

Relatório da Atividade 2:

Exclusão Mútua

Isabelle Ferreira de Oliveira
CES-27 - Engenharia da Computação 2020
Instituto Tecnológico de Aeronáutica (ITA)
São José dos Campos, Brasil
isabelle.ferreira3000@gmail.com

Resumo—Esse relatório documenta a implementação do algoritmo de Ricart-Agrawala, um algoritmo de exclusão mútua para sistemas distribuídos.

Index Terms—Algoritmo de Ricart-Agrawala, exclusão mútua, Relógio lógico vetorial, sistemas distribuídos

I. IMPLEMENTAÇÃO

A. Recurso Compartilhado

A implementação do recurso compartilhado se tratou principalmente da utilização do código disponibilizado no roteiro do laboratório. Dentro do loop principal, foi colocada a função de servidor, cuja implementação era simplesmente imprimir na tela qualquer mensagem recebida pela porta :10001.

Abaixo encontra-se os códigos da função main() e da função doServerJob().

```
func main() {
    Address, err := net.ResolveUDPAddr("udp",
        ":10001")
    Connection, err = net.ListenUDP("udp",
        Address)
    defer Connection.Close()

    for {
        doServerJob()
    }
}

func doServerJob() {
    buf := make([]byte, 1024)

    n, _, err := Connection.ReadFromUDP(buf)
    err = json.Unmarshal(buf[:n],
        &messageReceived)

    fmt.Println(messageReceived.Text)
}
```

A mensagem recebida era da forma de struct, com os atributos de id do processo que enviou a mensagem, o relógio lógico do processo que enviou a mensagem e, por fim, o texto a ser impresso no terminal.

Vale ressaltar também que, para esse código e todos os outros dessa atividade, sempre após a setagem da variável *err*, referente a um possível erro advindo de algumas funções,

também era chamada a função CheckError(err), que imprime o erro e interrompe o processo caso houvesse algum erro.

B. Processo

A grosso modo, o processo foi implementado de forma bastante semelhante ao feito no Laboratório 1, como pode ser observado na main() abaixo.

Sobre as diferenças, a existência de estados ("RELEASED", "WANTED" e "HELD"), que determinam a validade de uma entrada pelo terminal do processo, além participam na priorização de permanência na CS (assim como o timestamp das requests de entrada na CS, e os IDs dos processos). Além disso, mudanças nos comportamentos do servidor, além de mudanças nas mensagens trocadas entre os processos.

Conforme se pode ver na main() apresentada abaixo, ao receber uma mensagem pelo terminal, o processo verifica sua validade e, caso seja válida, passa para o estado "WANTED", enviando requests de entrada na CS para os demais processos. A partir daí, o processo em questão espera pelas replies dos demais processos, entrando na CS quando receber todas. Após sair da CS, o processo responde as requests dos demais processos que ele tiver adicionado a fila.

```
func main() {
    initConnections()
    setState("RELEASED")

    defer ServerConn.Close()
    for i := 0; i < nPorts; i++ {
        defer ClientsConn[i].Close()
    }

    go readInput(ch)

    for {
        go doServerJob()

        select {
        case textReceived, valid := <-ch:
            if valid {
                if myState == "WANTED" || myState ==
                    "HELD" {
                    fmt.Println(textReceived,
                        "invalido")
                } else {
                    if textReceived != myIDString {
                        messageSent.Text = textReceived
                    }
                }
            }
        }
    }
}
```

```

        setState("WANTED")
        myTimestamp = logicalClock
        request.Timestamp = myTimestamp

        for otherID := 1; otherID <=
            nPorts; otherID++ {
            if otherID != myID {
                go doClientJob(request,
                    otherID)
            }
        }
        go waitReplies()
    } else {
        // updating my clock
        logicalClock++
    }
}
} else {
    fmt.Println("Channel closed!")
}
default:
    time.Sleep(time.Second * 1)
}
}
}

```

Como servidor, ao receber uma mensagem de outros processos, o processo atualiza seu relógio lógico e verifica o tipo dessa mensagem, se é "request" ou "reply". Essa mensagem trata-se de uma struct, contendo os atributos de tipo ("request" ou "reply"), além de o timestamp (para o caso de request), o ID do processo que enviou e seu relógio lógico.

Caso receba um reply de que pode entrar na CS, um contador de replies é acrescido. Já caso a mensagem seja um request, é necessário verificar as condições de prioridade para responder esse request, ou adicioná-lo a uma fila com os demais requests a se responder. Enviar uma reply também incrementa o relógio lógico do processo.

