Sistemas Distribuidos

Práctica 1:

Introducción a Sistemas Distribuidos

Memoria

Grupo 4-8

Cristian Andrei Selivanov Dobrisan (816456@unizar.es)
Dorian Boleslaw Wozniak (817570@unizar.es)

Índice

1. Descripción del problema

El objetivo de esta práctica es crear una serie de sistemas distribuidos que resuelvan un problema sencillo, sin problemas de sincronización distribuida ni necesidad de acceso a sección crítica de forma distribuida, siguiendo los patrones de diseño conocidos.

El problema a resolver es el siguiente: un cliente solicita al sistema una lista con todos los números primos comprendidos en un intervalo (específicamente, en el rango [1000, 7000]). El sistema ya tiene implementadas las funciones para calcular dicho rango, por lo que el objetivo será implementar la comunicación entre sistemas implementando las siguientes cuatro arquitecturas:

- Cliente-servidor secuencial: donde el servidor trata las peticiones una a una conforme van llegando.
- Cliente-servidor concurrente: con un proceso de tratamiento por petición.
- Cliente-servidor concurrente con un *pool* fijo de procesos de tratamiento de peticiones.
- Master-worker: donde un proceso *master* actúa de coordinador entre el cliente y una serie de procesos *worker* situados en otras máquinas que realizan el trabajo solicitado.

Para probar dichos sistemas se dispone de las máquinas del laboratorio L1.02 de la universidad, sobre las cuales se aprovecha la capacidad de comprobar los equipos encendidos y arrancarlos remotamente a través del servidor *central*, la ejecución remota de ficheros mediante ssh, y el sistema NFS disponible en toda la red de prácticas de la EINA para compartir los binarios y *shell scripts* generados para la realización de la práctica. Se detallarán más adelante las decisiones específicas tomadas para el despliegue de los sistemas.

Por último, se ha utilizado para la realización de la práctica el lenguaje de programación Go, el cual tiene integrado la ejecución asíncrona como parte básica del lenguaje mediante las denominadas “*Gorutinas”*, hilos ligeros que se ejecutan a nivel de usuario diseñados para ser rápidos y fiables. La comunicación entre procesos asíncronos se realiza mediante “*channels”* (FIFO).

2. Diseño de las arquitecturas

A continuación se describen el diseño de cada una de las cuatro arquitecturas descritas anteriormente:

Cliente-servidor secuencial

El funcionamiento de este tipo de servidor es el siguiente: el servidor comienza a escuchar en una dirección y puerto dados. El servidor entra en un bucle donde espera a aceptar una conexión.

```
listener, err := net.Listen(CONN_TYPE, endpoint)

for {
   conn, err := listener.Accept()
   handleRequest(conn)
}
```

Una vez acepta una conexión, realiza el paso de mensajes y resuelve el problema pedido por el cliente. Una vez cerrada la conexión con el cliente tras cumplir su objetivo, acepta la conexión del siguiente cliente.

```
func handleRequest(conn net.Conn) {
   defer conn.Close()

   // Codificadores/decodificadores
   encoder := gob.NewEncoder(conn)
   decoder := gob.NewDecoder(conn)

   // Lee petición
   var request com.Request
   err := decoder.Decode(&request)

   // Lanza tarea
   primes := findPrimes(request.Interval)

   // Envía respuesta
   reply := com.Reply{Id: request.Id, Primes: primes}
   encoder.Encode(reply)
}
```

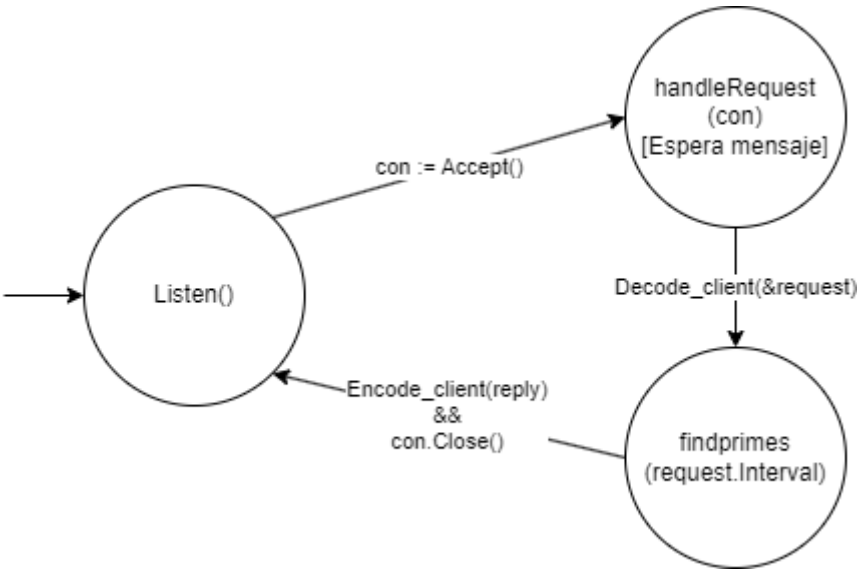

Fig. 1: Diagrama de estados de la arquitectura cliente-servidor secuencial

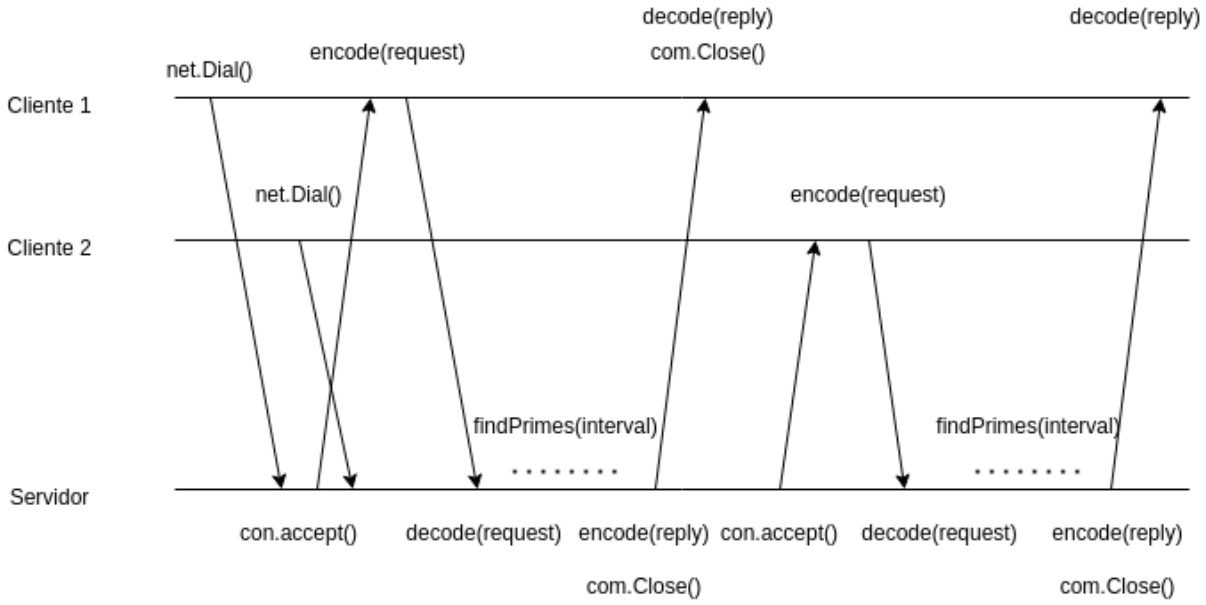

Fig. 2: Diagrama de secuencia de la arquitectura cliente-servidor secuencial

Cliente-servidor concurrente

Las desventajas del modelo anterior son evidentes: se malgasta la capacidad computacional del servidor, que no está asignando todos los recursos disponibles, y hace que aumente el *overhead* significativamente incluso atendiendo pocas solicitudes, al tener que esperar a que el servidor procese las solicitudes una a una.

