Práctica 3 - Tolerancia a fallos en Servidores Sin Estado

Sistemas Distribuidos 2021-2022

Jorge Lisa 774248
David Zandundo 780500

Índice

Introducción

En esta práctica se va a tratar la tolerancia a fallos en un sistema con arquitectura Master-Worker donde el worker es inestable. Este es capaz de retrasar su respuesta, omitírla durante más tiempo que el retraso, crashear su sistema o actuar de forma normal.

Objetivos

Desarrollar un sistema capaz de detectar los posibles fallos del worker y ser capaces de tratarlos en el menor tiempo posible, intentando no superar los 3 segundos de tiempo de respuesta.
También se pide intentar desarrollar un sistema flexible, capaz de quitar y añadir workers en tiempo de ejecución.

Metodología

Cambios al worker

El cambio más notable ha sido el cambio de paso de mensajes a RPC. Se ha hecho que en vez de que sea un bucle que siempre está escuchando a un cliente el encargado de enviar por el canal una tarea, lo haga en este sistema la llamada RPC.

```go
func (p *Primes) FindPrimes(interval com.TPInterval, primeList *[]int) error {
    resp := make(chan Respuesta)

    p.canal <- Mensaje{interval, resp} // Enviamos la petición a un worker
    respuesta := <-resp                // Esperamos a la respuesta del worker

    if respuesta.err != nil {
        *primeList = respuesta.reply // Devolvemos la respuesta
    }

    return respuesta.err
}
```

Como debe ser la misma llamada RPC la que le devuelva al cliente la respuesta, se debe esperar a que el worker termine y así devolverle al cliente su tarea realizada. En el anterior sistema, esto lo podía hacer el worker, ya que le pasábamos a este la conexión respectiva.

Cálculo aproximado del coste del intervalo

Tal y como se hizo en la práctica 1, con algunos retoques debido a errores en las escalas de la gráfica, se ha calculado un coste aproximado del tiempo de FindPrimes para un intervalo dado.
Esta función se ha obtenido de la misma manera, sin llegar a la integración, ya que esta servía para obtener un intervalo subóptimo.

$$\sum_{j=A}^{B} 0.00164 \cdot j^{0.9055}$$

Como el coste de calcular esta función podría ser muy elevado si el intervalo es muy grande, se ha decidido ir de 1000 en 1000 en vez de 1 en 1 a la hora de calcular el sumatorio. Esto da una menor precisión, pero no genera un coste lo suficientemente elevado como para considerarlo en el coste total de la ejecución del algoritmo.

Posibles formas actuar del worker

Como los workers no son consistentes, se ha calculado las probabilidades de que ocurra cada situación, siendo estas Normal, Omisión, Retraso y Crash.

Estado	Normal	Omisión	Delay	Crash	Total
Ocurrencias	1115	(351 + 195)*	678	17	2356
Porcentaje	47.33	(14.9 + 8.27)	28.78	0.72	100

* El estado de omisión puede tener 2 posibles actuaciones, que responda instantáneamente o que tarde 10 segundos. 351 de las veces que se ha omitido ha dado una respuesta instantánea y 195 veces ha tardado 10 segundos en hacerlo.

Como los estados Normales y de Crash se pueden detectar de forma instantánea, nos vamos a centrar en los de omisión y de delay, los cuales durante la ejecución, son indistinguibles.
Si normalizamos sus datos para solo tener en cuenta estos dos estados, obtenemos lo siguiente:

	Omisión	Delay	Total
Ocurrencias	(351 + 195)*	678	1224
Porcentaje	28.68 + 15.93	55.39	100

Posteriormente, se ha calculado la media armónica del retraso en las ocasiones de Delay, ya que en las de omisión, sabemos cuánto tarda.

