Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Cuadro de Mandos para Visualizar Algoritmos Distribuidos

Autor: Jan Cerezo Pomykol
Tutor: Fernando Pérez Costoya

Madrid, Abril - 2022

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática

Título: Cuadro de Mandos para Visualizar Algoritmos Distribuidos

Abril - 2022

Autor: Jan Cerezo Pomykol
Tutor: Fernando Pérez Costoya
Departamento de Arquitectura y Tecnología de Sistemas Informáticos
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

Cuando se habla de sistemas distribuidos, una de las propiedades fundamentales de los algoritmos de esta disciplina es que se ejecutan en múltiples máquinas (nodos) concurrentemente. Estos nodos interactúan de alguna manera entre ellos para lograr un objetivo determinado. Por lo general, la simultaneidad de los eventos causa dificultades a los estudiantes de la materia a la hora de entender el funcionamiento de estos algoritmos puesto que es complicado mantener una visión global del estado del sistema.

Este proyecto tiene como objetivo principal proporcionar una herramienta que facilite la comprensión del funcionamiento de los algoritmos de la asignatura de Sistemas Distribuidos. A diferencia de otras implementaciones que simplemente visualizan una simulación el algoritmo, en esta se pretende visualizar una ejecución real. De esta forma se podrá también ofrecer la posibilidad de interactuar con los nodos en tiempo de ejecución, pudiendo observar cómo el algoritmo reacciona ante distintas situaciones. Además de ofrecer una solución que facilita el aprendizaje en la asignatura de Sistemas Distribuidos, también se podría emplear como una herramienta de depuración de estos algoritmos.

Abstract

When it comes to distributed systems, one of the most important aspects of the algorithms is that they are executed simultaneously in various independent machines, commonly referred as nodes. These nodes interact with each other to achieve some objective. The simultaneousness of the events usually causes difficulties to the students when they try to understand the behavior of the algorithm. This is due to the fact that it is hard to maintain a global vision of the state of the distributed system.

The main goal of this project is to provide a tool that facilitates the understanding of the algorithms in the distributed systems discipline. Contrary to other existing implementations that only simulate a visualization of the algorithm, this one will visualize a real execution, permitting real-time interaction with the nodes. This allows to visualize how the algorithm reacts in different situations. Additionally, this tool provides a debugging environment for this type of algorithms.

Lista de figuras

Tabla de contenidos

Capítulo 1

Introducción

Uno de los aspectos más dificultosos para los estudiantes de la disciplina de sistemas distribuidos es comprender el modo de operación de los algoritmos propuestos en esta materia. La simultaneidad de los eventos en los nodos dificulta mantener una comprensión del estado global del algoritmo. A esto se le suma complejidad que supone la caída de nodos o el comportamiento anómalo de estos. Este proyecto proporciona una herramienta que facilita la comprensión del estado del algoritmo mediante la visualización de los nodos del sistema y los mensajes entre estos. Esta herramienta no sólo puede ser empleada con fines docentes, si no también con fines de depuración de algoritmos distribuidos.

1.1. Trabajos previos

En la actualidad ya existen numerosas implementaciones que logran este tipo de visualizaciones. No obstante, se pueden clasificar en dos conjuntos:

1. Las que ejecutan una simulación. Los nodos del sistema distribuido no son "reales", si no que se simula el comportamiento con fines meramente visuales. Algunos ejemplos de este tipo son [1], [2], [3].
2. Las que visualizan una ejecución real. En estas implementaciones los nodos sí son procesos reales que se ejecutan localmente o en un servidor. El entorno de visualización captura, de alguna manera, los mensajes entre nodos para visualizar el estado de cada uno. Un ejemplo de este tipo es [4].

Sin embargo, en lo que se refiere a las implementaciones del primer tipo, la ejecución *no real* del algoritmo no proporciona una representación realista del algoritmo, puesto que no se visualiza ningún proceso en tiempo de ejecución. Estas implementaciones por lo general tienen como objetivo único la visualización de la simulación de un algoritmo concreto, por lo que el algoritmo está integrado de alguna forma en la lógica de la aplicación. En estos casos se cumpliría el objetivo de facilitar la comprensión de los algoritmos, pero estos entornos presentan varias restricciones:

- Por lo general únicamente visualizan un algoritmo concreto, y este está fuertemente integrado con la interfaz. Por este motivo resulta complicado reemplazar el algoritmo en caso de que se quiera visualizar otro.
- La ejecución simulada de los eventos podría no representar los escenarios de comportamiento anómalo de nodos por el comportamiento impredecible de la red. La resistencia de fallos de los algoritmos distribuidos es una de las características más importantes de estos. Además suele ser de gran dificultad comprender este aspecto, por tanto es importante que el entorno sea capaz de visualizar cualquier tipo de situación que se pueda producir en una ejecución real.

Estas restricciones motivan, en parte, la creación de entornos del segundo tipo, que visualizan una ejecución real de un algoritmo. Estos tienen las siguientes ventajas comparados con los del primer tipo:

- La implementación de la interfaz de visualización del algoritmo suele ser independiente del propio algoritmo que se visualiza. Este desacoplamiento permite intercambiar el algoritmo a visualizar fácilmente sin tener que considerar muchos de los aspectos programáticos para visualizarlo.
- Dado que se muestra una ejecución real el entorno está implícitamente capacitado para representar cualquier situación que se pueda presentar.

1.2. Objetivos del proyecto

Teniendo en cuenta los puntos mencionados, se puede deducir que lo ideal es contar con un entorno de visualización con las siguientes características:

- Visualiza una ejecución real.
- La implementación del algoritmo es independiente de la implementación de la interfaz. En otras palabras, intercambiar el algoritmo a visualizar es sencillo.
- Se permite al usuario interactuar con los nodos, pausando o deteniendo su ejecución en tiempo real desde la interfaz.

Existen implementaciones que cumplen algunas de estas características, por ejemplo [4]. Esta permite visualizar una ejecución real, sin embargo no permite interactuar manualmente con cada nodo. Además, el diseño de la arquitectura de la implementación no permite un desacople completo del algoritmo con la interfaz. Para visualizar un algoritmo en este entorno es necesario implementar el algoritmo a visualizar en el lenguaje *Java*, y además una serie de métodos para visualizarlo correctamente.

Por lo general, ninguna implementación actual cumple estrictamente con los tres requisitos que se mencionan. El principal objetivo de este proyecto es proporcionar un entorno que los cumpla.

1.3. Estructura de la memoria

El contenido del resto del documento se organiza de la siguiente forma. En el capítulo de desarrollo se explica primero la arquitectura del entorno de visualización, seguida de la explicación detallada de la implementación y la justificación de las decisiones tomadas. Seguidamente se explica un caso de uso con una implementación concreta de un algoritmo (*Raft*). El funcionamiento detallado de este algoritmo se explica en el Anexo. Finalmente se describen los resultados obtenidos y el impacto, verificando que se han cumplido los objetivos del proyecto.

Capítulo 2

Arquitectura de la solución

En este capítulo se describe detalladamente la arquitectura de la aplicación, justificando cómo se cumplen los requisitos mencionados en la introducción.

2.1. Análisis del problema

Como se ha mencionado anteriormente, se desea visualizar una ejecución real del algoritmo, esto implica que debe existir algún medio de comunicación entre la interfaz y los nodos del mismo. Por tanto, es necesario que la aplicación que contiene la interfaz contenga además algún recurso capaz de interceptar o capturar los mensajes que se envían los nodos durante la ejecución del algoritmo. Este proceso *manager* tiene como objetivo modificar la visualización en pantalla en base a los eventos que detecta en los nodos. Existen diversas alternativas para conseguir que reciba información en tiempo real de los mensajes de los nodos.

2.2. Arquitectura de la interacción

La solución más sencilla a este problema consiste en modificar la funcionalidad del algoritmo que se ejecuta en los nodos, de tal forma que durante los envíos de mensajes se envíe también una copia al proceso *manager*. En este caso el proceso *manager* sería un simple servidor que recibe mensajes de los clientes (nodos del algoritmo) y actualiza la interfaz para representar los cambios de estado. Con esto sería necesario modificar ligeramente el algoritmo que se quiere visualizar. La arquitectura de esta solución se muestra en la Figura 2.1

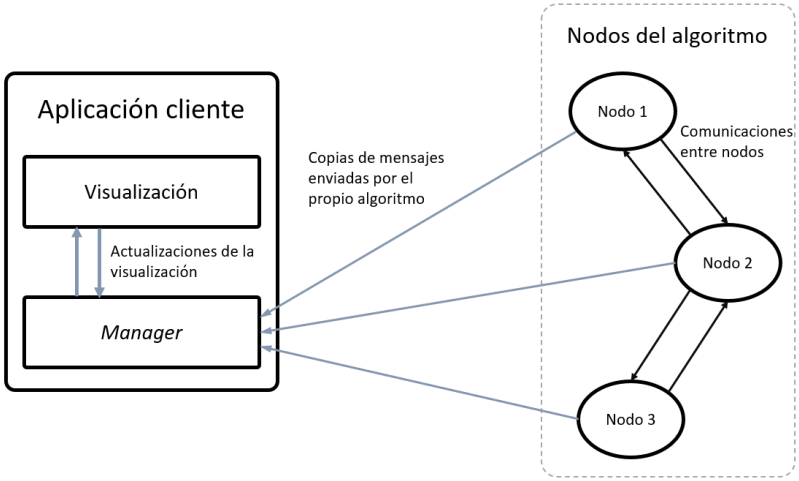

Figura 2.1: Diagrama de la primera arquitectura propuesta

En la Figura 2.1 se puede ver en la parte izquierda la aplicación cliente, que se compone de la interfaz, y del proceso *manager* que la actualiza. Ambas partes se ejecutan en la misma máquina. En la parte de la derecha se localizan los nodos del algoritmo, que pueden ejecutarse en un entorno distribuido. Las flechas entre nodos simbolizan las comunicaciones del algoritmo, y las flechas entre cada nodo y el proceso *manager* las copias de los mensajes.

Otra alternativa menos "intrusiva" en el algoritmo consiste en modificar de alguna manera las librerías que gestionan los envíos de paquetes de red. De tal forma que el algoritmo llama a la función de librería de envío de mensajes (`send`, `sendto`, `write`, etc) y esta envía una copia del mensaje al proceso *manager*. Esta solución permite no tener que modificar nada del algoritmo. La arquitectura (Figura 2.2) es muy parecida a la propuesta anteriormente, lo único que cambia es el origen del envío de las copias de mensajes.