Para solucionar el problema, se ha incluido una sola modificación al sistema anterior: cada vez que se acepta una solicitud, se lanza una nueva *Gorutina* de la función *handleRequest()*, tal que el bucle de aceptación no sea bloqueante y, al ser independiente una solicitud de las otras, poder procesarlas de forma concurrente.

```
for {
    conn, err := listener.Accept()
    go handleRequest(conn)
}
```

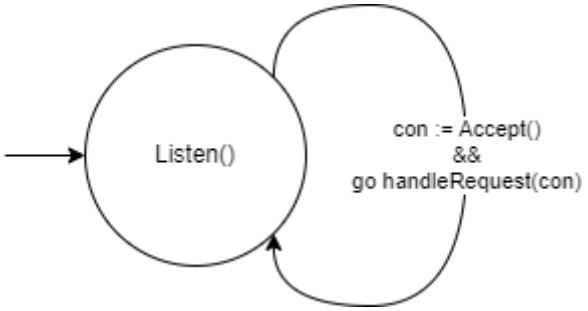

Fig. 3: Diagrama de estados del bucle principal de la arquitectura cliente-servidor concurrente

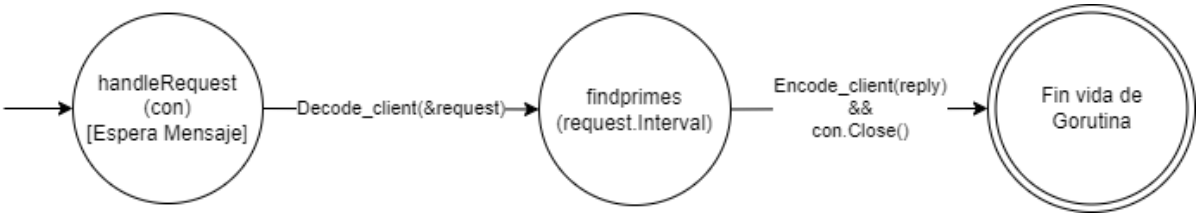

Fig. 4: Diagrama de estados del proceso asíncrono handleRequest() de la arquitectura cliente-servidor concurrente

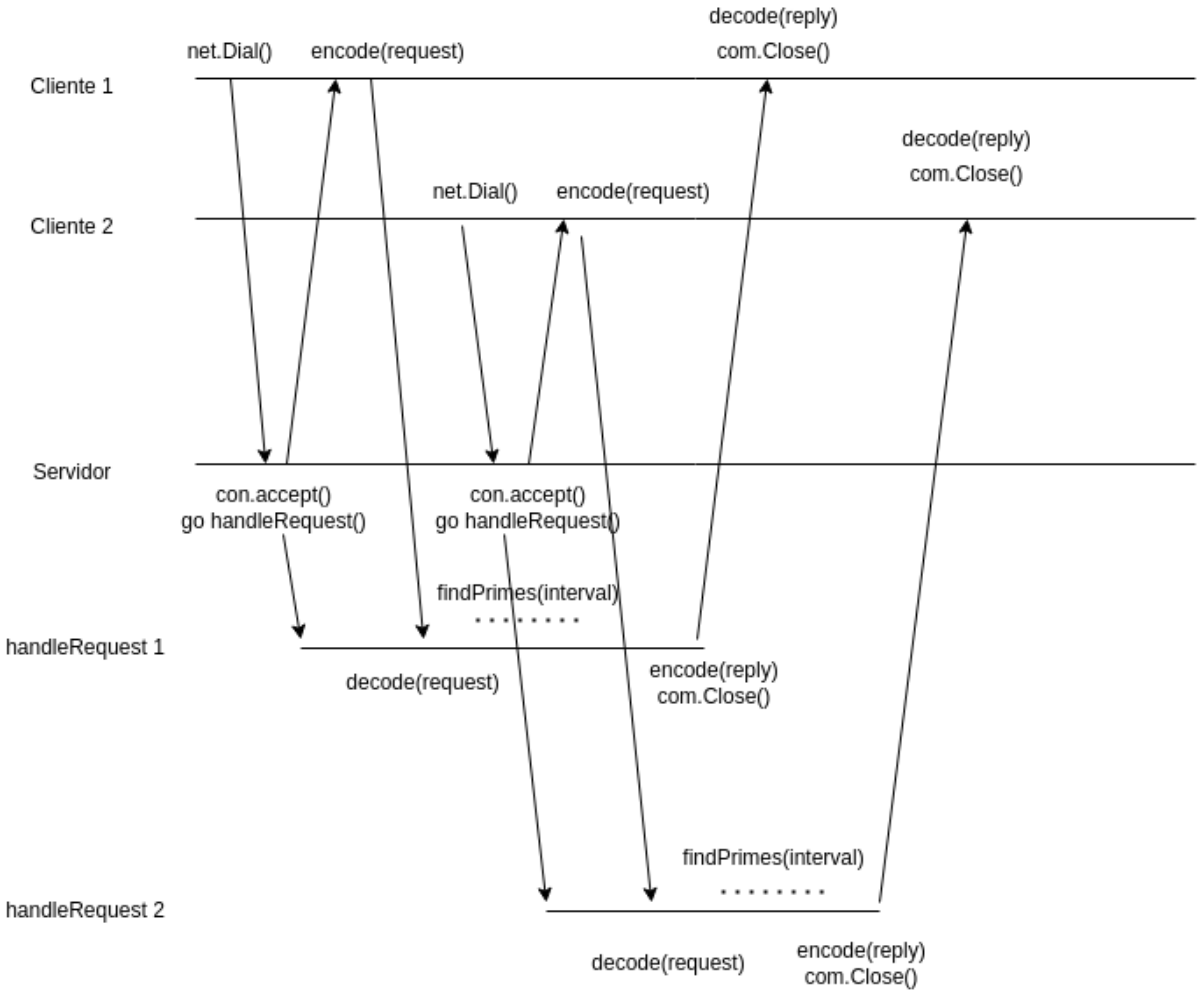

Fig. 5: Diagrama de secuencia de la arquitectura cliente-servidor concurrente

Cliente-servidor concurrente con pool fijo

La solución puramente concurrente no es la ideal: no en todas las situaciones es conveniente destinar todos los recursos de una máquina para un solo objetivo, y aunque las *Gorutinas* son suficientemente ligeras como para poder tener miles de ellas simultáneamente, podría acabar teniendo demasiadas solicitudes para poder mantener un nivel de respuesta aceptable entre clientes y el servidor.

Una posibilidad es limitar el número de procesos *handleRequest()* presentes en el sistema a un número fijo. Para ello, se lanzan primero dichas rutinas y luego, cada vez que se acepta una conexión, se añade a *ch*, un canal (cola) de conexiones por tratar. Los procesos *handleRequest()*, por su parte, se mantienen en un bucle donde primero esperan a que haya una conexión nueva disponible en el canal. Si un proceso consigue obtener una nueva conexión de *ch*, realiza la tarea y una vez que cierra la conexión con el cliente anterior trata de obtener la siguiente conexión.

```
    // Lanza pool de gorutinas
    ch := make(chan net.Conn)
    for i := 0; i < 6; i++ {
        go handleRequest(i, ch)
    }

    for {
        conn, err := listener.Accept()
        ch <- conn
    }
```

```
func handleRequest(i int, ch chan net.Conn) {
    for {
        // Obtiene siguiente tarea
        conn := <-ch

        // Codificadores/decodificadores
        encoder := gob.NewEncoder(conn)
        decoder := gob.NewDecoder(conn)

        // Lee petición
        var request com.Request
        err := decoder.Decode(&request)

        // Lanza tarea
        primes := findPrimes(request.Interval)

        // Envía respuesta
        reply := com.Reply{Id: request.Id, Primes: primes}
        encoder.Encode(reply)
```

```
        conn.Close()
    }
}
```

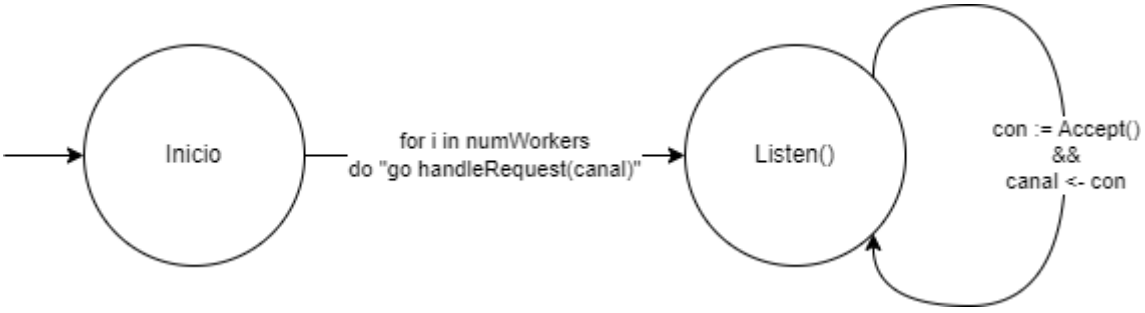

Fig. 6: Diagrama de estados del bucle principal de la arquitectura cliente-servidor concurrente con un pool fijo de procesos

