Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Implementación en Arduino de un protocolo de sincronización mediante consenso

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Sergio Sapiña España

Tutor: Jose Luís Poza Luján

Cotutor: Miguel Rebollo Pedruelo

2014-2015

Resumen

En este proyecto se ha implementado una red distribuida. Ha sido construida con placas Arduino Uno y conectada mediante el uso de shields Ethernet. El objetivo de la red es que los nodos que la componen lleguen al consenso de tres valores predefinidos para cada nodo, simulando una entrada de datos de un sensor de cualquier tipo.

Junto con la topología distribuida de la red, cada nodo sólo conoce la existencia de sus vecinos inmediatos y sólo es capaz de comunicarse con estos. Cada uno de los nodos van a estar compuestos, además de por una placa Arduino Uno con su correspondiente Shield Ethernet, de un led RGB, con el que se mostrará el valor de las variables con las que se está realizando el consenso.

Palabras clave: Arduino, Ethernet, red de consenso, red distribuida, sincronización.

Abstract

In this project, we have implemented a distributed network. The network has been built with Arduino Uno boards connected with Ethernet Shields. The objective of the network is that all the nodes involved find a consensus for three preset values on each node, simulating an input from a generic sensor.

With the distributed topology of the network, we have nodes that only know the existence of their immediate neighbors. Also, they are only able to communicate with them. Each of the nodes will be built, in addition of the Arduino Uno and the Ethernet shield, with a RGB led. The led has to show the values of the variables involved in the consensus.

Keywords: Arduino, Ethernet, consensus network, distributed network, synchronization.

Tabla de contenidos

1. Introducción

1.1 Motivación

El mundo está siendo monitorizado a través de sensores. Nosotros mismos nos convertimos en dispositivos móviles al estar permanentemente conectados lo que da la posibilidad de disponer de información sobre patrones de movilidad, consumo energético o de recursos, siendo ésta la base de las ciudades inteligentes. La principal limitación es que no se puede centralizar el procesamiento de toda la información debido al cambio en la disponibilidad o en la ubicación. Resultan más adecuados métodos distribuidos que permitan conocer en cada dispositivo la información global a partir de la información de los vecinos.

1.2 Objetivos y condiciones

El objetivo principal del presente trabajo es construir un prototipo en Arduino que permita a una red de sensores alcanzar un acuerdo en el valor de una variable común.

Para ello la red ha de contar con las siguientes características:

-Los nodos van a constar de un Arduino Uno con un Shield Ethernet y un led RGB.

-Los nodos sólo conocerán la existencia de sus vecinos inmediatos.

-La estructura de la red ha de ser una malla distribuida. La definición concreta de malla distribuida se va a comentar en el apartado 2.1.1 Redes distribuidas.

-Los nodos han de ser capaces de comunicarse y modificar el valor propio de la variable en relación al valor de sus vecinos.

-Los nodos han de ser capaces de mostrar el valor propio de la variable mediante un led RGB.

1.3 Estructura de la memoria

El contenido de la memoria se ha dividido en 7 capítulos, en los cuales se pretende explicar el trabajo realizado y todo lo que se ha necesitado para ello.

El objetivo del primer capítulo es mostrar en que consiste el proyecto, los objetivos propuestos y la motivación que llevó a la propuesta de este.

En el segundo capítulo vamos a analizar los distintos conceptos que se han utilizado en el proyecto, comparándolos con sus alternativas y mostrar algunos antecedentes.

En el tercero vamos a ver la especificación del proyecto, mostrando y explicando un diagrama de casos de uso y clasificando todas las funciones que se han implementado en el código realizado.

En el cuarto capítulo se va a analizar el diseño que se ha seguido para realizar el proyecto. Empezando por la comunicación entre los nodos, luego el comportamiento interno de los nodos y, por último, los bloqueos controlados que se han implementado para evitar condiciones de carrera.

En el quinto vamos a explicar cómo se ha construido la parte física del proyecto: el montaje de los nodos y la construcción del circuito. Y finalmente analizaremos la ejecución del algoritmo en distintas topologías y sus resultados.

En el sexto capítulo se expondrán las dificultades sufridas y las conclusiones a las que se ha llegado tras la realización del proyecto.

Para terminar, el capítulo siete contiene la bibliografía utilizada durante el proyecto.

2. Contexto

2.1 Conceptos

En este capítulo vamos a explicar con detalle los conceptos más importantes utilizados en el proyecto y su relación con éste. Inicialmente se revisarán las redes distribuidas para contextualizar el entorno de aplicación del proyecto. A continuación se introducirá el concepto de consenso empleado y la plataforma (Arduino) sobre la que se implementará la red de consenso.

2.1.1 Redes distribuidas

Para explicar en qué consiste una red distribuida es necesario primero explicar los tres grandes grupos de topologías que puede tener una red [1] [2].

El primer tipo de topología es la centralizada. En esta topología existen dos tipos de nodos, el nodo central o servidor y los nodos clientes. El nodo central es el encargado de traspasar toda la información a los clientes y estos sólo necesitan conocer la existencia del nodo servidor. La primera figura de la Imagen1 es un ejemplo de este tipo de red.

Esta topología se usa normalmente cuando muchos usuarios han de compartir un recurso, ya sea un servidor o incluso una impresora. Tiene el gran inconveniente de que el nodo central es absolutamente necesario en todas las conexiones, asumiendo la mayor parte de la carga de trabajo de la red y el riesgo de la total inutilización de la red en caso de que este falle. Por otra parte esta red apenas carga a los nodos cliente y no es necesaria la replicación de la información que posee el nodo servidor. Además es muy simple de instalar.

El segundo tipo de topología es la descentralizada. En esta topología también existen nodos cliente de la misma forma que en la topología anterior. Sin embargo en vez de tener un único nodo servidor, cuenta con varios de ellos, repartiendo la carga de clientes. Además de repartir la carga, esta topología cuenta con la ventaja de que, en caso de caída de un servidor, solo una parte de los clientes quedarán incomunicados, dejando la mayor parte de la red funcional. Tenemos un ejemplo de topología descentralizada en la segunda figura de la Imagen1.

La tercera y última forma de red, que ha sido el utilizado en el presente trabajo, es la topología distribuida. En esta topología puede decirse que no existe ningún nodo servidor, ya que todos los clientes tienen la capacidad de ejercer de servidor. En esta red las posibilidades de que algún cliente quede aislado son muy bajas. Sin embargo la complejidad y carga de los clientes aumenta, y hay que distribuir los recursos con los que cuenta este tipo de redes, por lo que suele usarse en redes basadas en información y

no suele tener utilidad en redes basadas en recursos físicos. Este tipo de red se corresponde con la tercera figura de la Imagen1.

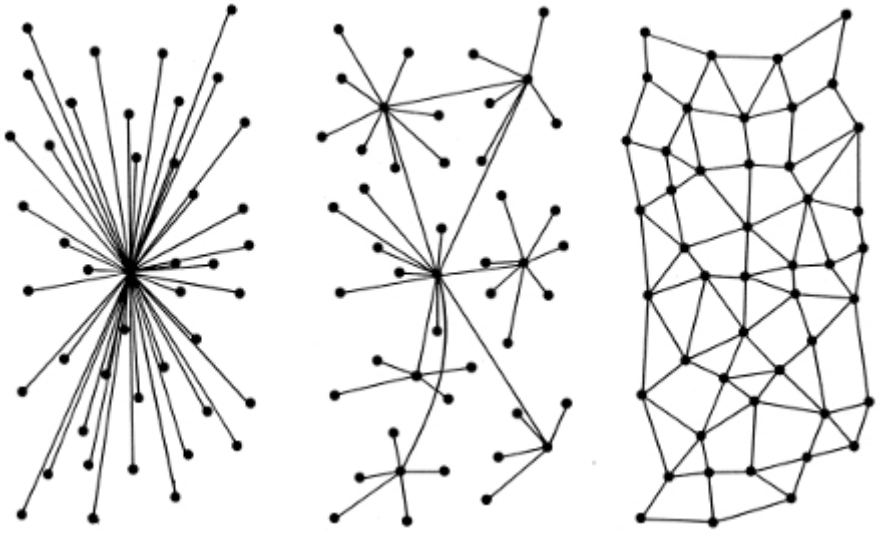

Imagen1: Tipos de topología

Para finalizar hemos dejado la característica más importante en nuestro trabajo y por la cual hemos requerido una red distribuida: la escalabilidad.

Una red escalable significa que es fácilmente ampliable en número de clientes. Esta característica es muy escasa en las redes centralizadas, pues al ir aumentando el número de clientes vamos sobrecargando el nodo servidor, llegando a ser insostenible. Las redes descentralizadas son algo más escalables al tener más servidores, pero el problema sigue ahí. Sin embargo, las redes distribuidas no cuentan un servidor que pueda ser sobrecargado de clientes ya que estos se conectan entre ellos ampliando homogéneamente la red, siendo el límite de escalabilidad la distancia máxima entre dos nodos, que puede llegar a ser un problema.

2.1.2 Redes de consenso

La forma más fácil de explicar este concepto es aplicar el significado de consenso a una red. Según el diccionario de la RAE el significado de consenso es: *"Acuerdo producido por consentimiento entre todos los miembros de un grupo o entre varios grupos"* [3]. Si lo aplicamos a redes compuestas por nodos podemos decir que se traduce en: "Acuerdo producido por consentimiento entre todos los nodos de una red".

Por tanto una red de consenso, como la implementada en este proyecto, es aquella cuyo objetivo es que todos los nodos que la componen negocien respecto a un dato común a todos los nodos, hasta que todos ellos lleguen a un acuerdo, normalmente el promedio del valor teniendo en cuenta o no una ponderación de los nodos de la red.

En nuestro caso, el consenso no va a estar ponderado, ya que todos los nodos tendrán el mismo peso y se va a realizar el consenso de tres valores.

Miguel Rebollo nos muestra en su estudio [4] que, una red de consenso con las características que hemos ido viendo hasta ahora puede ser de utilidad en muchos casos en los que se pueda aplicar la computación distribuida. Los ejemplos expuestos son: monitorización de estructuras, fusión de datos, control ambiental, formaciones autónomas, decisión colaborativa, análisis de sentimientos, reputación on-line y finalmente la gestión de la demanda de energía en la red eléctrica.

2.1.3 Arduino

Arduino es una plataforma hardware de código abierto [5] basada en una placa programable, con distintas versiones y diferentes características físicas. Las placas constan de un determinado número de entradas y salidas, lógicas y analógicas, a las que podemos conectar cualquier componente electrónico, siempre y cuando conozcamos sus características y sean compatibles. Luego podemos introducir un código en la placa que se relacione con estos componentes, o incluso que los relacione entre ellos.

Un gran punto a favor es que el hardware va siendo actualizado con el tiempo y con las nuevas tecnologías. Además siempre podemos contar con distintas opciones de compra, con placas con características específicas que se adaptan a todas las situaciones y, en caso de no hacerlo, se puede añadir uno o varios shields, distribuidos por la misma empresa, que añaden una gran funcionalidad a las placas.

Por si fuera poco, el código libre permite a la comunidad hacer sus modificaciones, ofreciendo la posibilidad de modificar cualquier parte del software. Y gracias a su simplicidad, bajo coste y que está orientado a la enseñanza ha recopilado una gran comunidad de usuarios, que contribuyen con sus proyectos y con sencillos tutoriales [6].

Los tipos más conocidos de placas [7], que no todos, son la Arduino Uno (Imagen2), placa estándar con características equilibradas, la placa Arduino Mega (Imagen3), con más capacidades, entradas y salidas que la anterior, pero de mayor tamaño y la Arduino Nano (Imagen4), con un tamaño reducido pero perdiendo algunas características, como las cómodas conexiones que tren sus dos hermanas mayores.

Todas estas características hacen de Arduino un medio perfecto para iniciarse en el mundo de la electrónica, programación o robótica, como ejemplos más destacados. Además se puede aprender poco a poco consultando la infinidad de tutoriales que hay en la red.

Imagen2: Arduino Uno

Imagen3: Arduino Mega

Imagen4: Arduino Nano

2.1.4 Shields Arduino

Después de aprender cómo funciona Arduino, el siguiente paso es añadirle funcionalidad. La empresa de Arduino ha decidido hacerlo mediante el desarrollo de diversos shields compatibles con Arduino. El uso de estos shields es fácil, conectas los pines del shield a la placa Arduino, instalamos la librería correspondiente si fuera necesario y usamos los pines libres que quedan exactamente igual que con sólo la placa básica. Siempre hay que tener en cuenta que algunas de las entradas/salidas del shield van a ser utilizadas por este, así que hay que consultarlo y evitar usarlas en nuestros programas.

Algunos de los shields más conocidos són el shield Ethernet, el shield Wifi, el shield Xbee y el Motor Shield. En nuestro proyecto hemos usado el shield Ethernet [8] (Imagen5), permitiendo que nuestras placas se conecten a la red mediante ethernet. Este shield suele utilizarse para montar un servidor web, aunque en este proyecto no se ha utilizado para conectarse a internet, sino para comunicar las placas entre ellas.

Imagen5: Arduino Ethernet Shield

2.2 Alternativas

En este capítulo vamos a comparar varias alternativas a las tecnologías usadas y explicar el porqué de la elección realizada. Empezaremos por las alternativas a la plataforma elegida, seguiremos con la versión de la plataforma y a continuación con el shield de comunicación más óptimo. Finalmente compararemos distintas librerías externas diseñadas para el shield elegido y, por último, el protocolo que usaremos en la comunicación.

2.2.1 Arduino

Para la realización de este proyecto contábamos con las placas y los shield Arduino desde el principio ya que la propuesta de la red era específica para Arduino. Sin embargo, en el mercado existen varias alternativas a Arduino [9] que vamos a comentar a continuación.

BeagleBone

BeagleBone, a diferencia de Arduino, es un ordenador completo que ejecuta un sistema operativo con mucha más capacidad que Arduino, pero por un precio bastante más alto, por lo que no vamos a tenerlo en cuenta.

Raspberry Pi

Raspberry Pi es, al igual que BeagleBone, un ordenador completo con sistema operativo. Sin embargo es más barato, aunque no tanto como Arduino. Por el precio y por la capacidad de ejecutar código paralelo es una gran alternativa a considerar frente a Arduino en proyectos como este. Además también está orientado a la enseñanza por lo que los tutoriales son abundantes.

Nanode

Nanode ha surgido como una evolución de Arduino, con el mismo entorno y calidad que Arduino cuenta con un precio más bajo por lo que también es una buena alternativa.

Libelium Waspmote

Por último, Libelium Waspmote está diseñado para crear redes inalámbricas de sensores, lo que sería perfecto como una futura ampliación inalámbrica del presente proyecto.