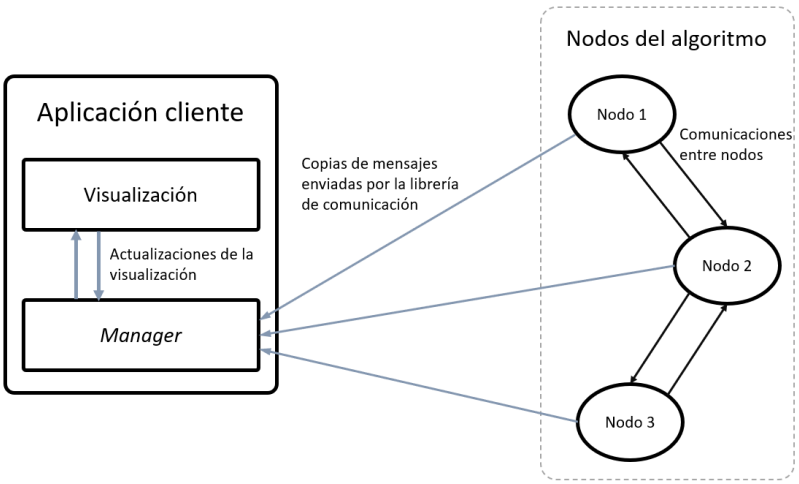

Figura 2.2: Diagrama de la segunda arquitectura propuesta

Sin embargo esta alternativa cuenta con el gran defecto de que no sería posible intercambiar el algoritmo por otro implementado en otro lenguaje de programación cualquiera. Sería necesario volver a modificar las librerías necesarias. Esta restricción, junto con el hecho de que puede ser complicado modificar la funcionalidad incluida en las librerías del lenguaje, motiva la siguiente alternativa.

La alternativa final cuenta con otro proceso capturador o *sniffer* que intercepta el tráfico de red a nivel de protocolo de un determinado proceso. Se ejecuta una instancia junto con cada nodo del algoritmo de tal forma que los mensajes capturados se reenvían al proceso *manager* para que este actualice la visualización. Esta captura de mensajes se lleva a cabo a nivel de protocolo, por lo que no es necesario modificar ningún aspecto del algoritmo que se quiere visualizar. En esta solución el cuadro de mandos se abstrae totalmente del algoritmo, de forma que permite visualizar implementaciones en distintos lenguajes, siempre que el medio de comunicación sea un protocolo conocido, por ejemplo TCP. Esta modificación se puede observar en la figura 2.3.

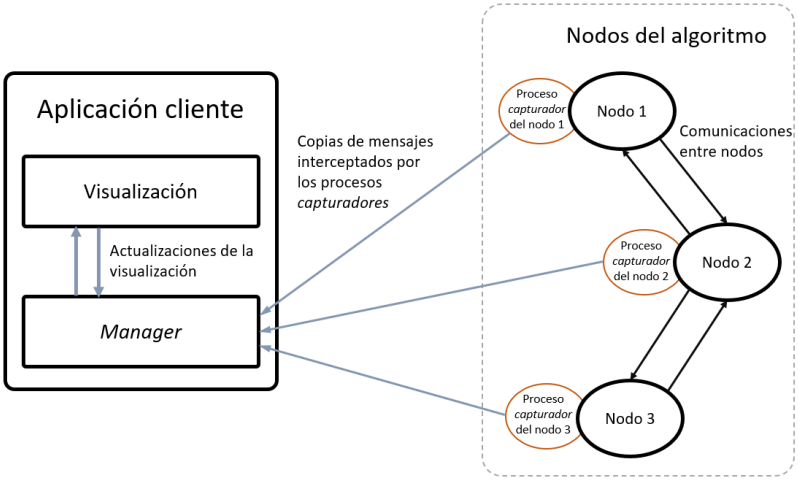

Figura 2.3: Diagrama de la tercera arquitectura propuesta

Por parte del proceso *manager*, el funcionamiento es el mismo que en las soluciones propuestas anteriormente. Otra ventaja de esta arquitectura es que soportaría también visualizar algoritmos cuyo medio de comunicación se basa en las llamadas a procedimientos remotos (RPC). En las soluciones propuestas anteriores únicamente se tienen en cuenta los algoritmos cuyo modelo de comunicación se basa en el envío de mensajes entre *sockets* (sería necesario modificar el proceso *manger* para admitir la comunicación mediante RPC). Sin embargo, dado que las librerías de llamadas a procedimientos remotos también se basan en la comunicación con algún protocolo (por lo general TCP), también se capturarían estos mensajes.

La ventaja principal de esta arquitectura comparada con las anteriores, es que permite un desacoplamiento total del algoritmo. Además de esto, satisface los requisitos que se mencionan en la introducción.

Capítulo 3

Desarrollo

A continuación se explicará la implementación detallada de cada uno de los componentes de la arquitectura. Puesto que el algoritmo a visualizar no es el contenido principal de este proyecto, se ha escogido uno cuya complejidad no sea excesiva. La elección ha sido el algoritmo *Raft* [5] principalmente porque es muy adecuado para comprobar el funcionamiento de la interfaz. A diferencia de otros algoritmos que envían mensajes hasta alcanzar un objetivo, en *Raft* se envían mensajes indefinidamente. Además de este aspecto, también es un algoritmo no demasiado complejo de comprender e implementar.

3.1. Implementación del algoritmo *Raft*

Para la implementación del algoritmo *Raft*, se ha escogido el lenguaje de programación *Go* [6], que proporciona numerosas comodidades y ventajas a la hora de programar aplicaciones distribuidas. Por otra parte, la propuesta original de *Raft* [1] sugiere que la comunicación entre los nodos se implemente mediante llamadas a procedimientos remotos (RPC) puesto que se elimina la complejidad que supone enviar y recibir mensajes con *sockets*. Sin embargo, en esta implementación, se ha diseñado la comunicación con mensajes tradicionales TCP. Además de que actualmente existen numerosas implementaciones del algoritmo empleando llamadas a procedimientos remotos [7], [8], la ventaja de los mensajes TCP es que es mucho más claro el proceso de envío y recepción de los mensajes.

Además de este aspecto, la implementación incluye la elección de líder completa y la replicación de *log* parcial. La replicación no está completa puesto que esta implementación tiene como objetivo visualizar los mensajes relacionados con el algoritmo, y no el contenido de los *logs* replicados. Por otra parte, la implementación está diseñada para interactuar con cada nodo mediante la entrada estándar del proceso. La función principal de la implementación (*Main*) contiene la lógica necesaria para interactuar con los procesos en segundo plano que ejecutan el algoritmo. Las funcionalidades o comandos que incluye esta implementación son las siguientes:

- `START`: Comenzar a ejecutar el algoritmo.
- `STOP`: Detiene la ejecución del algoritmo.
- `ADD`: Añade un nodo vecino. Se debe suministrar como parámetro la dirección válida del nodo vecino con formato `ip:puerto`.
- `PEERS`: Imprime la lista de nodos vecinos.

A continuación se describe la implementación de cada uno de los componentes del algoritmo. La descripción simplificada de *Raft* puede leerse en el Anexo, o en el artículo original [1] (versión completa y detallada). Por otra parte, sólo se muestran partes del código simplificadas que no contienen control de errores.

Cuando un nodo comienza a ejecutar el algoritmo, comienzan a ejecutarse dos procesos secundarios. El primer proceso ejecuta la función `raft`, que contiene la mayoría del funcionamiento del algoritmo, mientras que el segundo ejecuta la función `listener`, que es la encargada de recibir mensajes del *socket* TCP. La comunicación entre estas funciones se lleva a cabo mediante canales de *Go* [9], una de las mejores funcionalidades del lenguaje, que son similares a los *pipes* de Linux. El proceso que ejecuta la función `raft` recibe mensajes de este canal, y en función del estado del nodo y otros factores, actúa de una manera u otra. Por otra parte la función `listener`, entre otras, envía mensajes al canal.

3.1.1. Estado del algoritmo

El estado del algoritmo se almacena en una estructura con los siguientes campos (se muestran únicamente los relacionados con la descripción del algoritmo):

```
1  type NodeStatus struct {
2     peers               []NodeAddr     // Lista de nodos vecinos
3     received_votes      int8           // Numero de votos recibidos en una eleccion
4     voted_current_term  bool           // Indica si se ha votado en el term acutal
5     term                int            // Term actual
6     log                 []NodeLogEntry // Lista de entradas de log
7     raftStatus          uint8          // Estado de raft: follower | candidate | leader
8     electionTimeout     int64          // Tiempo de timeout para listener()
9     leaderHeartbeat     int64          // Intervalo de tiempo entre mensajes para
          leaderHeartbeats()
10    eventChan           chan Event     // Canal de comunicacion
11 }
```

3.1.2. Implementación de `listener`

La implementación de la función `listener` es simple:

```
func listener(l *net.TCPListener, status *NodeStatus) {
   for {
      var timeout time.Time
      // Calcular en que instante habra timeout
      timeout = time.Now().Add(time.Millisecond * time.Duration(status.electionTimeout))
      // Establecer el timeout para el socket
      l.SetDeadline(timeout)

      // Esperar a una nueva conexion
      conn, err := l.Accept()
      if err != nil {
         if errors.Is(err, os.ErrDeadlineExceeded) {
            if status.raftStatus == candidate || status.raftStatus == follower {
               // Si se ha superado el timeout y el estado es candidate o follower
               // Escribir en el canal el evento de timeout
               status.eventChan <- Event{
                  msg: NodeMsg{MsgType: followerTimeout},
               }
            } else {
               // Si es lider entonces este timeout no le afecta
            }
            continue
      }

      // Procesar la nueva conexion
      go responseHandler(conn, status)
   }
}
```

A esta función se le pasa como parámetro el estado del nodo, y el *socket* por el que el nodo recibe mensajes. A modo de resumen, la funcionalidad se basa en esperar a recibir una nueva conexión en el *socket* y procesar el mensaje con la función `responseHandler`, o bien enviar el evento de *timeout* al canal de comunicación en el caso de que no se inicie una conexión en un tiempo determinado.