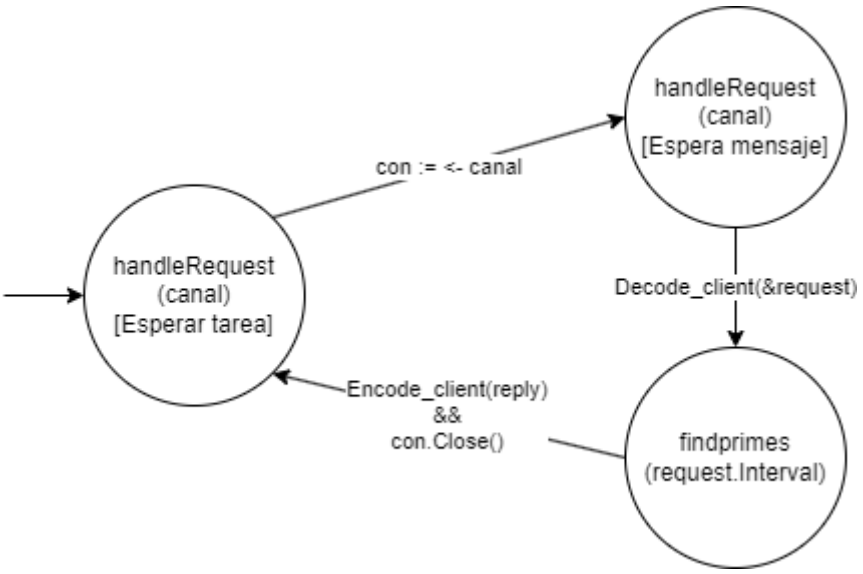

Fig. 7: Diagrama de estados del bucle de Gorutina handleRequest() de la arquitectura cliente-servidor concurrente con un pool fijo de procesos

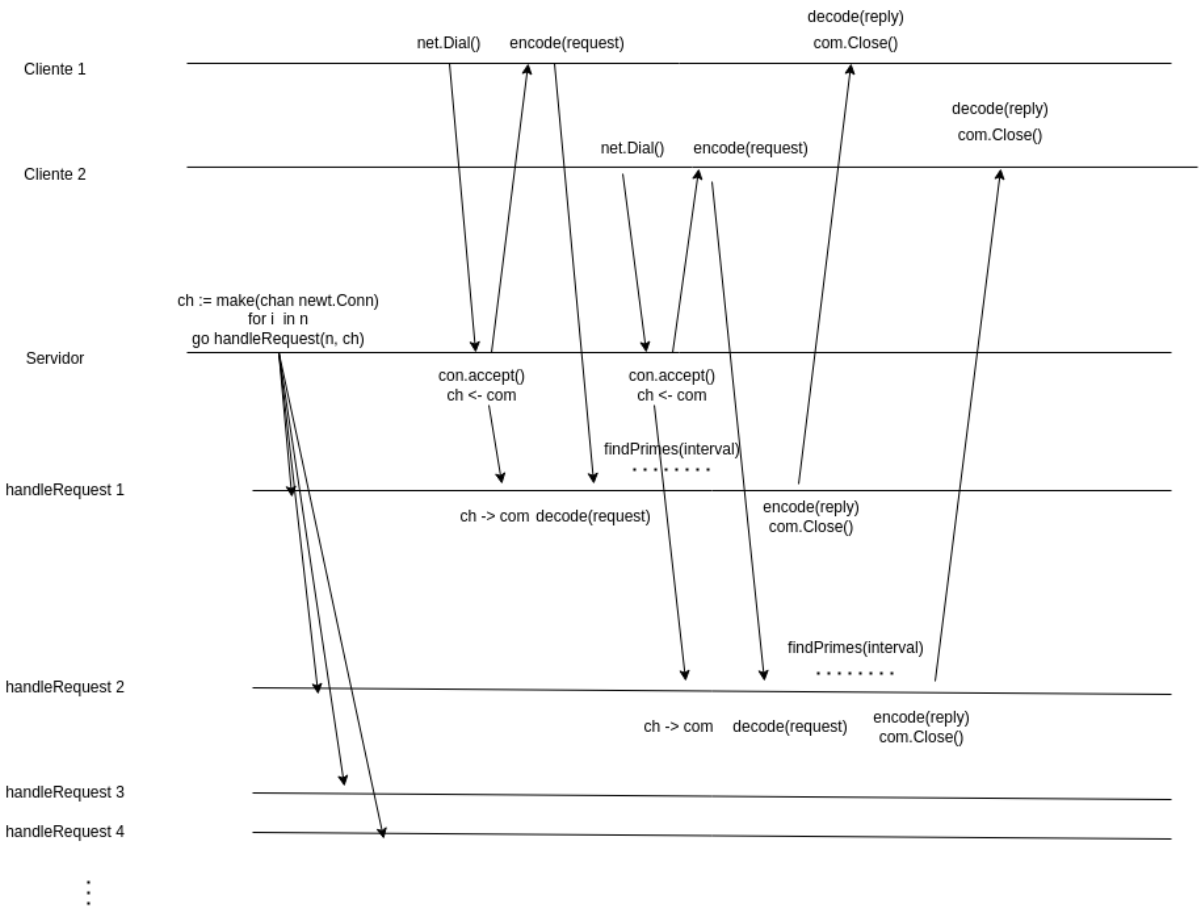

Fig. 8: Diagrama de secuencia de la arquitectura cliente-servidor concurrente con un pool fijo de procesos

Master-worker

En contraste a las las arquitecturas anteriormente descritas, en esta el proceso computacional para cumplir la solicitud de los clientes no se va a producir en la máquina con la que contactan los clientes, sino en una red de máquinas interconectadas a la máquina *master*, que actúa como coordinador entre el cliente y los procesos *worker*.

El proceso complica sustancialmente la tarea a realizar, pues ahora el *master* debe lanzar remotamente los procesos *worker* en otras máquinas interconectadas, y luego gestionar el paso de mensajes no solo entre el cliente, sino entre cada proceso *worker*: el cliente envía una petición, el *master* recibe la petición y se la pasa a un proceso que no tiene tareas pendientes, reenvía la petición al *worker* asociado (hay tantos procesos *handleRequest()* como *workers*), y reenvía la respuesta de su *worker* al cliente original.

```
func handleRequest(i int, endpoint string, ch chan net.Conn) {
   // Inicia conexion con worker
   connW, err := net.Dial(CONN_TYPE, endpoint)

   encoderW := gob.NewEncoder(connW)
   decoderW := gob.NewDecoder(connW)

   for {
       // Obtiene siguiente tarea
       conn := <-ch

       // Codificadores/decodificadores
       encoder := gob.NewEncoder(conn)
       decoder := gob.NewDecoder(conn)

       // Lee petición
       var request com.Request
       err = decoder.Decode(&request)

       // Reenvía petición a worker
       encoderW.Encode(request)

       // Espera respuesta del worker
       var reply com.Reply
       err = decoderW.Decode(&reply)

       // Reenvía respuesta
       encoder.Encode(reply)
       conn.Close()
   }
}
```

Para conectar con los *workers*, primero necesitará una lista de workers disponibles para el sistema. Estos se encuentran en un fichero llamado *workers*. El fichero tendrá el siguiente formato:

```
155.210.154.192:29310
155.210.154.193:29310
155.210.154.194:29310
155.210.154.195:29310
...
ip:puerto
```

Para arrancar los ficheros, se ha optado por implementar un *shell script* que arranque una serie de procesos ssh de forma concurrente (utilizando el comando junto a &) leyendo del mismo archivo «*workers*», que será ejecutado por el *master* al arrancar el sistema.

```
#!/bin/bash

while read endpoint || [ -n "$endpoint" ]
do
   ssh -n $(echo $endpoint | cut -d ':' -f 1)
"./$HOME/pratica1/server-worker $endpoint" &
done < "$HOME/practica1/workers"
```