2.2.2 Arduino Uno

Como hemos comentado en el apartado anterior, el material estaba fijado desde el inicio. Sin embargo existen alternativas dentro de las placas Arduino.

Tenemos las alternativas más potentes que Arduino Uno, como por ejemplo la placa Arduino Mega. Sin embargo esta placa cuenta con muchísimas más entradas y salidas que no vamos a utilizar, un tamaño más grande y un precio ligeramente más alto.

Por parte de las placas más pequeñas como la Arduino Nano, tenemos que no se puede "pinchar" directamente encima el Ethernet Shield y hay que conectarlo mediante cables. Además el tamaño reducido de estas placas no va a ser aprovechado ya que el shield sigue siendo igual de grande, por lo que esta opción no es viable.

2.2.3 Ethernet Shield

Para realizar la comunicación entre los nodos tenemos 2 principales alternativas al Ethernet Shield.

ArduinoWifiShield [10].

El funcionamiento de este shield es muy parecido al Ethernet pero sin los cables que este necesita. Además las librerías son extremadamente simples y parecidas a las de Ethernet por lo que sería una gran idea utilizar estos shields en lugar de los Ethernet si estamos dispuestos a pagar el doble por los shields. La Imagen6 es un ejemplo de shield Wifi.

Imagen6: Shield Wifi

Xbee [11]

Este shield permite conectar y comunicar las placas Arduino sin la necesidad de conectarse a un router de los dos anteriores. Además, al igual que el Wifi Shield, Xbee es inalámbrico. Por contra tenemos que la comunicación es muy diferente a Ethernet, por lo que habría que cambiar muchísimas partes del código y el precio es parecido al del Wifi Shield. En la Imagen7 podemos ver un ejemplo.

Imagen7: Shield Xbee

2.2.4 Librería para el chipset ENC28J60

Para la realización de este proyecto contábamos con varios Ethernet Shield compatibles con las librerías proporcionadas por la IDE de Arduino y varios que, al utilizar el chipset ENC28J60 requerían el uso de librerías externas.

Las tres librerías que se han considerado, teniendo en cuenta las sugerencias del siguiente artículo [12], son las siguientes:

Library ETHER_28J60

Pros:

-Realmente simple y facil de utilizar.

-Muy compacta, Arduino Uno tiene memoria bastante limitada por lo que este punto es una gran baza a favor.

Contras:

-El código no es compatible con el que utiliza la librería por defecto del IDE Arduino. Esto es realmente problemático, ya que en la red final utilizada sólo la mitad de los nodos iban a utilizar esta librería. Si se hubiera elegido esta librería se habrían tenido que desarrollar dos códigos completamente diferentes y, por tanto, doblado la cantidad de código a editar con cada modificación.

Library EtherCard

Pros:

-La más completa y con más funcionalidad de las 3.

Contras:

-Más extensa que ETHER_28J60, no prohibitivo pero es arriesgado usar mucha memoria.

-Muy compleja, ordenes simples como enviar un mensaje se vuelven complejas debido a su alta funcionalidad. No es necesario en este proyecto.

-El código tampoco es compatible con el que utiliza la librería por defecto del IDE Arduino.

Library UIPEthernet

Pros:

-Completamente compatible con la librería por defecto del IDE Arduino.

-La mayoría de ejemplos en la red también son compatibles con esta librería, ya que han sido diseñados para la librería por defecto.

Contras:

-Más extensa que ETHER_28J60.

Después de comparar las tres librerías nos decantamos por UIPEthernet. La funcionalidad abarca todo lo necesario para este proyecto y; aunque el tamaño de la librería podría ocasionar problemas de memoria; la posibilidad de realizar sólo un código para todos los nodos simplemente cambiando un include y la cantidad de ejemplos con los que cuenta sobrepasan con creces la desventaja de tener un poco menos de memoria disponible.

2.2.5 TCP

Para la realización de este proyecto también estaba libre la decisión del protocolo de envío de mensajes. Tanto la librería ofrecida por el IDE Arduino como UIPEthernet cuentan con los protocolos TCP y UDP por lo que hemos podido elegir entre ellos.

TCP está orientado a conexión, estableciendo una entre emisor y receptor antes de enviar el mensaje y esperando una confirmación. Proporciona confiabilidad en la red.

UDP en cambio envía un mensaje al destino sin comprobar su disponibilidad y sin comprobar que se ha recibido. A cambio, el protocolo es mucho más ligero y sobrecarga mucho menos la red.

En nuestro caso hemos decidido utilizar TCP por la capacidad de comprobar si un mensaje ha sido recibido correctamente. Como el tráfico en nuestra red no es un problema, podemos cargarlo un poco más a cambio de la fiabilidad que TCP nos proporciona.

2.3 Sistemas similares

En este capítulo vamos a comentar brevemente el proyecto Tangible Networks, , también conocido como TN, un proyecto conjunto entre investigadores de la BCCS y Espen Knoop; y el proyecto kilobot.

Tangible Networks [13] consta de un conjunto de herramientas educativas con el objetivo de ayudar a explicar ideas y el funcionamiento de redes a personas sin formación científica o matemática.

Este conjunto de herramientas consta de nodos programables con código abierto (Imagen8), cuyo núcleo es capaz de parpadear, cambiar de color y emitir sonidos por medio del código instalado. Además estos nodos pueden conectarse entre ellos, y con interruptores o diales (Imagen9). Esto permite modificar en vivo la red, y aprender muchos conceptos sobre las redes simplemente cambiando conexiones y observando que ocurre.

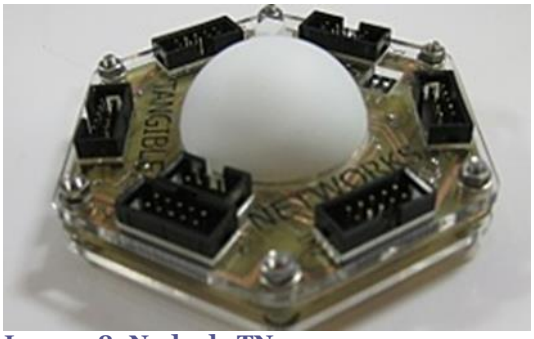

Imagen8: Nodo de TN

Imagen9: Red con TN

El proyecto kilobot [14], desarrollado por investigadores de la universidad de Harvard, consiste en unos robots pequeños (33mm de diámetro) como el que vemos en la Imagen10 capaces de moverse y comunicarse entre ellos a corta distancia. De este modo son capaces de crear una red dinámica variable, cuyos nodos y, por tanto sus conexiones, pueden moverse entrando y saliendo de la red (Imagen11).

Aunque se puede establecer un líder, el proyecto ha sido más enfocado a evitar esta característica y centrado en la inteligencia colectiva, también llamada inteligencia de enjambre. No tener un líder nos genera una escalabilidad de red altísima y este ha sido uno de los requisitos tanto en el proyecto kilobot como en el presente proyecto.

En la web podemos ver algunos comportamientos con los que han experimentado, obteniendo resultados muy interesantes y con muchas posibles aplicaciones, por lo que seguir desarrollando la inteligencia enjambre puede traernos cambios importantes en el futuro.

Imagen10: Kilobot

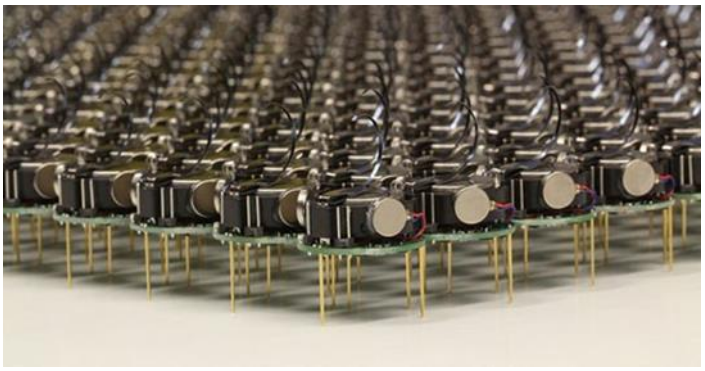

Imagen11: Conjunto Kilobots

3. Especificación

3.1 Diagrama de casos de uso

En este capítulo vamos a ver el comportamiento del sistema respecto a los distintos eventos que van a ocurrir. Es en este capítulo donde también se deberían ver las interacciones del usuario con el sistema, sin embargo en nuestro caso el sistema fluye automáticamente en busca de un estado final, por lo que los únicos actores que influyen al sistema están incluidos en el. Esto es posible mediante varios timeouts que se van a activar periódica y aleatoriamente, desencadenando todos los eventos necesarios para que funcione todo el sistema. Para más información sobre la implementación de los timeouts consultar el apartado 4.3 Bloqueos.

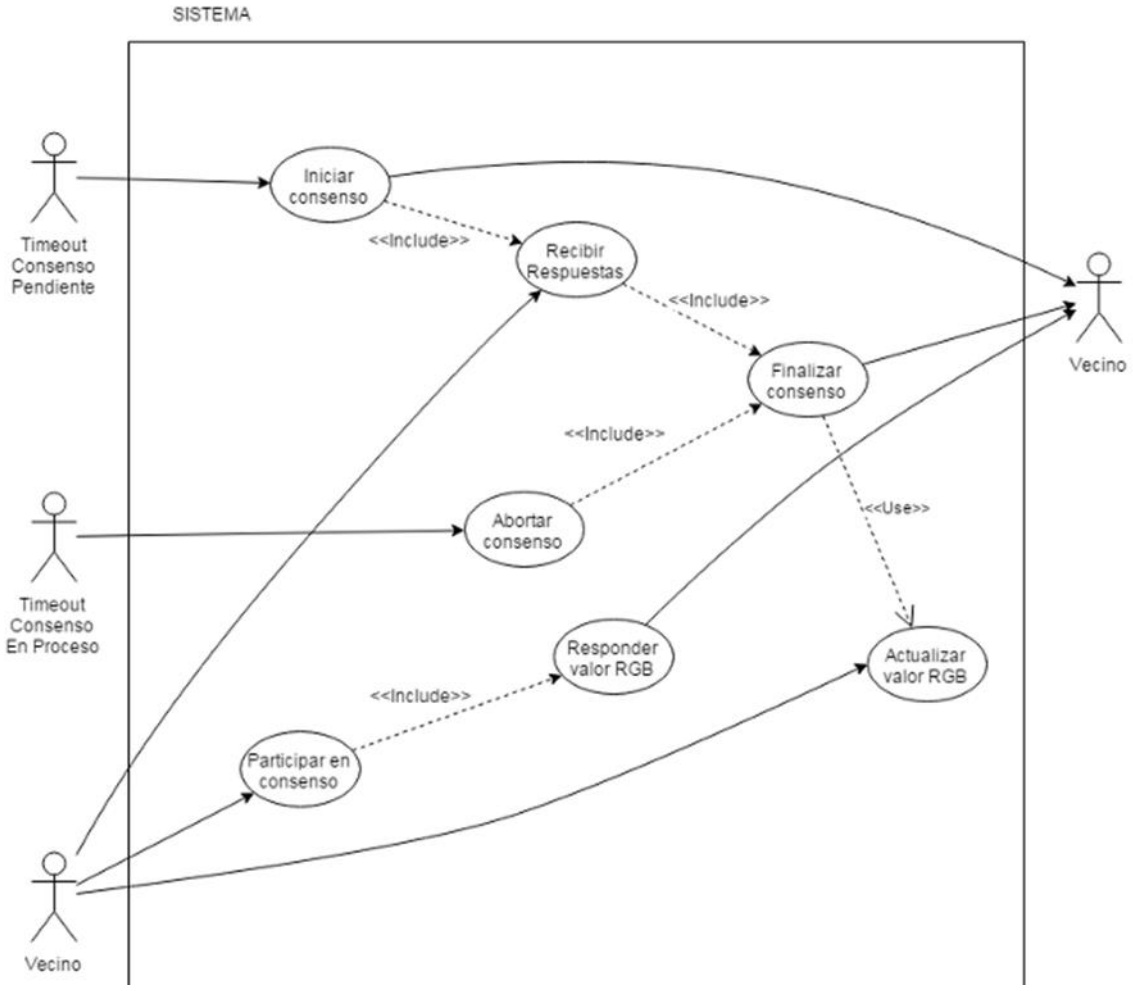

Imagen12: Diagrama de casos de uso

Como podemos ver en el diagrama de la Imagen12, en nuestro sistema vamos a considerar los siguientes actores:

-Timeout Consenso Pendiente: Este primer timeout va a ser el que se active aleatoriamente después de haber realizado, con o sin éxito, un consenso.

-Timeout Consenso en Proceso: Este segundo timeout tiene un tiempo de activación fijo, pero está relacionado con el anterior timeout. Finalizará un consenso activo pasado un tiempo si éste no lo ha hecho en el tiempo esperado.

-Vecino: Este actor representa a todos los posibles nodos conectados al nodo que van a tener la capacidad de enviar y recibir mensajes. Se ha decidido dividir el agente en dos partes para mejorar la claridad del diagrama.

El actor Vecino tiene la particularidad de que se va a comportar exactamente igual que el propio sistema, dado que realmente un Vecino del sistema es otro sistema con la misma funcionalidad. Es por esto que el mismo número de casos de uso que puede generar un Vecino en el sistema es igual al número de casos de uso que son capaces de enviar información a un Vecino. De esta forma se van a conectar los nodos mediante estos tres casos (Imagen13).

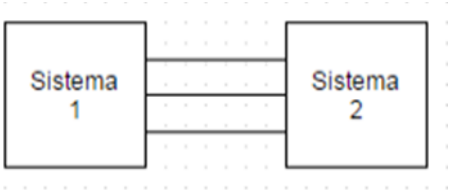

Imagen13: Conexión de nodos

Para aclarar este último concepto podemos decir que:

- Si el sistema ejecuta "Iniciar consenso" en el vecino va a llegar como "Participar en consenso".
- Si el sistema ejecuta "Responder valor RGB" en el vecino va a llegar como "Recibir Respuestas".
- Si el sistema ejecuta "Finalizar consenso" en el vecino va a llegar como "Actualizar valor RGB".

Estas tres comunicaciones entre vecinos van a ser la definición de la comunicación que va a haber entre los nodos mediante cinco tipos de mensajes como se verá en la sección 4.1 Mensajes.

3.2 Funcionalidades

En esta sección vamos a clasificar todas las funciones que se han implementado en los nodos para que realicen correctamente su trabajo. Vamos a agruparlas en tres bloques dependiendo de su funcionalidad.

3.2.1 Funciones de uso general

Estas funciones son básicamente de apoyo a las demás y la mayoría son utilizadas tanto por la parte cliente de los nodos como por la parte servidor.