3.1.3. Implementación de `responseHandler`

La implementación de la función `responseHandler` es la siguiente:

```
func responseHandler(conn net.Conn, status *NodeStatus) {
   decoder := json.NewDecoder(conn)
   defer conn.Close()

   var msg NodeMsg
   decoder.Decode(&msg)

   status.eventChan <- Event{
      msg:    msg,
      sender: conn.RemoteAddr().(*net.TCPAddr).IP.String(),
   }
}
```

Esta función recibe del *socket* un mensaje, lo *deserializa* a un `struct` de *Go* [10] y envía este objeto al canal.

3.1.4. Implementación de `raft`

La función que lee y procesa los mensajes del canal es `raft`, que implementa la mayoría de las funcionalidades del algoritmo.

```
1  func raft(wg *sync.WaitGroup, status *NodeStatus) {
2     // Leer eventos del canal
3     for event := range status.eventChan {
4        switch event.msg.MsgType {
5        // Si otro nodo ha comenzado una votacion
6        case requestVote:
7           // Independientemente del estado, convertirse en follower si el term del
                mensaje es superior y si no se ha votado en el term actual
8           if event.msg.Term > status.term && !status.voted_current_term {
9              status.voted_current_term = true
10             status.raftStatus = follower
11             status.term = event.msg.Term
12             go sendMsg(
13                NodeMsg{MsgType: grantVote, Term: event.msg.Term},
14                event.sender,
15                event.msg.SenderPort,
16             )
17          }
18       // Si es un mensaje de voto aceptado
19       case grantVote:
20          switch status.raftStatus {
21          case follower:
22             // No hace nada porque ya se termino la eleccion
23          case candidate:
24             // Mirar si es un voto a nuestra eleccion
25             if event.msg.Term == status.term {
26                status.received_votes++
27                if int(status.received_votes) > (len(status.peers)+1)/2 {
28                   // Si se han recibido suficientes votos, convertirse en lider
29                   status.raftStatus = leader
30                   go leaderHeartbeats(status)
31                }
32             }
33          case leader:
34             // No hacer nada porque ya se ha conseguido ser lider
35          }
36       // Si es un mensaje con entradas de log
37       case appendEntries:
38          if event.msg.Term > status.term {
39             // Si el term del mensaje es superior, convertirse en follower
40             status.voted_current_term = false
41             status.raftStatus = follower
42             status.term = event.msg.Term
43          }
44       // Si es un evento de timeout
45       case followerTimeout:
46          switch status.raftStatus {
47          case follower, candidate:
48             // Comenzar eleccion si es follower o candidate
49             go startElection(status)
50          case leader:
51             // Ignorar evento
52          }
53       }
54    }
55 }
```

Como se puede observar, esta función altera el estado del algoritmo en función de los mensajes que recibe del canal y del estado en el que se encuentra:

- Cuando el mensaje que recibe es de tipo `requestVote`, significa que algún otro nodo ha iniciado una votación. En este escenario, el nodo se convertirá en seguidor si el nuevo *term* es superior, y si no ha votado todavía en el *term* actual.

- En el caso de que el mensaje recibido sea de tipo `grantVote`, significa que el propio nodo ha iniciado una votación anteriormente, y algún otro nodo ha aceptado el voto. Si el nodo está en el estado *follower*, se ignora la petición, puesto que la votación se ha terminado y no se ha conseguido el liderazgo. En el caso en el que el estado del nodo sea *candidate*, hay que comprobar si el voto aceptado se corresponde con la votación actual. En el caso de que lo sea, el nodo se convierte en líder ejecutando la función `leaderHeartbeats` si ha recibido suficientes votos. En el caso de que el estado sea *leader*, se ignora el mensaje, puesto que ya se ha conseguido el liderazgo.

- En el caso de que se reciba un mensaje de tipo `appendEntries`, el nodo se convierte en seguidor si el *term* del mensaje es superior al *term* actual.

- Si el tipo del mensaje es `followerTimeout`, significa que la función `listener` no ha recibido una nueva conexión en el tiempo del *timeout*. Este escenario implica que el nodo inicie una votación (función `startElection`) en el caso de que sea seguidor o candidato.

3.1.5. Implementación de `leaderHeartbeats`

La función `leaderHeartbeats` se ejecuta cuando el estado del nodo es *leader*. El objetivo es enviar periódicamente mensajes de tipo `appendEntries` a los nodos vecinos para que continúen en el estado *follower*. Solamente termina cuando el estado del nodo deja de ser *leader*.

```
1  func leaderHeartbeats(status *NodeStatus) {
2     for {
3        if status.raftStatus != leader {
4           return
5        }
6
7        msg := NodeMsg{
8           MsgType:    appendEntries,
9           Term:       status.term,
10          Entries:    nil,
11       }
12       timeout := status.leaderHeartbeat
13
14       // Enviar appendEntries a los vecinos
15       for _, p := range status.peers {
16          go sendMsg(msg, p.ip, p.port)
17       }
18
19       // Esperar para enviar la siguiente rafaga de appendEntries
20       <-time.After(time.Duration(timeout) * time.Millisecond)
21    }
22 }
```

3.1.6. Implementación de `startElection`

Por último, la función `startElection` es la encargada de enviar mensajes de tipo `requestVote` a los nodos vecinos cuando se inicia una nueva elección.

```
1  func startElection(status *NodeStatus) {
2     status.raftStatus = candidate
3     status.term++
4     status.received_votes = 1 // Votar por uno mismo
5
6     msg := NodeMsg{
7        MsgType:    requestVote,
8        Term:       status.term,
9        Entries:    nil,
10    }
11
12    for _, p := range status.peers {
13       go sendMsg(msg, p.ip, p.port)
14    }
15 }
```

3.2. Implementación del proceso *capturador*

Como se describe en la sección 2.2, el proceso capturador debe capturar los mensajes de red que envía el nodo y reenviarlos al proceso manager. Además de esto, para permitir una comunicación bidireccional, debe ser capaz de interactuar con el proceso que ejecuta el algoritmo. Cabe destacar además que la comunicación entre los nodos del algoritmo no necesariamente debe ser a través de un mismo puerto siempre. En la implementación de *Raft* de este proyecto, se reciben siempre desde el mismo puerto, pero se desconocen los puertos desde los que se envían los mensajes.

Actualmente existen múltiples librerías de programación de captura de mensajes de red, algunos ejemplos son [11], [12], [13]. Sin embargo todas cuentan con una restricción, no se permite actualizar el filtro de paquetes de red dinámicamente. Esto es un inconveniente únicamente por la funcionalidad de añadir nodos vecinos durante la ejecución. Al añadir un nodo vecino, es necesario actualizar el filtro para incluir el puerto del nuevo nodo para capturar los mensajes enviados a ese nodo. Con las librerías existentes sería necesario detener la captura de mensajes, actualizar el filtro, y finalmente volver a comenzar la captura. Esta pausa en la escucha de mensajes de red puede suponer que no se capturen los mensajes en este intervalo de tiempo. Es por este motivo por el que se ha implementado un proceso capturador sin emplear librerías de este tipo.

El lenguaje de programación escogido para implementar este módulo es *Python* [14], dado que permite implementar las funcionalidades necesarias de una forma sencilla comparado con otros lenguajes. La implementación se compone de dos partes, una con la función `sniffer` que captura los mensajes de red y los retransmite al proceso *manager*, y otra con la funcionalidad relacionada con arrancar el nodo del algoritmo y comunicarse con él. Más adelante se detalla la implementación de estos módulos, comenzando por la función `sniffer`.

3.2.1. Captura de mensajes

Como se ha indicado anteriormente, las librerías existentes de captura de mensajes de red no permiten actualizar el filtro dinámicamente. Es por esto por lo que se ha implementado este módulo de captura personalizado. A continuación se muestra parte de la función `sniffer`.

```
def sniffer():
    conn = socket.socket(socket.AF_PACKET, socket.SOCK_RAW, socket.ntohs(3))

    while True:
        # Get new packet
        raw_data, addr = conn.recvfrom(65535)

        # Get destination MAC, source MAC and ethernet protocol
        dest_mac, src_mac, eth_proto = struct.unpack('! 6s 6s H', raw_data[:14])
        eth_proto = socket.htons(eth_proto)
        data = raw_data[14:]

        # If packet is IPv4
        if eth_proto == 8:
            version_header_length = data[0]
            version = version_header_length >> 4
            header_length = (version_header_length & 15) * 4
            ttl, ip_proto, src_ip, target_ip = struct.unpack('! 8x B B 2x 4s 4s', data
                [:20])
            data = data[header_length:]

            # If TCP protocol
            if ip_proto == 6:
                src_port, dest_port, sequence, ack, offset_reserved_flags = struct.
                    unpack('! H H L L H', data[:14])
                offset = (offset_reserved_flags >> 12) * 4
                flag_urg = (offset_reserved_flags & 32) >> 5
                flag_ack = (offset_reserved_flags & 16) >> 4
                flag_psh = (offset_reserved_flags & 8) >> 3
                flag_rst = (offset_reserved_flags & 4) >> 2
                flag_syn = (offset_reserved_flags & 2) >> 1
                flag_fin = offset_reserved_flags & 1
                data = data[offset:]

                # If packet has content for the application
                if flag_psh == 1 and dest_port in ports:
                    captured_msg = json.loads(data.decode())

                    captured_msg["SrcPort"] = captured_msg["SenderPort"]
                    captured_msg["SrcIp"] = "localhost"
                    captured_msg["DstPort"] = dest_port
                    captured_msg["DstIp"] = "localhost"

                    app_server.sendto(bytes(json.dumps(captured_msg), encoding='utf-8'),
                         manager_addr)
```

Al principio se crea un *socket* de red de tipo `SOCK_RAW`, que recibe mensajes a nivel de protocolo *ethernet*. Seguidamente se comienza un bucle infinito que ejecuta lo siguiente:

1. Guarda un nuevo mensaje en un buffer de máximo tamaño.
2. Obtiene las direcciones MAC origen y destino, y el protocolo *ethernet*.
3. Comprueba si el paquete ha sido enviado desde el protocolo IPv4.