Media: 2.921 s

Tratamiento de los Fallos

Hay 3 tipos de fallos de los cuales 2 de ellos son indistinguibles durante la ejecución, siendo estos la omisión y el retraso.
Para tratar estos primeros fallos, se ha decidido que cuando el sistema tarda más del coste aproximado del cálculo del intervalo sumado al coste medio anteriormente calculado, se enviará la tarea a otro worker para que este sea él que realice el cálculo, ya que es más probable que el worker actual esté en el estado de omisión de 10 segundos que en el de delay.

```
select {
case mensajeWorker := <-callChan: // Recepción del mensaje a tiempo
    respuesta.err = mensajeWorker.Error

    if mensajeWorker.Error != nil { // CRASH
        primes.canal <- msj
        primes.lanzarWorker(id)
        time.Sleep(10 * time.Second)
        return

    } else {
        msj.resp <- respuesta
    }
    break

case <-time.After(MAX_TIME + aprxDur): // Más retraso del permitido
    // Le pasamos el mensaje a otro worker y esperamos 0.250s
    primes.canal <- msj
    time.Sleep(250 * time.Millisecond)
    break
}
```

Esto se hace con una alarma y un select que puede ser llamado antes de que el worker le responda,

Para tratar el crash, se ha decidido enviar la tarea de nuevo por el canal de nuevo antes de realanzar el worker que se ha caído (cláusula “else” de la anterior imágen).

Flexibilidad del Sistema

No se ha conseguido crear un código funcional que consiga este objetivo, pero se ha programado un coordinador de forma parcial que con mínimos cambios sería capaz de conseguir esto.

Primero necesitamos conocer el estado del sistema actual. Esto se ha hecho con un struct llamado Estado.

```
type Estado struct {
    // Info del estado actual del sistema
    actual_thoughput float64 // En milisegundos
    mutex            sync.Mutex
    estadoWorker     [com.POOL]bool
    workersActivos   int

    // Info del usuario para lanzar workers
    user string
    pass string
}
```

Este guarda el throughput actual, un mutex para el acceso a estas variables, el estado de cada worker (en ejecución o no) y cuantos workers hay activos.

Antes de introducir en el canal del Máster, se llamaría con RPC al coordinador, el cual ejecutaría el siguiente código:

```
func (e *Estado) NuevaEntrada(interval com.TPInterval, noReturn *interface{}) error {
    e.mutex.Lock()
    e.actual_thoughput += aproxThr(interval)
    estado := e.checkWorkers()
    e.mutex.Unlock()

    if estado == POCOS_WORKERS {
        e.relanzarWorker()
    } else if estado == MUCHOS_WORKERS {
        e.terminarWorker()
    }
    return nil
}
```

Esta suma el throughput aproximado tras la introducción de una de estas tareas al sistema. Este throughput se chequea en la función checkWorkers, la cual devuelve pocos workers si no hay un worker que puede calcular una tarea de máximo coste sin tenerla que encolar; muchos workers sí se pueden introducir 2 o más tareas de máximo coste sin problema o suficientes worker si estamos en un punto intermedio.
Con tarea de máximo coste nos referimos al intervalo entre 1000 y 70000.
Si hay pocos workers se lanzaría uno y si hay muchos se quitaría uno.

```go
func (e *Estado) checkWorkers() int {
    retVal := SUFIC_WORKERS
    // Si no podemos meter una tarea de máximo coste introducimos un worker
    if e.actual_thoughput+THR_PEOR > float64(e.workersActivos)*THR_PEOR {
        retVal = POCOS_WORKERS

        // Si podemos meter +2 tareas de máximo coste terminamos un worker
    } else if e.actual_thoughput+2*THR_PEOR < float64(e.workersActivos)*THR_PEOR {
        retVal = MUCHOS_WORKERS
    }
    return retVal
}
```

