Soluciones distribuidas para una aplicación cliente/servidor de simulación.

Tesis de Máster en Computación Paralela y Distribuida de:
Andrés Villar Monserrat

Dirigida por:
Francesc Daniel Muñoz Escoí
Codirigida por:
Stefan Beyer

A mis compañeros de clase, del trabajo y profesores... por tener esa santa paciencia conmigo y darme las oportunidades que me han dado. Gracias.

Índice general

Capítulo 1

Introducción

1.1. Metas de la presente tesis

Este documento presenta una aplicación real de simulación de regatas de barcos, implementada en Java y de código propietario, que sigue una arquitectura cliente-servidor; así como los problemas que presenta y una serie de soluciones propuestas para solventarlos.

Esta aplicación, que será denominada `BogReg`, se presenta como un simulador de regatas de barcos on-line en el que varias personas compiten con un determinado modelo de barco, siguiendo un determinado circuito y con unas condiciones climáticas dadas. Estas competiciones se realizan en una serie de carreras aisladas, que consisten en dar un determinado número de vueltas a un circuito, o en una regata, que se compone de una o varias carreras. En cada carrera, todos los barcos que compiten son del mismo tipo, existiendo cuatro tipos diferentes de barco, cada uno con características propias.

Además de la funcionalidad de simulador que posee la aplicación, también engloba un panel de avisos que muestra tanto mensajes del sistema como una suerte de chat interno de cada circuito para que los participantes se puedan comunicar entre ellos. El sistema de chat puede ser empleado tanto por participantes como por espectadores, sin tener en cuenta el tipo de cliente que se está ejecutando.

Existen dentro de la aplicación diferentes tipos de cliente que se pueden conectar a un servidor:

- Cliente *MASTER*: Consiste en un tipo de cliente que tiene además acceso a los parámetros de configuración de la carrera o de la regata. Este cliente se emplea actualmente para testeo de actualizaciones y no tienen acceso los usuarios normales.

- Cliente *CLIENT*: Es el modo de cliente normal. Cuando se introduce en una regata, se le asigna directamente un barco para competir, en caso de que el aforo máximo de la regata lo permita.
- Cliente *JURY*: Este tipo de cliente posee la misma funcionalidad que el cliente de tipo *CLIENT* con la diferencia de que inicialmente no se le asigna un barco para competir, sino que empieza como un espectador de la regata y posteriormente puede solicitarlo.
- Cliente *WATCHER*: Un modelo de cliente en modo espectador, al que no se le asigna barco, y que únicamente puede interactuar con el chat de la aplicación y observar el desarrollo de la regata.

Por otro lado, la aplicación, como se ha indicado anteriormente, se basa en una arquitectura de cliente-servidor. Dentro de este contexto, cada cliente se puede conectar únicamente a un servidor, mientras que un servidor puede albergar varios clientes, con un mínimo de uno y un máximo de 32 actualmente. Este hecho se debe a que, al existir un único servidor central, la carga computacional que soporta es limitada, por lo que con un número mayor de clientes puede saturarse.

Para acceder a una regata, los clientes se conectan al servidor y, en función del tipo de cliente que tengan, usualmente tipo *CLIENT*, pueden recibir un barco si lo solicitan en la interfaz de la aplicación (botón *Get-a-boat*. El motivo de comenzar con este tipo de cliente es que permite que los usuarios puedan entrar en una regata sin necesidad de tener un barco asignado desde un primer momento, pudiendo quedarse como simples espectadores o tomando partido en la regata.

En toda regata existe un período máximo de 350 segundos para que se conecten todos los participantes de la regata, aunque este período se puede reducir a 120 segundos o se puede reiniciar. Además de los participantes humanos de la regata, el servidor se puede configurar de modo que también incluya participantes controlados por la máquina.

Para lograr el progreso de la regata correctamente, tanto la parte servidora de la misma como la parte cliente implementan un simulador del sistema, con la diferencia de que el simulador de la parte cliente de la aplicación únicamente implementa un simulador ligero con los cálculos menos costosos de la aplicación, recayendo en la parte servidora la responsabilidad de avanzar todo el estado, realizando todos los cálculos necesarios, y mantener a los clientes sincronizados.

Entrando más en detalle, el servidor, cuando inicialmente crea una regata nueva, carga ciertos valores desde fichero, como, por ejemplo, las boyas

del circuito, los vientos y las corrientes marinas. Posteriormente, crea, en caso de que existan, los barcos controlados por él, e inicializa los hilos de progreso del estado y de comunicaciones. Mientras no se conecta ningún cliente el estado no avanza, así como una vez desconectado el último cliente, el estado se resetea y detiene su avance. Por otro lado, deja un hilo de escucha a la espera de mensajes y un hilo de envío a la espera de destinatarios.

Una vez se conecta un cliente, el servidor se encarga de evolucionar el estado de la regata, así como de atender todas las peticiones que le envían los clientes asociados a él. Cada vez que se modifica el estado en el servidor, con una frecuencia máxima de 3 veces por segundo (unos 400 ms aproximadamente), se envía un mensaje de sincronización a todos los clientes. Adicionalmente, en caso de que durante un período prolongado no se produzcan cambios de estado, se envían sincronizaciones cada 3-4 segundos.