4. Extrae la información de la cabecera del paquete IP. Nótese que es necesario recurrir a operaciones de desplazamiento de bits dado que Python no permite indexar un byte a nivel de bit.
5. Comprueba si el paquete se ha enviado con el protocolo TCP.
6. Se extraen los datos de la cabecera TCP, entre los cuales se encuentran el *flag* PSH, el puerto destino, el tamaño de la cabecera y el contenido del paquete.
7. Se comprueba si el puerto destino está en la lista de puertos de los nodos vecinos, y si el *flag* PSH está a 1. Este *flag* indica si el mensaje se envía al nivel de aplicación, es decir, indica si contiene datos de la aplicación.
8. Se envía un mensaje al proceso *manager* con la información del mensaje capturado.

Cabe destacar que la librería `socket` de *Python* no incluye ninguna definición de sockets de este tipo para el sistema operativo *Windows* [15]. Además de este aspecto, dado que el socket se ha declarado para recibir todos los mensajes *ethernet*, es necesario ejecutar el código como *superusuario*.

3.2.2. Interacción con el proceso del algoritmo

En cuanto a la gestión del proceso que ejecuta el algoritmo, se ha planteado la siguiente solución. Se ejecuta el proceso en un proceso hijo, de tal forma que se tiene acceso a distintos recursos, por ejemplo la entrada y la salida estándar. El código simplificado se muestra a continuación:

```
1 process = subprocess.Popen('go run ./node/node.go', text=False, stdin=subprocess.PIPE,
      stdout=subprocess.PIPE, stderr=subprocess.PIPE, universal_newlines=False, shell=True
      )
2
3 while process.poll() is None:
4     line = process.stdout.readline().decode()
5     if line != "":
6         print(">>> ", line, sep='', end='')
7
8 process.wait()
9 print("return_code:", process.returncode)
```

La primera línea inicializa la ejecución del algoritmo *Raft*, indicando que se quiere acceder a la salida y entrada estándar. Seguidamente se comienza un bucle que imprime todos los mensajes que el proceso hijo muestra por la salida estándar. Este termina cuando el proceso hijo escribe en la salida estándar el carácter EOF (final de fichero). Finalmente se espera a que termine el proceso hijo para imprimir el valor de terminación.

3.2.3. Interacción con el proceso *manager*

Por último, queda por explicar la forma en la que el proceso *manager* se comunica con el proceso que ejecuta el algoritmo. Esto se consigue con la función `udp_listener` (no confundir con la función `listener` del algoritmo *Raft*).

```
def udp_listener(process, node_port):
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as listener:
        listener.bind(("", 0)) # random port
        while True:
            msg_raw, addr = listener.recvfrom(2048)
            msg = json.loads(msg_raw.decode())
            final_msg = b'%s\n' % bytes(msg["args"], encoding='utf-8')

            process.stdin.write(final_msg)
            process.stdin.flush()

            if msg["type"] == "ADD":
                addPort(int(getPeerPort(final_msg)))
```

Como se puede observar, al comenzar se declara un socket de tipo UPD asignando un puerto aleatorio. En esta implementación se ha escogido el protocolo UDP para la comunicación con el proceso *manager*. Se podría haber empleado cualquier otro tipo de protocolo de envío de mensajes de red. Sin embargo se ha escogido el protocolo UDP puesto que es un protocolo más rápido que TCP. La velocidad de transmisión de mensajes puede reducir la latencia en la visualización de mensajes en la interfaz.

El resto de la función consiste en un bucle infinito que recibe mensajes del proceso *manager* y los escribe en la entrada estándar del proceso hijo (que ejecuta el algoritmo). Por otra parte, si el mensaje recibido de la interfaz indica que se quiere agregar un nuevo nodo, entonces se agrega el puerto a la lista de puertos que emplea el filtro de la función `sniffer`, descrita anteriormente.

3.3. Descripción del cliente

A continuación se describe la implementación del cliente. A modo de recordatorio la interfaz debe ser capaz de visualizar tanto los nodos del algoritmo como los mensajes que se envían en tiempo real, permitiendo al usuario interactuar de distintas formas. Para visualizar una red distribuida, sería necesario representar en un grafo las máquinas como nodos, y las conexiones como vértices. Las funcionalidades de representar estos elementos en pantalla y ofrecer la posibilidad arrastrar nodos para cambiar la distribución en pantalla, entre otras, son bastante complejas de implementar y no forman parte de los objetivos de este proyecto. Es por esto por lo que la elección del lenguaje de implementación se ha visto marcado por la existencia de alguna librería de programación que implemente estas funcionalidades. El lenguaje escogido es *JavaScript* por los siguientes motivos:

- *JavaScript* [16] es el lenguaje que cuenta con más librerías a fecha de escritura [17], por tanto es más probable que contenga una que cumpla con los requisitos.
- Además de ser el lenguaje con más librerías, es empleado extensamente en el desarrollo web, y por tanto está capacitado para implementar la lógica necesaria para controlar la interfaz de usuario.

- Existen diferentes librerías que cumplen con los requisitos mencionados para representar grafos, algunos ejemplos son [18], [19], [20].

Por otra parte, la interfaz no ha sido implementará como una aplicación *web* convencional, si no como una aplicación de *electron*. Esto es imprescindible puesto que desde la versión *web* de *JavaScript* no es posible enviar mensajes de red, por motivos de seguridad. *Electron*[21] es un *framework* que permite desarrollar aplicaciones de escritorio multiplataforma empleando recursos de desarrollo *web*, por ejemplo *HTML*, *CSS* y *JavaScript*.

En cuanto a la librería de visualización de grafos, se ha escogido *Vis.js* [18], puesto que además de ser gratuita y de código abierto, permite visualizar animaciones que recorren los vértices del grafo. Estas animaciones se emplean para representar el tráfico de red entre los nodos del algoritmo. En lo restante de este apartado se detalla el funcionamiento de los botones de la interfaz, y cómo interactúa el cliente con el proceso *capturador*. La interfaz final de la aplicación puede apreciarse en la imagen de la figura 3.1.

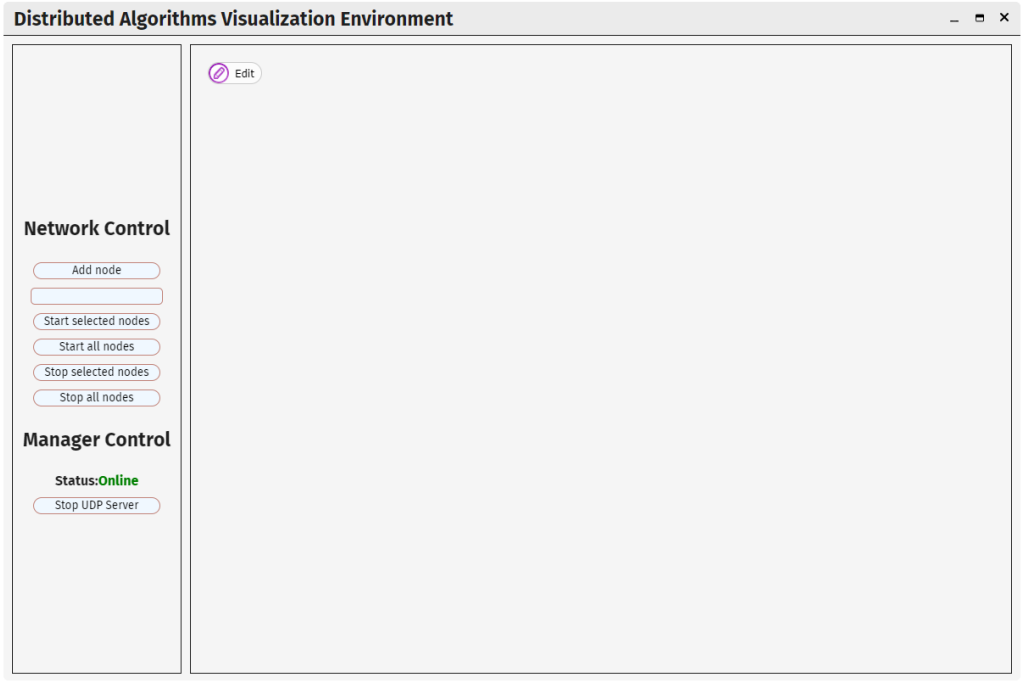

Figura 3.1: Captura de pantalla de la interfaz

Como se puede observar, en la parte izquierda de la pantalla se localizan los controles, y en la derecha el marco donde se dibujará el grafo del algoritmo. Los controles se dividen en dos partes, una que controla los nodos, y otra que controla el proceso *manager*. Como se ha descrito anteriormente, el proceso capturador recibe mensajes a través de un *socket* de red, al mismo tiempo que los retransmite al proceso que ejecuta el algoritmo. Estos mensajes, se envían desde la interfaz pulsando alguno de los siguientes botones:

- Botones "*Start selected nodes*" y "*Start all nodes*": comienzan la ejecución de los nodos seleccionados o de todos los nodos, respectivamente. Se envía un mensaje correspondiente al comando `START`, que se describió durante la sección 3.1.
- Botones "*Stop selected nodes*" y "*Stop all nodes*": tienen el mismo comportamiento que el último, excepto que detienen la ejecución de uno o más nodos enviando el mensaje correspondiente al comando `START`.
- Botón "*Add node*": añade un nodo a la visualización. Se debe suministrar la dirección de nodo a añadir.

Por la parte de la sección de control del proceso *manager*, se puede detener y arrancar el proceso. Pudiendo observar el estado de este en la parte superior al botón. La interfaz es completamente "*responsive*" como se dice en el ámbito del desarrollo web, lo cual significa que el tamaño de los elementos en la pantalla se adapta automáticamente para mostrar todo el contenido cuando se cambia de tamaño la ventana.

Más adelante, en el siguiente apartado, se explica la funcionalidad del proceso *manager* y cómo interactúa con la visualización para mostrar los eventos que se producen en el algoritmo.

3.4. Implementación del *manager*

Como se ha descrito anteriormente, el proceso *manager* debe contener toda la funcionalidad relacionada con recibir mensajes de todos los procesos capturadores, y modificar la visualización de acuerdo con el origen, destino y contenido de estos. Puesto que es un proceso que debe interactuar con la visualización de la red que se visualiza, es necesario que sea implementado junto con la interfaz para tener acceso a las variables del programa necesarias.