Al lanzarse el proceso *lanzar_master_worker.sh* se utiliza el método Start() en vez del método Run() pues los procesos que ejecutan los *workers* remotos no acaban. También se espera durante un segundo antes de lanzar las rutinas para contactar con los workers para asegurarse que están en ejecución al salir de la espera (puesto que la puesta en marcha de los procesos remotos puede tardar).

```
   // Ejecuta script para lanzar procesos remotos
   err = exec.Command(".././lanzar_master_worker.sh").Start()
   time.Sleep(time.Duration(1000) * time.Millisecond)

   // Obtiene lista de workers con los que conectar
   f, err := os.Open("../workers")
   FScan := bufio.NewScanner(f)
   FScan.Split(bufio.ScanLines)
   var lines []string
   for FScan.Scan() {
       lines = append(lines, FScan.Text())
   }
   f.Close()

   // Lanza gorutinas
   ch := make(chan net.Conn)
```

```
    i := 1
    for _, endpoint := range lines {
        go handleRequest(i, endpoint, ch)
        i++
    }

    for {
        conn, err := listener.Accept()
        ch <- conn
    }
```

Por otro lado, el worker es similar al servidor secuencial anteriormente descrito, pero con la diferencia de que solo acepta una vez conexión y el intercambio de mensajes con el master se produce en un bucle en *handleRequest()*

```
func handleRequest(conn net.Conn) {
    // Codificadores/decodificadores
    encoder := gob.NewEncoder(conn)
    decoder := gob.NewDecoder(conn)

    for {
        // Lee petición
        var request com.Request
        err := decoder.Decode(&request)

        // Lanza tarea
        primes := findPrimes(request.Interval)

        // Envía respuesta
        reply := com.Reply{Id: request.Id, Primes: primes}
        encoder.Encode(reply)
    }
}

func main() {
    listener, err := net.Listen(CONN_TYPE, os.Args[1])
    conn, err := listener.Accept()
    handleRequest(conn)
}
```

Es importante destacar dos cuestiones importantes:

- Las conexiones ssh se establecen mediante autenticación con clave pública-privada. Utilizando *ssh-keygen* y *ssh-copy-id*, se ha creado un par de claves pública-privada y se ha instalado la clave pública en los equipos utilizados. Este proceso se tiene que realizar solo una vez desde cualquier equipo de la red de ordenadores y servidores

de prácticas de la universidad, siendo el resultado el acceso sin contraseña desde un equipo del laboratorio a cualquier otro equipo del laboratorio sin necesidad de contraseña. Se ha realizado un proceso similar para las máquinas personales de los integrantes y los dispositivos de la red de la Universidad.

- **La ejecución correcta de la práctica es dependiente de dónde se encuentran los ficheros requeridos.** Esto es, al ejecutar el *shell script*, espera que los binarios para los workers se encuentren en *$HOME/practica1/bin/server-worker*. Así mismo, el binario server-master se espera que *lanzar_master_worker.sh* esté en el directorio padre de *bin*, por lo que se debe ejecutar el comando desde dicho directorio, no mediante *./bin/server-master*

La estructura del proyecto entregado es el siguiente:

```
practica1
| - bin
|     | - server-secuencial
|     | - server-concurrente
|     | - server-pool
|     | - server-master
|     | - server-worker
|
| - lanzar_master_worker.sh
| - workers
```

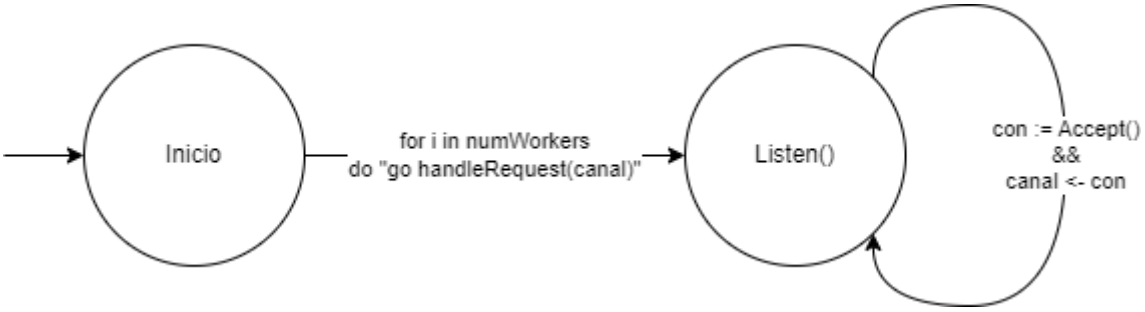

Fig. 9: Diagrama de estados del bucle principal de la arquitectura master-worker

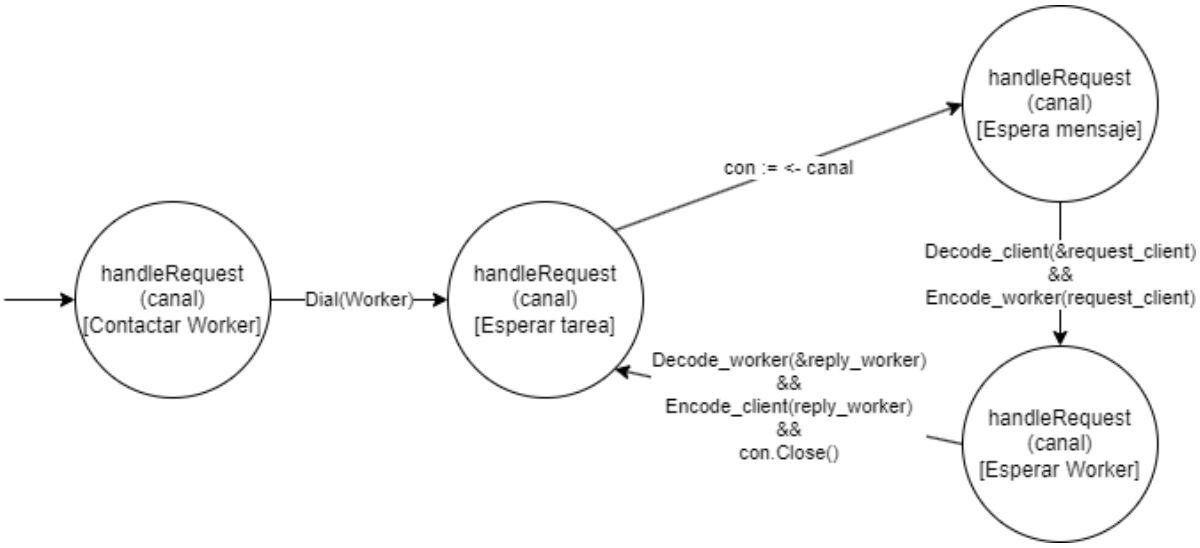

Fig. 10: Diagrama de estados del proceso handleRequest() de la arquitectura master-worker

