MAC0219 - Relatório Mini-EP 8

Nathália Yukimi Uchiyama Tsuno (14600541)

November 2024

1 Tarefa 1

1.1 Algoritmo

```
package main

import (
    "encoding/json"
    "log"

    maelstrom "github.com/jepsen-io/maelstrom/demo/go"
)


func main() {
        n := maelstrom.NewNode()

        n.Handle("echo", func(msg maelstrom.Message) error {
                // Unmarshal the message body as an loosely-typed map.
                var body map[string]any
                if err := json.Unmarshal(msg.Body, &body); err != nil {
                        return err
                }

                // Update the message type to return back.
                body["type"] = "echo_ok"

                // Echo the original message back with the updated message type.
                return n.Reply(msg, body)
        })

        if err := n.Run(); err != nil {
                log.Fatal(err)
        }

}
```

1.2 Como um Nó É Representado no Maelstrom?

Nós são binários que recebem mensagens em JSON via STDIN e enviam outras mensagens via STDOUT.

No maelstrom, os nós são representados como pontos numa rede, que podem se comunicar entre si por meio de mensagens. Podemos ter vários nós que podem realizar tarefas independentes ou que exijam interação com outrém.

1.3 O que faz o método `Node.Reply()`?

O método `Node.Reply()`, é executado após a recepção e atualização de uma mensagem, e tem a função de retorná-la ao nó que fez a solicitação inicial.

1.4 Explique brevemente o método `Node.Run()`

O método `Node.run()` é responsável por iniciar um nó, configurá-lo para interagir com outros nós na rede simulada, processar eventos e responder às mensagens de outros nós durante o teste, dentro do ambiente adequado.

Dentro do Maelstrom, o método `Node.run()` é uma função que inicia a execução de um nó dentro de um ambiente de teste. Cada nó no Maelstrom representa uma instância de um componente do sistema distribuído sendo testado. Durante a execução, o nó escuta as mensagens que são trocadas entre os nós do sistema, conforme o teste vai acontecendo. E finaliza, quando o teste for concluído ou quando o nó terminar sua execução.

2 Tarefa 2

2.1 Item 1

2.1.1 Algoritmo

```
package main

import (
        "encoding/json"
        "log"
        "os"
        "sync"

        maelstrom "github.com/jepsen-io/maelstrom/demo/go"
)

type Recurso struct {
        counter int
        mux     sync.Mutex
}

func main() {
        n := maelstrom.NewNode()

        recurso := &Recurso{
                counter: 0,
```

```
        }

        // Register a handler for the "generate" message that responds with
    // an "generate_ok" and a unique ID.

        n.Handle("generate", func(msg maelstrom.Message) error {
                // Unmarshal the message body as an loosely-typed map.
                var body map[string]any
                if err := json.Unmarshal(msg.Body, &body); err != nil {
                        return err
                }

                // Get current counter value and increment it.
                body["id"] = getAndIncrementCounter(recurso) // Operacao atomica

                // Update the message type.
                body["type"] = "generate_ok"

                return n.Reply(msg, body)
        })

        // Execute the node's message loop. This will run until STDIN is closed.
        if err := n.Run(); err != nil {
                log.Printf("ERROR: %s", err)
                os.Exit(1)
        }
}

func getAndIncrementCounter(recurso *Recurso) int {
        recurso.mux.Lock()
        defer recurso.mux.Unlock() // Destrava o Mutex, apos o retorno

        id := recurso.counter
        recurso.counter++

        return id
}
```

2.1.2 Explique em poucas palavras as principais alterações no código e por que elas resolvem o problema.

As principais alterações do código foram a transformação da variável global contadora para uma estrtura que contenha sincronizadores e barreiras, em especial, o Mutex, além da atomização da operação de leitura e escrita.

A priori, quando implementamos uma estrtura que contenha o mutex, bloqueamos o acesso e escrita múltiplos a um mesmo recurso compartilhado (o contador para atribuição de id's). Assim, coordenamos as requisições e operamos de modo que nenhuma duas mensagens tenham o mesmo id.

Ademais, paralelamente a isso, quando tornamos a operação de obtenção e incremento algo atômico, evitamos que, enquanto uma mensagem obtém o id, outra incremente simultaneamente, gerando problemas na unicidade do id.

2.2 Item 2

2.2.1 Algoritmo

```
package main

import (
        "encoding/json"
        "log"
        "os"
        "fmt"
        "sync"
        "time"

        maelstrom "github.com/jepsen-io/maelstrom/demo/go"
)

type Recurso struct {
        nodeID  string
        counter int
        mux     sync.Mutex
}

func main() {
        n := maelstrom.NewNode()

        recurso := &Recurso{
                nodeID:  n.ID(),
                counter: 0,
        }

        // Register a handler for the "generate" message that responds with an "generate_
        n.Handle("generate", func(msg maelstrom.Message) error {
                // Unmarshal the message body as an loosely-typed map.
                var body map[string]any
                if err := json.Unmarshal(msg.Body, &body); err != nil {
                        return err
                }

                // Get current counter value and increment it.
                body["id"] = getAndIncrementCounter(recurso) // Operacao atomica

                // Update the message type.
                body["type"] = "generate_ok"

                return n.Reply(msg, body)
        })

        // Execute the node's message loop. This will run until STDIN is closed.
        if err := n.Run(); err != nil {
                log.Printf("ERROR: %s", err)
                os.Exit(1)
```

```
        }
}

func getAndIncrementCounter(recurso *Recurso) string {
        recurso.mux.Lock()
        defer recurso.mux.Unlock() // Destrava o Mutex, apos o retorno

        timestamp := time.Now().UnixNano()

        //id := recurso.counter
        id := fmt.Sprintf("%s-%d-%d", recurso.nodeID, recurso.counter, timestamp)
        recurso.counter++

        return id
}
```

2.2.2 Explique sucintamente sua solução.

Nesta modalidade, são executados e operados três nós, pontos da rede, que enviam e recebem, simultaneamente mensagens entre si. Nesse viés, precisamos coordená-las, uma vez que cada nó emite seus próprios ids.

Assim, precisamos que cada nó atribua um id único e que nenhum outro nó pense em fazer igual. Em especial, cada nó possui um id, obtido pela função `NodeID()`. Para evitar conflitos entre os ids dos nós, podemos concatenar, a cada contador individual, o NodeID mensageiro. Ainda assim, isso não é suficiente, mesmo aplicado aos mutexes, pois ainda podem haver colisões.

Para isso, vamos aplicar a técnica do timestamp, incluindo o tempo de geração do ID para garantir que, mesmo se os contadores locais se cruzarem, a ordem temporal irá diferenciar os IDs. É o desempate.

Assim, criamos ids da forma: `<NodeID>-<counter>-<timestamp>`, garantindo a unicidade de IDs para sistemas particionados.