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 determianado. 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 of the distributed system 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.

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

Vocabulario

En este capítulo se definen los diferentes términos que se emplean a lo largo del documento:

- *manager* (gestor): hace referencia al proceso que gestiona la actividad de mensajes en la interfaz.
- *sniffer*: proceso que captura el tráfico de red de la máquina.
- *log*: se refiere a un registro de información.
- *Main*: función que se ejecuta a lo largo de la vida de un proceso.
- *listener* (escuchador): proceso que espera nuevas conexiones a partir de un *socket*.
- *socket*: recurso del sistema operativo que permite enviar información a otros *sockets*.

Capítulo 3

Desarrollo

En este capítulo se describe detalladamente tanto la arquitectura de la aplicación, justificando cómo se cumplen los requisitos mencionados en la introducción, como la implementación propiamente dicha de cada uno de los elementos. También se incluye un caso de uso con un algoritmo concreto: Raft.

3.1. 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 informacion en tiempo real de los mensajes de los nodos.

3.2. 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 3.1

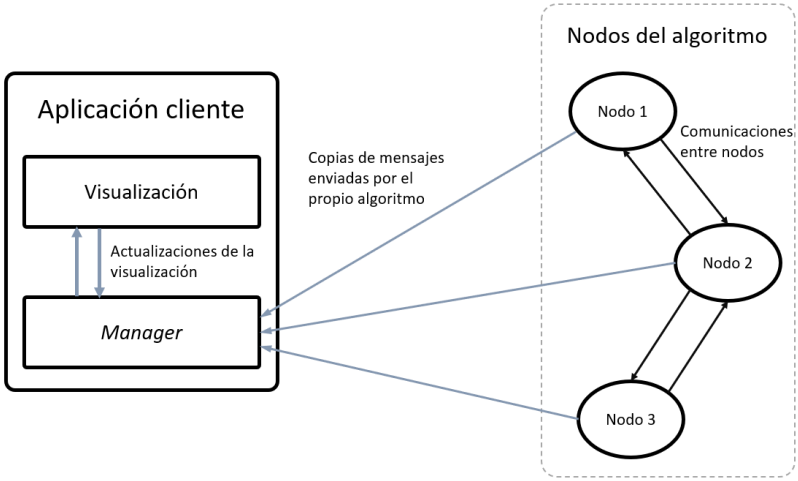

Figura 3.1: Diagrama de la primera arquitectura propuesta

En la Figura 3.1 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 3.2) es muy parecida a la propuesta anteriormente, lo único que cambia es el origen del envío de las copias de mensajes.

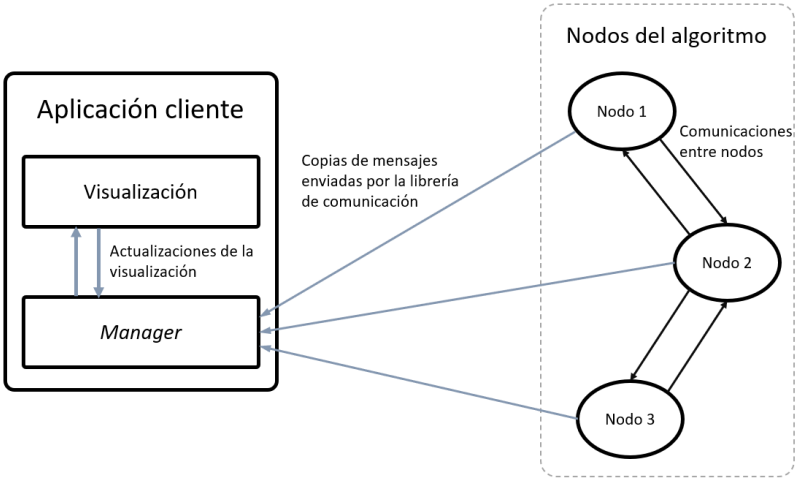

Figura 3.2: Diagrama de la segunda arquitectura propuesta

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 resticción, junto con el hecho de que puede ser complicado modificar la funcionalidad incluída 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 algororitmo de tal forma que los mensajes capturados se reenvían al proceso *manager* para que este actualize 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 distrintos lenguajes, siempre que el medio de comunicación sea un protocolo conocido, por ejemplo TCP. Esta modificación se puede observar en la figura 3.3.