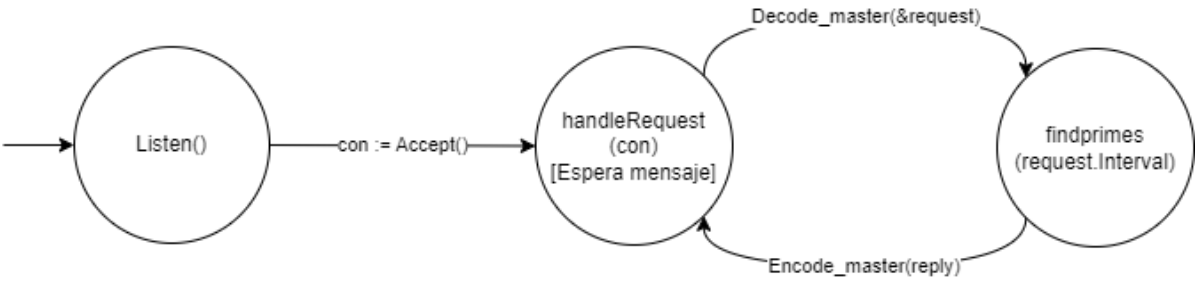

Fig. 11: Diagrama de estados del proceso worker de la arquitectura master-worker

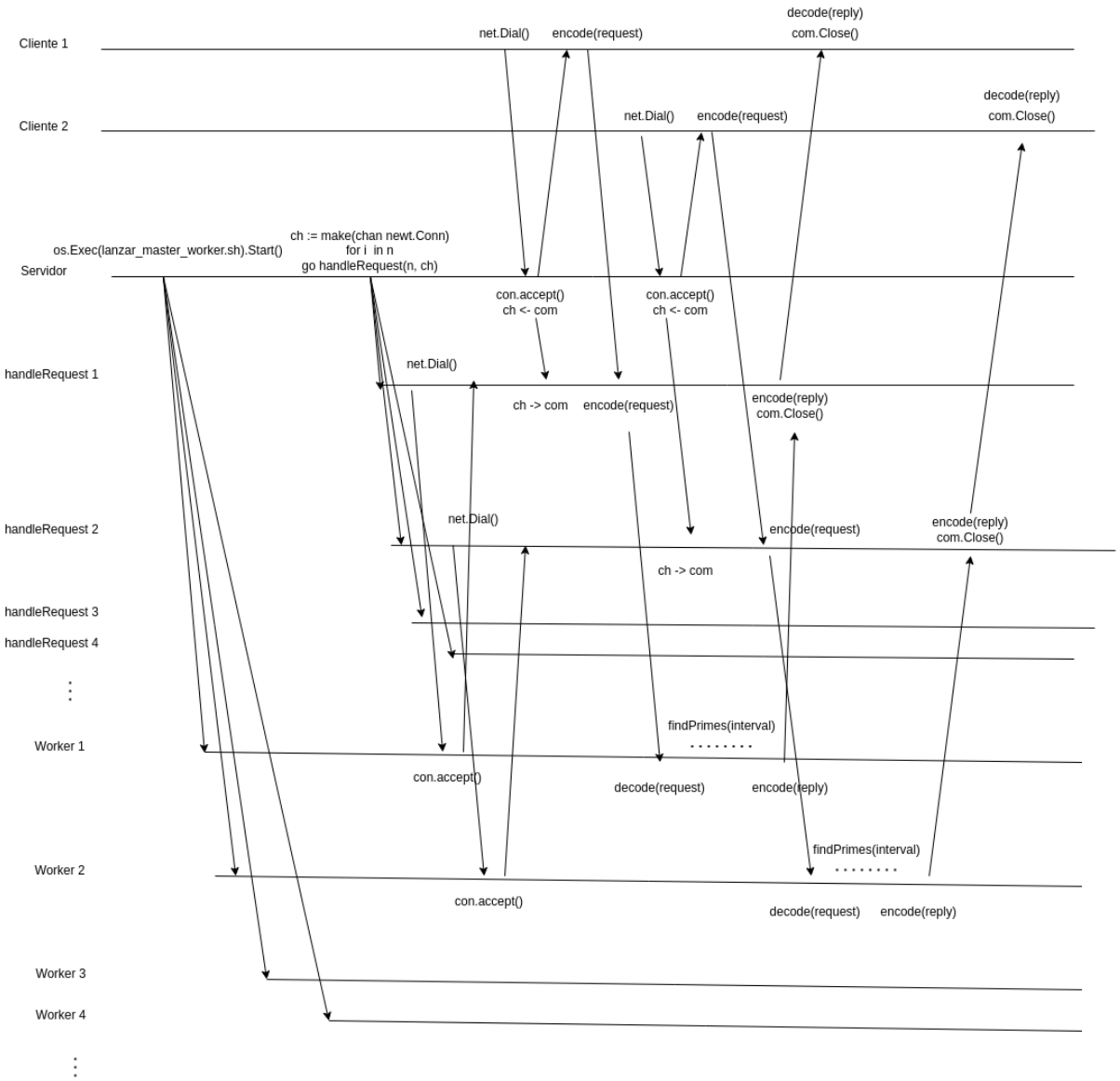

Fig. 12: Diagrama de secuencia de la arquitectura master-worker

3. Análisis de carga de trabajo teórica máxima

Dada la métrica "*QoS (Quality Of Service)"*, utilizada para determinar las cargas de trabajo máximas de un sistema distribuido:

$$QoS: (t_{EX} + t_{CON} + t_{OH}) \leq 2 * t_{EX}$$

Se pretende calcular las peticiones por segundo / cargas máximas teóricas, de cada una de las arquitecturas diseñadas, teniendo en cuenta los siguientes parámetros:

$$t_{EX} \approx 20\,ms \qquad t_{CON} \approx 0\,ms \qquad t_{OH} \approx 0\,ms\ (en\ casos\ ideales)$$

Donde *"t_{EX}"* se obtendría a partir de realizar la media con los tiempos resultantes al ejecutar el algoritmo de búsqueda de números primos en los intervalos [1000, 7000] *(dados por el cliente entregado con el enunciado)*. Por un lado *"t_{CON}"* se considerará insignificante ya que los equipos se comunican entre sí a través de una red local con tiempos muy bajos (A excepción del caso master-worker). Por otro lado, el *"Overhead"* / *"t_{OH}"* por lo general se considerará insignificante si estamos en casos ideales, salvo casos puntuales comentados en las situaciones específicas.

En el caso de la arquitectura **secuencial**:

$$peticiones\ por\ segundo = 1000\,ms\ /\ (20\,ms + 0\,ms + 0\,ms) = 50$$

Habiendo como mucho solo un proceso simultáneo, se procesarán 50 peticiones por segundo en el mejor de los casos. Este caso ideal empeora a medida que vayan aumentando el número de clientes, solicitando más peticiones que se acumulan, incumpliendo el *"QoS"*, ya que los tiempos de respuesta por parte del servidor aumentan de forma lineal:

$$t_{OH} \approx (t_{EX} * N_{clientes} - 1)\,ms$$

$$QoS:\ 20\,ms + 0\,ms + (20 * N_{clientes} - 1)\,ms \geq 2 * 20\,ms,\ si\ N_{clientes} \geq 2$$

$$peticiones\ por\ segundo = 1000\,ms\ /\ (20\,ms + 0\,ms + (20 * N_{clientes} - 1)\,ms) = 50$$

Se puede observar que para más de dos clientes la arquitectura incumple el *"QoS"*.

En el caso de la arquitectura **concurrente**:

$$peticiones\ por\ segundo\ =\ (1000\ ms\ /\ (20\ ms\ +\ 0\ ms\ +\ t_{OH}))\ *\ N_{workers}$$

Donde se permitirá tener tantas Gorutinas simultáneas como se pueda mientras el *"Overhead"* no sea significativo, aumentando a medida que más clientes soliciten el servicio, creando más Gorutinas en el proceso, estando limitado por el hardware del servidor en cuestión. Básicamente cuando el *"Overhead"* sea significativo, se dejará de cumplir *"QoS"*.

En el caso de la arquitectura **pool de workers**:

$$peticiones\ por\ segundo\ =\ (1000\ ms\ /\ (20\ ms\ +\ 0\ ms\ +\ t_{OH}))\ *\ N_{pool}$$

Depende del tamaño dado para el pool de workers encargado de atender las peticiones provenientes del cliente por parte del servidor, el *"Overhead"* aumenta si las peticiones superan el número de workers de los que se dispone, ya que se provocan esperas y por tanto, dejará de ser insignificante, llegando a un momento en el que se incumple *"QoS"*.

En el caso de la arquitectura **master-worker**:

$$peticiones\ por\ segundo\ =\ (1000\ ms\ /\ (20\ ms\ +\ t_{CON}\ +\ t_{OH}))\ *\ N_{workers}$$

Se puede apreciar, que sería similar al concurrente / pool de workers, con la diferencia de que el tiempo de conexión "t_{CON}" se duplicaría, ya que ahora el servidor se ha de comunicar con los distintos equipos de los workers.