Nombre: sendMsg
Descripción: Envía un mensaje a un destinatario; en caso de fallar, insiste hasta darse por vencido y aborta el envío.
Entrada: Mensaje, id destinatario
Salida: -
Requisitos: Destinatario accesible, destinatario conectado.

Nombre: creaMsg
Descripción: Crea un mensaje con el formato aceptado por el sistema.
Entrada: Id propia, orden, valor R, valor G, valor B
Salida: Mensaje
Requisitos: -

Nombre: rango
Descripción: Evita valores inadecuados vara evitar fallos.
Entrada: Valor
Salida: Valor entre 0 y 255
Requisitos: -

Nombre: iniServ
Descripción: Inicia el servidor con MAC y IP.
Entrada: MAC, IP
Salida: -
Requisitos: -

Nombre: color
Descripción: Actualiza el valor de las salidas conectadas al led RGB con los valores internos del nodo.
Entrada: valor R, valor G, valor B
Salida: -
Requisitos: -

3.2.2 Funciones del servidor

Este grupo de funciones sólo son utilizadas por la parte servidor de los nodos. Su objetivo es la recepción de mensajes y su procesado, realizando las acciones que estos mensajes indiquen. Algunas acciones requeridas como responder a un mensaje también están incluidas en el servidor.

Nombre: servidor
Descripción: Proceso principal en modo servidor, recoge mensajes recibidos validos.
Entrada: -
Salida: -
Requisitos: Mensaje en el buffer de entrada

Nombre: procesaMsg
Descripción: Recibe un mensaje, lo desglosa y llama a la funcion de procesado pertinente.
Entrada: Mensaje
Salida: -
Requisitos: -

Nombre: sGet
Descripción: Procesa mensajes tipo GET
Entrada: Id solicitante.
Salida: -
Requisitos: Mensaje tipo GET, key libre, consenso no activo

Nombre: sRes
Descripción: Procesa mensajes tipo RES
Entrada: Id solicitante, valor R, valor G, valor B
Salida: -
Requisitos: Mensaje tipo RES, respuestas recibidas < nodos consultados

Nombre: sOcu
Descripción: Procesa mensajes tipo OCU
Entrada: Id solicitante
Salida: -
Requisitos: Mensaje tipo OCU, consenso activo

Nombre: sPut
Descripción: Procesa mensajes tipo PUT
Entrada: Id solicitante, valor R, valor G, valor B
Salida: -
Requisitos: Mensaje tipo PUT, key bloqueada por solicitante

Nombre: sNot
Descripción: Procesa mensajes tipo NOT
Entrada: Id solicitante
Salida: -
Requisitos: Mensaje tipo NOT, key bloqueada por solicitante

3.2.3 Funciones del cliente

Estas funciones forman parte sólo de la parte cliente de los nodos. Su objetivo está fuertemente relacionado con el inicio de la comunicación con los vecinos, dado que el servidor no es capaz de iniciar una comunicación. También son las encargadas de decidir el curso de los consensos.

Nombre: solicitarConsenso
Descripción: Activa en el nodo el modo consenso, iniciando comunicación con los vecinos accesibles.
Entrada: -
Salida: -
Requisitos: key libre, consenso no activo

Nombre: aplicarConsenso
Descripción: Envia el resultado del consenso a todos los vecinos y finaliza el modo consenso en el nodo.
Entrada: -
Salida: -
Requisitos: consenso activo, mensajes recibidos = nodos consultados

Nombre: difNuevoValor
Descripción: Envia el valor calculado en el consenso a todos los vecinos
Entrada: valor R, valor G, valor B
Salida: -
Requisitos: -

Nombre: devolverKeys
Descripción: Envia mensajes NOT a todos los vecinos.
Entrada: -
Salida: -
Requisitos: Consenso abortado por timeout o consenso finalizado con nuevo valor igual a valor anterior

Nombre: finConsenso
Descripción: Reinicia las variables utilizadas por el nodo en modo consenso.
Entrada: -
Salida: -
Requisitos: -

Nombre: sigConsenso
Descripción: Reinicia el timeout para activar el modo consenso.
Entrada: -
Salida: -
Requisitos: -

4. Diseño

4.1 Mensajes

En este capítulo vamos a explicar cómo se han diseñado el sistema de mensajería desarrollado en el proyecto, ya que la comunicación es una parte esencial en cualquier red. Para ello debe haber un mínimo protocolo que seguir para evitar errores y simplificar el envío y la recepción de la información.

Por tanto se ha decidido que los mensajes cumplan con dos requisitos:

-Todos han de tener el mismo tamaño y estructura.

-El tipo de mensajes va a ser de 5, con un significado propio bien definido.

4.1.1 Estructura

Para definir la estructura de los mensajes se ha tenido en cuenta lo siguiente:

1. Todos los mensajes han de tener la misma longitud. Para simplificar la extracción de los mensajes del buffer de entrada y para filtrar posibles mensajes corruptos, aunque en una red tan controlada es muy difícil que esto llegue a pasar.

2. Todos los mensajes van a tener los mismos campos. Para simplificar al máximo el código encargado de leer los mensajes y interpretar la información recibida.

3. Por tanto todos los mensajes tendrán los siguientes 5 campos:
   -Id: emisor del mensaje.
   -Orden: tipo de mensaje,
   -Valor R: De 0 a 255.
   -Valor G: De 0 a 255.
   -Valor B: De 0 a 255.
4. Los mensajes empezarán por "__" y sus campos estarán separados por "_", para facilitar el desglose de los mensajes y, principalmente, facilitar la lectura del flujo de mensajes durante la etapa de testeo.

5. En algunos mensajes los campos RGB no son necesarios por tanto, y para evitar incumplir los puntos anteriores, estos campos se dejarán a 0.

6. En algunos casos los campos RGB serán valores de 1 o 2 cifras. En estos casos los valores RGB serán rellenados con ceros para cumplir el tamaño especificado.

Con todo esto obtenemos la siguiente estructura para los mensajes:

__N_AAA_RRR_GGG_BBB*

N – ID
A – Orden
R – Valor R
G – Valor G
B – Valor B

*El significado de los campos que tienen 3 letras es que ocupan 3 caracteres en el mensaje.

4.1.2 Tipos de mensaje

Como hemos comentado anteriormente se han desarrollado 5 tipos de mensajes que equivalen a las 5 posibles peticiones que un nodo puede hacer a otro. Son las siguientes:

GET – Petición, reserva la key* y dame tu valor. Este tipo de este mensaje es siempre para que el nodo receptor participe en el consenso que está iniciando el emisor. Sin embargo, como veremos más adelante, esto no siempre va a ser posible.

RES – Petición aceptada, mi valor es el siguiente. Este tipo de mensaje es una respuesta afirmativa al anterior, añadiendo a la respuesta el valor con el que el nodo va a participar en el consenso.

OCU – Petición denegada, mi key* está bloqueada. Este mensaje es el opuesto al anterior. En este caso el nodo ha decidido que no va a participar en el consenso y informa de ello.

PUT – Libera la key*, hay que modificar tu valor. Este mensaje se corresponde con una finalización exitosa del consenso. Al mensaje se le añade el nuevo valor que los que participaron en el deben actualizar.

NOT – Libera la key*, no hay que modificar tu valor. Este mensaje es el contrario al anterior. Se informa a los participantes del consenso que este ha sido abortado, ya sea por algún fallo o por que el consenso no era necesario.

*El uso de las keys está ampliamente explicado en el punto 4.3 Bloqueos.

Los tipos de mensajes pueden organizarse en relación "respuesta de" de la forma mostrada en la Imagen14.

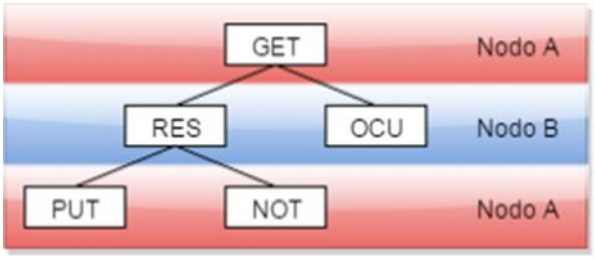

Imagen14: Arbol de mensajes

Por tanto podemos observar que las comunicaciones siempre empiezan con un mensaje tipo GET por parte de un nodo, por ejemplo el nodo A, que ha iniciado un consenso con sus vecinos inmediatos.

A esta petición los vecinos le contestarán con un RES o con un OCU dependiendo de su estado. Si el nodo que ha recibido el GET, por ejemplo el nodo B, está libre, mandará como respuesta un RES junto con los valores RGB locales y se bloqueará en espera de la siguiente orden. En el caso contrario en el que dicho nodo B ya está bloqueado por otro vecino, el nodo responderá con un mensaje OCU finalizando la comunicación con el nodo A.

Si el nodo A recibió un mensaje RES añadirá el valor RGB del nodo B en su consenso y cuando haya obtenido el resultado se lo devolverá al nodo B. Si es necesario que el nodo B modifique el valor el nodo A enviará un mensaje PUT junto con el nuevo valor RGB. Si por el contrario el valor no ha de ser modificado se enviará un mensaje NOT. En ambos casos se da por finalizada la comunicación y se libera la key del nodo A.

Además, el mensaje NOT tiene un uso adicional. Tiene el papel de cancelar una difusión abortada por timeout, devolviendo las keys a todos los nodos que estén involucrados en dicha difusión.

Por seguridad, un nodo solo aceptará un mensaje PUT o NOT si su key está siendo bloqueada por el nodo emisor de ese mensaje, evitando que un nodo se desbloquee si uno de sus vecinos empieza a funcionar de forma anómala.

El significado de los mensajes para el nodo que los ha recibido es el siguiente:

__0_GET_000_000_000

El nodo con identificador 0 ha iniciado un consenso y pide que se participe en él.

__1_RES_123_005_155

El nodo con identificador 1 ha aceptado participar en el consenso que este nodo ha iniciado y participa con el valor R=123, G=5, B=155.

__1_OCU_000_000_000

El nodo con identificador 1 ha rechazado participar en el consenso que este nodo ha iniciado.

__0_PUT_080_055_105

El nodo con identificador 0 ha finalizado el consenso, concluyendo que el nuevo valor para los vecinos es R=80, G=55, B=105.

__0_NOT_000_000_000

El nodo con identificador ha abortado el consenso y por tanto no se va a modificar el valor de los vecinos.

4.1.3 Ejemplos de comunicación

En este capítulo se van a mostrar varios ejemplos para explicar todos los casos que pueden darse al realizar un consenso. En estos ejemplos se muestran 4 nodos. El nodo 1 empieza a hacer un consenso y los nodos 2, 3 y 4 son los vecinos directos del nodo 1.

4.1.3.1 Consenso completo

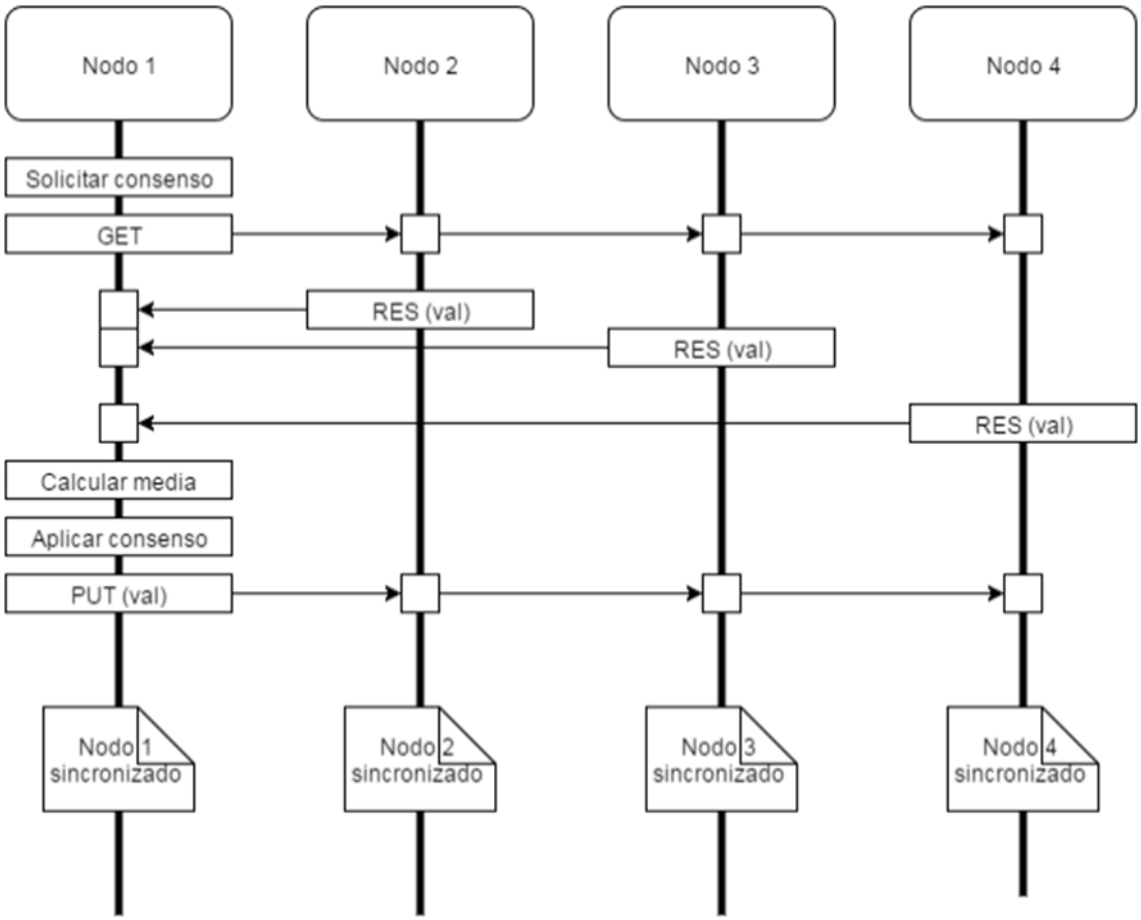

Imagen15: Consenso completo

En este primer diagrama (Imagen15) podemos ver que el Nodo 1 activa Solicitar consenso, por lo que hace una difusión de peticiones GET a todos sus vecinos. Todos ellos reciben el mensaje correctamente y responden confirmando el consenso y comunicando su valor RGB local. A continuación, el Nodo 1 que está en espera de las respuestas, las va recibiendo y procesando una a una. En el momento en que recibe la última, calcula la media y ejecuta Aplicar consenso. Esta función comprueba si es necesario o no hacer una difusión del nuevo valor. En este caso sí que es necesario por lo que manda un mensaje PUT con el nuevo valor a todos los nodos implicados en el consenso, finalizándolo con todos los nodos sincronizados.

