chamada, um retorno irá entregar o número total de palavras e a segunda chamada irá entregar um dicionário em que as keys serão as palavras e o valor será a frequência em que elas apareceram.

Nos workers, ele divide o texto através da função split, que gera um array das palavras contidas no texto, inicializa um dicionário vazio, percorre palavra por palavra checando se ele já está presente no dicionário, caso não esteja ele vai adicionar no dicionário já checando quantas vezes ele aparece no array com a função count do python. O código abaixo:

```python
class WordCount(wordcount_pb2_grpc.WordCountServicer):
    def SendText(self, request, context):
        words = request.text.split()
        words_freq = {}

        for word in words:
            if word not in words_freq.keys():
                words_freq[word] = words.count(word)      You, 3 days ago • Finish gRPC …

        number_words = len(request.text.split())

        return wordcount_pb2.WordCountResponse(number_words=number_words, words_freq=words_freq)
```

No client, ele subdivide o arquivo em dois e faz a leitura de cada arquivo armazenando em uma variável diferente, e com o objetivo de implementar o paralelismo dos workers foi utilizado de threads em que cada thread fica responsável por mandar e esperar a resposta do servidor, após isso o cliente pega os dados, mescla e imprime a contagem total e a frequência total do arquivo.

```python
print("Dividing example.txt file...")
divide_file()

text1 = read_file(worker=1)
text2 = read_file(worker=2)

print("Will try to count words ...")

threads = []
responses = []

thread = Thread(target=request_to_worker, args=(text2, 'grpc_worker2', 8089, responses))
threads.append(thread)
thread = Thread(target = request_to_worker, args=(text1, 'grpc_worker1', 8089, responses))
threads.append(thread)
```

```python
for thread in threads:
    thread.start()

for thread in threads:
    thread.join()

number_words = 0
words_freq = {}

for response in responses:
    number_words = number_words + response.number_words
    words_freq.update(response.words_freq)

print(f"WordCount received: {number_words}")
for word in words_freq.keys():
    print(f"{word}: {words_freq[word]}")
```

```python
def request_to_worker(text, worker_ip, worker_port, responses):
    with grpc.insecure_channel(f'{worker_ip}:{worker_port}') as channel:
        stub = wordcount_pb2_grpc.WordCountStub(channel)
        response = stub.SendText(wordcount_pb2.WordCountRequest(text=text))
        responses.append(response)
```

OPINIÃO GERAL

Denniel: O projeto ensinou a implementar desde uma linguagem c até um mais alto nível como python, assim como a aplicação de gRPC deu para entender todo um mercado por trás devido a quantidade de material que se atualiza diariamente. Contudo devido ao semestre reduzido, o projeto em si ficou extenso ainda mais por concorrência com outras disciplinas, tanto que a ideia de projeto sobre o experimento 2 estava e como foi iniciada mas não pode ser concluída exatamente por essa dificuldade que foi a extensão e concorrência com outras disciplinas também. Participei de ambas implementações e em uma nota de 0 a 10, me avaliaria em 10 devido ao entendimento e esforço mesmo não tendo conseguido atingir o experimento 2.

Carlos: O laboratório foi muito bem vindo de forma a aplicar os conhecimentos adquiridos em sala de aula, mas devido ao atual semestre reduzido e ao paralelismo com outras disciplinas não foi possível elaborar todos os itens pedidos na especificação. Entre aprendizados novos posso citar o fato de utilizar uma forma mais

“alto nível” de comunicação entre processos, possibilitando focar mais no problema, diferente do que acontecia com o uso de sockets. Durante o desenvolvimento fiquei “travado” e senti dificuldades para usar workers em máquinas diferentes, o que acabou me tirando muito tempo. Minha participação foi principalmente focada na solução B1, mas também contribui com a solução B3. De forma a dar nota para minha participação uma nota 8 condiz com o meu aprendizado