4. Experimentos de carga máxima práctica

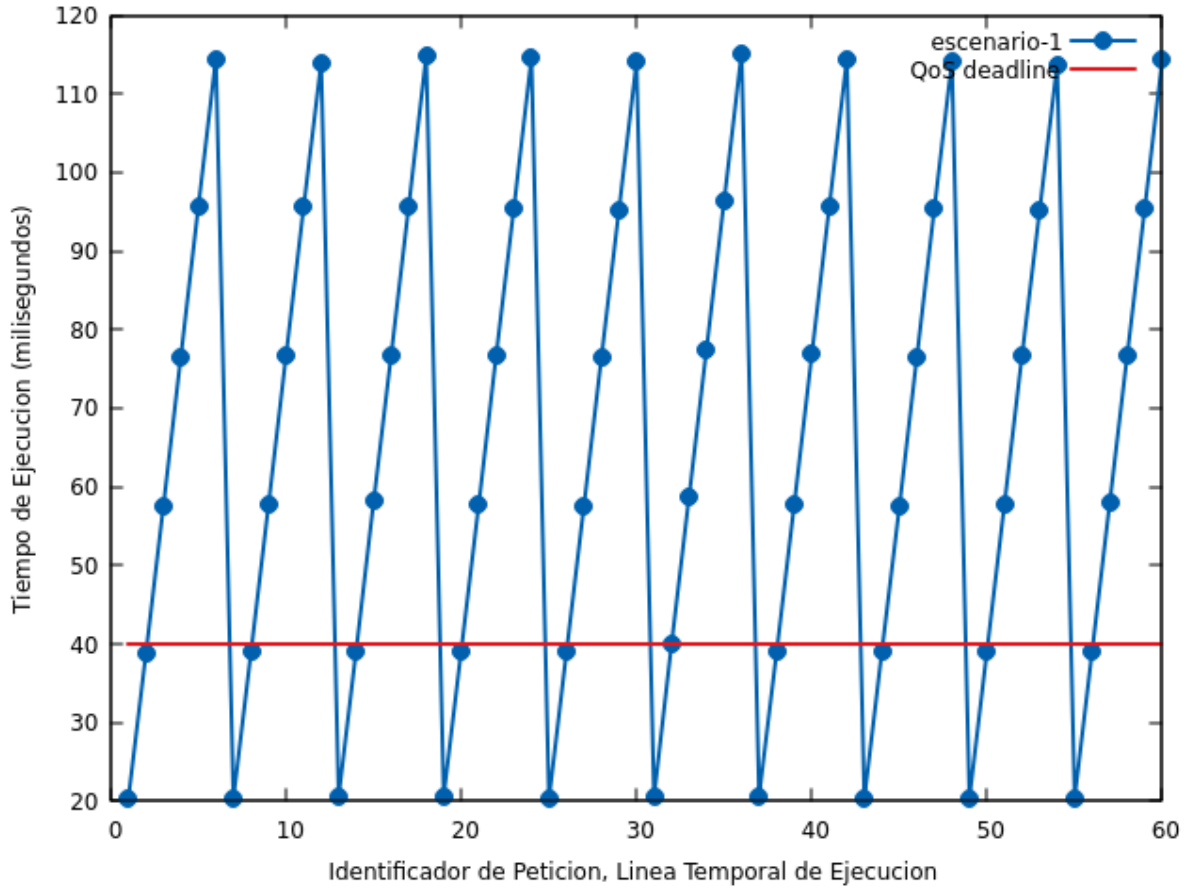

Fig. 13: Gráfica de tiempo de ejecución de 60 peticiones de un cliente en ráfagas de 6 para una arquitectura cliente-servidor secuencial

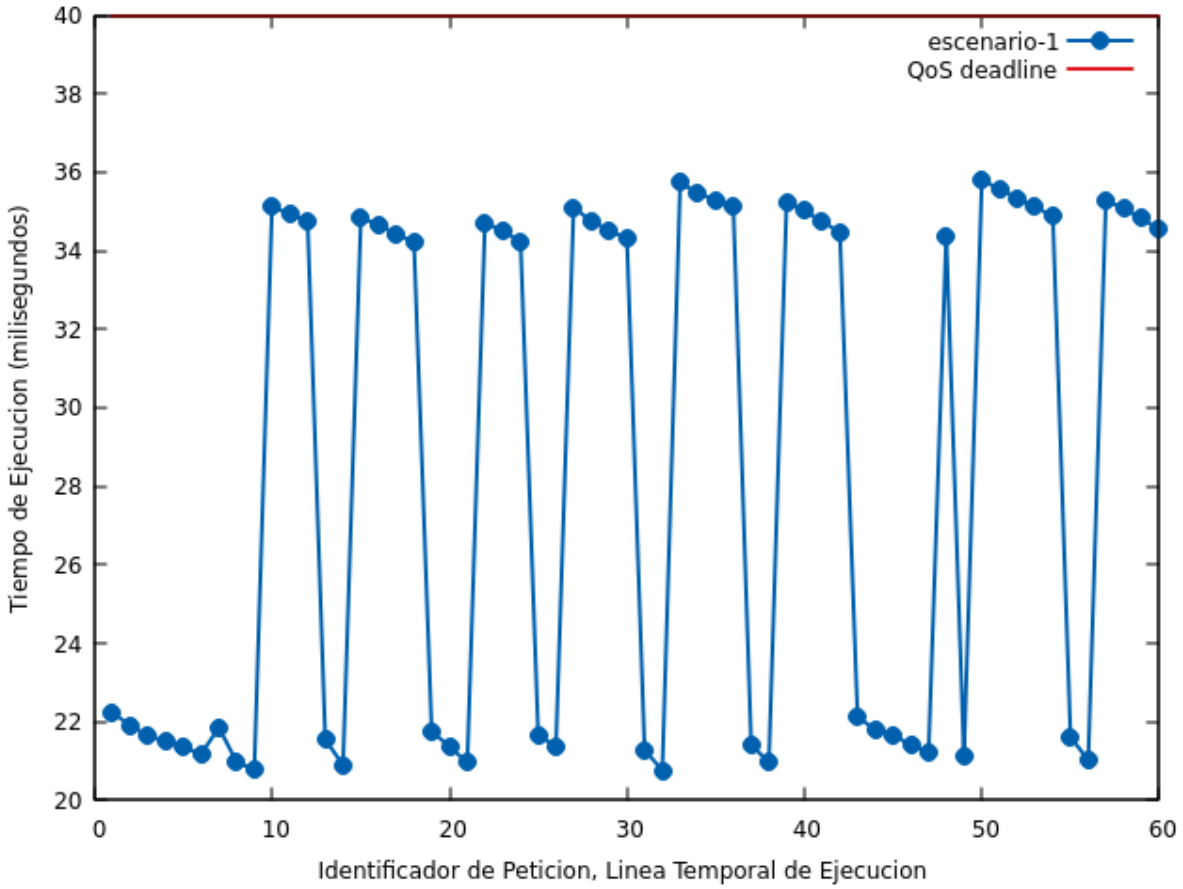

Fig. 14: Gráfica de tiempo de ejecución de 60 peticiones de un cliente en ráfagas de 6 de una arquitectura cliente-servidor concurrente

