

Relatório da Atividade 3:

Map Reduce

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 um trabalho com o modelo de programação MapReduce de forma sequencial e distribuída.

Index Terms—Map Reduce, Golang, sistema sequencial, sistema distribuído

I. IMPLEMENTAÇÃO

A. Parte 1: Trabalhando em modo sequencial

1) Tarefa 1.1: blabla

```
func mapFunc(input []byte) (result
[]mapreduce.KeyValue) {

    // [...]

    var item mapreduce.KeyValue

    for _, word := range words {
        word = strings.ToLower(word)
        item.Key = word
        item.Value = "1"
        result = append(result, item)
    }

    return result
}
```

2) Tarefa 1.2: blabla

```
func reduceFunc(input []mapreduce.KeyValue)
(result []mapreduce.KeyValue) {

    var mapAux map[string]int =
        make(map[string]int)
    var value int

    for _, item := range input {
        value, _ = strconv.Atoi(item.Value)
        _, ok := mapAux[item.Key]
        if ok {
            mapAux[item.Key] += value
        } else {
            mapAux[item.Key] = value
        }
    }

    var itemAux mapreduce.KeyValue

    for key, value := range mapAux {
        itemAux.Key = key
        itemAux.Value = strconv.Itoa(value)
```

```
        result = append(result, itemAux)
    }

    return result
}
```

3) Tarefa 1.3:

4) Tarefa 1.4:

B. Parte 2: Trabalhando em modo distribuído

1) Tarefa 2.1: Blabla

```
func (master *Master) handleFailingWorkers() {

    for elem := range master.failedWorkerChan {
        master.workersMutex.Lock()
        delete(master.workers, elem.id)
        master.totalWorkers--
        master.workersMutex.Unlock()

        fmt.Println("Removing worker", elem.id,
            "from master list")
    }
}
```

2) Tarefa 2.2: Blabla

```
type Master struct {
    // [...]

    // Fault Tolerance
    failedOperationChan chan *Operation
}

func (master *Master)
runOperation(remoteWorker *RemoteWorker,
operation *Operation, wg *sync.WaitGroup)
{

    // [...]

    if err != nil {
        log.Printf("Operation %v '%v' Failed.
            Error: %v\n", operation.proc,
            operation.id, err)

        master.failedWorkerChan <- remoteWorker
        master.failedOperationChan <- operation
    } else {
```

```

    wg.Done()
    master.idleWorkerChan <- remoteWorker
}
}

```

```

func (master *Master)
    handleFailingOperations(wg
        *sync.WaitGroup) {

    var operation *Operation
    var worker *RemoteWorker
    var ok bool

    for {
        operation, ok =
            <-master.failedOperationChan
        if !ok { break }

        worker, _ = <-master.idleWorkerChan

        go master.runOperation(worker, operation,
            wg)
    }
}

```

```

func (master *Master) schedule(task *Task,
    proc string, filePathChan chan string)
    int {

    var (
        wg      sync.WaitGroup
        filePath string
        worker *RemoteWorker
        operation *Operation
        counter int
    )

    master.failedOperationChan = make(chan
        *Operation, RETRY_OPERATION_BUFFER)

    go master.handleFailingOperations(&wg)

    // [...]
}

```

3) Tarefa 2.3:

4) Tarefa 2.4:

II. RESULTADOS E CONCLUSÕES

A. Teste 1

Este foi o caso com um processo solicitando a CS e, depois que ele liberasse, outro processo solicitando a CS, sugerido no roteiro do laboratório. O esquema do resultado esperado foi apresentado na Figura 1.

Na Figura 1, para P1, as setas azuis representam requests e as verdes, replies; para P3, essas setas são rosas e cinzas, respectivamente. Nos requests, a mensagem enviada é da forma "relógio lógico, < timestamp, id >"; já no reply, a forma é "relógio lógico, 'reply'". Além disso, a linha amarela representa o processo no estado WANTED; e a linha vermelha, no estado HELD, ou seja, na CS.