4.1.3.2 Consenso innecesario

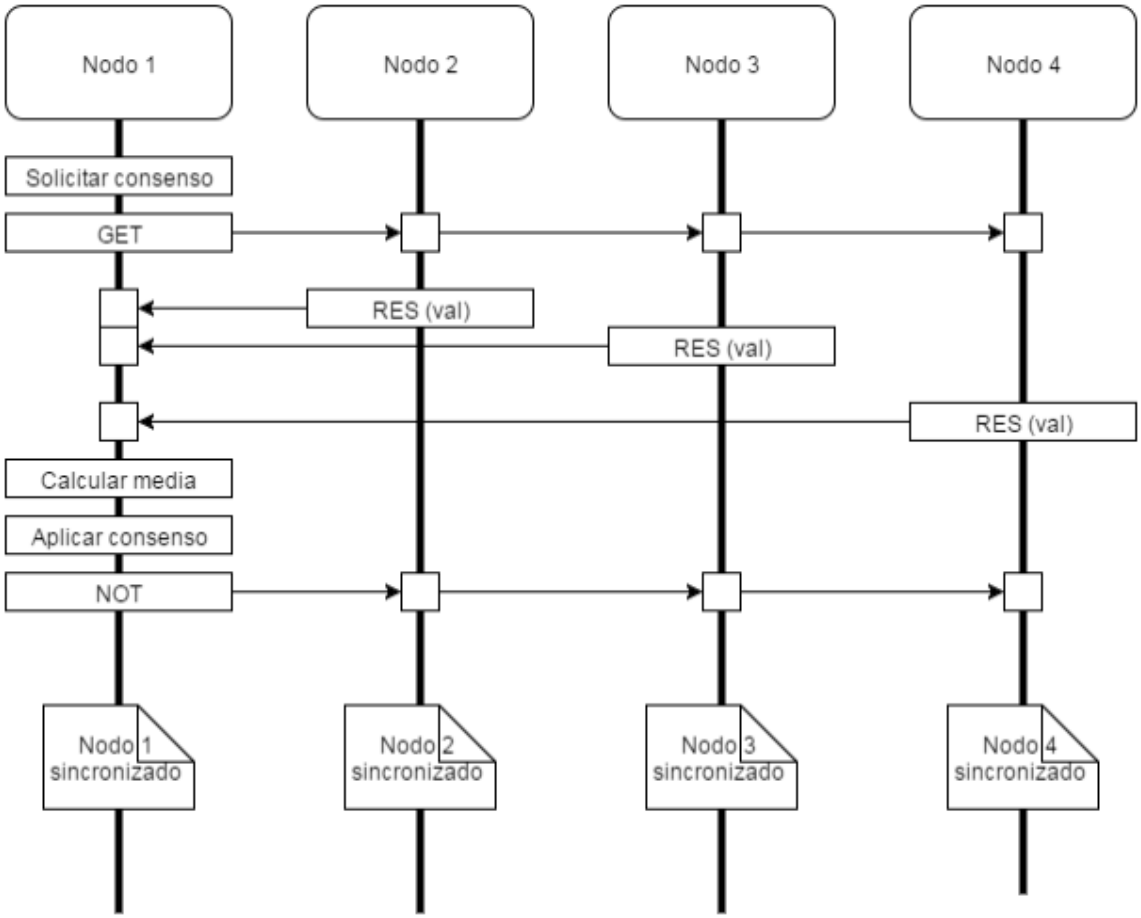

Imagen16: Consenso innecesario

En este segundo diagrama (Imagen16) nos encontramos con el mismo caso que en el diagrama anterior salvo por una diferencia: el segundo mensaje que difunde el Nodo 1 en un mensaje NOT en lugar de un mensaje PUT. Esto se debe a que al realizar la operación para calcular el nuevo valor RGB el Nodo 1 ha considerado que no se ha modificado lo suficiente el valor como para actualizar el de sus vecinos. Sin embargo, no se puede detener la comunicación en ese punto, pues sus vecinos están bloqueados en espera de una respuesta, por lo que una buena opción es enviar un mensaje vacio cuya función es la de liberar a dichos nodos y que se sigan comunicando con sus respectivos vecinos.

4.1.3.3 Consenso parcial

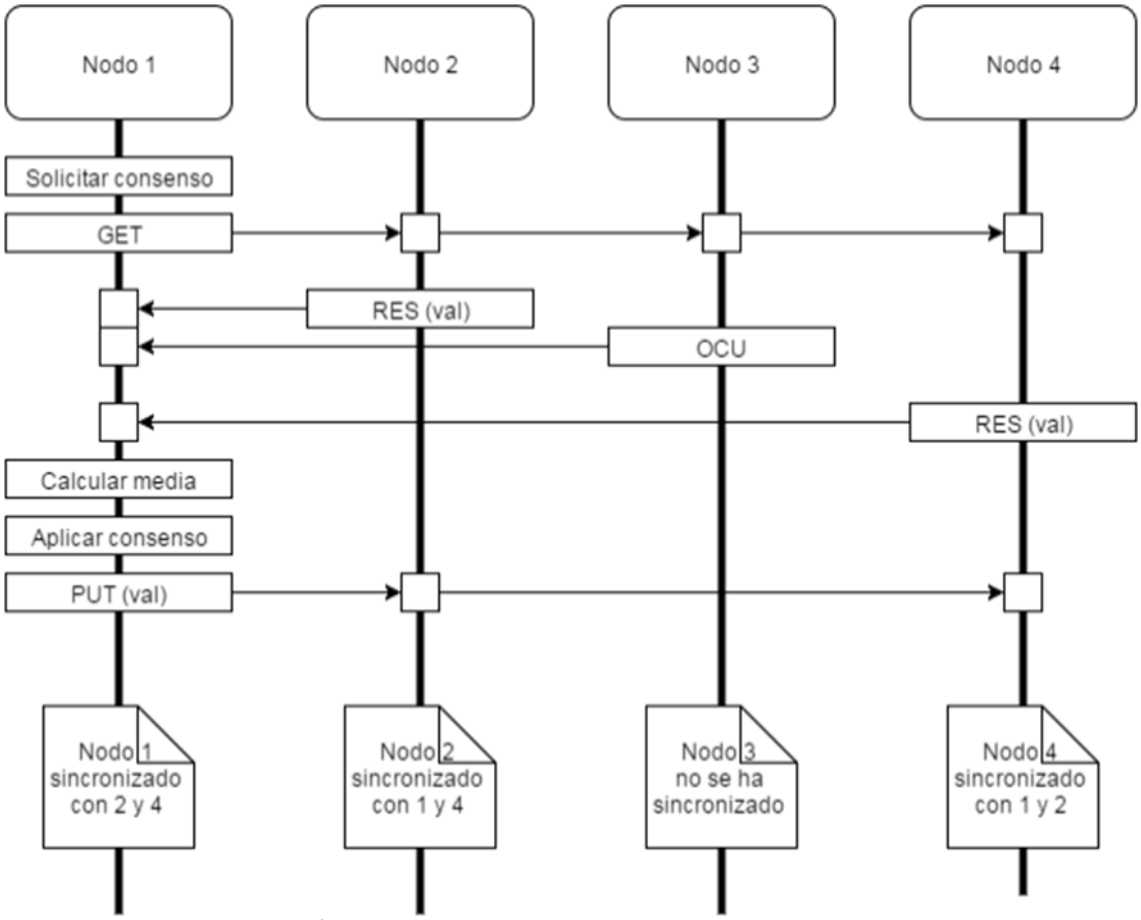

Imagen17: Consenso parcial

En este tercer caso (Imagen17) podemos ver la posibilidad de que un nodo decida no participar en un consenso. Concretamente el Nodo 3 ha respondido a la petición con un mensaje OCU. Esto puede deberse a dos razones, que el nodo haya sido bloqueado por uno de sus vecinos que también está haciendo un consenso o porque el mismo Nodo 3 está intentando realizarlo. En cualquiera de los dos casos el Nodo 3 no debe participar en el consenso pues podría modificarse su valor RGB dando lugar a una condición de carrera. Por tanto una buena solución es que este nodo responda con un mensaje distinto, indicando que no puede participar, y que el Nodo 1 realice el consenso con el resto de vecinos.

Nótese además que, aunque el Nodo 1 mande el mensaje a los 3 vecinos al finalizar el consenso, el Nodo 3 va a desechar ese mensaje pues no ha sido bloqueado por el emisor del susodicho.

4.1.3.4 Consenso abortado

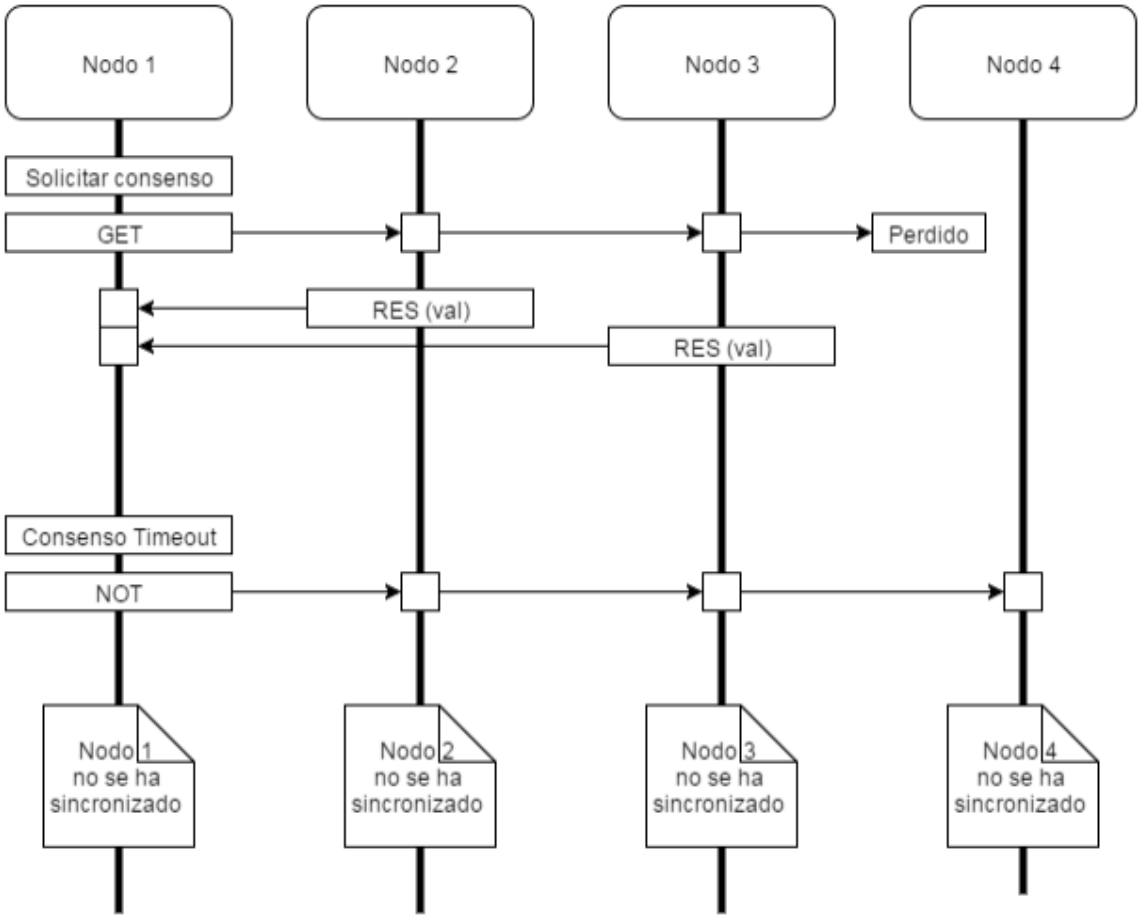

Imagen18: Consenso abortado

En este último ejemplo (Imagen18) contemplamos un caso indeseado que por mala fortuna ocurre impredeciblemente y por ello ha de tratarse con cuidado. En este ejemplo vemos que el mensaje difundido por el Nodo 1 se pierde por causas de fallo en la comunicación. Además no somos capaces de saber si el fallo ha ocurrido antes de que sea procesado por el Nodo 4 o después de que este se haya bloqueado por lo que tenemos que analizar si podemos o no llevar a cabo el consenso con las dos posibilidades.

Caso 1. El mensaje no ha llegado a ser leído por el Nodo 4 por lo que suponemos que este haya caído o simplemente siga funcionando sin haber bloqueado la key, ajeno a que el Nodo 1 ha intentado realizar un consenso. En este caso se podría finalizar el consenso con los otros dos nodos restantes ya que al enviar el mensaje PUT el Nodo 4 lo rechazaría en caso de recibirlo.

Caso 2. El mensaje ha sido procesado por el Nodo 4 pero se ha perdido la respuesta que este ha enviado. En este caso el Nodo 4 si está preparado para modificar su valor, sin embargo el Nodo 1 no tiene información suficiente para realizar el consenso. Podría realizarse un consenso parcial añadiendo complejidad a la parte servidor de los nodos pero por problemas de espacio de memoria esta opción ha sido descartada.

Por tanto llegamos a la conclusión que la mejor opción es abortar el actual consenso, devolver las keys a todos los vecinos, esperar un tiempo y repetir el consenso.

4.2 Nodos

En este capítulo, en la que ya sabemos qué es lo que están haciendo los nodos respecto a comunicarse entre ellos, vamos a ver cómo funciona cada nodo individualmente. Aunque hay que tener en cuenta que los nodos implementados no tienen ningún sentido sin la capacidad de comunicarse con el resto. Empezaremos viendo el funcionamiento general de las iteraciones de los nodos y luego detallaremos cada una de las acciones que lo componen.

4.2.1 General

Recordando que el funcionamiento de las placas Arduino está basado en una parte estática y a continuación un "loop" infinito, vamos a empezar viendo de forma general cómo se suceden las distintas acciones que realizan los nodos y, más adelante, profundizaremos más en que hace exactamente cada una.

En este diagrama (Imagen19) observamos fácilmente cuales son las acciones que hace cada nodo continuamente desde que es puesto en ejecución. El grupo de bloques del 3 al 6 van a ser explicados en detalle en las siguientes secciones.

1. Crear el servidor y crear el cliente

Esto es necesario para que haya una comunicación entre los nodos. Sólo hay que hacerlo una vez por lo que se hace antes de que empiece el bucle infinito.

2. Loop

Empieza el bucle. A partir de este punto todas las acciones se van a repetir mientras el nodo esté activo.

3. Servidor

Este bloque es el encargado de leer y procesar los mensajes que reciba el nodo del exterior. Ha de lanzarse siempre ya que, aunque el nodo esté bloqueado, ha de ser capaz de contestar a sus vecinos para evitar errores.