La aplicación servidora además posee un hilo para atender a los mensajes de keep-alive. En caso de que el tiempo de keep-alive venza, lanza un thread para liberar la conexión del cliente y liberar el barco asociado en caso de que lo hubiese.

Por su parte, la aplicación cliente posee otro simulador, aunque éste es más ligero, dado que no realiza cálculos de alta carga computacional como pueden ser las colisiones entre barcos, etc. En primer lugar envía un mensaje de inicialización con los valores facilitados por el usuario, donde recibirá como respuesta si puede o no puede participar en la regata y, en caso afirmativo, los valores de las corrientes, vientos, boyas, etc. No será hasta un segundo mensaje de inicialización cuando reciba los datos pertenecientes a la flota participante en la regata.

Cuando un usuario desea realizar una operación, el cliente la captura y la aplica localmente, para, posteriormente, enviar un mensaje con los cambios locales al servidor. No existe un lapso mínimo entre acciones, poniendo el límite las que pueda procesar el sistema. Por otro lado, cuando se recibe un mensaje de sincronización, el cliente lo analiza y sustituye en su estado local los valores enviados por el servidor. Adicionalmente, cada vez que el cliente envía un mensaje al servidor, se procesa como si fuese además un mensaje de keep-alive, existiendo también un thread de envío de keep-alives para mantener el cliente con actividad y que no sea desconectado de la regata.

BogReg, como aplicación cliente-servidor centralizada, presenta una serie de problemas en su versión inicial, que serán objeto de estudio y punto de partida para las aportaciones propuestas en el presente documento, entre las que se destacan las siguientes:

- Desorganización y dificultades en la legibilidad del código: la aplicación inicialmente se presenta como una única clase que engloba toda la aplicación, omitiendo el paradigma de la orientación a objetos.
- Falta de trasparencia de ubicación: los envíos y recepciones de mensajes entre el cliente y el servidor son explícitos en el código, estando las capas de aplicación y de comunicaciones entremezcladas. Si se desea obtener un sistema distribuido real desde este planteamiento, las comunicaciones deben ser transparentes para la aplicación, al menos a nivel de código.
- Retardos de comunicaciones elevados: partiendo de que únicamente existe un servidor por regata, los tiempos de transmisión de los mensajes variarán en función de la distancia entre el servidor y los clientes conectados pudiendo dar lugar a "saltos" en la visión de la regata que tienen los clientes, lo que reduce la jugabilidad.
- Falta de escalabilidad: al existir un servidor centralizado, la capacidad de cómputo de éste es limitada, por lo que puede llegar un momento en que no pueda gestionar un número elevado de conexiones. Actualmente el límite de participantes en una regata es de 32.
- Nula tolerancia a fallos: en caso de que el servidor tuviese un fallo que le impidiera continuar funcionando normalmente, la regata sería interrumpida.

1.2. Visión general

La presente tesis muestra diferentes vías a seguir con el propósito de obtener una aplicación descentralizada, a partir de una aplicación con arquitectura cliente-servidor centralizada. Dicha aplicación se compone inicialmente de:

- **Usuarios:** personas físicas que acceden a la aplicación.
- **Cliente:** máquina física que emplea cada uno de los usuarios para conectarse al servidor web mediante un navegador y acceder a la aplicación mediante el applet.
- **Applet:** aplicación empleada por el cliente para acceder a la aplicación. Contiene la interfaz con el usuario mostrándole la evolución de la regata y atendiendo a sus acciones. Se comunica con el servidor de regata, al que le envía las acciones y del que recibe periódicamente una sincronización con el estado oficial de la regata. Una vez inicializado por el servidor con su estado, mientras no reciba ningún estado entre sincronizaciones, es capaz de hacer evolucionar su estado local por

sí mismo, aunque sin tener en cuenta los cálculos más pesados, como colisiones entre barcos, etc.

- **Servidor Web:** permite al usuario registrarse, autenticarse (si lo desea) y seleccionar la regata en la que quiere participar. Cuando un usuario selecciona una regata, se encarga de obtener el servidor que la simula y enviar al usuario un applet configurado para acceder a él.
- **Catálogo:** sabe qué regatas están activas y qué servidor de regata se encarga de cada una.
- **Servidor de Regata:** simula una y sólo una regata, y contiene su estado oficial, siendo capaz de hacerlo evolucionar en base al paso del tiempo y a las acciones que los clientes le transmiten de sus usuarios. Cada cierto tiempo difunde el estado del juego a todos sus clientes. Los servidores de regata se hospedan en un host.
- **Host:** capaz de hospedar múltiples servidores de regata siempre que usen puertos distintos.