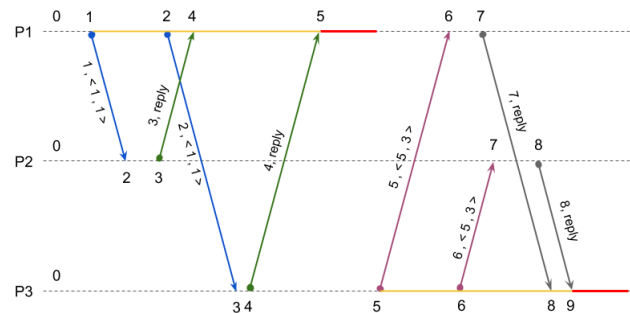


Figura 1. Funcionamento esperado para a tarefa com 3 processos.

Foi simulada essa situação acima com o código implementado no laboratório, tendo os resultados apresentados nas Figuras de 2 a 5. Na simulação, ao invés de o P3 fazer a requisição, é o P2 a faz, mas isso não prejudica o teste.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-
$ go run Process.go 1 :10006 :10007 :10008
oi
Entrei na CS
Sai da CS

```

Figura 2. Exemplo do funcionamento da tarefa com 3 processos. Tela do processo 1.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-
$ go run Process.go 2 :10006 :10007 :10008
xau
Entrei na CS
Sai da CS

```

Figura 3. Exemplo do funcionamento da tarefa com 3 processos. Tela do processo 2.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-julia
$ go run Process.go 3 :10006 :10007 :10008

```

Figura 4. Exemplo do funcionamento da tarefa com 3 processos. Tela do processo 3.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-
└─$ go run SharedResource.go
oi
xau

```

Figura 5. Exemplo do funcionamento da tarefa com 3 processos. Tela do SharedResource.

A fim de entender melhor cada estágio do funcionamento, foram realizados os mesmos testes novamente, agora com prints de debug. Dessa forma, esses resultados estão apresentados nas Figuras de 6 a 9.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-
└─$ go run Process.go 1 :10006 :10007 :10008
Estado: RELEASED
oi
Estado: WANTED
Multicast request to all processes
logicalClock atualizado: 1
Request enviado: 1 , < 0 , 1 >
Esperando N-1 respostas
logicalClock atualizado: 2
Request enviado: 2 , < 0 , 1 >
Reply recebido: 3 reply
logicalClock atualizado: 4
Reply recebido: 4 reply
logicalClock atualizado: 5
Estado: HELD
Entrei na CS
Request recebido: 6 , < 4 , 3 >
logicalClock atualizado: 7
Sai da CS
Estado: RELEASED
Reply enviado: 8 reply

```

Figura 6. Exemplo do funcionamento da tarefa com 3 processos. Tela do processo 1.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-
└─$ go run Process.go 2 :10006 :10007 :10008
Estado: RELEASED
Request recebido: 1 , < 0 , 1 >
logicalClock atualizado: 2
Reply enviado: 3 reply
logicalClock atualizado: 3
Request recebido: 6 , < 4 , 3 >
logicalClock atualizado: 7
Reply enviado: 8 reply
logicalClock atualizado: 8

```

Figura 7. Exemplo do funcionamento da tarefa com 3 processos. Tela do processo 2.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-
└─$ go run Process.go 3 :10006 :10007 :10008
Estado: RELEASED
xauRequest recebido: 2 , < 0 , 1 >
logicalClock atualizado: 3
Reply enviado: 4 reply
logicalClock atualizado: 4

Estado: WANTED
Multicast request to all processes
Esperando N-1 respostas
logicalClock atualizado: 6
Request enviado: 6 , < 4 , 3 >
logicalClock atualizado: 5
Request enviado: 6 , < 4 , 3 >
Reply recebido: 8 reply
logicalClock atualizado: 9
Reply recebido: 8 reply
logicalClock atualizado: 10
Estado: HELD
Entrei na CS
Sai da CS
Estado: RELEASED

```

Figura 8. Exemplo do funcionamento da tarefa com 3 processos. Tela do processo 3.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-
└─$ go run SharedResource.go
Received {1 5 oi}
oi
Received {3 10 xau}
xau

```

Figura 9. Exemplo do funcionamento da tarefa com 3 processos. Tela do SharedResource.

