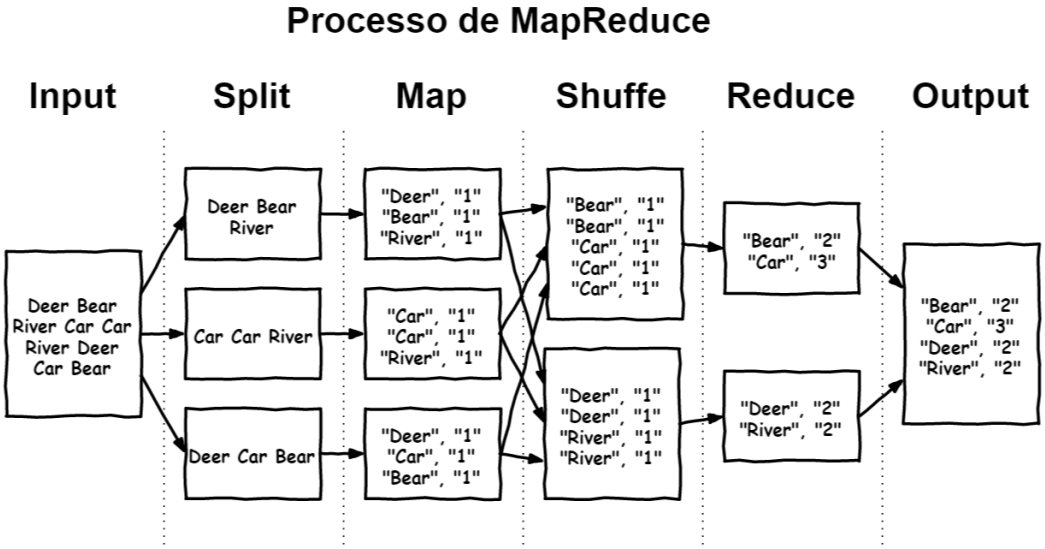
3º ATIVIDADE de CES-27 / 2019

CTA - ITA - IEC

Prof Hirata e Prof Juliana

Objetivo: Trabalhar com o modelo de programação MapReduce de forma sequencial e distribuída.

Entregar (através do GoogleClassroom): Códigos finais (arquivos .go) e relatório. O relatório deve indicar detalhes particulares/críticos do código, além apresentar as tarefas realizadas e comentar resultados.



Primeiramente vamos compreender como usar o código fornecido.

- Baixar labMapReduce.zip com os códigos fornecidos.

- Descompactar essa pasta no seu `$GOPATH/src`
  - `$GOPATH` é o diretório onde você armazena seus códigos Go.
  - Dentro de `$GOPATH/src` deve então ficar a pasta `labMapReduce` com duas outras dentro (`mapreduce` e `wordcount`)
  - `mapreduce` é o pacote que implementa a framework MapReduce e deve se abster de detalhes da operação a ser realizada.
  - `wordcount` é o pacote que implementa a operação Contador de Palavras, e deve importar e entender as operações da API provida pelo pacote acima.

- Instalar a package `mapreduce` fornecida:
  - Entrar em `$GOPATH/src/labMapReduce/mapreduce`
  - Executar: `go install`
  - Verificar que aparece `mapreduce.a` num subdiretório `/pkg` de `$GOPATH`
  - Esse passo é necessário para os arquivos do package `wordcount` poder importar "labMapReduce/mapreduce" corretamente.

Atenção: Essa parte não vai funcionar agora porque os códigos das funções *map* e *reduce* estão incompletos. Mas vale olharmos para entender um pouco mais.

- Para compilar:
  - Entrar em `$GOPATH/src/ labMapReduce/wordcount`
  - Executar: `go build`
  -
- Para rodar:
  - Entrar em `$GOPATH/src/labMapReduce/wordcount`
  - Executar: `wordcount.exe -mode sequential -file files/teste.txt`
  - Obs: Esse arquivo `teste.txt` é o que será processado. Temos dois modos de execução `sequential` e `distributed`. Veremos em seguida cada um.

