Relatório
Algoritmo de Eleição de Líder
Chang Roberts

Universidade Federal de Santa Catarina

19 de Setembro de 2021

Samuel Cardoso
19100544

Nicolas Vanz
19100541

Como executar o programa

Para a execução correta do programa é necessário que o sistema operacional do computador utilizado seja Linux, já que usamos o módulo *fcntl* do Python que é funcional apenas neste SO. Dentro da pasta do projeto execute um dos seguintes comandos no Terminal:

```
python main.py <process_number>

python main.py <process_number> <break_process>
```

Onde <process_number> é um inteiro de 3 à 9 que representa o número de processos que estarão sendo executados no ambiente. Para a simulação de erro em um processo durante a execução, pode ser especificado <break_process>, que é um inteiro de 0 à 8 que representa qual processo terá chance (20%) de parar de responder. Mais para frente iremos explicar por qual motivo limitamos em 9 processos.

Estrutura do projeto

O projeto do algoritmo foi desenvolvido na linguagem Python e em sistema operacional Linux. Cada nó do anel é retratado como um processo Python. Para isso foi utilizado o módulo *multiprocessing* do Python. Cada nó gerencia 2 *sockets*, um para recebimento de dados vindos do nó anterior no anel e outro *socket* para o envio de dados ao nó posterior no anel. Essa arquitetura de *sockets* garante um sistema de comunicação unidirecional e circular, que é característico do algoritmo *Chang e Roberts*. Além disso, os *sockets* operam de maneira não bloqueante para garantir que a falha de um processo não bloqueie a execução de todos os outros processos.

A implementação tem como limitação a execução de no máximo 9 processos, pois para a comunicação precisamos informar quantos *bytes* o *socket* alvo deve receber. Atualmente os *bytes* estão fixos em 5, sendo que apenas um é destinado ao índice do processo, o que restringe o índice a apenas um caracter. Dessa forma o maior valor possível de ser representado é 9. Não seria difícil aprimorar isto no código, mas como não é o foco do trabalho não achamos necessário realizar tal alteração.

Partes importantes do código

As mensagens trocadas entre os nós são de 4 tipos: (i) *Election*: Indica que uma eleição está acontecendo. (ii) *Winner*: Indica que um líder foi eleito. (iii) *Alive*: Indica que o nó anterior está operando normalmente. (iv) *Restore*: Indica que um nó parou de respopnder.

A função responsável pelo recebimento de dados é a parte principal do projeto. Através dela, os processos recebem os dados através de *sockets* não bloqueantes. São realizadas *ntrials* tentativas de leitura. Duas situações podem acontecer:

1. Há dados no *socket* e 5 *bytes* são lidos. Isso porque as mensagens trocadas sempre possuem 5 caracteres, os quais identificam o tipo da mensagem, o campo *greatest_process* e o campo *sender*. A leitura de apenas 5 *bytes* garante que apenas uma mensagem é lida do *socket* por vez.

2. Não há dados no socket. Nesse caso, o processo tenta ler algumas vezes o *socket* até ler algum dado (com intervalo de um segundo a cada leitura). Caso nenhum dado seja lido em $2 * numero\_de\_processos$ iterações, o processo avisa ao nó posterior do anel que houve um atraso de comunicação, impedindo que todos os processos julguem que seus nós anteriores estão com mal funcionamento. Caso $4 * numero\_de\_processos$ tentativas de leitura sejam realizadas sem sucesso, o processo julga que o nó anterior está inoperante e envia uma mensagem de restauração do anel, excluindo o nó anterior.

A Classe Message, além de guardar os dados da mensagem como citado acima, também é responsável por codificar e decodificar a mensagem de forma a facilitar o uso dos dados dentro do código e torna possível o envio das informações através do *socket*. A função *encode()* cria uma *string* do tipo "<code>#<greatest_process>#<sender>". Já a função *decode()* pega uma mensagem criada pelo encode e insere os dados dentro de um Objeto Message.

É válido notar que ambas as funções chamam funções de mesmo nome *encode()* e *decode()*, estas nativas do Python que convertem uma *string* para uma *byte string* e uma *byte string* para uma *string* respectivamente.

```
    def encode(self):
        return f'{self.code}#{self.greatest_process}#{self.sender}'.encode()

    def decode(self, message):
        message = message.decode()
        if (message == ''):
            return
        code, greatest_process, sender = message.split("#")
        self.code = code
        self.greatest_process = int(greatest_process)
        self.sender = int(sender)
```

Dificuldades da implementação

A primeira dificuldade encontrada foi garantir a conexão entre cada par de *sockets*. Inicialmente, um *socket* poderia tentar se conectar com outro *socket* antes mesmo deste abrir um *socket* de entrada de dados. Isso prejudicava a criação do anel. Esse problema foi resolvido com a utilização de uma estrutura *Barrier* compartilhada entre os processos. A estrutura garante sincronização na criação do anel e apenas permite a conexão entre *sockets* quando todos os nós já criaram seus *sockets* de entrada de dados. Além disso, tivemos que intercalar a conexão dos *sockets*, isto é, dividir os processos em 2 grupos: (i) os que conetam com outro processo para envio de dados e depois aceitam sua conexão para entrada de dados. (ii) os que aceitam a conexão para entrada de dados e depois se conectam com outro processo para envio de dados.

Em um segundo momento, as mensagens por vezes paravam de ser enviadas. Após um bom tempo, percebeu-se que o problema estava no fato de que um Nodo poderia enviar uma mensagem do tipo *alive* e logo em seguida receber uma mensagem. Isto poderia induzir o Nodo alvo a acumular mais de uma mensagem dentro do *socket*, o que causaria um *decode* errado da mensagem, já que várias mensagens seriam lidas como se fossem apenas uma. Para resolver isto, decidiu-se fixar o tamanho das mensagens recebidas e enviadas em 5 *bytes*.

Realização do trabalho

Para este trabalho, ambos os integrantes estiveram integralmente durante a implementação, por tanto ambos sabem tudo sobre tal. Foi utilizada a ferramenta (extensão do VSCode) LiveShare que permite com que várias pessoas editem um mesmo código e até mesmo executem ele através um de um Shared Terminal. O Relatório foi escrito através do Overleaf e a comunicação realizada através do Discord.

Conclusões

Para a conclusão do trabalho foi necessário o entendimento do algoritmo, bem como a utilização de sistemas de comunização assícrona e sincronização entre processos. Dessa forma, pôde-se implementar um ambiente de eleição de líder com tolerência a falhas, o que promoveu aos estudantes maior compreensão a respeito dos conteúdos abordados durante a disciplina.