En *electron* se puede implementar fácilmente un proceso servidor con el módulo `dgram`, que permite crear y manejar *sockets* UDP. La implementación básica de un servidor de este tipo se muestra a continuación:

```
1  const dgram = require('dgram');
2  var server;
3
4  function start_server() {
5      server = dgram.createSocket('udp4');
6
7      server.on('error', function(err) {
8          // Error
9      });
10
11     server.on('message', function(msg, rinfo) {
12       // Procesar mensaje
13     });
14
15     server.on('listening', function() {
16         // Socket listo para recibir mensajes
17     });
18
19     server.on('close', function() {
```

```
20          // El socket se ha cerrado
21      });
22
23      // Asignar un puerto de escucha
24      server.bind(3333);
25  }
```

Como se puede observar, la implementación en *JavaScript* de un servidor UDP es sencilla. Por cada evento del *socket* (`error`, `message`, `listening` y `close`) se asocia una función que se ejecuta cada vez que el evento se produce. En el caso de este Trabajo de Fin de Grado, la más importante claramente es la función asociada al evento `message`, que se ejecuta cada vez que se recibe un mensaje. Esta función debe ser capaz de actualizar la interfaz de acuerdo con el contenido del mensaje. A continuación se muestra parte de la implementación final del servidor:

```
1  function start_server() {
2      server = dgram.createSocket('udp4');
3
4      server.on('error', function(err) {
5          print('start_server', err.stack)
6          server.close();
7      });
8
9      server.on('message', function(msg, rinfo) {
10         print('start_server', `msg from ${rinfo.address}:${rinfo.port}\n${msg}`);
11         var parsed_msg = JSON.parse(msg);
12         var e = `${parsed_msg.SrcIp}:${parsed_msg.SrcPort}_${parsed_msg.DstIp}:${
               parsed_msg.DstPort}`
13
14         // Animate message
15         console.log(`Edge: ${e}`);
16         network.animateTraffic([{
17             edge: e,
18             trafficSize: 5,
19         }]);
20
21         // Update node labels
22         var aux_id = `${parsed_msg.SrcIp}:${parsed_msg.SrcPort}`;
23         network_nodes.update([{
24             id: aux_id,
25             label: formatLabel(
26                 parsed_msg.SrcIp,
27                 nodes[aux_id].port,
28                 parsed_msg.SrcPort,
29                 parsed_msg.Term
30             ),
31         }]);
32     });
33
34     server.on('listening', function() {
35         const address = server.address();
36         print('start_server', `listening on ${address.address}:${address.port}`);
37     });
38
39     server.on('close', function() {
40         print('start_server', 'Closed server');
41     });
42
43     server.bind(3333);
44 }
```

En la implementación del cliente se emplean varias funciones auxiliares. Una de ellas es `print`, que imprime mensajes de log en la consola formateados para su lectura fácil. Recibe como primer parámetro la función que la llama, y como segundo el mensaje. De esta manera es posible determinar el origen de los mensajes de log. Por otra parte se emplea la función `formatLabel`, que formatea varios parámetros a mostrar en la interfaz de cada nodo.

Como se puede observar la función asociada al evento `message` averigua, según la información sobre la conexión proporcionada por el *socket*, el vértice del grafo que se corresponde con la conexión entre el nodo origen. Además de esto, también obtiene los identificadores de ambos nodos implicados en la comunicación.

La gestión de animaciones de la interfaz que representan los mensajes se lleva a cabo con la función `animateTraffic`, incluida con la librería de visualización de grafos. Admite como parámetro una lista de vértices en los que mostrar una animación, pero en el caso de este proyecto se indica sólo uno dado que se visualiza únicamente un mensaje por cada evento `message`.

3.5. Caso de uso

En este apartado se demuestra la funcionalidad completa de la aplicación con un algoritmo concreto: *Raft*. Cabe destacar que los pasos mostrados a continuación son únicamente un ejemplo concreto. Como es lógico, no es posible incluir tantas imágenes como combinaciones posibles de acciones del usuario.

En primer lugar, es necesario ejecutar por una parte la aplicación de escritorio y por otra todas las instancias que se deseen del proceso capturador. Una vez inicializadas se introduce en el campo asociado con el botón "*Add node*" de la interfaz, una a una cada dirección de cada instancia. El proceso capturador muestra convenientemente, por la salida estándar, la dirección del nodo con el formato adecuado para que el usuario únicamente tenga que copiar y pegar la información, sin necesidad de escribir manualmente. El resultado de insertar varios nodos en el sistema de visualización es se puede apreciar en la captura de pantalla de la figura 3.2.

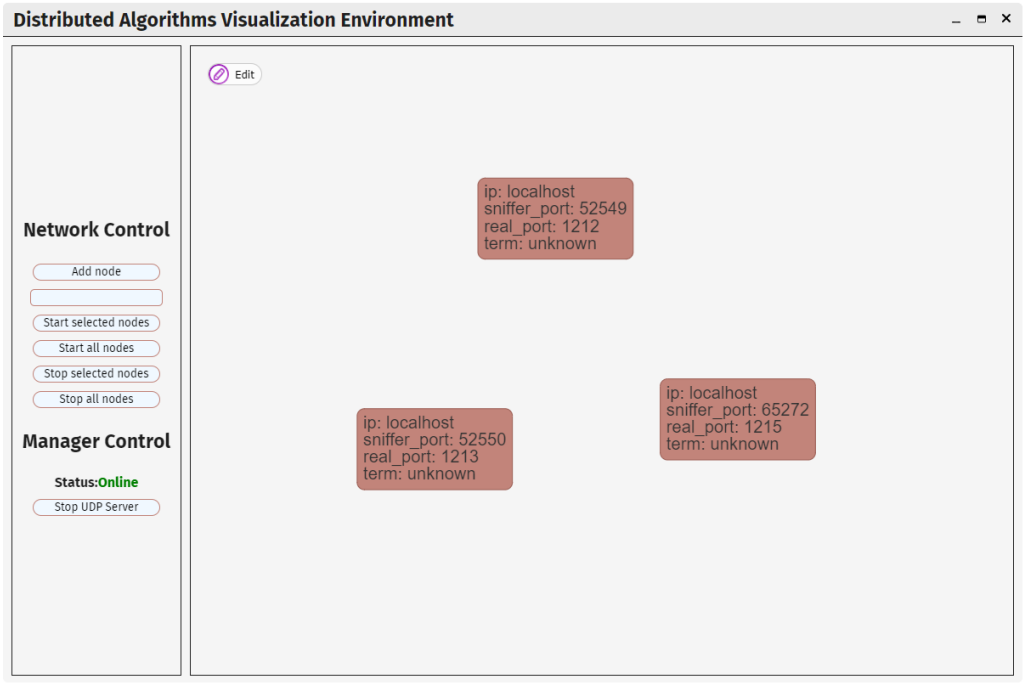

Figura 3.2: Resultado de añadir varios nodos en la visualización

Los contenidos visualizados de cada nodo son los siguientes: puerto del algoritmo, puerto del proceso capturador, dirección IP del nodo, y *term* actual. Como se puede observar, se desconoce el *term* de cada nodo. Esto se debe a que la visualización del algoritmo únicamente conoce el estado a partir del contenido de los mensajes que captura el proceso capturador. Puesto que todavía no se han enviado mensajes, esta información es desconocida por el momento.

En las figuras 3.3, 3.4 y 3.5 se muestra el procedimiento seguido para añadir una nueva conexión. En primer lugar, se pulsa en el botón de edición para acceder al menú de conexiones, esto se puede observar en la figura 3.3. Seguidamente se pulsa el botón "*Add Connection*". A continuación se pulsa en un nodo y, manteniendo la pulsación, se arrastra el puntero hasta otro nodo donde se suelta (Figura 3.4). El resultado son dos vértices entre ambos nodos, cada uno simboliza la conexión en un sentido. El resultado final de este paso se muestra en la figura 3.5. A continuación se sigue el mismo procedimiento para añadir las conexiones restantes que se deseen. Un ejemplo de configuración final se puede apreciar en la figura 3.6.