Obs: Na função `main.go`, está definido como default 5 *reducejobs* (ou seja, simularemos 5 tarefas de *reduce*) e *chunksize* de 1024 bytes (ou seja, esse é o tamanho máximo de cada "pedaço" do arquivo original que será "fatiado"). Esses parâmetros poderão ser alterados. Por exemplo, executando:

```
wordcount.exe -mode sequential -file files/teste.txt -chunksize
100 -reducejobs 2
```

PARTE 1: Trabalhando em modo sequencial

Para o modo sequencial somente alguns arquivos fornecidos são realmente usados. Vejamos:

- `mapreduce\`
  - `common.go`
  - `mapreduce.go`
- `wordcount\`
  - `main.go`
  - `data.go`
  - `wordcount.go`
  - `files\`
    - `pg1342.txt`
    - `teste.txt`

Explicação geral:

`common.go`: Contém as definições dos dados que as aplicações devem estender para realizar uma operação de MapReduce. Além disso contém funções internas compartilhadas pelo framework.

`mapreduce.go`: Contém o código de inicialização da operação de MapReduce.

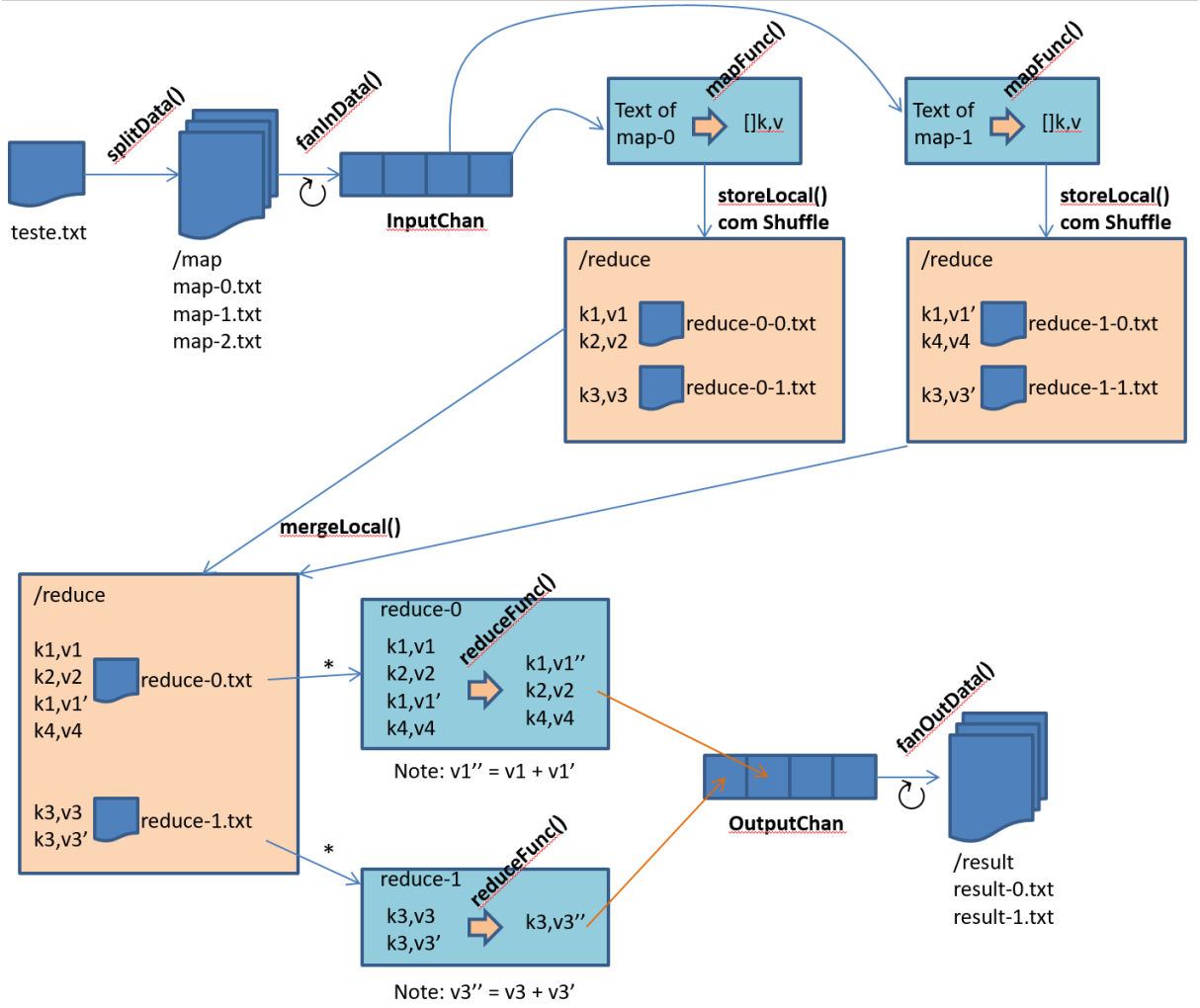
`main.go`: Contém o código de inicialização do programa.