Como esses resultados foram condizentes com os resultados esperados, leva-se a perceber que a implementação da Tarefa foi feita corretamente. Mas, antes de concluir-se algo, foi-se realizado um segundo teste.

B. Teste 2

Este foi o caso com processos solicitando a CS "simultaneamente", sugerido no roteiro do laboratório. O esquema do resultado esperado foi apresentado na Figura 10.

Análogo ao teste 1, na Figura 10, para P1, as setas azuis representam requests e as verdes, replies; e para P4, essa setas são rosas e cinzas, respectivamente. Nos requests, a mensagem enviada também é da forma "*relógio lógico, < timestamp, id >*"; assim como no reply, que a forma também é "*relógio lógico, 'reply'*". Além disso, nesse caso também a linha amarela representa o processo no estado WANTED; e a linha vermelha, no estado HELD, ou seja, na CS.

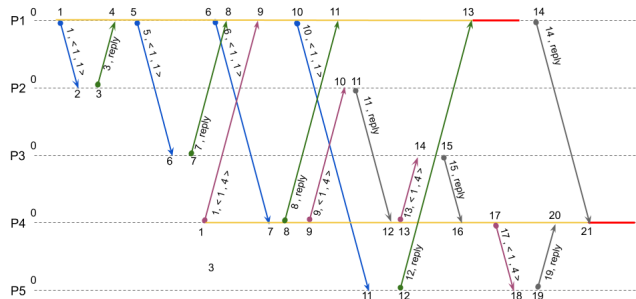


Figura 10. Funcionamento esperado para a tarefa para 5 processos.

Foi simulada essa situação acima com o código implementado no laboratório, tendo os resultados apresentados nas Figuras de 11 a 16. Durante o período na CS, também foi digitado mensagens no terminal dos processos, para que se verificasse o funcionamento da validação da mensagem. Assim, para essas mensagens, foi dado o feedback de mensagem inválida.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-CES-27/Lab2 <master*>
$ go run Process.go 1 :10004 :10003 :10002 :10005 :10006
oi
Entrei na CS
a
a invalido
Sai da CS
```

Figura 11. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 1.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-CES-27/Lab2 <master*>
$ go run Process.go 2 :10004 :10003 :10002 :10005 :10006
```

Figura 12. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 2.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-CES-27/Lab2 <master*>
$ go run Process.go 3 :10004 :10003 :10002 :10005 :10006
```

Figura 13. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 3.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-CES-27/Lab2 <master*>
$ go run Process.go 4 :10004 :10003 :10002 :10005 :10006
xau
Entrei na CS
a
a invalido
4
4 invalido
Sai da CS
```

Figura 14. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 4.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-CES-27/Lab2 <master*>
$ go run Process.go 5 :10004 :10003 :10002 :10005 :10006
```

Figura 15. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 5.