Posteriormente tenemos Nueva salida, que hace lo contrario a nueva entrada y se ejecutaría tras terminar de calcular satisfactoriamente una tarea por parte de cualquier worker. Esta quita el throughput aproximado de la tarea y recalcula el estado del sistema.

```go
func (e *Estado) NuevaSalida(interval com.TPInterval, noReturn *interface{}) error {
    e.mutex.Lock()
    e.actual_thoughput -= aproxThr(interval)
    estado := e.checkWorkers()
    e.mutex.Unlock()

    if estado == POCOS_WORKERS {
        e.relanzarWorker()
    } else if estado == MUCHOS_WORKERS {
        e.terminarWorker()
    }
    return nil
}
```

Si se cae un worker, habría que informar. Esto se haría a través de InformarWorkerCaido, el cual quitaría 1 worker activo del estado actual del sistema y miraría si hace falta volverlo a lanzar con SSH.

```go
func (e *Estado) InformarWorkerCaido(id int, workIniciado *bool) error {
    e.mutex.Lock()
    e.workersActivos--
    e.mutex.Unlock()

    *workIniciado = false
    var err error = nil

    if e.checkWorkers() == POCOS_WORKERS {
        err = e.LanzarWorker(id, workIniciado)
    }

    return err
}
```

Como el Master ya no sabe qué worker está activo o no, este debería pedir acceso a través de una función. Esta le devuelve el estado del worker que se desea.
Para terminar, tenemos las funciones que levantan y eliminan workers de forma dinámica. Estas no tratan de hacer kill o run de los procesos, lo cual sería lo idóneo para evitar el gastos innecesarios de energía o recursos del sistema. En vez solo ponen a true o false las variables que permiten acceder al worker:

```go
func (e *Estado) PedirWorker(id int, accesible *bool) error {
    e.mutex.Lock()
    *accesible = e.estadoWorker[id]
    fmt.Println(e.estadoWorker[id])
    e.mutex.Unlock()
    return nil
}
```

Se ha decidido que el worker que se debería levantar sería el de menor id y se eliminaría el de mayor. Con esto imponemos un sistema de prioridades y podríamos poner las máquinas más potentes las primeras para optimizar el sistema.

```go
// Relanzamos el worker con menor id
func (e *Estado) relanzarWorker() {
    done := false
    e.mutex.Lock()
    for i := 0; i < com.POOL && !done; i++ {
        if !e.estadoWorker[i] {
            e.estadoWorker[i] = true
            done = true
        }
    }
    if done {
        e.workersActivos++
    }
    e.mutex.Unlock()
}
```

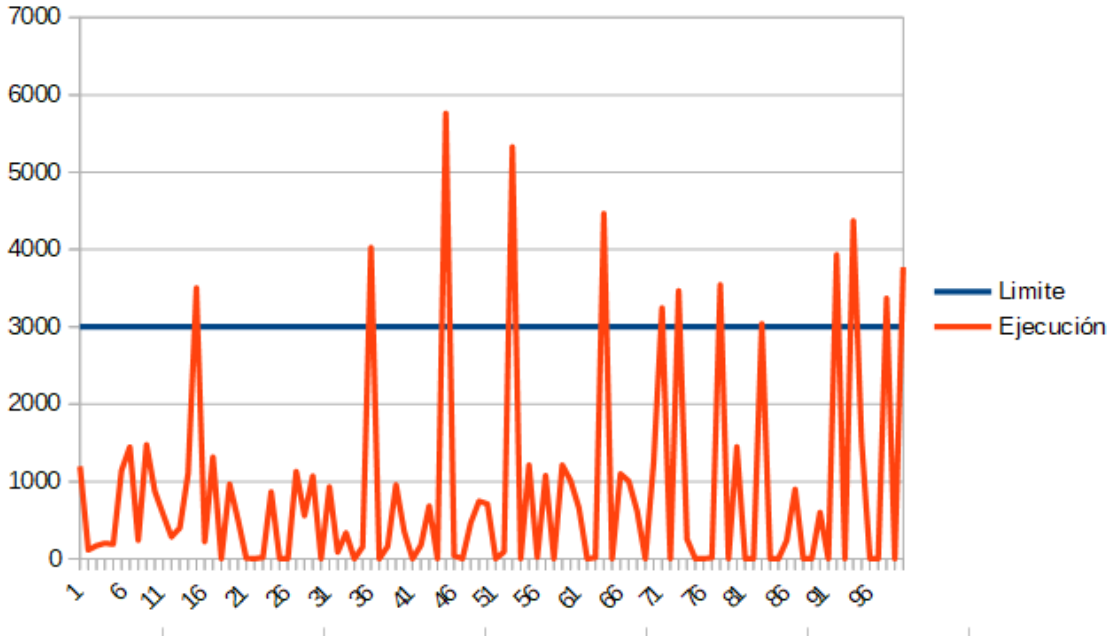
```go
// Terminamos el worker con mayor id
func (e *Estado) terminarWorker() {
    done := false
    e.mutex.Lock()
    for i := com.POOL - 1; i >= 0 || done; i-- {
        if e.estadoWorker[i] {
            fmt.Printf("WORKER %d TERMINADO\n", i)
            e.estadoWorker[i] = false
            done = true
        }
    }
    if done {
        e.workersActivos--
    }
    e.mutex.Unlock()
}
```