4. Solicitar consenso

Este bloque sólo se va a activar si se reúnen ciertas condiciones. Estas son que el timer de consenso pendiente haya llegado a 0, indicando que se necesita consultar a los vecinos y, además, que el propio nodo no tenga la llave bloqueada por un vecino.

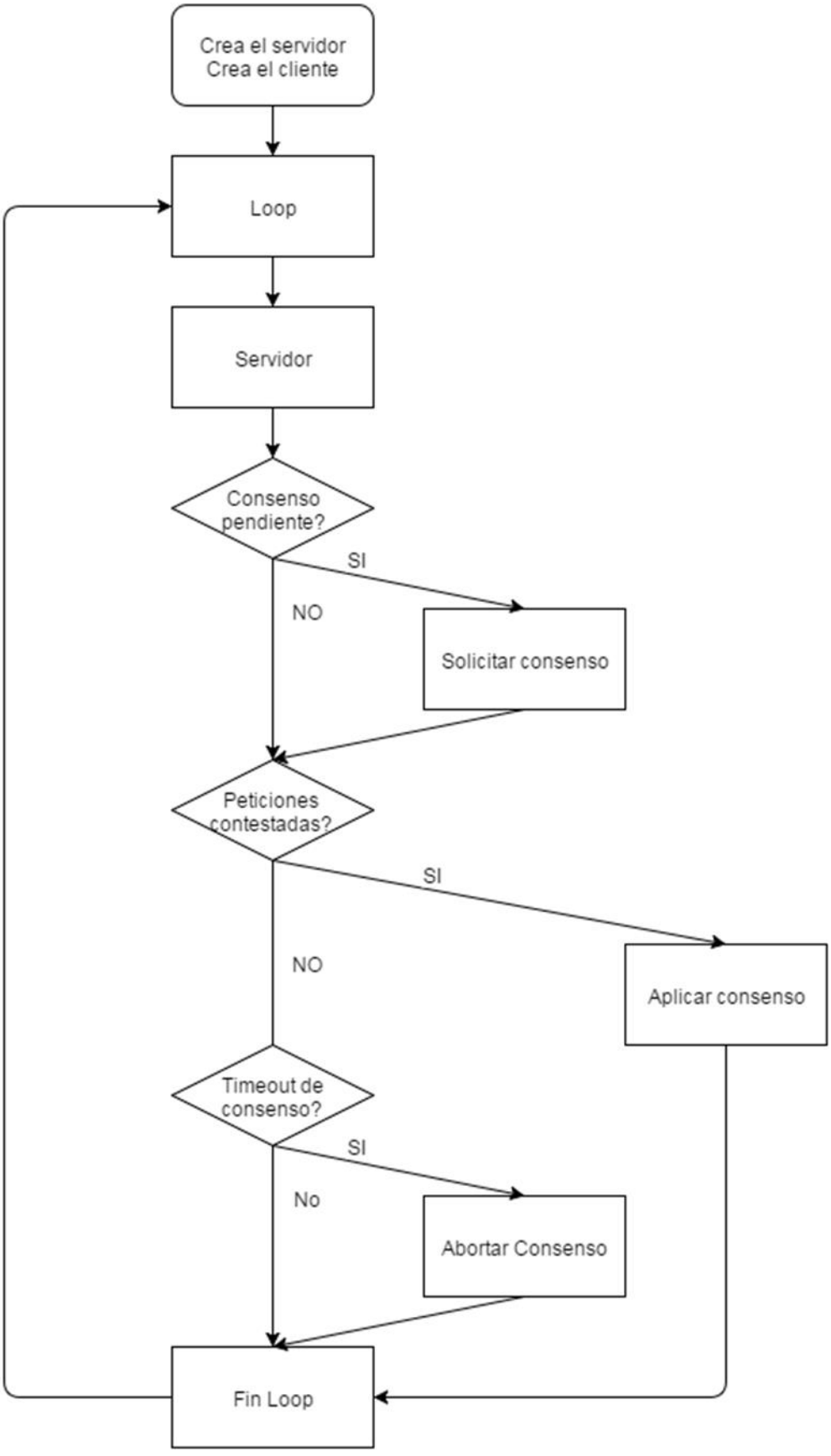

Imagen19: Diagrama de flujo general

5. Aplicar consenso

Para que se active este bloque también hacen falta varias condiciones. Obviamente necesita que el nodo haya iniciado un consenso previamente, pero más importante es que todos los nodos a los que les mandó la difusión le hayan contestado, aunque sea para comunicar que no participan en el consenso. Además debe estar activo el timeout que se lanza al iniciar el consenso.

6. Abortar consenso

En este bloque se activa sólo si se ha iniciado un consenso pero no ha llegado a finalizar dentro del plazo impuesto por el timeout.

7. Fin Loop

Finaliza una iteración para volver a empezar. Realiza algunas acciones de depuración.

4.2.2 Servidor

El bloque servidor (Imagen20) es el más complejo de todos, pero es difícil de reducir en partes más pequeñas sin que se pierda claridad. A continuación se va a comentar su funcionamiento de fuera hacia adentro:

1. Primero nos encontramos con una comprobación muy simple. Esta sólo sirve para desactivar el servidor rápidamente en caso de que no se haya recibido ningún mensaje, por lo que si no se cumple saltará directamente a Desactiva Servidor, finalizando el bloque Servidor.

2. La siguiente comprobación forma parte de un bucle encargado de procesar los mensajes uno a uno.

3. A continuación tenemos una comprobación múltiple, ya que el servidor tiene que realizar acciones diferentes dependiendo de qué mensaje se ha recibido. Esta parte del servidor ha sido explicada en el punto 4.1 Mensajes por lo que no es necesario entrar en más detalle.

Después del punto 3 volvemos al punto 2 con la comprobación del bucle y ya por último se desactiva el servidor hasta la siguiente iteración en la que posiblemente haya más mensajes que procesar.

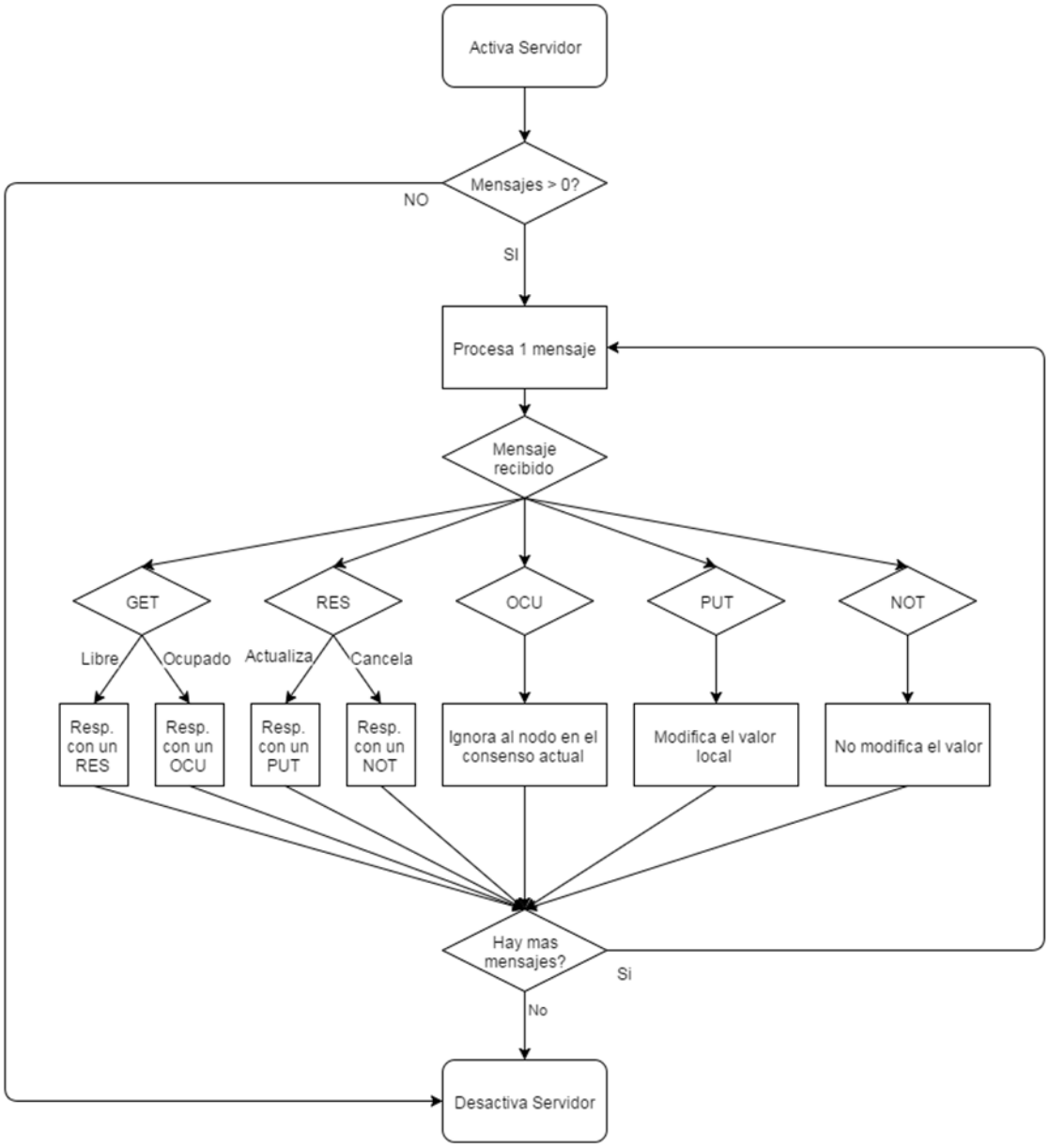

Imagen20: Diagrama de flujo servidor

4.2.3 Solicitar consenso

Este bloque, como podemos ver en la Imagen21, es muy simple ya que simplemente consta con varias acciones que siempre ha de realizar, pero a continuación se va a explicar de la necesidad de cada una de ellas:

1. Bloquear entrada de peticiones de consenso

Si el nodo actual va a realizar un consenso lo primero que tiene que hacer es evitar que otros nodos intenten iniciar un consenso en el que el propio nodo esté implicado.

2. Activar consenso en curso

A continuación el nodo va a activar el timeout de consenso, necesario para saber cuándo hay que interrumpir el consenso si este lleva mucho tiempo en espera y no recibe las respuestas que ha solicitado a sus vecinos. Además va a reiniciar las variables que se utilizan durante el consenso, aunque solo es por precaución ya que también se limpian al final del consenso.

3. Enviar petición de consenso a todos los vecinos

Después de realizar los preparativos ya puede empezar realmente la solicitud de consenso enviando mensajes GET a todos los vecinos accesibles.

Imagen21: Diagrama de flujo Solicitar consenso

4.2.4 Aplicar consenso

Este bloque, cuyo diagrama vemos en la Imagen22, es realmente la parte final del bloque anterior cuando el consenso tiene éxito. Sus dos funciones básicas son las siguientes:

Por una parte decidir si el consenso ha modificado suficiente el valor como para modificarlo en los vecinos. En caso afirmativo procederá con una difusión a todos los

vecinos del nuevo valor y, en caso contrario, difundirá un mensaje NOT para que los demás nodos sepan que el consenso ha concluido.

Y por otra parte volver a dejar el nodo preparado para recibir nuevas peticiones de consenso de los vecinos. Para ello tiene que deshacer lo hecho por el bloque Solicitar Consenso, explicado en el punto anterior.

Imagen22: Diagrama de flujo Aplicar consenso

4.2.5 Abortar consenso

Finalmente el último bloque que vamos a explicar (Imagen23) es la contrapartida del anterior. Este también es la parte final del bloque Solicitar consenso pero en este caso el consenso ha terminado abruptamente a causa del vencimiento del timeout. Por tanto el nodo tiene que difundir mensajes NOT a todos sus vecinos y finalizar el consenso exactamente igual que el bloque anterior. Los motivos por los cuales se ha de abortar el consenso están explicados en el punto 4.3 Mensajes por lo que no es necesario entrar en más detalle.

Imagen23: Diagrama de flujo Abortar consenso

4.3 Bloqueos

Para finalizar el apartado de diseño, en este capítulo vamos a profundizar un poco en el uso que se les ha dado a las keys mediante algunos ejemplos y a los dos timeouts implementados. Con esto podremos consolidar lo que se ha visto anteriormente respecto a estos.

4.3.1 Keys

Como se ha ido viendo a lo largo de los puntos anteriores, se ha utilizado una key en cada nodo para indicar que el nodo está libre para colaborar en un consenso, o indicar que el nodo está participando en un consenso iniciado por otro nodo.

Por tanto, la key se va a representar mediante un entero. Si el valor de la key es el identificador del nodo significará que el nodo está libre. Por el contrario cualquier otro valor significará que el nodo con dicho identificador está realizando un consenso en el cual el nodo propietario de la key está implicado.

Los siguientes son los mensajes que, al ser recibidos, interactúan con las keys:

GET. Si la key está libre, la bloquea con el identificador del emisor del mensaje.

PUT. Si la key está bloqueada por por el emisor del mensaje, libera la key.

NOT. Tiene el mismo efecto que los mensajes PUT.

A continuación vamos a ver ejemplos de comunicaciones y el estado de las keys. El color de la key representa que el nodo representado con ese color está bloqueando dicha key. El color gris significa que el nodo está bloqueado porque ha iniciado una difusión y no acepta difusiones de sus vecinos. Este tipo de bloqueo se va a ver en detalle en el siguiente apartado 4.3.2 Timeouts.

4.3.1.1 Ejemplo 1

Este primer ejemplo (Imagen24) muestra una red de 3 nodos conectados en serie. Podemos observar una difusión correctamente ejecutada por el Nodo 1 y una difusión que no llega a comunicarse con el nodo 2.

Esto se debe a que el Nodo 1 ha bloqueado al Nodo 2 con su petición de consenso y el Nodo 3, que intenta hacer lo mismo con el Nodo 2, le manda el mensaje mientras este está esperando que termine el consenso iniciado por el otro nodo.