`data.go`: Contém o código que lida com os dados que serão fornecidos e retornados pela operação de MapReduce.

`wordcount.go`: Contém o código que implementa as operações *map* e *reduce*.

`files\`: Diretório com arquivos para testarmos MapReduce.

Esquema para ajudar a compreender o funcionamento do programa:



Obs 1: Aqui consideramos que o arquivo inicial foi "quebrado" em 3 *chunks* e que temos 2 *reduce jobs*.

Obs 2:
reduce-0-0.txt são dados do **map-0.txt** a serem trabalhados pelo **reduce worker 0**
reduce-0-1.txt são dados do **map-0.txt** a serem trabalhados pelo **reduce worker 1**
reduce-1-0.txt são dados do **map-1.txt** a serem trabalhados pelo **reduce worker 0**
reduce-1-1.txt são dados do **map-1.txt** a serem trabalhados pelo **reduce worker 1**
reduce-0.txt é a junção de **reduce-0-0.txt e reduce-1-0.txt**
reduce-1.txt é a junção de **reduce-0-1.txt e reduce-1-1.txt**

Obs 3: Na versão distribuída, usa-se uma fila para intermediar *. Assim os *workers* vão pegando *reduce jobs*.

Tarefa 1.1. Complete a função `mapFunc` (no arquivo `wordcount.go`) para identificar as palavras de cada fatia do arquivo inicial. Para facilitar, toda palavra deve ser convertida para minúsculo. Vamos trabalhar da seguinte forma: registrar cada palavra com valor um. Não precisa ter ordenação alfabética entre as palavras.
Ex:

```
<"car",1> <"ball", 1> <"car",1>
```

Tarefa 1.2. Implemente a função `reduceFunc` (no arquivo `wordcount.go`) para consolidar a contagem das palavras. Novamente não precisa ter ordenação alfabética entre as palavras.
Ex:

```
<"car",2> <"ball", 1>
```

Tarefa 1.3. Rode o programa com o arquivo `teste.txt`. Comente sobre os resultados/arquivos (parciais e finais) indicando se estão coerentes com o esperado. Use o seguinte comando para rodar:

```
wordcount.exe -mode sequential -file files/teste.txt -chunksize 100 -reducejobs 2
```

Tarefa 1.4. Rode o programa com outro arquivo de entrada. Comente sobre os resultados/arquivos (parciais e finais) indicando se estão coerentes com o esperado. Use diferentes valores para `chunksize` e `reducejobs`.

PARTE 2: Trabalhando em modo distribuído

Nesta segunda parte, temos pronto um processo de MapReduce que já opera de forma distribuída. A implementação considera que todos *maps* são feitos para depois fazer todos *reduces*.

Curiosidade: Aqui usa-se RPC (*remote procedure call*).