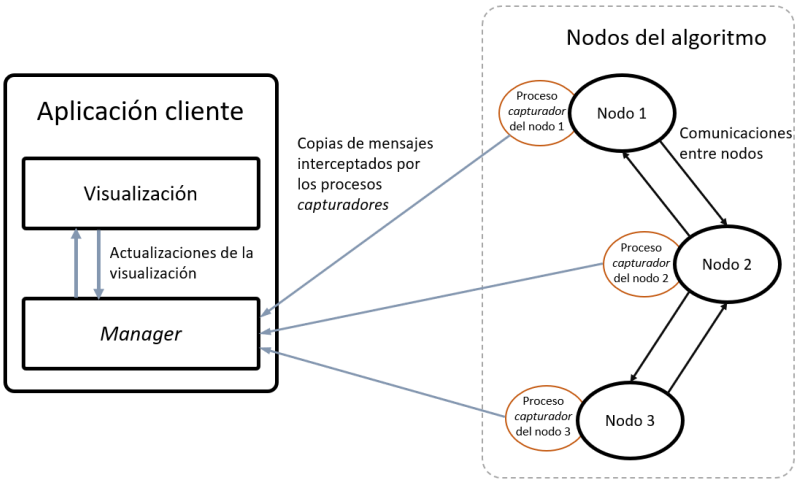

Figura 3.3: Diagrama de la tercera arquitectura propuesta

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 anterior, es que permite un desacoplamiento total del algoritmo. Además de esto, satisface los requisitos que se mencionan en la introducción. A continuación se explicará la implementación detallada de cada uno de los elementos de la arquitectura.

3.3. Implementación del algoritmo `Raft`

Para la implementación del algoritmo Raft, se ha escogido el lenguaje de programación *Go* CITAR, que proporciona numerosas comodidades y ventajas a la hora de programar aplicaciones distribuidas. Por otra parte, la propuesta original de *Raft* 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 TCP. Además de que actualmente existen numerosas implementaciones del algoritmo empleando llamadas a procedimientos remotos, 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.

La descripción simplificada del algoritmo *Raft* puede leerse en el Anexo, o en el artículo original CITAR (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*. La comunicación entre estas funciones se lleva a cabo mendiante canales de Go, 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. 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 }
```

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 reciba una conexión en un tiempo determinado. 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 y envía este objeto al canal. La función que lee y procesa los mensajes del canal es `raft`, que implementa la mayoría de las funcionalidades del algoritmo.

```
func raft(wg *sync.WaitGroup, status *NodeStatus) {
   // Leer eventos del canal
   for event := range status.eventChan {
      switch event.msg.MsgType {
      // 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 inciado 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 es de la votación actual. En el caso de que lo sea, el nodo se convierte en líder ejecutando la función `leaderHeartbeats`. 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.

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*. Sólamente 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 }
```

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 }
```

3.4. Implementación del proceso *capturador*

Como se describe en la sección 3.2, el proceso capturador debe capturar los mensajes de red que envía el nodo y reenviarlos al proceso manager. Además de esto, para permitir una comuincación bidireccional, debe ser capaz de interactuar con el proceso que ejecuta el algoritmo.

- Explicar que no se pueden hacer sockets raw de python en windows
- Poner el codigo de sniffer.py
- Explicar los mensajes de ADD

3.5. Implementación y descripción del cliente

El cliente es la aplicación que ejecuta la interfaz de usuario.

- Explicar eleccion de elecctron/js
- Explicar libreria vis.js
- Explicar restricciones de vis.js
- Explicar funcionamiento

- Explicar la relacion entre los botones y los mensajes enviados al sniffer.

3.6. Implementación del *manager*

- Explicar que esta dentro de la interfaz
- Poner el codigo y explicar

3.7. Caso de uso

En este apartado se demuestra la funcionalidad completa de la aplicación con un algoritmo concreto: Raft. El funcionamiento de este algoritmo se puede encontrar en el Anexo de este documento. A modo de resumen, Raft es un algoritmo de consenso empleado para replicar información en conjunto de nodos.

Capítulo 4

Análisis de impacto

En este capítulo se anañizará 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.

- **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.
- **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.
- **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.

En 2015 el Ministerio de Derechos Sociales publicó la *Agenda 2030*, basada en los Objetivos de Desarrollo Sostenible 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 .[Ed]ucación 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 tabién 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

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

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 CITAR 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 empleado para mantener un *log* replicado en múltiples máquinas. Garantiza la integraidad de la información y también ofrece resistencia a comportamiento anóamlo de nodos o caída de estos. Los creadores de este algoritmo, ... y ..., tuvieron como objetivo principal diseñar un algoritmo cuyo funcionamiento sea simple. Previamente a su invención, existían otros algoritmos de este tipo, por ejemplo Paxos CITAR, 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 *candidate*, 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 antiguedad de los datos de un nodo.

.0.1. 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 varios decisegundos. Estos mensajes contienen las nuevas entradas de *log* que recibe el algoritmo para replicar en todos los nodos.