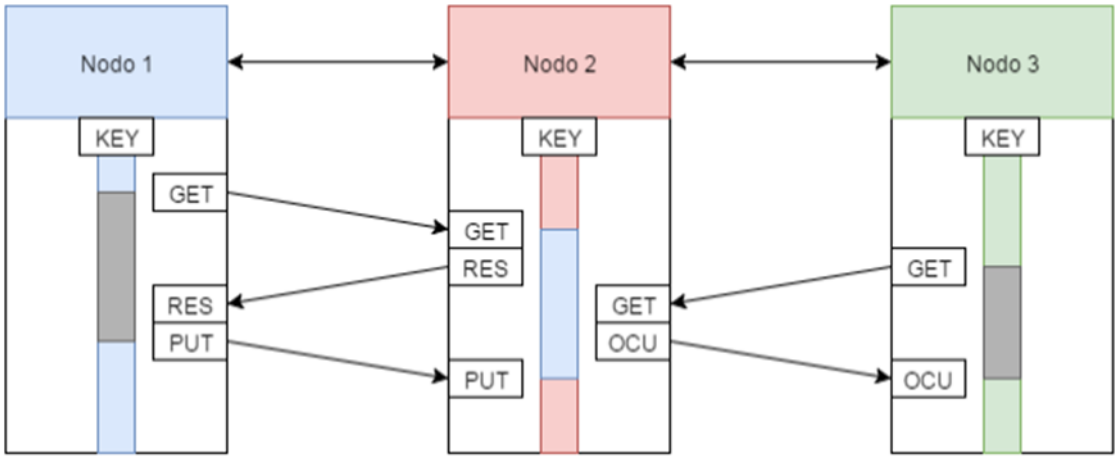

Imagen24: Ejemplo 1

4.3.1.2 Ejemplo 2

En este segundo ejemplo (Imagen25) podemos observar una correcta difusión con sus dos vecinos por parte del Nodo 2. Podemos ver que envía mensajes GET a todos sus vecinos, bloqueándose y bloqueándolos; y a continuación espera a que estos le respondan con su valor para poder difundir el nuevo valor y quitar todos los bloqueos.

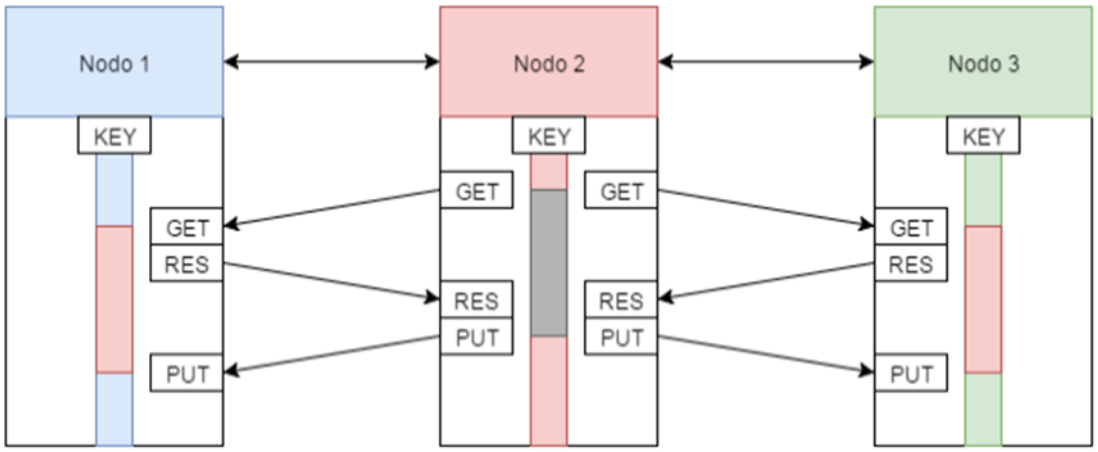

Imagen25: Ejemplo 2

4.3.1.3 Ejemplo 3

En este último ejemplo (Imagen26) podemos observar un caso indeseado. Esta vez los dos nodos tratan de iniciar una difusión al mismo tiempo. Esto se debe a que los mensajes tienen un retardo al ser enviados por la red y, cuanto más alto sea ese retardo, más posibilidades habrán de producir este interbloqueo.

Sin embargo este problema se resuelve colateralmente con el uso de las keys (recordemos que el objetivo inicial de las keys es evitar las condiciones de carrera), que van a evitar que haya comunicación entre los Nodos 1 y 2, pero dejando que estos sigan la difusión con sus respectivos vecinos. La sincronización entre los Nodos 1 y 2 se deja para el siguiente consenso.

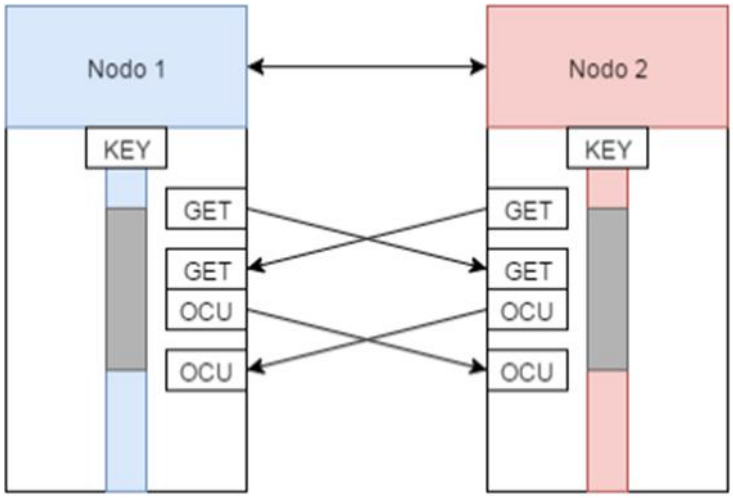

Imagen26: Ejemplo3

4.3.2 Timeouts

Para evitar las condiciones de carrera se han implementado, además de las keys, dos timeouts. Uno de ellos ya se ha visto por encima anteriormente pero en este capítulo se va a explicar un poco más profundamente su necesidad

4.3.2.1 Timeout consenso

Como hemos visto anteriormente, este timeout se usa junto a las keys pero, en vez de bloquear los nodos involucrados indirectamente en el consenso, este timeout bloquea al nodo que inició el consenso.

A priori podría haberse implementado exactamente igual que las otras keys pero se le ha añadido funcionalidad, exigiendo que sea algo totalmente diferente, pasando de ser una simple key a un timeout.

Esta funcionalidad consiste en abortar un consenso si lleva mucho tiempo activo y no se reciben los mensajes que se deberían, indicando que alguno de los nodos ha fallado, obligando al consenso activo a abortarse.

El tiempo medido por este timeout, al igual al que vamos a explicar a continuación, son iteraciones del bucle principal. Este tipo de timeouts son los más utilizados en la plataforma Arduino dada su simple implementación. Hay que tener en cuenta que el tiempo que tarda este timeout en vencer es un tiempo fijo.

4.3.2.2 Timeout siguiente consenso

Este otro timeout tiene la función de hacer que los nodos inicien difusiones. Sin este timeout no se inicia ningún tipo de comunicación.

Para ello se activa por primera vez en la parte estática del código de los nodos y luego se va reactivando cada vez que se hace un consenso.

Y por último y, al contrario que en el timeout anterior, el tiempo de espera de este timeout es variable. Esto se debe a que hay que evitar que despues de un consenso todos los nodos que participaron en el decidan hacer un consenso exactamente al mismo tiempo, generando un interbloqueo múltiple si esos nodos se pueden comunicar entre ellos. Además evitamos que después de un interbloqueo como el visto en el Ejemplo 3 del punto 4.3.1 Keys, donde los dos nodos vuelvan a intentar realizar en consenso a la vez, resultando en infinitos intentos de consensos y que ninguno se resuelva nunca.

En la Imagen27 podemos ver el problema que conllevaría que el timer SiguienteConsenso fuera siempre el mismo. Siempre que el nodo 1 complete con éxito un consenso los nodos 2 y 3 van a bloquearse mutuamente. Sin embargo si el timeout es diferente uno de ellos realizará antes el consenso y bloqueará al otro, realizando el consenso exitosamente.

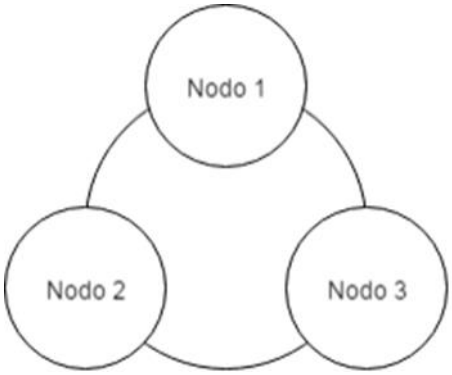

Imagen27: Red conflictiva

5. Implementación

5.1 Nodos

En este capítulo vamos a ver rápidamente cómo se han construido y conectado los nodos.

Para cada uno de los nodos necesitamos una placa Arduino Uno, en shield Ethernet compatible con la placa anterior y los respectivos cables USB, para comunicar la placa Arduino con el PC, y el cable Ethernet, con el que se conectará el shield al router o switch utilizado. Podemos ver ejemplos en las Imágenes 28 a 33.

El cable USB se puede sustituir por una batería aunque es más recomendable usar un cargador. Y también es buena idea usar un Hub Usb.

Imagen28: Arduino Uno

Imagen29: Ethernet Shield

Imagen30: Cable USB type A/B

Imagen31: Cable Ethernet

Imagen32: Cargador compatible Arduino

Imagen33: HUB USB

La construcción de un nodo es simple y no se puede cometer ningún error ya que ningún cable encaja donde no debe.

Pero antes de empezar a conectar los cables hay que encajar el shield Ethernet en la placa Arduino Uno. Solo encajan de una única manera pero hay que tener cuidado de no doblar ningún pin del shield (Imagen34). Después de tener las placas acopladas ya se pueden conectar el cable USB con el PC, el HUB o el cargador y, el cable ethernet con el router o switch (Imagen35).

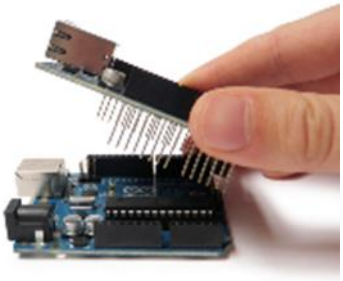

Imagen34: Acoplar el shield Ethernet

Imagen35: Conectar cables USB y Ethernet

Solo nos queda repetir el proceso para cada nodo y ya estarán preparados para recibir el código desde el PC.

Es recomendable que antes de conectar los nodos al PC se haya abierto un IDE de Arduino. De este modo podemos observar al conectar los nodos de uno en uno a que puerto se vinculan. También es buena idea utilizar posits pegados a cada nodo o a alguno de sus cables indicando a que puerto se ha conectado.

5.2 Circuito

A continuación vamos a ver como montar el circuito conectado a cada uno de los nodos. La única función de este circuito es mostrar el valor RGB interno de los nodos mediante un led RGB.

Por cada uno de los nodos que pongamos en la red necesitamos 1 led RGB de 4 patas (Imagen 36), 3 resistencias (Imagen37) y 4 cables (Imagen38). Se recomienda el uso de una protoboard para realizar las conexiones (Imagen39).

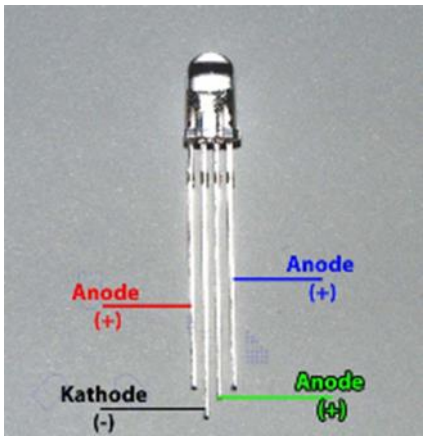

Imagen36: LED RGB 4 patas

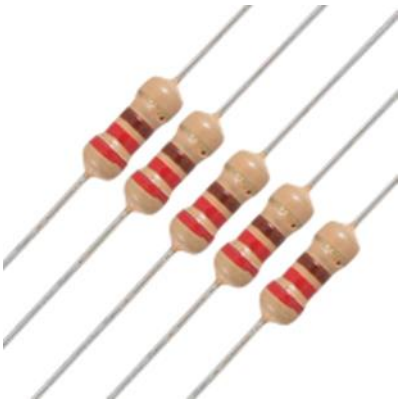

Imagen37: Resistencias 220 ohmnios

Imagen38: Cable Arduino Macho-Macho

Imagen39: Protoboard

Con todo el material preparado el montaje es bastante sencillo si se tienen en cuenta un par de cosas que explicaremos más adelante. Puede que la longitud de los cables y el espacio reducido sean un poco molestos, pero se pueden usar cables Macho-Hembra para alargar los cables Macho-Macho.

Lo primero que tenemos que tener en cuenta son las patas del led RGB. Como podemos ver en la imagen anterior el más largo de todos será el cátodo común de los 3 led que tiene internamente, y deberemos conectarlo directamente a 5V. Las otras 3 patas són los ánodos de cada led, el que se queda solo es el del led rojo, y los otros dos son, el más exterior el azul y el que queda el verde. Estas 3 patas necesitan una resistencia para evitar que se quemen. Si elegimos una resistencia muy alta los leds no brillarán lo suficiente, lo ideal son las típicas de 220Ω.

Ahora solo queda conectar las 3 resistencias con los puertos 5 a 7. El 5 con la resistencia del led rojo, el 6 con la verde y el 7 con la azul. Si repetimos todo el proceso para más nodos debería quedar algo como en las imágenes 40 y 41, en la que se muestra el montaje de 4 nodos en una protoboard.