- Os *workers* se registram no *master* chamando uma função (`Register`) do *master* via RPC.
- O *master* passa *jobs* (*map* e *reduce*) para os *workers* através da chamada de funções (`RunMap` e `RunReduce`) nos *workers* via RPC.
- Mais detalhes sobre RPC: https://golang.org/pkg/net/rpc/

Primeiramente vamos compreender como usar o código fornecido.

Para executar o código, precisamos executar múltiplos processos (cada uma em um terminal). Temos um ***master*** (com porta fixa 5000) e um ou vários ***workers*** (com porta definida ao executar o código com atributo `-port`). O ideal é subir primeiro os *workers* e no final o *master*, assim o *master* terá todos seus *workers* para trabalhar. Lembrado que esses *workers* podem fazer tarefas de *map* e/ou *reduce*.

Exemplo com um *worker* e um *master*:

```
wordcount.exe -mode distributed -type worker –port 50001

wordcount.exe -mode distributed -type master -file files/pg1342.txt -
chunksize 102400 -reducejobs 5
```

Obs: O arquivo `pg1342.txt` já foi fornecido.
Obs: Como resultado, vemos que o *master* usou somente o *worker* 0 para realizar todos os jobs de *map* e *reduce*.

Exemplo com dois *workers* e um *master*:

```
wordcount.exe -mode distributed -type worker –port 50001

wordcount.exe -mode distributed -type worker –port 50002

wordcount.exe -mode distributed -type master -file files/pg1342.txt -
chunksize 102400 -reducejobs 5
```

Obs: Como resultado, vemos que o *master* usou os *workers* 0 e 1 para realizar os jobs de *map* e *reduce*.

Introduzindo falhas

Vamos introduzir agora um processo *worker* no qual vamos induzir uma falha.

Vamos usar um *worker* (que falha na execução de sua terceira tarefa) e um *master*:

```
wordcount.exe -mode distributed -type worker –port 50001 –fail 3

wordcount.exe -mode distributed -type master -file files/pg1342.txt -
chunksize 102400 -reducejobs 5
```

Note que a execução do *worker* encerra.
Já a execução do *master* trava:

```
Operation Worker.RunMap '1' Failed. Error: read tcp 127.0.0.1:58204->127.0.0.1:5001: wsarecv: An existing connection
was forcibly closed by the remote host.
```

Se subirmos outro *worker*...

```
wordcount.exe -mode distributed -type worker -port 50002
```

Vemos que o *master* continua o processamento dos jobs faltantes (alocando tudo para o novo *worker*).

Para entender melhor vamos olhar os trechos de código a seguir.
Todas essas estruturas são utilizadas para gerenciar os `Workers` no `Master`:

```
// master.go
type Master struct {
    (...)
    // Workers handling
    workersMutex sync.Mutex
    workers      map[int]*RemoteWorker
    totalWorkers int // Used to generate unique ids for new workers

    idleWorkerChan   chan *RemoteWorker
    failedWorkerChan chan *RemoteWorker
    (...)
}
```

A primeira coisa que acontece, quando um *worker* é inicializado, é a chamada da função `Register` no `Master` (Obs. Essa função na verdade é um método da classe `Master`):

```
// master_rpc.go
func (master *Master) Register(args *RegisterArgs, reply *RegisterReply) error {
    (...)
    master.workersMutex.Lock()

    newWorker = &RemoteWorker{master.totalWorkers, args.WorkerHostname, WORKER_IDLE}
    master.workers[newWorker.id] = newWorker
    master.totalWorkers++

    master.workersMutex.Unlock()

    master.idleWorkerChan <- newWorker
    (...)
}
```

Observe que, quando um novo `Worker` se registra no `Master`, ele adiciona esse *worker* na lista `master.workers`.
Para evitar problemas de sincronia no acesso dessa estrutura usamos o `mutex` `master.workersMutex`.