```
isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-CES-27/Lab2 <master*>
$ go run SharedResource.go
oi
xau
```

Figura 16. Exemplo do funcionamento da tarefa com 5 processos. Tela do SharedResource.

A fim de entender melhor cada estágio do funcionamento, foram realizados os mesmos testes novamente, agora com prints de debug. Dessa forma, esses resultados estão apresentados nas Figuras de 17 a 22. Como é difícil simular com exatidão os instantes dos envios de request dos processos simultaneamente, foi feito da seguinte forma: o processo P4 entra no estado WANTED durante o estado HELD do processo P1. Isso é um pouco diferente do apresentado na Figura 10, mas não prejudica os testes. Contudo, isso altera um pouco a ordem dos envios de mensagens, e, consequentemente, os valores relógios lógicos.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana
└─$ go run Process.go 1 :10004 :10003 :10002 :10005 :10006
Estado: RELEASED
oi
Estado: WANTED
Multicast request to all processes
Esperando N-1 respostas
logicalClock atualizado: 1
Request enviado: 2 , < 0 , 1 >
logicalClock atualizado: 2
Request enviado: 2 , < 0 , 1 >
logicalClock atualizado: 3
Request enviado: 3 , < 0 , 1 >
logicalClock atualizado: 4
Request enviado: 4 , < 0 , 1 >
Reply recebido: 5 reply
logicalClock atualizado: 6
Reply recebido: 6 reply
logicalClock atualizado: 7
Reply recebido: 4 reply
logicalClock atualizado: 8
Reply recebido: 4 reply
logicalClock atualizado: 9
Estado: HELD
Entrei na CS
Request recebido: 6 , < 4 , 4 >
logicalClock atualizado: 10
Sai da CS
Estado: RELEASED
Reply enviado: 11 reply

```

Figura 17. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 1.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana
└─$ go run Process.go 4 :10004 :10003 :10002 :10005 :10006
Estado: RELEASED
xauRequest recebido: 2 , < 0 , 1 >
logicalClock atualizado: 3
Reply enviado: 4 reply
logicalClock atualizado: 4

Estado: WANTED
Multicast request to all processes
logicalClock atualizado: 6
Request enviado: 6 , < 4 , 4 >
logicalClock atualizado: 7
Esperando N-1 respostas
logicalClock atualizado: 8
Request enviado: 8 , < 4 , 4 >
logicalClock atualizado: 5
Request enviado: 8 , < 4 , 4 >
Reply recebido: 10 reply
logicalClock atualizado: 11
Request enviado: 7 , < 4 , 4 >
Reply recebido: 10 reply
logicalClock atualizado: 12
Reply recebido: 9 reply
logicalClock atualizado: 13
Reply recebido: 11 reply
logicalClock atualizado: 14
Estado: HELD
Entrei na CS
Sai da CS
Estado: RELEASED

```

Figura 20. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 4.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana
└─$ go run Process.go 2 :10004 :10003 :10002 :10005 :10006
Estado: RELEASED
Request recebido: 3 , < 0 , 1 >
logicalClock atualizado: 4
Reply enviado: 5 reply
logicalClock atualizado: 5
Request recebido: 8 , < 4 , 4 >
logicalClock atualizado: 9
Reply enviado: 10 reply
logicalClock atualizado: 10

```

Figura 18. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 2.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana
└─$ go run Process.go 5 :10004 :10003 :10002 :10005 :10006
Estado: RELEASED
Request recebido: 2 , < 0 , 1 >
logicalClock atualizado: 3
Reply enviado: 4 reply
logicalClock atualizado: 4
Request recebido: 8 , < 4 , 4 >
logicalClock atualizado: 9
Reply enviado: 10 reply
logicalClock atualizado: 10

```

Figura 21. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 5.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana-CES-27/L
└─$ go run SharedResource.go
Received {1 9 oi}
oi
Received {4 14 xau}
xau

```

Figura 22. Exemplo do funcionamento da tarefa com 5 processos. Tela do SharedResource.

```

isabelle@isabelle-Inspiron-5448 ~/Graduacao/hirata-juliana
└─$ go run Process.go 3 :10004 :10003 :10002 :10005 :10006
Estado: RELEASED
Request recebido: 4 , < 0 , 1 >
logicalClock atualizado: 5
Reply enviado: 6 reply
logicalClock atualizado: 6
Request recebido: 7 , < 4 , 4 >
logicalClock atualizado: 8
Reply enviado: 9 reply
logicalClock atualizado: 9

```

Figura 19. Exemplo do funcionamento da tarefa com 5 processos. Tela do processo 3.

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