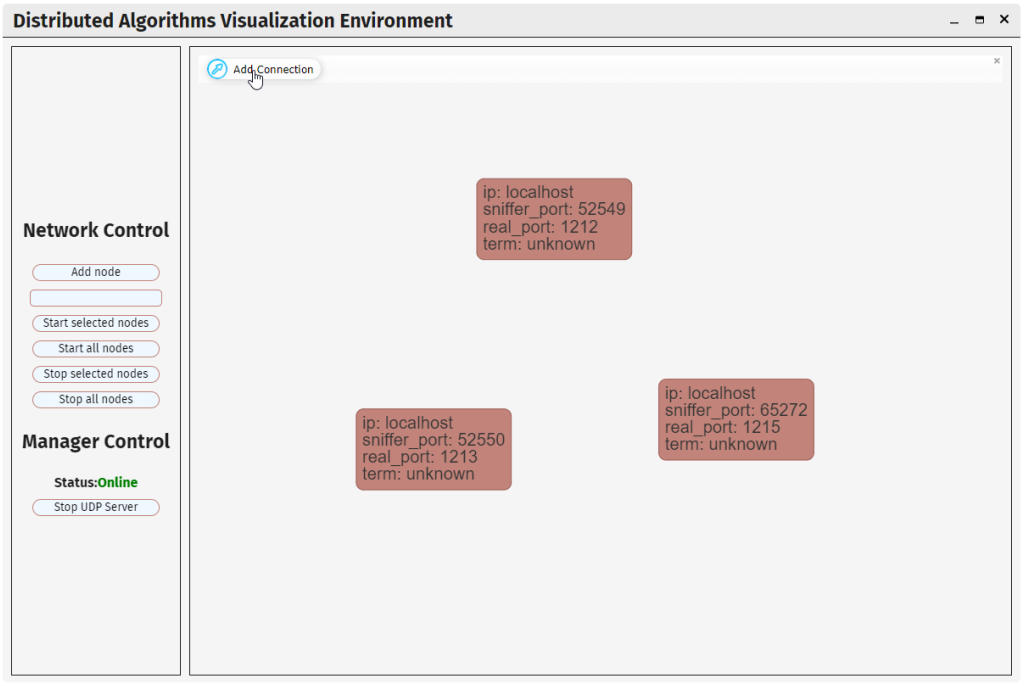

Figura 3.3: Menú de conexiones

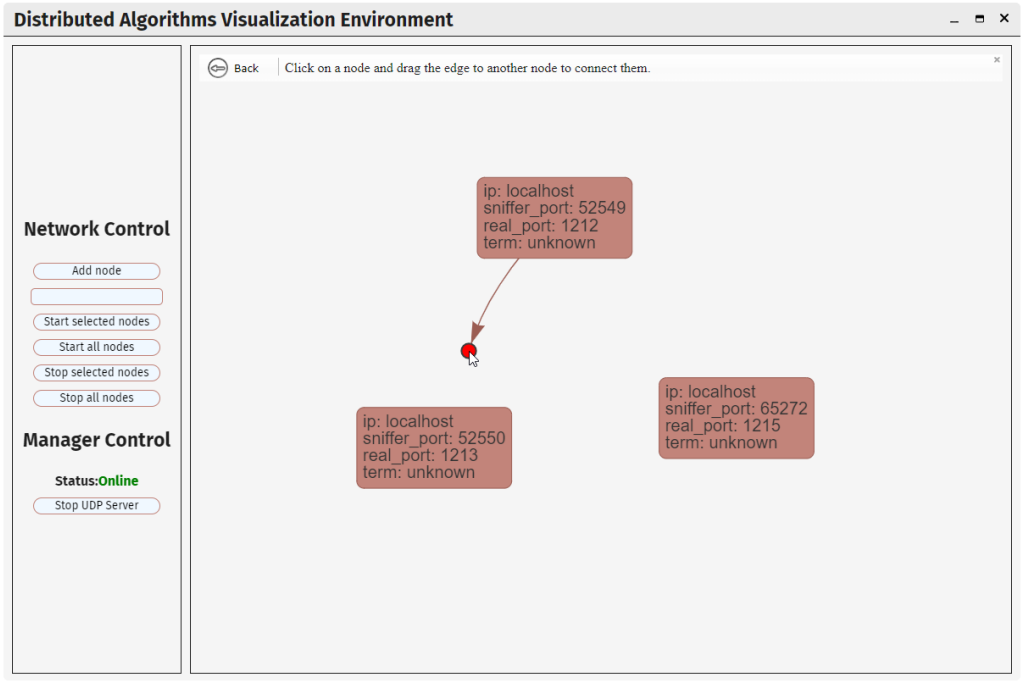

Figura 3.4: Proceso de crear una nueva conexión

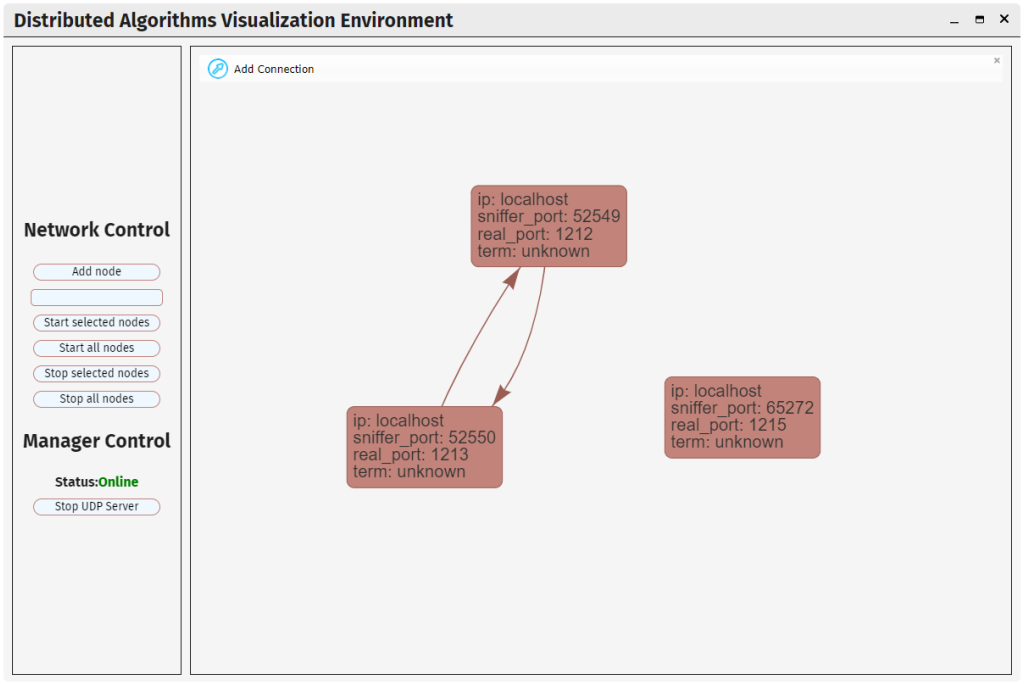

Figura 3.5: Resultado de añadir una nueva conexión

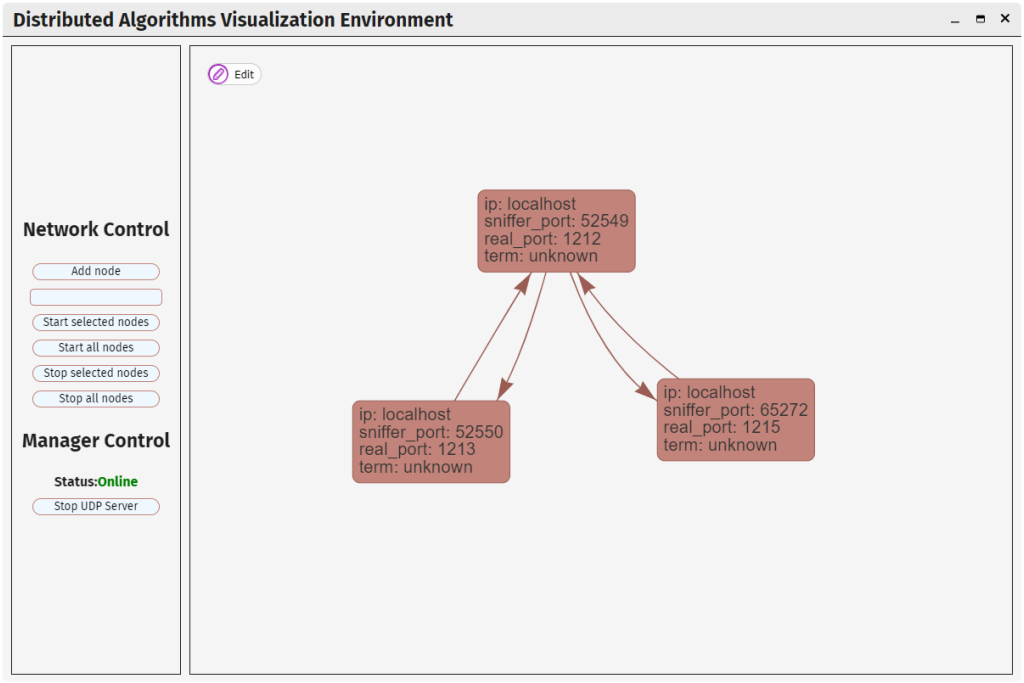

Figura 3.6: Resultado de añadir todas las conexiones

Ahora que se han añadido los nodos y conexiones deseados, se puede proceder a iniciar la ejecución de cada uno de los nodos. Para ello se pulsará sobre un nodo para seleccionarlo, y posteriormente se pulsará el botón "*Start selected nodes*". Seguidamente se seleccionarán los nodos restantes y se pulsará el mismo botón de nuevo. Esto provocará que el primer nodo sea líder puesto que comenzará la ejecución antes que los demás. A partir de este momento, los mensajes entre nodos se mostrarán como círculos azules recorriendo los vértices. Como es lógico, en la imagen no se muestra la animación. Cabe destacar que el campo *term* cuyo valor antes era desconocido, ahora se conoce puesto que se extrae de los mensajes. El resultado final se puede apreciar en la figura 3.7.

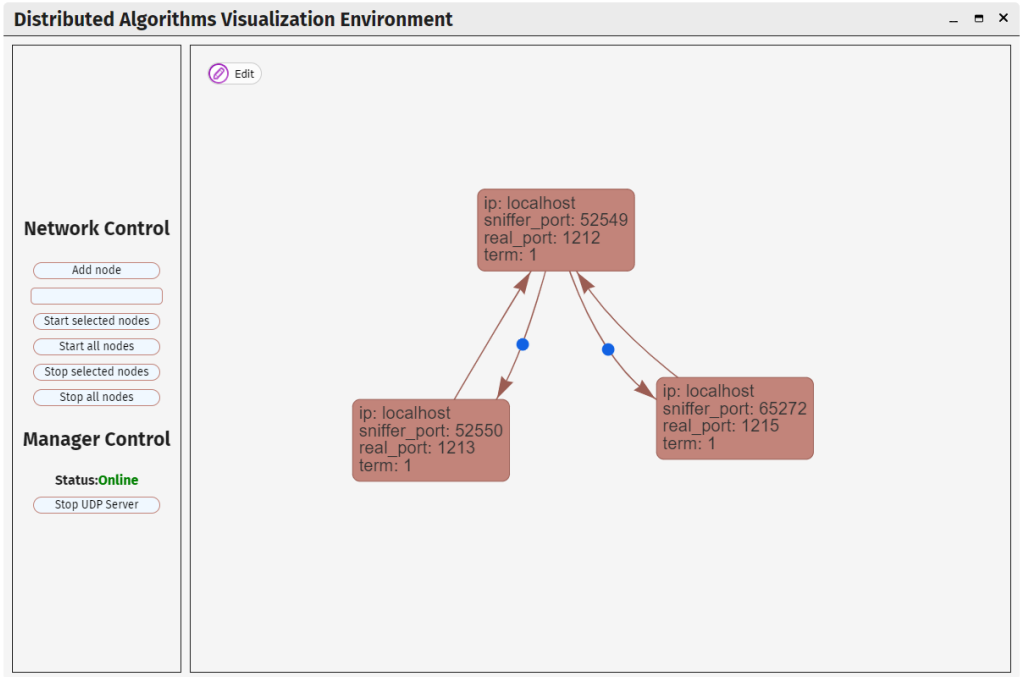

Figura 3.7: Visualización del tránsito de mensajes

Cabe destacar que no es necesario añadir todos los nodos y las conexiones antes de arrancar la ejecución de estos. Se pueden añadir sin problema durante la ejecución. Un ejemplo de esto se muestra en las figuras 3.8 y 3.9, donde se añade otro nodo y se conecta a uno de los nodos existentes.