Um ponto interessante é a última linha acima, onde o *worker* que acabou de ser registrado é colocado no canal de *workers* disponíveis.
A saída desse canal está na função `schedule`, que aloca as operações (de *map* ou *reduce*) nos *workers*. Vide abaixo. Veja que cada alocação é uma *goroutine* lançada (com a função `runOperation`).

```go
// master_scheduler.go
func (master *Master) schedule(task *Task, proc string, filePathChan chan string) {
    (...)
    counter = 0
    for filePath = range filePathChan {
        operation = &Operation{proc, counter, filePath}
        counter++

        worker = <-master.idleWorkerChan
        wg.Add(1)
        go master.runOperation(worker, operation, &wg)
    }

    wg.Wait()
    (...)
}
```

Voltando ao nosso exemplo, vejamos porque o *master* consegue continuar só com o novo *worker*, após o primeiro ter falhado:

```go
// mapreduce.go
    (...)
    go master.acceptMultipleConnections()
    go master.handleFailingWorkers()

    // Schedule map operations
    master.schedule(task, "Worker.RunMap", task.InputFilePathChan)
    (...)
```

Observe que a linha `master.acceptMultipleConnections()` é colocada em uma goroutine separada. Já na linha `master.schedule(...)`, fazemos a execução na *goroutine* atual. Dessa forma essas operações estão acontecendo concorrentemente, e o `master.idleWorkerChan` é o canal de comunicação entre elas. Quando a operação `Register` escreve no canal, a operação `schedule` é informado de que um novo *worker* está disponível e continua a execução.

Ainda no nosso exemplo, vejamos porque o *master* travou logo após seu primeiro e único *worker* falhar:

```go
func (master *Master) runOperation(remoteWorker *RemoteWorker, operation *Operation, wg *sync.WaitGroup) {
    (...)
    err = remoteWorker.callRemoteWorker(operation.proc, args, new(struct{}))

    if err != nil {
        log.Printf("Operation %v '%v' Failed. Error: %v\n", operation.proc, operation.id, err)
        wg.Done()
        master.failedWorkerChan <- remoteWorker
    } else {
        wg.Done()
        master.idleWorkerChan <- remoteWorker
    }
    (...)
}
```

Quando um *worker* completa uma operação corretamente, ele cai no `else` acima e o *worker* que a executou é colocado de volta no canal `master.idleWorkerChan` (e que vai ser lido pelo *scheduler* mostrado anteriormente).

Entretanto, no caso de falha, o *worker* é colocado no canal `master.failedWorkerChan`. Atualmente ninguém trata esse canal! É por isso que a execução trava. O *scheduler* vai esperar infinitamente por um *worker* (que falhou).

Para concluir a execução do nosso exemplo, nós adicionamos um segundo *worker* e as operações (de *map* e *reduce*) foram ser retomadas. No fim da tarefa de MapReduce, o *master* informa a todos os *workers* que a tarefa foi finalizada. Neste caso recebemos o seguinte erro (dado que temos um *worker* falho):

```
Closing Remote Workers.
Failed to close Remote Worker. Error: dial tcp [::1]:50001: connectex: No connection could be made because the target machine actively refused it.
Done.
```

Tarefa 2.1. Complete o código da função `handleFailingWorkers` (do arquivo `master.go`).

Essa rotina é executada em uma *goroutine* separada (no arquivo `mapreduce.go`, logo abaixo de `go master.acceptMultipleConnections()`). Você deve alterá-la de forma que toda vez que um *worker* falhar durante uma operação, ele seja corretamente tratado.

Num ambiente real, existem várias possibilidades, como informar ao processo que gerencia a inicialização dos *workers* o endereço do *worker* falho; ou verificar se o *worker* ainda está vivo (isso pode acontecer no caso de uma falha de rede, por exemplo).

No nosso caso, não vamos tentar retomar o processo. Vamos **apenas registrar que o processo não está mais disponível**.

> Dica 1: A função deve monitorar o canal `master.failedWorkerChan`. Para isso, é interessante observar o uso da operação `range` em estruturas do tipo `channel`. https://gobyexample.com/range-over-channels.
>
> Dica 2: Para remover elementos em estruturas do tipo `map` (no caso, o `master.workers`), utilizar a operação `delete`. https://gobyexample.com/maps.
>
> Dica 3: Para garantir a sincronia da estrutura, utilizar o `mutex master.workersMutex` para proteger o acesso à estrutura `master.workers`. https://gobyexample.com/mutexes.