Segue a seguir o código da função doServerJob().

```

func doServerJob() {
    buf := make([]byte, 1024)

    n, _, err := ServerConn.ReadFromUDP(buf)
    var message RequestReplyStruct
    err = json.Unmarshal(buf[:n], &message)

    msgType := message.Type
    msgLogicalClock := message.LogicalClock
    msgTimestamp := message.Timestamp

    // updating clocks
    logicalClock = max(msgLogicalClock,
        logicalClock) + 1

    if msgType == "request" {
        msgId := message.Id

        if myState == "HELD" ||
            ( myState == "WANTED" && (
                msgTimestamp < myTimestamp ||
                ( msgTimestamp == myTimestamp &&
                    msgId < myID ))) {

```

```

            requestsQueue = append(requestsQueue,
                msgId)
        } else {
            // updating clocks
            logicalClock++
            reply.LogicalClock = logicalClock

            jsonReply, err := json.Marshal(reply)
            _, err =
                CliConn[msgId-1].Write(jsonReply)
        }
    } else if msgType == "reply" {
        nReplies++
    }
}

```

Por fim, esperar as replies dos demais processos se trata de uma thread presa em loop com a condição de o contador de replies chegar ao total esperado. Após isso, o processo passa para o estado "HELD", adentra a CS e a usa, posteriormente saindo da CS e passando para o estado "RELEASED". Como explicado anteriormente, dentro da CS, o processo envia uma mensagem ao SharedResource com o texto digitado no terminal do processo e dorme por alguns segundos; após sair da CS, o processo responde aos requests restantes que estiverem enfileirados.

Os códigos de waitReplies() e useCS() estão apresentados a seguir.

```

func waitReplies() {
    for nReplies != nPorts-1 {}
    nReplies = 0

    setState("HELD")
    useCS()
    setState("RELEASED")

    // reply requests
    for _, element := range requestsQueue {
        logicalClock++
        reply.LogicalClock = logicalClock
        jsonReply, err := json.Marshal(reply)
        _, err =
            ClientsConn[element-1].Write(jsonReply)
    }
    requestsQueue = make([]int, 0)
}

```

```

func useCS() {
    fmt.Println("Entrei na CS")
    messageSent.LogicalClock = logicalClock

    jsonMessage, err :=
        json.Marshal(messageSent)
    _, err =
        SharedResourceConn.Write(jsonMessage)

    time.Sleep(time.Second * 10)
    fmt.Println("Sai da CS")
}

```

Após isso, é iniciada uma thread para ler as entradas do usuário a partir da função `readInput()`.

Em seguida, é iniciado o loop de fazer o trabalho de servidor (ou seja, atualiza-se o relógio lógico caso chegue uma mensagem de outro processo, por meio da thread `doServerJob()`) e fica-se esperando uma mensagem do usuário no canal criado *ch*.

Ao se receber esse input do usuário, atualiza-se o relógio lógico e, a partir daí, duas ações podem ser tomadas a depender do conteúdo da entrada:

- faz-se o trabalho de cliente (enviando uma mensagem a um outro processo por meio da thread `doClientJob()`) caso a entrada seja o ID de outro processo;
- ou apenas imprime-se o valor atual do relógio lógico caso a entrada seja o próprio ID desse processo.

Abaixo, segue-se o código dessa função `initConnections()`.

```
func initConnections() {
    nPorts = len(os.Args) - 2

    // my process
    logicalClock = 0
    auxMyID, err := strconv.Atoi(os.Args[1])
    myID = auxMyID
    myPort = os.Args[myID+1]

    // Server
    ServerAddr, err :=
        net.ResolveUDPAddr("udp", myPort)
    aux, err := net.ListenUDP("udp", ServerAddr)
    ServConn = aux

    // Clients
    for i := 0; i < nPorts; i++ {
        aPort := os.Args[i+2]

        ServerAddr, err :=
            net.ResolveUDPAddr("udp", "127.0.0.1"
                + aPort)
        LocalAddr, err :=
            net.ResolveUDPAddr("udp",
                "127.0.0.1:0")
        auxConn, err := net.DialUDP("udp",
            LocalAddr, ServerAddr)
        AllConn = append(AllConn, auxConn)
    }
}
```

Vale ressaltar também que, para esse código e todos os outros dessa atividade, sempre após a setagem da variável *err*, referente a um possível erro advindo de algumas funções, também era chamada a função `CheckError(err)`, que imprime o erro e interrompe o processo caso houvesse algum erro.

Essas chamadas de funções foram suprimidas do relatório a fim de simplificar a apresentação dos códigos, e por entender-se que não se trata da ideia principal dos códigos desenvolvidos.