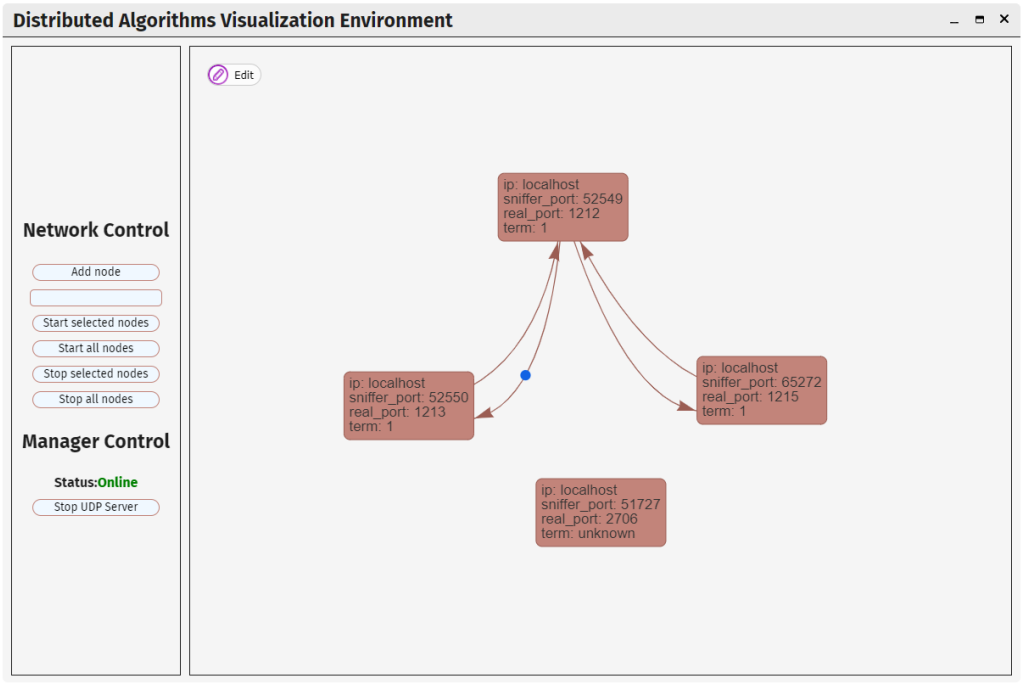

Figura 3.8: Añadir nodo nuevo durante la ejecución de otros nodos

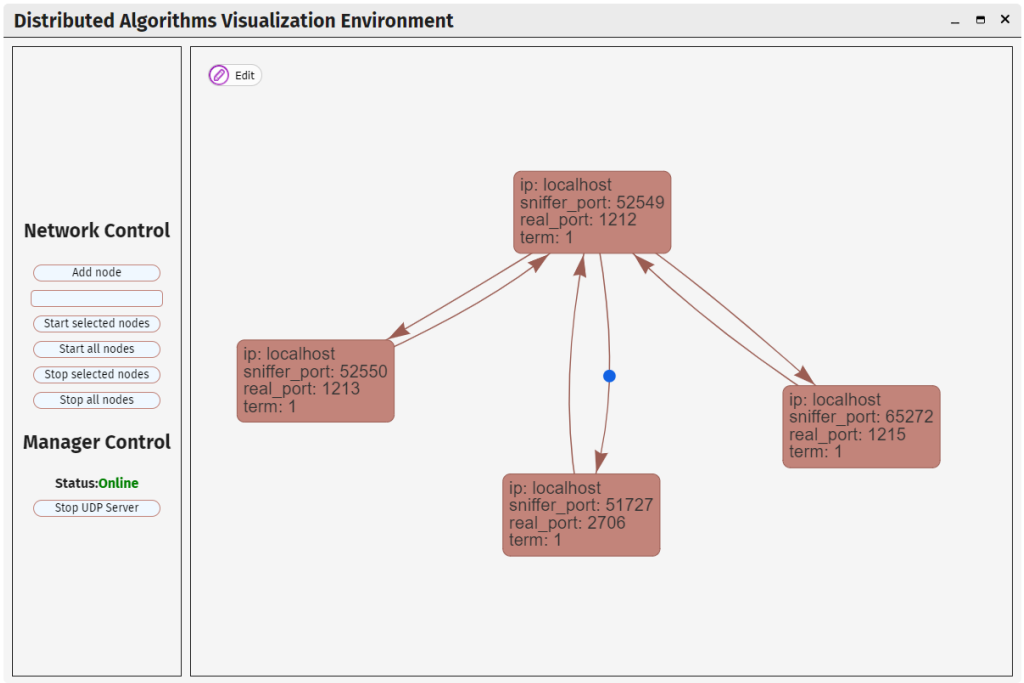

Figura 3.9: Visualización del tránsito de mensajes con el nuevo nodo

3.6. Comentarios sobre la implementación

Como se ha descrito en la introducción, uno de los objetivos de este proyecto es proporcionar una herramienta que permita visualizar algoritmos distribuidos, permitiendo la posibilidad de intercambiar el algoritmo a visualizar. Tanto la arquitectura propuesta como la implementación permiten la visualización de otros algoritmos, sin embargo hay ciertos aspectos que es necesario tener en cuenta. No existe ningún formato universal de mensajes entre nodos de cualquier algoritmo distribuido, por tanto hay ciertos parámetros que se tienen que modificar para admitir el nuevo algoritmo.

- En primer lugar, se debe definir el formato de los mensajes para modificar debidamente las funcionalidades de la interfaz que muestran información sobre estos. Estas modificaciones se harían en el proceso *manager*, dado que es el proceso que modifica la visualización del algoritmo.
- También es necesario modificar el proceso capturador de tal manera que se comunica de la forma adecuada con el nuevo algoritmo. En el caso de esta implementación, el algoritmo se controla por la entrada estándar, pero podría darse el caso en el control se llevase a cabo desde un dispositivo o desde un *socket*.

En lo restante de esta sección se comentarán tanto los requisitos para ejecutar el sistema, como las dependencias de cada módulo.

3.6.1. Dependencias

Para las máquinas que ejecuten el algoritmo, se requieren las siguientes dependencias:

- *Go* - versión 1.17+. Para soportar la ejecución del algoritmo *Raft*.
- *Python* - versión 3.9+. Para ejecutar el proceso capturador.
- Sistema operativo: basado en *Linux*. Esta restricción viene de que la función `sniffer` del proceso capturador requiere componentes de la librería `socket` de *Python* que no están disponibles en otro tipo de sistemas operativos.

En cuanto a la ejecución de la interfaz, se requieren las siguientes dependencias:

- *Node.js*.
- *electron*. Este paquete se puede instalar con el administrador de paquetes de *Node.js*, con el mandato `npm install –save-dev electron`.

3.6.2. Ejecución del entorno

La ejecución del entorno se divide en dos partes. Por una parte la ejecución de las instancias del proceso *capturador*, y por otra la ejecución de la aplicación de escritorio. Cada instancia del proceso capturador se inicia ejecutando el mandato `python sniffer.py` desde el directorio raíz del proyecto. La interfaz se puede ejecutar de dos formas: o bien desde el ejecutable generado por *electron*, o con el mandato `npm run start` desde el directorio raíz de la implementación de la interfaz.

Capítulo 4

Análisis de impacto

En este capítulo se analizará primero el impacto de los objetivos que se han logrado con este proyecto, clasificándolos según el ámbito al que afectan, y posteriormente se relacionarán con los Objetivos de Desarrollo Sostenible de la ONU.

4.1. Impacto personal

A nivel personal el impacto de este Trabajo de Fin de Grado se ve reflejado sobre todo en la cantidad de conocimientos adquiridos durante la realización del mismo. Estos conocimientos no se basan únicamente en el uso de tecnologías nuevas, si no también en la experiencia que ha supuesto implementar todos los componentes.

4.2. Impacto empresarial y económico

La implementación propuesta es de utilidad para los trabajadores de las empresas que implementan sistemas distribuidos, puesto que facilita el proceso de depuración. Esto implica que los trabajadores pueden completar las tareas de este tipo con mayor velocidad, lo cual tendría una repercusión positiva sobre la empresa.

4.3. Impacto cultural

Además de facilitar la labor de los programadores de aplicaciones distribuidas, este proyecto también mejora la calidad de la docencia en las asignaturas de esta materia. Disponer de un recurso que facilite la explicación de los algoritmos de la materia supone que los estudiantes tendrán más facilidades para comprenderlos.

4.4. Relación con los Objetivos de Desarrollo Sostenible

En 2015 el Ministerio de Derechos Sociales publicó la *Agenda 2030* [22], basada en los Objetivos de Desarrollo Sostenible [23] de la ONU, cuya finalidad es mejorar las condiciones de vida de los ciudadanos. Su nombre se deriva de que en los próximos 15 años (a partir de 2015) el Gobierno debe adoptar medidas para luchar contra el cambio climático y erradicar la pobreza, entre otros. La *Agenda 2030* recoge en 17 apartados los diferentes Objetivos de Desarrollo Sostenible.

Analizando los objetivos del proyecto, se puede deducir que están estrechamente relacionados con el punto "Educación de calidad" de la *Agenda 2030*. Uno de los objetivos principales de este Trabajo de Fin de Grado es proporcionar una herramienta que facilite la docencia en la asignatura de Sistemas Distribuidos y facilite la depuración el ámbito de la programación distribuida. Esto por una parte implicaría una docencia de mayor calidad en las asignaturas de esta materia, y por otra parte también mejoraría el desarrollo de aplicaciones distribuidas. Este último aspecto también está relacionado con el punto "Trabajo Decente y Crecimiento Económico" de los objetivos de Desarrollo Sostenible, dado que la solución propuesta en este proyecto es una herramienta de apoyo para los programadores de aplicaciones distribuidas, lo cual contribuye a un trabajo de mejor calidad.

Capítulo 5

Conclusiones y trabajo futuro

En este último capítulo se describen las conclusiones personales con respecto a la realización de este proyecto. Recapitulando los objetivos del proyecto indicados en la introducción, se pretende implementar una aplicación que permita visualizar una ejecución real de un algoritmo distribuido, permitiendo interactuar con los nodos. Además de ofrecer la posibilidad de intercambiar fácilmente el algoritmo a visualizar. Como se ha justificado en los apartados del desarrollo, el diseño y la implementación cumple con estos tres requisitos del proyecto.