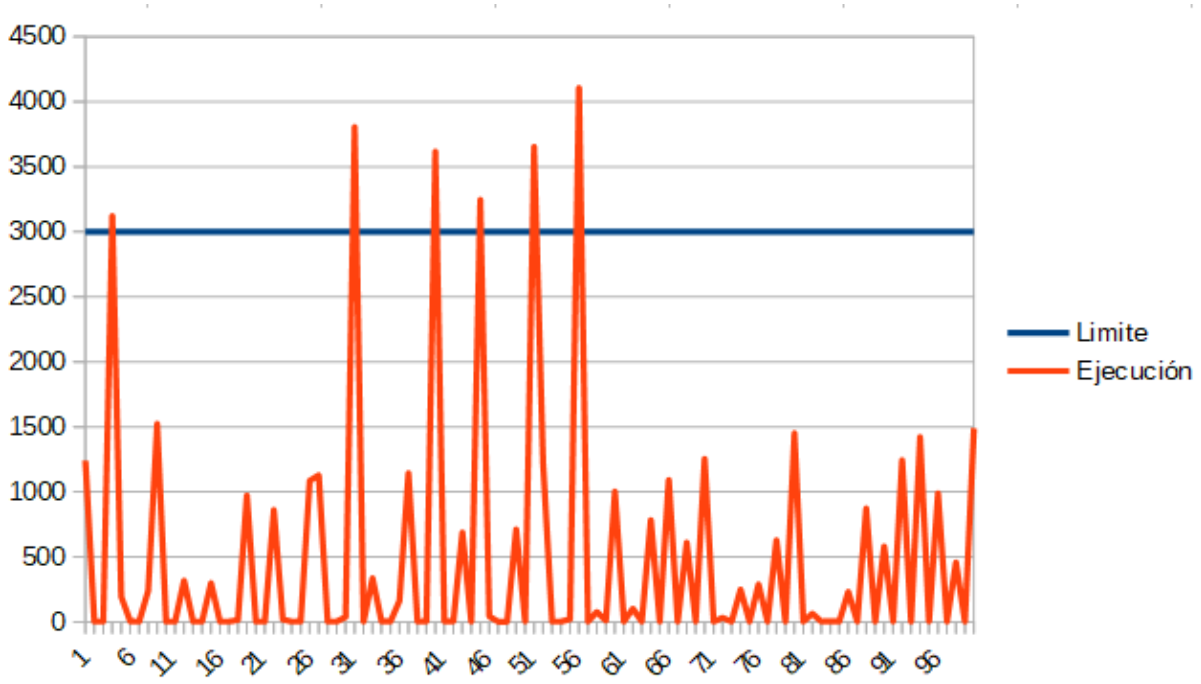
Resultados

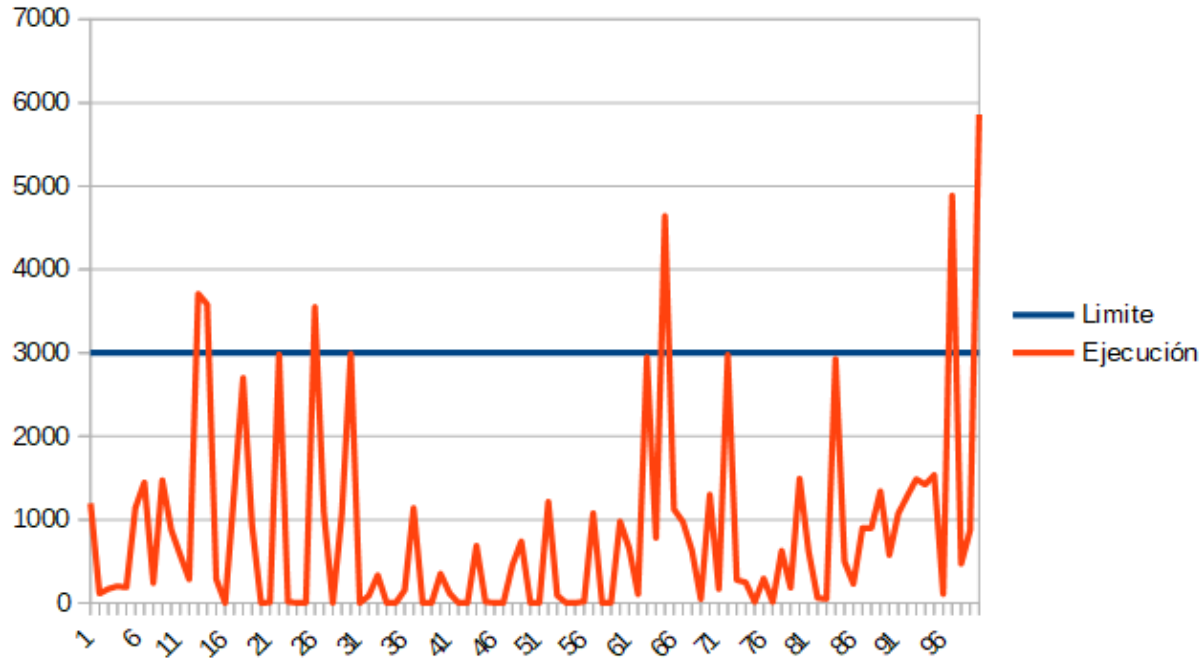
Se ha ejecutado 3 veces el código del cliente y se han obtenido 3 conjuntos de datos los cuales se han graficado para una mayor visibilidad.

En todas las pruebas se han usado solo 4 workers. Si se hubiesen usado más, los resultados probablemente serían mejores, pero como cada vez que se hace una llamada RPC se crea una gorutina y solo tenemos 6 núcleos, no queríamos poner el cuello de botella en la comunicación en vez de en el cálculo.

Con esto, nosotros consideramos que tenemos un sistema decente si se rompe el QoL menos veces que las ocurrencias de Crash anteriormente medidas. Esto se debe a que cuando un worker Crashea u Omite, esta máquina está inutilizable durante 10 segundos. Si esto ocurre cuando varias de las otras máquinas entran en problemas, el coste del cálculo de la siguiente tarea será bastante elevado.







El QoL se rompe 13, 6 y 6 veces respectivamente. Los tiempos que están rozando la línea tienen entre 2900 y 2990 ms de coste. La primera ejecución, ya que incluso los tiempos que rozan el límite son mayores a 3 segundos.

Conclusiones

Con más máquinas y más pruebas se podría haber obtenido incluso una menor cantidad de picos en las gráficas. Aún así, que 6 de cada 100 ejecuciones superen el QoL, de las cuales la mitad es menos de un segundo, se puede considerar viable para un sistema tan inconsistente.