Resultado esperado ao remover o *worker* falho da lista (a linha em azul foi inserida!):

```
Running Worker.RunMap (ID: '2' File: 'map\map-2' Worker: '0')
Operation Worker.RunMap '2' Failed. Error: read tcp 127.0.0.1:56282->127.0.0.1:50001: wsarecv: An existing connection was forcibly closed by the remote host.
Removing worker 0 from master list.
```

Resultado esperado no final: mesmo com falhas, os *workers* são finalizados corretamente (a linha em azul aparece devido a correta execução!):

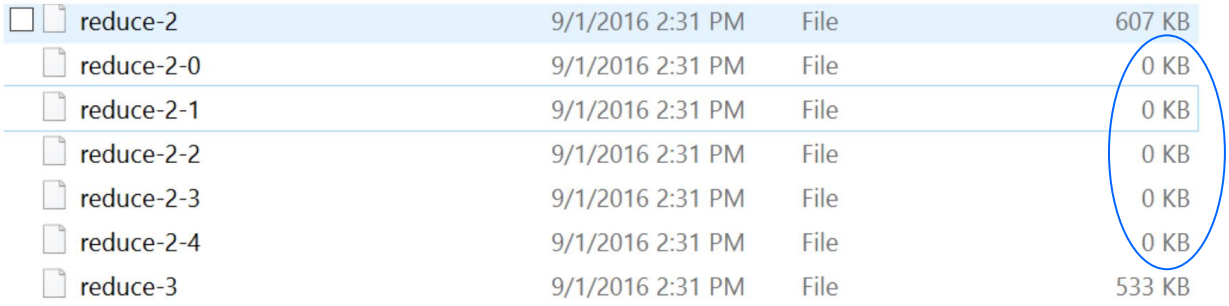
```
Closing Remote Workers.
Done.
```

Recuperando após falhas

Na tarefa acima, fizemos com que o nosso código terminasse de forma natural mesmo que *workers* falhassem. Entretanto, a operação falha nunca foi realizada, e por conta disso, o nosso MapReduce está incorreto.

No nosso exemplo, induzimos falha na 3 operação do *worker 0*. O *master* informou a operação que falhou, veja: `Operation Worker.RunMap '2' Failed.` Analisando os arquivos na pasta `reduce/`, vemos que realmente essa operação falhou: resultando em arquivos vazios!

> Obs: Olhamos a pasta `reduce/` em caso de falha de uma operação *map*. Olhamos a pasta `result/` em caso de falha de uma operação reduce.



reduce-2	9/1/2016 2:31 PM	File	607 KB
reduce-2-0	9/1/2016 2:31 PM	File	0 KB
reduce-2-1	9/1/2016 2:31 PM	File	0 KB
reduce-2-2	9/1/2016 2:31 PM	File	0 KB
reduce-2-3	9/1/2016 2:31 PM	File	0 KB
reduce-2-4	9/1/2016 2:31 PM	File	0 KB
reduce-3	9/1/2016 2:31 PM	File	533 KB

Para começar a entender, vejamos o código abaixo:

```
master_scheduler.go
func (master *Master) schedule(task *Task, proc string, filePathChan chan string) {
    var (
        wg        sync.WaitGroup
    (...)
    counter = 0
    for filePath = range filePathChan {
        operation = &Operation{proc, counter, filePath}
        counter++

        worker = <-master.idleWorkerChan
        wg.Add(1)
        go master.runOperation(worker, operation, &wg)
    }

    wg.Wait()
    (...)
}
```

Temos a seguinte lógica: Os nomes dos arquivos que devem ser processados são obtidos através de uma leitura em um canal (`filePath = range filePathChan`). Em seguida, uma operação é criada e um *worker* é obtido através do canal `master.idleWorkerChan`. Em seguida, 1 delta é adicionado ao `WaitGroup wg` e uma nova goroutine é executada com a chamada de `runOperation`.