Dejando a un lado los objetivos mencionados en la introducción, este proyecto también ha supuesto un reto en otros aspectos. Para empezar, todos los lenguajes de implementación excepto *Python* eran desconocidos. La implementación de todos los módulos ha supuesto aprender *HTML*, *CSS*, *JavaScript* y *Go* en unos 4 meses. Por lo general ha sido bastante desafiante, sobre todo la parte del desarrollo *web*. Sin duda la mayoría de los problemas surgidos durante el desarrollo han sido causados por la falta de experiencia de programación en estos lenguajes. Además de aprender sobre lenguajes concretos, también se han empleado herramientas y *frameworks* de estos, por ejemplo *electron* [21] o *Node.js* [24], que también requieren un periodo de formación.

Por otra parte, la formación en estos lenguajes de programación supone un buen complemento para el Currículum Vitae, puesto que son tecnologías muy demandadas en el mercado.

5.1. Trabajo futuro

Uno de los factores más limitantes en el desarrollo de este proyecto ha sido la cantidad de tiempo que se ha tenido que dedicar a la formación previa. Sin embargo, pese a que se han completado los objetivos principales del proyecto, no se han implementado varias funcionalidades que serían de gran utilidad.

Un ejemplo sería la posibilidad de pausar la ejecución de un nodo, simulando la ejecución lenta de un proceso. Esta funcionalidad no se ha implementado puesto que es complejo pausar la ejecución del algoritmo sin alterar el código

fuente de este. Una posible solución sería ejecutar el proceso en un contenedor que permita pausar temporalmente la ejecución. Otra funcionalidad interesante sería ralentizar o pausar el envío de mensajes, de tal forma que el nodo origen percibe que se ha enviado el mensaje, pero el nodo destino no lo recibe.

Por otra parte se podrían mejorar o completar otro tipo de aspectos, algunos de estos son los siguientes.

1. En la implementación de *Raft*, completar la replicación de *log*, y publicar la solución en la lista oficial de implementaciones [1].
2. Mejorar la apariencia de los nodos en la interfaz, cambiando el formato del texto.
3. Añadir iconos para que los controles sean más claros.
4. Completar la documentación sobre el uso de la aplicación.

Bibliografía

[1] D. Ongaro. The raft consensus algorithm. [Online]. Available: https://raft.github.io

[2] B. Johnson. Raft - understandable distributed consensus. [Online]. Available: http://thesecretlivesofdata.com/raft

[3] J. Martin. Raft distributed consensus algorithm visualization. [Online]. Available: http://kanaka.github.io/raft.js

[4] Y. Moses, Z. Polunsky, A. Tal, and L. Ulitsky, "Algorithm visualization for distributed environments," *Journal of Visual Languages and Computing*, vol. 15, no. 1, pp. 97–123, 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1045926X03000569

[5] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. USA: USENIX Association, 2014, p. 305–320.

[6] Go programming language. [Online]. Available: https://go.dev

[7] G. Lee, X. Li, and A. Romano. Distributed reliable key-value store for the most critical data of a distributed system. [Online]. Available: https://github.com/etcd-io/etcd

[8] E. Bendersky. Eli bendersky's raft implementation. [Online]. Available: https://eli.thegreenplace.net/2020/implementing-raft-part-0-introduction

[9] Go channels. [Online]. Available: https://golangdocs.com/channels-in-golang

[10] Go structs. [Online]. Available: https://golangdocs.com/structs-in-golang

[11] Packet capture library. [Online]. Available: https://wiki.wireshark.org/libpcap

[12] Python wrapper for tshark, allowing python packet parsing using wireshark dissectors. [Online]. Available: https://pypi.org/project/pyshark

[13] A simplified object-oriented python wrapper for libpcap. [Online]. Available: https://pypcap.readthedocs.io/en/latest

[14] Python programming language. [Online]. Available: https://www.python.org

[15] Python socket library. [Online]. Available: https://docs.python.org/3/library/socket.html#socket.AF_PACKET

[16] Javascript programming language. [Online]. Available: https://www.javascript.com

[17] E. DeBill. Module count of programming languages. [Online]. Available: http://www.modulecounts.com

[18] A dynamic, browser based visualization library. [Online]. Available: https://visjs.org

[19] A web framework for rapidly building interactive diagrams. [Online]. Available: https://gojs.net/latest/index.html

[20] M. Franz, C. T. Lopes, G. Huck, Y. Dong, O. Sumer, and G. D. Bader, "Cytoscape.js: a graph theory library for visualisation and analysis," *Bioinformatics*, vol. 32, no. 2, pp. 309–311, 09 2015. [Online]. Available: https://doi.org/10.1093/bioinformatics/btv557

[21] C. Zhao and K. Sawicki. A framework for building desktop applications using javascript, html, and css. [Online]. Available: https://www.electronjs.org

[22] Estrategia de desarrollo sostenible 2030. [Online]. Available: https://www.mdsocialesa2030.gob.es/agenda2030/index.htm

[23] Objetivos de desarrollo sostenible. [Online]. Available: https://www.un.org/sustainabledevelopment/es

[24] A javascript runtime environment built on chrome's v8 javascript engine. [Online]. Available: https://nodejs.org/en

[25] L. Lamport. Paxos algorithm. [Online]. Available: https://www.cs.yale.edu/homes/aspnes/pinewiki/Paxos.html

Anexos

Anexo I - Algoritmo *Raft*

En materia de Sistemas Distribuidos, uno de los aspectos más dificultosos es mantener la consistencia de los datos en diferentes máquinas. Para lograr esto, se han diseñado numerosos algoritmos, como por ejemplo *Paxos* [25] o *Raft* [1]. Para este proyecto se ha escogido la implementación de *Raft* puesto que la implementación, que no deja de ser sencilla, es más sencilla que otros algoritmos, y además la implementación del algoritmo a visualizar no es el objetivo principal. *Raft* es un algoritmo que tiene dos objetivos:

- Elegir un líder en un conjunto de nodos.
- Mantener información (*logs*) replicada en múltiples máquinas.

Este algoritmo garantiza la integridad de la información y también ofrece resistencia a comportamiento anómalo de nodos o caída de estos. Los autores de este algoritmo tuvieron como objetivo principal diseñar un algoritmo cuyo funcionamiento sea simple. Previamente a su invención, existían otros algoritmos de este tipo, por ejemplo *Paxos* [25], pero la complejidad de estos impedía que se emplearan extensivamente.

Los nodos que ejecutan el algoritmo *Raft* pueden pertenecer a tres estados: *leader* (líder), *candidate* (candidato) y *follower* (seguidor). Los nodos comienzan su ejecución en el estado *follower*, y dependiendo del estado en el que se encuentre un nodo, ejecuta determinadas operaciones. Por otra parte, cada nodo guarda el *term* en el que se encuentra. Este valor sirve, entre otras cosas, para determinar la antigüedad de los datos de un nodo.

Estado *leader*

Este estado es el más sencillo, dado que lo único que realiza el nodo es enviar mensajes de tipo *AppendEntries* a los nodos vecinos cada varias décimas de segundo. A este intervalo de tiempo se le denomina "*leader heartbeat timeout*". Estos mensajes contienen las nuevas entradas de *log* que recibe el algoritmo para replicar en todos los nodos. Un nodo deja de ser líder cuando recibe mensajes de otros nodos con un *term* superior. Este escenario significa que el nodo guarda datos antiguos.

Estado *follower*

Los nodos en el estado *follower* reciben continuamente mensajes de tipo `AppendEntries` del nodo líder. También cuentan con otro tiempo de *timeout* denominado "*follower timeout*". Este tiempo determina el tiempo máximo que puede transcurrir entre dos recepciones de mensajes. Cuando un nodo *follower* deja de recibir mensajes de tipo *AppendEntries* significa que, o bien el líder ha dejado de funcionar, o bien se ha perdido el contacto. Independientemente de la causa del problema, el nodo *follower* pasará al estado *candidate* iniciando una votación. Se podría decir que los nodos en este algoritmo quieren a toda costa convertirse en líderes. Cabe destacar que el tiempo de *timeout* del líder (*leader heartbeat timeout*) debe ser inferior al tiempo de *timeout* del *follower* (*follower timeout*). Esto es necesario porque si no fuera así, el nodo líder no enviaría a tiempo mensajes `AppendEntries` a los nodos vecinos, llevándose a cabo una elección de líder nueva en cada instante.

Además de los mensajes de tipo `AppendEntries`, también se pueden recibir mensajes de tipo `RequestVote`. Estos mensajes implican que el nodo acepta la petición de la votación respondiendo con `GrantVote` si no ha votado ya en el *term* actual.

Por otra parte, si un proceso *follower* recibe mensajes de otro nodo con un *term* superior, se convierte seguidor del nuevo *term*.

Estado *candidate*

Cuando un nodo *follower* inicia una votación, transita al estado *candidate*, incrementando el *term* actual y enviando mensajes `RequestVote` a los nodos vecinos. Cuando el nodo recibe $\frac{N+1}{2}$ mensajes de tipo `GrantVote`, donde N es el número de nodos vecinos, significa que ha ganado la elección. En este escenario el nodo transita al estado *leader*.

Si en este estado no se reciben mensajes en un intervalo de tiempo inferior al "*follower timeout*", entonces se inicia otra votación nueva incrementando el *term*.

En el caso de que un nodo en el estado *candidate* recibe un mensaje de tipo `AppendEntries` o `RequestVote` con un *term* superior, el nodo pasaría inmediatamente al estado *follower*.

Comentarios sobre el algoritmo

Sobre este algoritmo cabe destacar ciertos aspectos. En primer lugar, dado que cada nodo comienza una votación cada vez que deja de recibir mensajes del líder, es posible que dos nodos en estado *follower* comiencen una votación simultáneamente. Dependiendo del número de nodos del sistema (por ejemplo 4 nodos), se podría dar el caso en el que dos nodos se disputan por ser el líder dado que no reciben suficientes votos, esto iniciaría otra votación y se repetiría el problema indefinidamente. Es por esto por lo que normalmente se establece

aleatoriamente el tiempo "*follower timeout*". Cuando cada nodo tiene este valor distinto, es muy poco probable que ocurra el escenario mencionado.

Por otra parte se podría dar el caso en el que la red distribuida se divide en dos conjuntos de nodos. En este escenario uno de los conjuntos (que contiene el líder previo) seguiría una ejecución normal. Sin embargo el otro conjunto elegiría un nuevo líder con un *term* superior. Si se diera el caso en el que los dos conjuntos se conectaran otra vez, el último líder elegido pasaría a ser el líder del grupo, puesto que tiene un *term* superior.