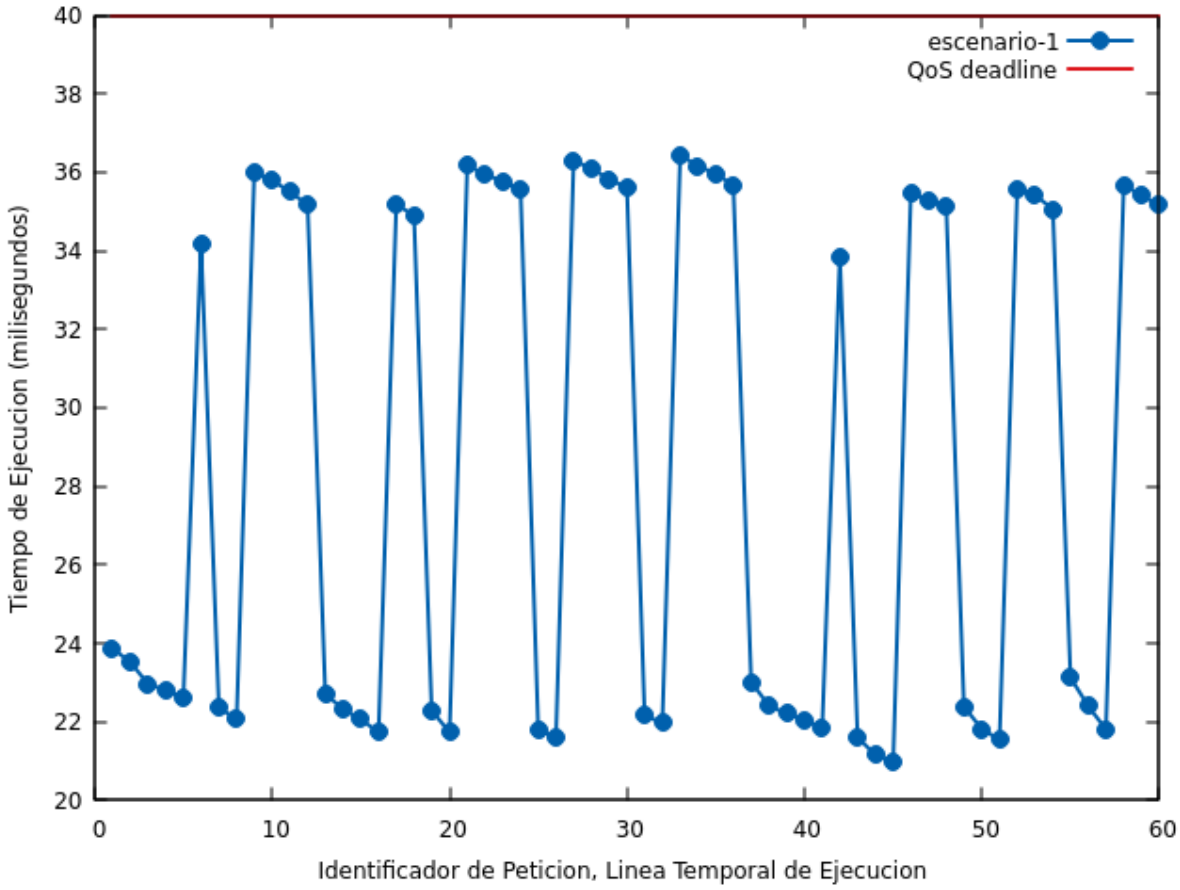

Fig. 15: Gráfica de tiempo de ejecución de 60 peticiones de un cliente en ráfagas de 6 de una arquitectura cliente-servidor concurrente con un pool fijo de 6 procesos

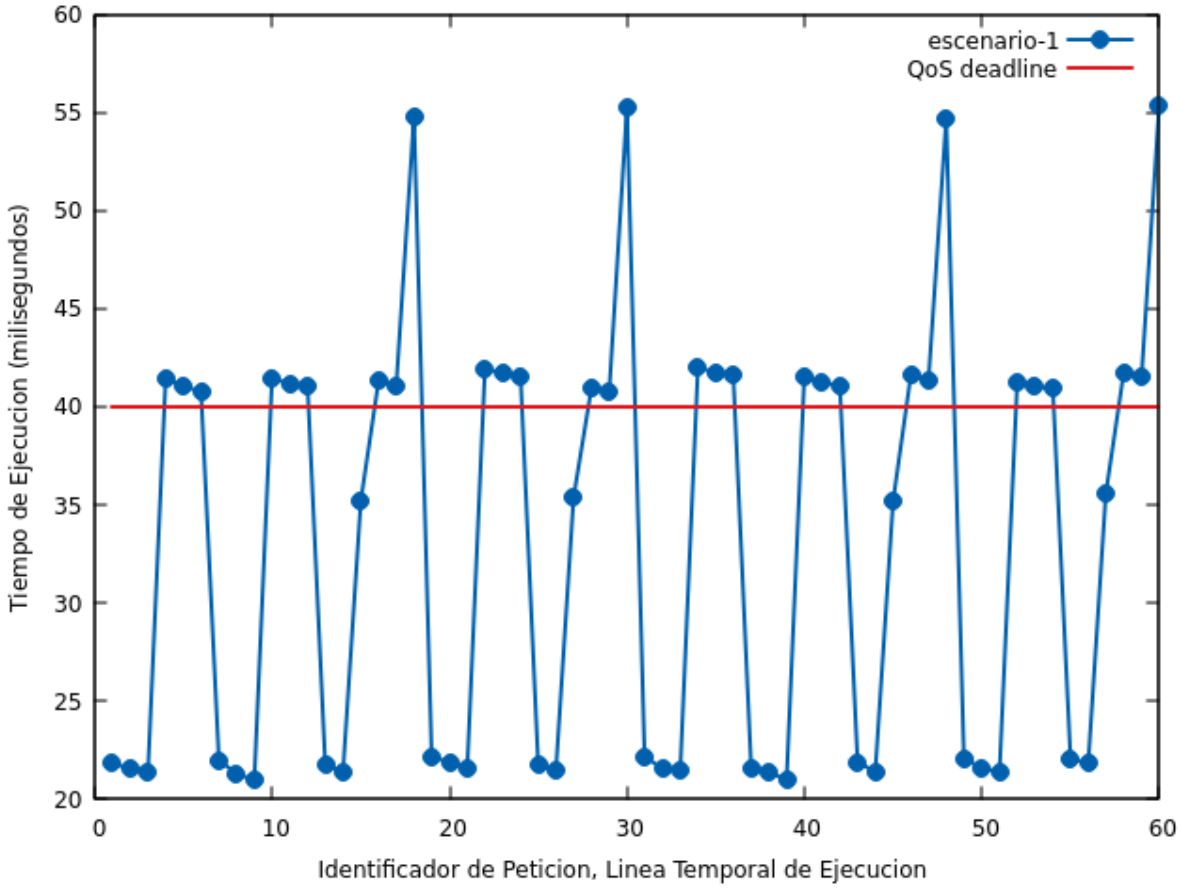

Fig. 16: Gráfica de tiempo de ejecución de 60 peticiones de un cliente en ráfagas de 6 de una arquitectura cliente-servidor concurrente con un pool fijo de 3 procesos

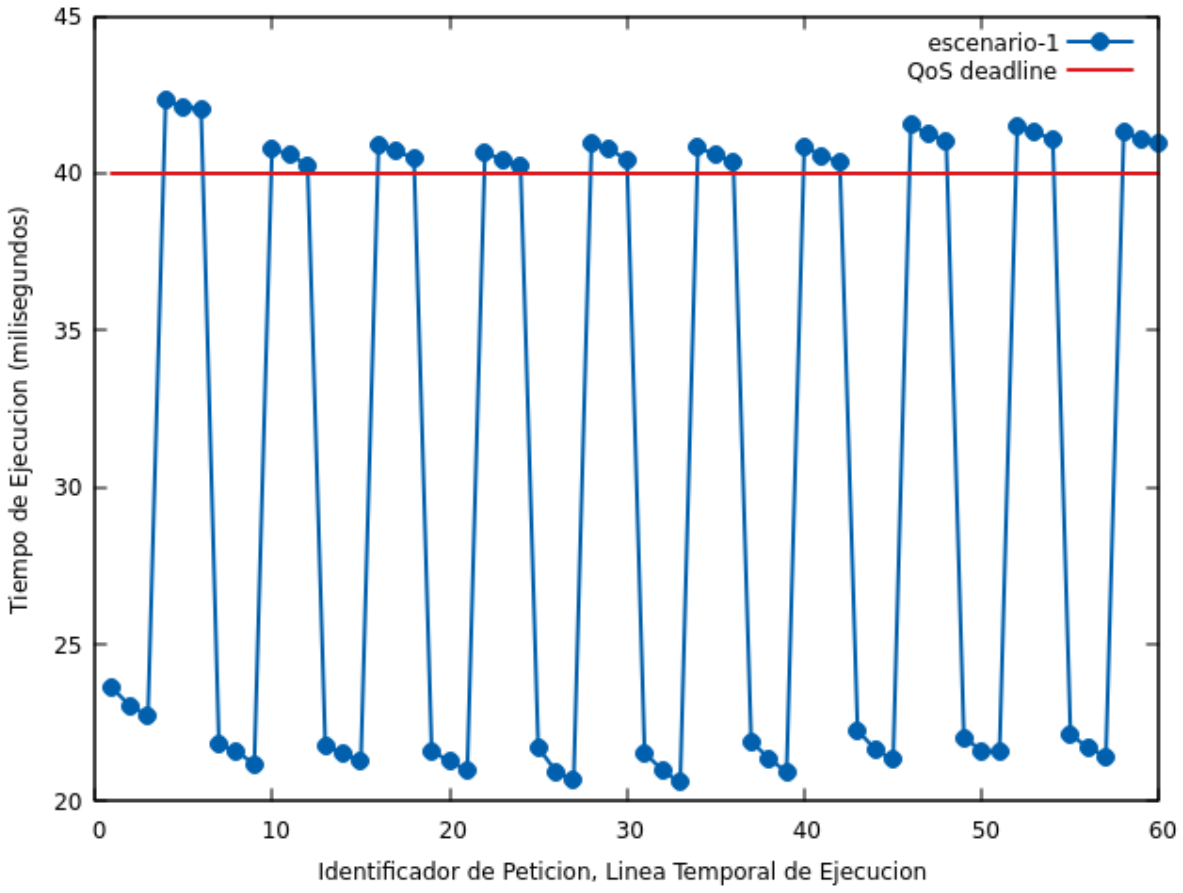

Fig. 17: Gráfica de tiempo de ejecución de 60 peticiones de un cliente en ráfagas de 6 de una arquitectura master-worker con 3 workers asociados