`WaitGroup` é um mecanismo de sincronização de um número variável de *goroutines*. Neste caso, toda vez que uma operação é executada em uma nova *goroutine*, adicionamos 1 ao contador no `WaitGroup`. Desta forma, na linha `wg.Wait()` o código vai ficar bloqueado até que *n goroutines* tenham chamado `wg.Done()`, onde *n* é o total de deltas adicionado ao `WaitGroup`. O código abaixo indica a chamada de `wg.Done()`:

```go
// master_scheduler.go
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)
        wg.Done()
        master.failedWorkerChan <- remoteWorker
    } else {
        wg.Done()
        master.idleWorkerChan <- remoteWorker
    }
}
```

Tarefa 2.2. Altere o código fornecido de forma que as operações que falhem sejam executadas. Uma solução simples utiliza canais de forma eficiente para comunicar operações que falharam entre *goroutines*. É indicado que alterações sejam feitas apenas nos seguintes arquivos:

```go
// master.go
    (...)
    ////////////////////////////////
    // ADD EXTRA PROPERTIES HERE //
    ////////////////////////////////
    // Fault Tolerance
    (...)
```

Dicas:
- Ao adicionar propriedades no `Master struct`, pode ser necessário inicializá-las na função `newMaster`.
- A constante `RETRY_OPERATION_BUFFER` já está definida.

```go
// master_scheduler.go
func (master *Master) schedule(task *Task, proc string, filePathChan chan string) {
    (...)
    ////////////////////////////////
    // YOU WANT TO MODIFY THIS CODE //
    ////////////////////////////////
    (...)
}

func (master *Master) runOperation(remoteWorker *RemoteWorker, operation *Operation, wg *sync.WaitGroup) {
    (...)
    ////////////////////////////////
    // YOU WANT TO MODIFY THIS CODE //
    ////////////////////////////////
    (...)
}
```

É importante observar que o código de `runOperation` é sempre executado em uma *goroutine* distinta:

```
master_scheduler.go
func (master *Master) schedule(task *Task, proc string, filePathChan chan string) {
    (...)
    for filePath = range filePathChan {
        (...)
        go master.runOperation(worker, operation, &wg)
    }
    (...)
}
```

É necessário desenvolver uma forma de compartilhar a informação de que uma `Operation` não foi concluída corretamente com a execução do método `schedule` (o responsável por alocar as operações nos *workers*). A dica é usar um *channel* dedicado para isso.

> Obs: Por que criar um novo canal? Por que não reusar o canal `filePathChan`?
> O canal `filePathChan` vai retornar todos os arquivos criados pelo `splitData` apenas uma vez, preparando para começar as operações *map*. Depois, esse canal é usado para guardar os arquivos consolidados de *reduce* (ex: `reduce-0.txt`, etc.), preparando para começar as operações *reduce*. Esse canal é controlado por métodos externos. Reutilizar esse canal pode causar problemas (por exemplo, o canal pode ser encerrado pelo código externo, e ao tentar escrever nele, uma chamada de *panic* será feita).

Após sua implementação, seguindo o nosso exemplo, o *master* deve então conseguir rodar a operação falha depois e os arquivos em `reduce/` ficarão coerentes.

```
Running Worker.RunMap (ID: '2' File: 'map\map-2' Worker: '1')
```

Tarefa 2.3. Rode o programa com o arquivo `teste.txt`. Comente sobre os resultados/arquivos (parciais e finais) indicando se estão coerentes com o esperado. Utilize um *worker* falho e outro normal. Emule falha numa operação *map*. Em outro teste, emule falha numa operação *reduce*.

Tarefa 2.4. Rode o programa com outro arquivo de entrada. Comente sobre os resultados/arquivos (parciais e finais) indicando se estão coerentes com o esperado. Use diferentes valores para `chunksize` e `reducejobs`. Use um *master* e dois (ou mais) *workers* falhos.

Bom trabalho!