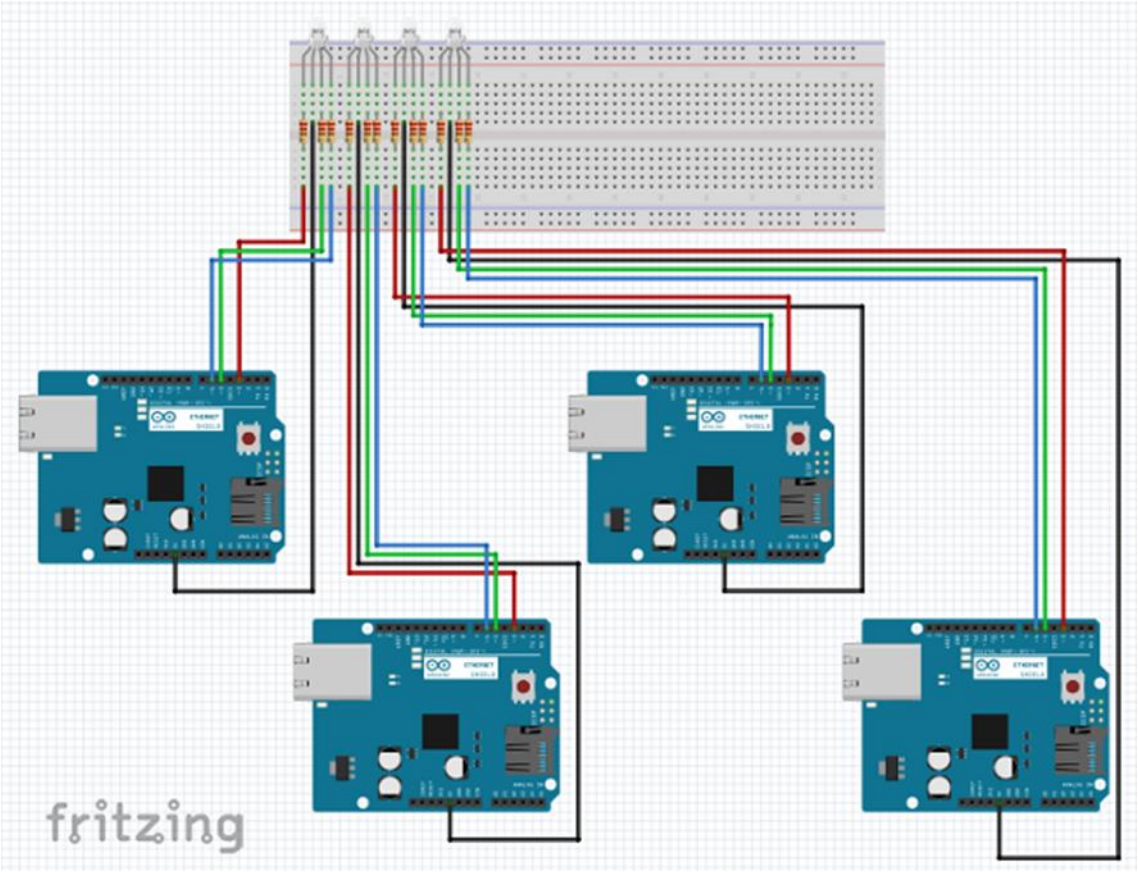

Imagen40: Esquema del circuito realizado en Fritzing [15]

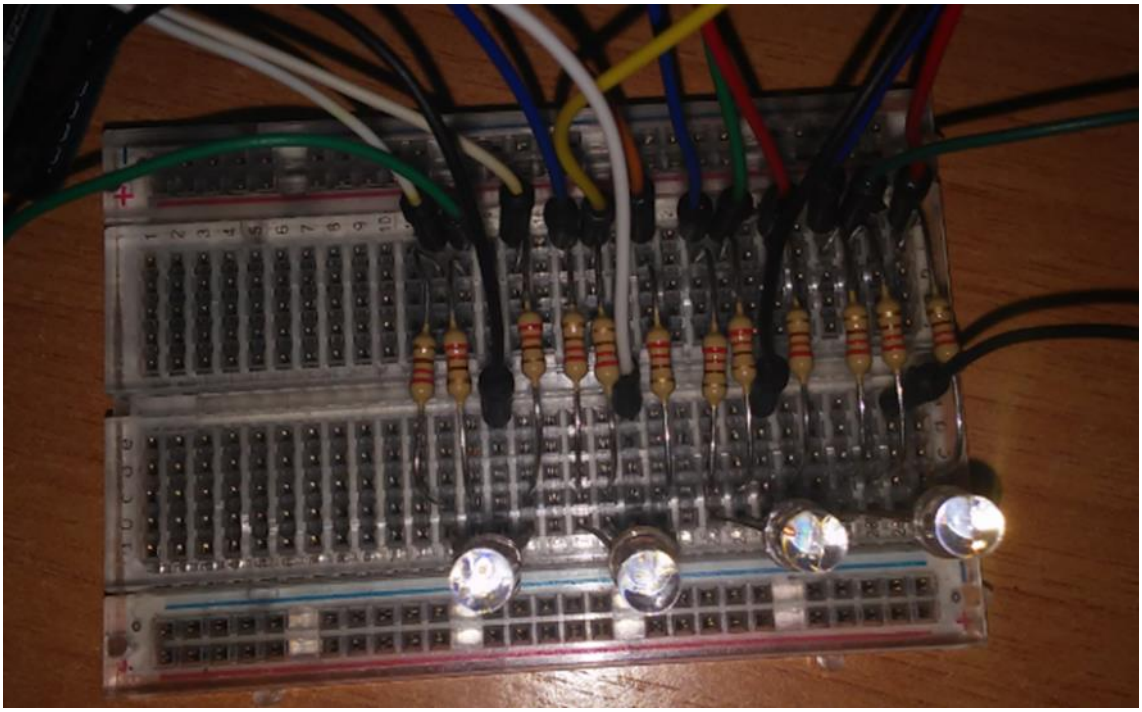

Imagen41: Circuito para 4 nodos

5.3 Ejecución

En el siguiente capítulo vamos a mostrar detalladamente un ejemplo de ejecución de un sistema con cuatro nodos y, para finalizar, unos rápidos ejemplos con distintas topologías. Luego se van a comparar los resultados obtenidos en cada una de las redes, teniendo en cuenta el tiempo en que se llega a consenso, el número de consensos realizados, los mensajes tipo OCU enviados, los consensos finalizados por timeout y los mensajes fallidos. Todos estos valores teniendo en cuenta el número de conexiones que posee la red y la distancia máxima entre dos nodos.

5.3.1 Ejemplo 1

En el primer ejemplo hemos implementado una red en serie (Imagen42). Esto significa que el nodo 0 está conectado con el 1, el 1 con el 2 y el 2 con el 3. Esta red tiene la mayor distancia máxima entre dos nodos con 3 conexiones y un destacado cuello de botella entre los nodos 2 y 3.

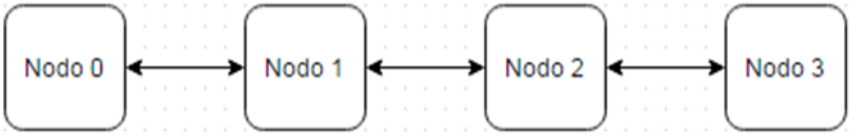

Imagen42: Red 1

Para implementar esta red se ha editado en cada nodo el vector activos[] de la siguiente forma:

Nodo 0: {true, true, false, false}
Nodo 1: {true, true, true, false}
Nodo 2: {false, true, true, true}
Nodo 3: {false, false, true, true}

El primer campo del vector se utiliza para considerar si el nodo con identificador 0 es alcanzable o no, el siguiente campo con el nodo con identificador 1, etcétera.

Nótese que no importa qué valor tenga el campo con el identificador del nodo actual. Por ejemplo el Nodo 0 nunca va a intentar comunicarse con el mismo.

Además se han fijado los siguientes valores iniciales RGB para una mejor visualización del cambio de color:

Nodo 0:	R = 0	G = 0	B = 100
Nodo 1:	R = 0	G = 100	B = 0
Nodo 2:	R = 100	G = 0	B = 0
Nodo 3:	R = 0	G = 0	B = 0

Valores por encima de 200 no se ven correctamente al hacerles una foto debido a la abundancia de luz.

Con los valores anteriores el sistema empieza como en la Imagen43.

Imagen43: Leds RGB al inicio de la ejecución

Estos valores iniciales nos llevan al estado final en el que todos los nodos tienen R = 25, G = 25, B = 25 (Imagen44). Aunque puede que difiera ligeramente debido a que los valores han de ser enteros y el resultado de una división puede resultar en decimales. Además se ha implementado un umbral para determinar una diferencia mínima entre dos valores. De este modo si dos valores son muy cercanos serán considerados iguales.

Imagen44: Leds RGB al final de la ejecución

Durante el proceso de intercambio de mensajes podemos ver el estado de cada nodo y el tráfico que genera y recibe mediante el monitor serie (Imagen45). El valor k es el estado de la key del nodo, los valores C son los dos timeouts y por último tenemos los valores RGB.

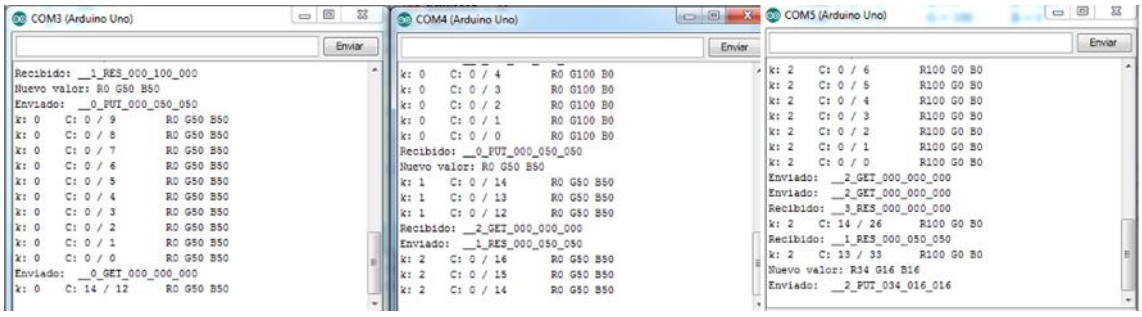

Imagen45: Ejemplo de del monitor serie de tres nodos durante la ejecución.

A continuación vamos a mostrar una sucesión de imágenes (de la imagen 46 a la 49) en las que se muestra cómo los cuatro nodos han llegado a consensuar un color para los leds.

Imagen46: Consenso de 4 leds estado 1

Nodo 3	Nodo 2	Nodo 1	Nodo 0
R = 0	R = 100	R = 0	R = 0
G = 0	G = 0	G = 100	G = 0
B = 0	B = 0	B = 0	B = 100

Imagen47: Consenso de 4 leds estado 2

Nodo 3	Nodo 2	Nodo 1	Nodo 0
R = 0	R = 100	R = 0	R = 0
G = 0	G = 0	G = 50	G = 50
B = 0	B = 0	B = 50	B = 50

Imagen48: Consenso de 4 leds estado 3

Nodo 3	Nodo 2	Nodo 1	Nodo 0
R = 33	R = 33	R = 33	R = 0
G = 16	G = 16	G = 16	G = 50
B = 16	B = 16	B = 16	B = 50

Imagen49: Consenso de 4 leds estado 4

Nodo 3	Nodo 2	Nodo 1	Nodo 0
R = 33	R = 22	R = 22	R = 22
G = 16	G = 27	G = 27	G = 27
B = 16	B = 27	B = 27	B = 27

5.3.2 Ejemplo 2

En este segundo ejemplo (Imagen50) tenemos una red igual que la primera con la gran diferencia que los nodos que antes eran extremos ahora están comunicados entre ellos. De este modo el cuello de botella desaparece y la nueva distancia máxima entre nodos se reduce a dos conexiones.

El vector activos[] en este ejemplo es:

Nodo 0: {true, true, false, true}
Nodo 1: {true, true, true, false}
Nodo 2: {false, true, true, true}
Nodo 3: {true, false, true, true}

Los colores iniciales se han dejado igual que en el ejemplo 1.

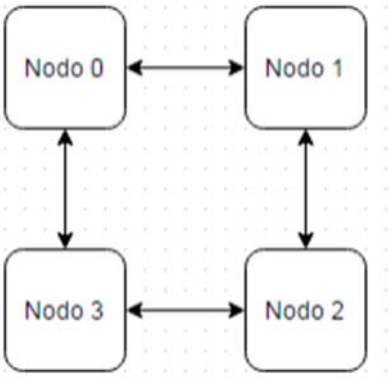

Imagen50: Red 2

5.3.3 Ejemplo 3

Este tercer ejemplo (Imagen51) tenemos la red del ejemplo 2 pero añadiendo las dos conexiones que faltaban, creando una red completa en la que tampoco hay cuello de botella y todos los nodos pueden comunicarse con cualquier otro nodo con una sola conexión.

El vector activos[] en este ejemplo es:

Nodo 0: {true, true, true, true}
Nodo 1: {true, true, true, true}
Nodo 2: {true, true, true, true}
Nodo 3: {true, true, true, true}

Los colores iniciales se han dejado igual que en el ejemplo 1.

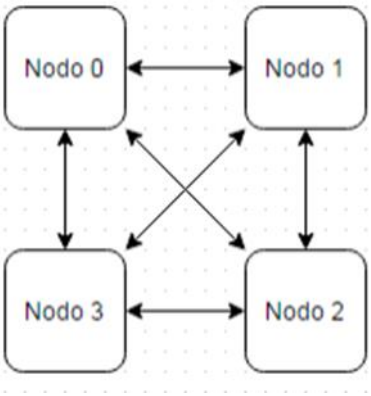

Imagen51: Red 3

5.3.4 Ejemplo 4

En el ejemplo 4 (Imagen52) hemos implementado una red con forma de estrella. Este tipo de redes es más común en redes centralizadas. La distancia máxima entre nodos es de dos conexiones y el cuello de botella es el nodo 1.

El vector activos[] en este ejemplo es:

Nodo 0: {true, true, false, false}
Nodo 1: {true, true, true, true}
Nodo 2: {false, true, true, false}
Nodo 3: {false, true, false, true}

Los colores iniciales se han dejado igual que en el ejemplo 1.

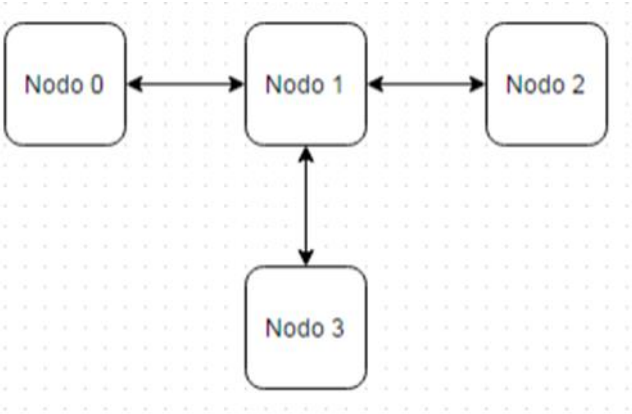

Imagen52: Red 4

5.3.5 Ejemplo 5

En el ejemplo 5 (Imagen53) hemos implementado una red con forma irregular. La distancia máxima entre nodos es de dos conexiones y el cuello de botella es el nodo 3.

El vector activos[] en este ejemplo es:

Nodo 0: {true, true, false, true}
Nodo 1: {true, true, false, true}
Nodo 2: {false, false, true, true}
Nodo 3: {true, true, true, true}

Los colores iniciales se han dejado igual que en el ejemplo 1.

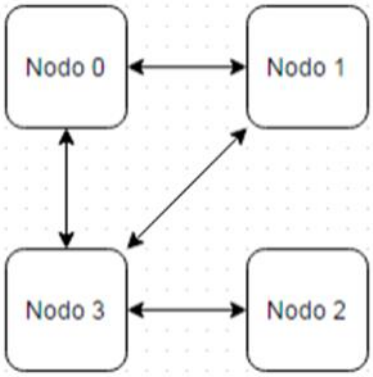

Imagen53: Red 5

5.3.6 Resultados

Se ha realizado un pequeño estudio con las cinco distintas redes vistas anteriormente.

Antes de empezar a observar los resultados hay que tener en cuenta que, dado el factor aleatorio con el que cuenta el sistema, algunas ejecuciones se ejecutan de manera ideal mientras que otras se bloquean unas cuantas veces. Además, debido a las propias placas Arduino de vez en cuando algún nodo se desconecta sin ningún motivo y detiene la comunicación con sus vecinos, rompiendo la red.