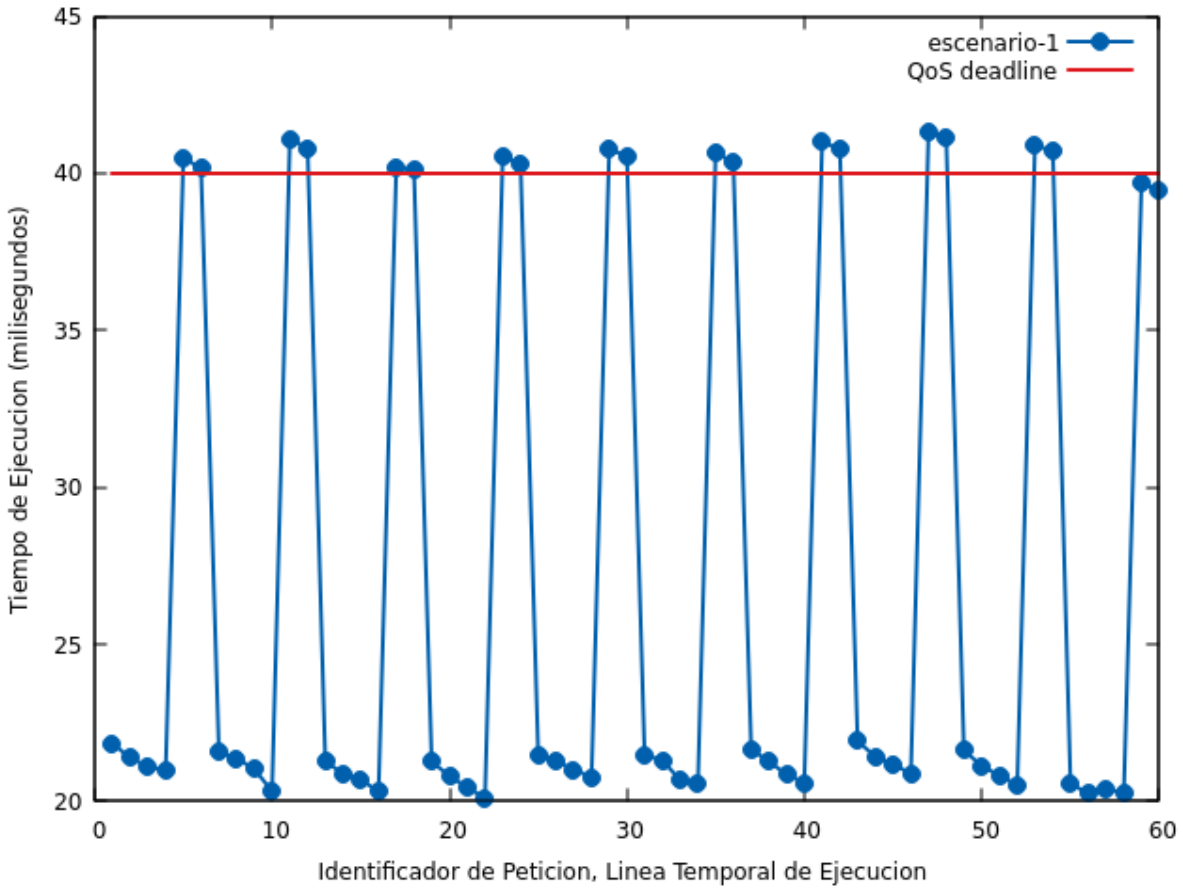

Fig. 17: Gráfica de tiempo de ejecución de 60 peticiones de un cliente en ráfagas de 6 de una arquitectura master-worker con 4 workers asociados

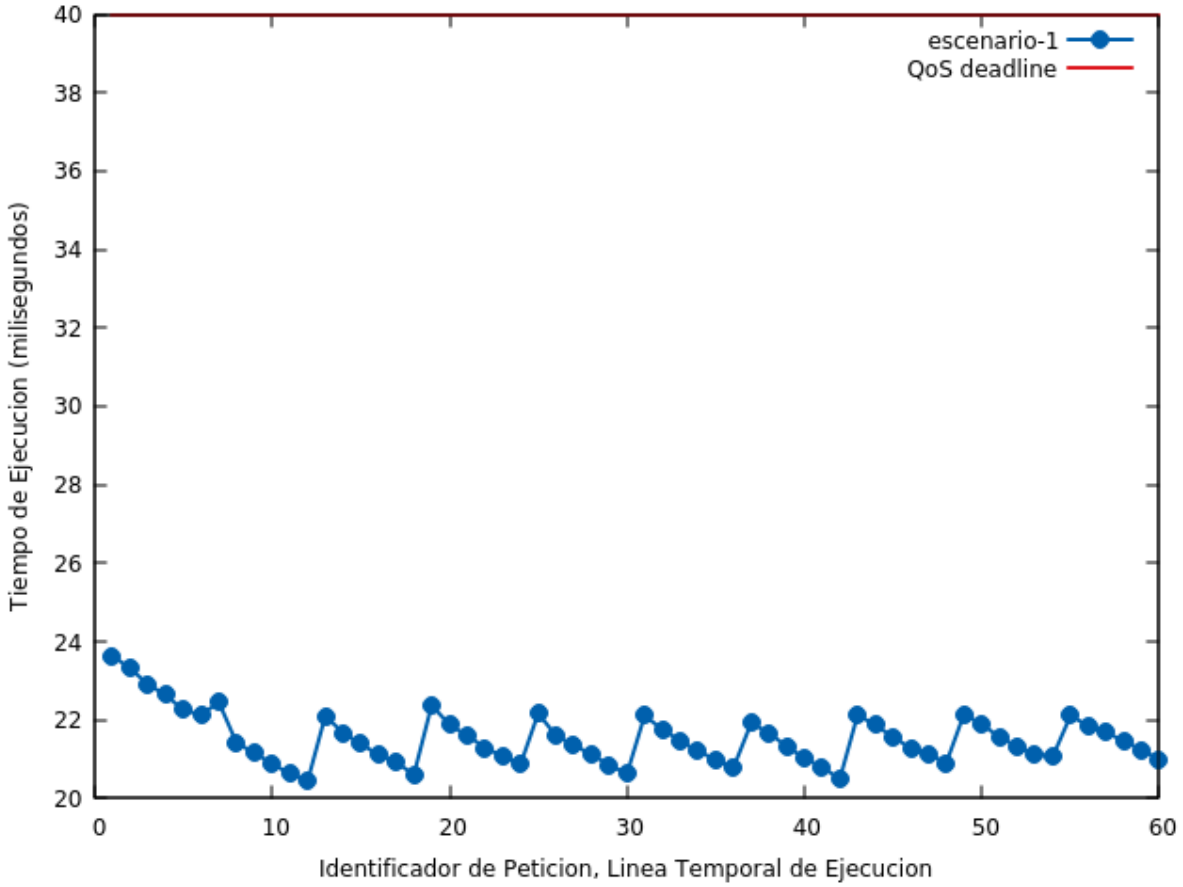

Fig. 18: Gráfica de tiempo de ejecución de 60 peticiones de un cliente en ráfagas de 6 de una arquitectura master-worker con 6 workers asociados