A função `readInput()` segue bem semelhante àquela apresentada na Dica 3 do roteiro, com a diferença de aceitar um canal de inteiro ao invés de um canal de string. Assim, a função é capaz de ler o ID que o usuário digitar.

Já a função `doServerJob()`, apresentada abaixo, também segue bem semelhante à apresentada na função `main()` do código do servidor fornecido na Dica 1. A diferença está na retirada do loop *for* e do fechamento da conexão, uma vez que essas etapas se equivalem aos apresentados na `main()` da atividade (função já apresentada acima). Outra diferença também é a impressão do relógio lógico recebido por mensagem e, em seguida, a impressão do valor atualizado. Segue abaixo o código descrito.

```
func doServerJob() {
    buf := make([]byte, 1024)

    n, _, err := ServConn.ReadFromUDP(buf)

    aux := string(buf[0:n])
    otherLogicalClock, err := strconv.Atoi(aux)
    fmt.Println("Received", otherLogicalClock)

    // updating logical clock
    logicalClock = max(otherLogicalClock,
        logicalClock) + 1
    fmt.Println("logicalClock atualizado:",
        logicalClock)
}
```

Por fim, a função `doClientJob()` também seguiu de forma semelhante ao código apresentado na função `main()` do código do cliente fornecido na Dica 1. As alterações também foram semelhantes: retirouse o loop *for* e o fechamento da conexão. Além disso, o conteúdo da mensagem a ser enviada foi alterado para o relógio lógico atual do processo em questão. Segue abaixo o código descrito.

```
func doClientJob(otherProcessID int,
    logicalClock int) {
    otherProcess := otherProcessID - 1

    msg := strconv.Itoa(logicalClock)
    buf := []byte(msg)

    _, err := AllConn[otherProcess].Write(buf)

    time.Sleep(time.Second * 1)
}
```

A segunda etapa se tratou da implementação da simulação para o Relógio Lógico Vetorial, segundo o algoritmo descrito nos slides da aula e conforme o solicitado no roteiro da atividade. Essa implementação foi feita de forma bastante análoga à da Tarefa 1.

As principais alterações na implementação foram:

- mudança do `logicalClock` de inteiro para uma *struct* com o ID do processo atual e um vetor com os clocks de todos os processos;
- alteração das mensagens recebidas e enviadas (de inteiros para *jsons* contendo as *structs*);
- alteração na lógica de atualização do vetor de clocks.

O impacto dessas mudanças nas funções no código desenvolvido na Tarefa 2 em relação a Tarefa 1 foram:

- em `initConnections()`, a iniciação do `logicalClock` se trata de setar o atributo `myId` e criar o vetor de clocks da struct com valores iniciais zero;
- as mensagens (structs) trocadas entre processos foram encapsuladas na forma de json a partir da função `json.Marshal()` e desencapsuladas com a função `json.Unmarshal()`;
- ao se receber a struct `logicalClock` de outro processo, (além de incrementar o clock referente ao processo atual) os valores de clocks eram atualizados para o máximo entre o clock armazenado e o clock recém recebido.

Como as principais alterações foram pequenas e já foram descritas acima, não se considerou necessário colocar nesse relatório as partes referentes do código da Tarefa 2. Caso seja necessário, pode-se também consultar o código enviado como anexo a essa atividade.

II. RESULTADOS E CONCLUSÕES

A. Tarefa 1

A implementação mostrou-se correta, uma vez que os resultados dos testes e das simulações se mostraram condizentes com o esperado.

O primeiro teste executado foi apresentado na Figura ??, e se tratava do exemplo apresentado nos slides da aula. Eram três processos (P_1 , P_2 e P_3), e os acontecimentos se deram da seguinte maneira:

- P_1 realizou seu primeiro evento, incrementando seu relógio lógico para 1;
- P_3 realizou seu primeiro evento, incrementando seu relógio lógico para 1;
- P_1 mandou uma mensagem para P_2 , incrementando seus relógios lógicos para 2 e 3, respectivamente;
- P_2 mandou uma mensagem para P_3 , incrementando seus relógios lógicos para 4 e 5, respectivamente.

Como esses resultados, conforme na Figura ??, foram exatamente os esperados de acordo com os slides da aula, tem-se que esse teste foi satisfatório.