Por estos motivos se ha decidido que, en vez de calcular la media de varias ejecuciones con cada topología, se ha elegido un caso representativo de cada una de ellas. El caso elegido para cada topología ha sido el que ocurre con mucha más frecuencia que el caso "ideal" o, por el contrario, el caso en que se desconecta algún nodo.

	Ejemplo 1	Ejemplo 2	Ejemplo 3	Ejemplo 4	Ejemplo 5
Conexiones	3	4	6	3	4
Distancia máxima entre nodos	3	2	1	2	2
Tiempo	39s	24s	9s	65s	33s
Consensos	10	5	1	14	8
Mensajes OCU	5	4	0	8	4
Consensos fallidos	0	0	0	1	0

Después de estas aclaraciones pasemos a analizar la tabla:

El ejemplo 1 ha tardado 39 segundos en llegar al estado final y ha realizado un total de 10 consensos, mientras 5 nodos han decidido no participar en alguno de ellos. Vamos a tomar este caso como base para comparar los demás.

El ejemplo 2 ha tardado 24 segundos en llegar al estado final, realizando 5 consensos y con 4 nodos que han decidido no participar en algún consenso. En este caso la reducción de la distancia máxima entre nodos y la supresión del cuello de botella ha sido determinante, ya que en el ejemplo anterior los dos extremos requerían una larga cadena de consensos para intercambiar información, cargando a los nodos centrales.

El ejemplo 3 ha tardado sólo 9 segundos y ha realizado la media en tan solo 1 consenso. Es lo lógico que va a pasar en una red completa, pues si el primer consenso no es bloqueado por nadie lleva el sistema directamente al estado final. Aunque este caso es muy poco realista dado que pocas veces nos encontramos con redes totalmente conectadas.

El ejemplo 4 ha tardado 65 segundos, 14 consensos con 8 envíos de mensajes OCU y encima uno de los consensos ha finalizado abruptamente por timeout. Lo que pasa en esta red es que el nodo central está constantemente recibiendo peticiones y, mientras realiza el consenso con uno de los vértices, los otros dos se quedan ociosos. En esta red se puede dar el caso que el nodo central realice un consenso a la vez con los tres vértices, llegando al estado final directamente como el ejemplo 3. Sin embargo después de 20 ejecuciones, en ninguna se ha dado el caso.

Finalmente el ejemplo 5 ha tardado 33 segundos, 8 consensos y ha sufrido 4 mensajes OCU. Este caso se comporta a mitad de camino entre el ejemplo 1 y el ejemplo 2, además de que tiene la posibilidad de terminar en solo 1 consenso realizado por el nodo 3.

Como conclusión a este pequeño estudio podemos decir que se podría mejorar el código para tener en cuenta el caso de la red en estrella. Una posible solución sería ampliar el cálculo del siguiente consenso, añadiendo a la parte aleatoria un tiempo en relación a la cantidad de vecinos del nodo. Con esto podríamos conseguir que nodos con muchos vecinos sean muy activos, y nodos con pocos vecinos no originen muchos bloqueos. Aunque esto plantea un incremento teórico de interbloqueos en redes muy pobladas, ya que dos nodos muy activos juntos tenderían a bloquearse continuamente.

5.4 Dificultades

En este capítulo vamos a exponer las dificultades sufridas durante la realización de este proyecto y la solución que se ha llevado a cabo para solucionarlos.

5.4.1 Chipset y librerias

Descripción

La primera dificultad sufrida durante la realización de este proyecto fue el desconocimiento de las distintas librerías que gestionan los shield Ethernet de Arduino. Los primeros shield con los que se iniciaron las pruebas de comunicación tienen el chipset ENC28J60, que no es compatible con las librerias que trae por defecto la IDE Arduino para el shield Ethernet. Utilizar la librería errónea provoca que el código compile correctamente, ya que realmente no hay ningún error en el código de la placa Arduino pero, por otra parte, el shield Ethernet no tiene código ejecutable, fallando todos los mecanismos que este dispone: lanzar servidor, crear clientes, enviar mensajes...

Solución

La solución es muy simple, cambiar la librería. Elegimos la librería UIPEthernet como ya explicamos en el apartado 2.2.4 Librerias para el chipset ENC28J60.

5.4.2 Retraso de mensajes

Descripción

Con el uso del chipset ENC28J60 surgió otro problema a parte del de las librerías. Los mensajes no llegan al destino hasta que no se cierra la conexión TCP. No se ha encontrado información al respecto ya que hay poca gente utilizando el chipset ENC28J60 y este comportamiento parece que no ocurre con los demás chipsets.

Solución

La solución no ha sido la más óptima pero ha servido para realizar correctamente la comunicación. Esta consiste simplemente en cerrar la conexión después del envío de cada mensaje.

5.4.3 Bloqueo de nodos

Descripción

Este problema es bastante simple. Hay veces que el shield recibe información errónea del buffer de entrada, creyendo que hay información en espera cuando en realidad este está vacío. Esto deja al nodo bloqueado intentando leer el buffer e ignorando sus demás funciones.

Solución

Timeout de lectura. Añadiendo una simple variable al bucle (Imagen54) de lectura forzando la salida si el nodo intenta leer más de 18 caracteres del buffer, ya que los mensajes tienen ese tamaño.

```
void servidor(){
  int count = 0;
  EthernetClient client = server.available();
  String msg = "";
  if(client){
    while ((client)&&(count <= 18)) {
      count++;
      char c = client.read();
      msg = msg + c;
    }
    ////////////
    Serial.println("Recibido: " + msg);
    ////////////
  }
```

Imagen54: Solución para el bloqueo de nodos

5.4.4 Condiciones de Carrera

Descripción

Aunque Arduino funcione secuencialmente tenemos ejecutando varios de ellos y además tienen la capacidad de modificar el valor de sus vecinos. Esto ocasiona una alta posibilidad de condiciones de carrera si, por ejemplo, un nodo recibe dos órdenes de lectura de dos nodos distintos y a continuación ambos intentan actualizar el valor del primer nodo. Podemos ver un ejemplo en la Imagen55.

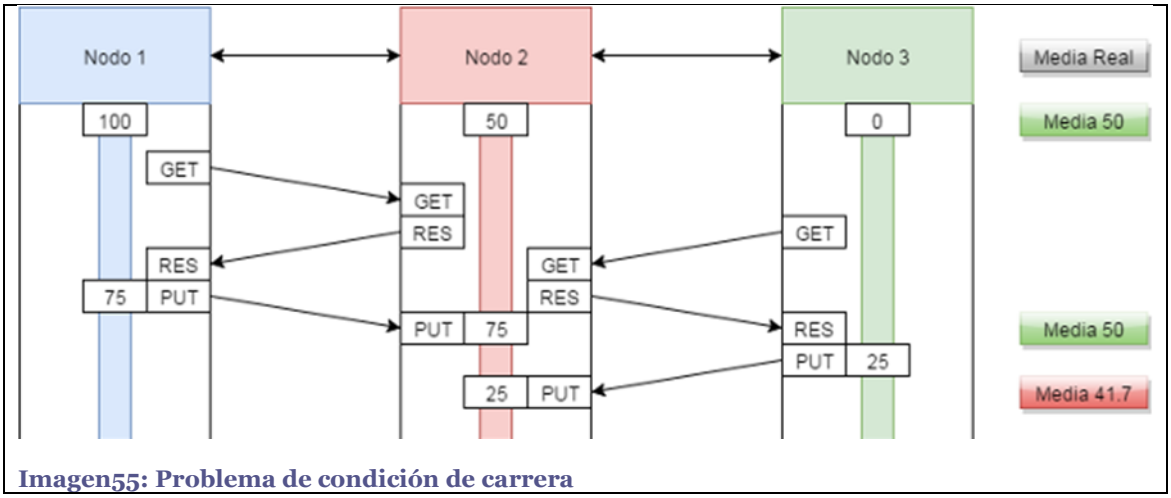

Imagen55: Problema de condición de carrera

Solución

La más sencilla de implementar y la que se ha utilizado es el uso de una key por cada nodo. Esta key es "tomada" por otro nodo al recibir una consulta de valor y "devuelta" al recibir la orden de actualizar el valor. Mientras un nodo se encuentre con la key bloqueada responderá a peticiones con un mensaje de ocupado. El uso de las keys está explicado en el punto 4.3.1 Keys.

5.4.5 Memoria insuficiente

Descripción

Este problema (Imagen56) se ha tenido en el punto de mira casi desde el inicio del proyecto. Finalmente casi al final de este se manifestó este error, ya que varias de las placas más viejas de Arduino Uno llegaron al 75% de capacidad, ocasionando que el nodo no tenia espacio suficiente para crear los mensajes y terminaba enviando mensajes incompletos, vacios o leyendo los mensajes recibidos a medias.

```
Sketch uses 26.890 bytes (83%) of program storage space. Maximum
is 32.256 bytes.

Global variables use 1.633 bytes (79%) of dynamic memory,
leaving 415 bytes for local variables. Maximum is 2.048 bytes.

Low memory available, stability problems may occur.
```

Imagen56: Problema de memoria insuficiente

Solución

Llegado a este punto solo hay tres soluciones:

La primera y sólo viable como último recurso. Cambiar la librería por la más compacta de las 3 nombradas anteriormente, forzando a modificar y rehacer prácticamente todo el código ya implementado.

La segunda sería cambiar los Arduino Uno de versiones anteriores por las más nuevas, que tienen el doble de memoria disponible. Esta opción requiere comprar las nuevas placas y esperar el tiempo que tardan en llegar, lo que también la hace poco viable.

Y la última opción, eliminar las partes no esenciales del código. Al estar probando errores y afinando el funcionamiento el código contaba con cantidad de mensajes de depuración bastante complejos y concretos. Mediante la simplificación o la eliminación de estos mensajes se ha podido liberar gran parte de la memoria utilizada, evitando tener que cambiar la librería y todo lo que ello conlleva.

6. Conclusiones

6.1 Trabajo desarrollado y aportaciones

En este capítulo vamos a recopilar todo el trabajo que se ha realizado durante la realización de este proyecto y para finalizar las aportaciones que se han recibido.

Para empezar se reunió todo el material necesario para la construcción física del proyecto, desde las placas y los shields hasta los cables, leds y resistencias. Todo el material ha sido nombrado en los capítulos 5.1 Nodos y 5.2 Circuito, junto con su correspondiente montaje.

A continuación se realizaron numerosas pruebas de comunicación para determinar cómo se realizaría y todo lo que se iba a necesitar para que no hubiera errores. Prueba a prueba se ha ido aumentando el código con todas las características que se han necesitado, como el uso de las librerías específicas para cada nodo con distinto chipset (Punto 2.2.4 Librerías para el chipset ENC28J60), el uso de keys y timeouts (Punto 4.3 Bloqueos). Durante las pruebas también se definieron las necesidades y las características que iban a tener los mensajes para que el sistema funcionara como se esperaba (Punto 4.1 Mensajes) y como iban los nodos a gestionar la recepción, el envío y todas las operaciones necesarias en los consensos (Punto 4.2 Nodos).

Con todo el sistema montado y funcionado se empezaron a realizar pruebas para refinar el código, tratando de mejorar su funcionamiento y se realizó un pequeño estudio con la red de cuatro nodos para poder mostrar los resultados (Punto 5.3 .6 Resultados).

Finalmente, quiero agradecer personalmente la ayuda de Ángel Rodas Jordá por la ayuda prestada al inicio del proyecto, sugiriendo que los errores en los envíos eran cosa del uso de una librería incorrecta, y prestando una placa con un chipset diferente para confirmar que no era problema del código.

6.2 Ampliaciones

Como posibles ampliaciones a este proyecto nos gustaría proponer que se realice el mismo proyecto pero probando otras tecnologías de las que se propusieron en el apartado 2.2 Alternativas. Concretamente tengo principal interés en el uso de Raspberry Pi, ya que poder realizar este trabajo con la posibilidad de hilos concurrentes debe ser una mejora considerable, y también la utilización de Xbee, ya que es una tecnología diferente a la que estamos acostumbrados y promete ser de mucha utilidad la posibilidad de gestionar la red sin necesidad de un router o switch.

Otra posible ampliación sería la realización de una red mucho más grande que la construida en este proyecto. La repercusión del aumento del tamaño de la red, tanto en longitud entre nodos como en número de vecinos por nodo, afectará el tiempo de espera requerido en los timers, pero lamentablemente no ha podido ser testeado en el presente proyecto.

7. Bibliografía

[1] Topologias de red. Disponible en <http://lapautaqueconecta.blogspot.com.es/2009/08/redes-centralizadas-descentralizadas-y.html>. Fecha de consulta agosto de 2015.

[2] Topologias de red. Disponible en< http://civercibilian.blogspot.com.es/p/el-presente-libro-contiene-tan-solo.html>. Fecha de consulta agosto de 2015.

[3] Definición de consenso. Disponible en <http://lema.rae.es/drae/?val=consenso>. Fecha de consulta agosto de 2015.

[4] Consenso en redes <https://consenet.wordpress.com/2013/04/04/demo-consenso/>. Fecha de consulta agosto de 2015.

[5] Arduino <https://www.arduino.cc/>. Fecha de consulta agosto de 2015.

[6] Comunidad de Arduino <https://forum.arduino.cc/>. Fecha de consulta agosto 2015.

[7] Productos Arduino < https://www.arduino.cc/en/Main/Products>. Fecha de consulta agosto 2015.

[8] Ethernet Shield <https://www.arduino.cc/en/Main/ArduinoEthernetShield>. Fecha de consulta agosto 2015.

[9] Alternativas a Arduino <http://blogthinkbig.com/4-alternativas-arduino-beaglebone-raspberrypi-nanode-waspmote/>. Fecha de consulta agosto de 2015.

[10] Wifi Shield <https://www.arduino.cc/en/Guide/ArduinoWiFiShield>. Fecha de consulta agosto 2015.

[11] Xbee Shield <https://www.arduino.cc/en/Main/ArduinoXbeeShield>. Fecha de consulta agosto 2015

[12] Librerias para el chip ENC28J60 <http://www.tweaking4all.com/hardware/arduino/arduino-enc28j60-ethernet/>. Fecha de consulta agosto de 2015.

[13] Tangible Networks <http://www.bristol.ac.uk/bccs/outreach/tangible-networks/>. Fecha de consulta agosto de 2015.

[14] Proyecto Kilobot <http://www.eecs.harvard.edu/ssr/projects/progSA/kilobot.html>. Fecha de consulta agosto de 2015.

[15] Fritzing <http://fritzing.org/learning/get-started>. Fecha de consulta agosto de 2015.