Outro teste executado foi o apresentado nas Figuras ?? e ??. Essas duas figuras apresentam o mesmo teste, dessa vez para quatro processos, sendo a Figura ?? referente aos processos 1 e 3, e a Figura ?? aos processos 2 e 4. Os eventos aconteceram da seguinte maneira:

- P_1 realizou seus três primeiros eventos, incrementando seu relógio lógico para 3;
- P_2 realizou seu primeiro evento, incrementando seu relógio lógico para 1;
- P_4 mandou uma mensagem para P_1 , incrementando seus relógios lógicos para 1 e 4, respectivamente;
- P_4 realizou cinco eventos, incrementando seu relógio lógico para 6;
- P_4 mandou uma mensagem para P_1 , incrementando seus relógios lógicos para 7 e 8, respectivamente;
- P_2 mandou uma mensagem para P_3 , incrementando seus relógios lógicos para 2 e 3, respectivamente;

- P_2 mandou uma mensagem para P_1 , incrementando seus relógios lógicos para 3 e 9, respectivamente;
- P_2 mandou uma mensagem para P_4 , incrementando seus relógios lógicos para 4 e 8, respectivamente;
- P_2 realizou um evento, incrementando seu relógio lógico para 5;
- P_2 mandou uma mensagem para P_3 , incrementando seus relógios lógicos para 6 e 7, respectivamente;
- P_3 realizou três eventos, incrementando seu relógio lógico para 10;
- P_3 mandou uma mensagem para P_1 , incrementando seus relógios lógicos para 11 e 12, respectivamente.

Como esses resultados também foram condizentes com os resultados esperados, conclui-se que a implementação da Tarefa 1 foi feita corretamente.

B. Tarefa 2

A implementação também mostrou-se correta, uma vez que os resultados dos testes e das simulações se mostraram condizentes com o esperado. Os testes foram exatamente os mesmos da Tarefa 1, com a mudança apenas do tipo de relógio lógico estava sendo enviado como mensagem.

O primeiro teste executado foi apresentado nas Figuras ?? e ??, e se tratava do exemplo apresentado nos slides da aula. Eram três processos (P_1 , P_2 e P_3), e os acontecimentos se deram da seguinte maneira:

- P_1 realizou seu primeiro evento, incrementando seu relógio lógico para (1, 0, 0);
- P_3 realizou seu primeiro evento, incrementando seu relógio lógico para (0, 0, 1);
- P_1 mandou uma mensagem para P_2 , incrementando seus relógios lógicos para (2, 0, 0) e (2, 1, 0), respectivamente;
- P_2 mandou uma mensagem para P_3 , incrementando seus relógios lógicos para (2, 2, 0) e (2, 2, 2), respectivamente.

Como esses resultados, conforme nas Figuras ?? e ??, foram exatamente os esperados de acordo com os slides da aula, tem-se que esse teste foi satisfatório.

Outro teste executado foi o apresentado nas Figuras ?? e ??. Essas duas figuras apresentam o mesmo teste, dessa vez para quatro processos, sendo a Figura ?? referente aos processos 1 e 2, e a Figura ?? aos processos 3 e 4. Os eventos aconteceram da seguinte maneira:

- P_1 realizou seus três primeiros eventos, incrementando seu relógio lógico para (3, 0, 0, 0);
- P_2 realizou seu primeiro evento, incrementando seu relógio lógico para (0, 1, 0, 0);
- P_4 mandou uma mensagem para P_1 , incrementando seus relógios lógicos para (4, 0, 0, 1) e (0, 0, 0, 1), respectivamente;
- P_4 realizou cinco eventos, incrementando seu relógio lógico para (0, 0, 0, 6);
- P_4 mandou uma mensagem para P_1 , incrementando seus relógios lógicos para (5, 0, 0, 7) e (0, 0, 0, 7), respectivamente;

- P_2 mandou uma mensagem para P_3 , incrementando seus relógios lógicos para (0, 2, 0, 0) e (0, 2, 1, 0), respectivamente;
- P_2 mandou uma mensagem para P_1 , incrementando seus relógios lógicos para (0, 3, 0, 0) e (6, 3, 0, 7), respectivamente;
- P_2 mandou uma mensagem para P_4 , incrementando seus relógios lógicos para (0, 4, 0, 0) e (0, 4, 0, 8), respectivamente;
- P_2 realizou um evento, incrementando seu relógio lógico para (0, 5, 0, 0);
- P_2 mandou uma mensagem para P_3 , incrementando seus relógios lógicos para (0, 6, 0, 0) e (0, 6, 2, 0), respectivamente;
- P_3 realizou três eventos, incrementando seu relógio lógico para (0, 6, 5, 0);
- P_3 mandou uma mensagem para P_1 , incrementando seus relógios lógicos para (0, 6, 6, 0) e (7, 6, 6, 7), respectivamente.

Como esses resultados também foram condizentes com os resultados esperados, conclui-se que a implementação da Tarefa 2 foi feita corretamente.