Como se ha mencionado, `BogReg` es una aplicación que simula regatas, es decir, competiciones de barcos. En dichas regatas, los usuarios pueden entrar bien como espectadores, bien tomando partido en ellas. Una regata se compone de uno o más circuitos que deben ser completados por todos los participantes en ella. Dependiendo del orden de clasificación de cada uno de ellos, se les asignará una puntuación.

Para participar, cada usuario debe acceder a la página web de la aplicación, registrarse (aunque no es necesario autenticarse) y acceder a la colección de regatas activas donde, en caso de querer participar, tiene la opción de conseguir un barco vitual. Aquellos usuarios que tengan una identidad en el sistema, pueden hacer un seguimiento de sus participaciones y formar parte de una clasificación global.

Las regatas se componen de diferentes factores: por un lado están los factores invariantes, es decir, los que no evolucionan con el tiempo. En este grupo tenemos, por un lado, las corrientes marinas y el viento que, debido a la duración de una carrera, aproximadamente 10-15 minutos, no sufren gran variación (es raro que en 10-15 minutos cambie un viento de 20 km/h dirección norte a uno de 190 km/h dirección oeste, por poner un ejemplo extremo), por lo que no es necesario que se vayan generando en tiempo de ejecución, lo cual complicaría el motor físico de la aplicación; y por otro lado, las boyas ubicadas en el terreno que definen la configuración de un circuito, la cual es invariante para cada carrera. Todos estos factores son cargados desde fichero y posteriormente enviados al cliente, quien no los

modifica. Por otro lado, existen los factores dinámicos, que pueden variar con el tiempo. En este grupo se incluyen los barcos participantes, tanto los que son comandados por clientes reales (uno por cliente) como los que son comandados por el sistema (bots).

1.2.1. Funcionamiento de la aplicación

Inicialmente, un usuario accede al servidor web mediante el navegador de su máquina cliente. Una vez registrado, obtiene las regatas activas y selecciona aquella en la que quiere participar. Habiéndola seleccionado, el servidor web genera un applet configurado para acceder a su servidor de regata asociado. Es menester resaltar que existe una relación 1 a 1 entre regata y servidor de regata, es decir, un servidor ejecuta una sola regata, y una regata es ejecutada en un solo servidor.

Una vez el cliente recibe el applet, lo usa para conectarse al servidor de la regata y obtener el estado actual del juego. A partir de ese momento, el applet actúa de interfaz con el usuario, mostrándole la evolución de la regata y aplicando las acciones que éste realiza (p.e.: virar). Aplicar acciones implica tanto reflejarlas en el estado local del applet como enviarlas al servidor de la regata para que las aplique al estado oficial. Ambos, servidor y applet, son también capaces de hacer evolucionar el estado en base al tiempo.

Las diferencias fundamentales entre applet y servidor radican en que el applet es incapaz de realizar los cálculos más costosos como las colisiones entre barcos y las penalizaciones, además de que sólo aplica directamente las acciones de su propio usuario. El resto de acciones las aplicará cuando reciba el estado oficial del servidor, el cual lo difunde cada cierto tiempo. A su vez, el applet envía regularmente mensajes de `KEEP_ALIVE` al servidor para indicar que sigue vivo.

1.3. Problemas a Resolver

El modelo de `BogReg` descrito en la secciones anteriores actualmente se encuentra en producción y funciona correctamente, aunque presenta una serie de problemas que serán abordados en esta tesis, los cuales ya han sido esbozados en la sección 1.1. De los cuales, la falta de legibilidad y la nula transparencia de ubicación quedan relegados a un segundo plano, siendo el cuerpo principal de la tesis los siguientes:

- **Tolerancia a fallos:** el modelo actual se basa en una arquitectura cliente-servidor centralizada, es decir, un único servidor se encarga de proporcionar una regata en particular. Este hecho propicia que, en caso

de fallo del servidor, al no existir más réplicas, se deje de proporcionar servicio. La solución evidente de este problema sería replicar el servidor para que, en caso de caída, existiera una o más réplicas de apoyo que prosiguieran con su función.

- **Retardos elevados:** al existir un único servidor, todos los clientes deben conectar con él para tener servicio, lo que provoca que existan diferentes retardos, y que los clientes geográficamente más alejados del servidor obtengan unos retrasos mayores, originando "saltos" durante la ejecución de la aplicación. La forma de solventar directamente este problema sería acercar los servidores a los clientes lo máximo posible.
- **No escalable:** al existir un único servidor que procese todas las operaciones de todos los clientes, existe un punto en que el servidor se colapsa debido al crecimiento de clientes, al no poder aplicar todas las operaciones que le llegan, por lo que el límite de clientes viene dado por la capacidad del servidor de aceptarlos. Dos soluciones simples serían bien aumentar la capacidad de carga del servidor, la cual llegaría un punto en el que no pudiera aumentar más, o bien repartir la carga de trabajo entre diferentes servidores.

1.4. Soluciones existentes

En la solución actual, dada una regata, sólo existe un servidor que la simule. No obstante, en las soluciones propuestas en esta sección y desarrolladas en posteriores capítulos, esta restricción se relaja, por lo que se asume que hay, al menos, un servidor por regata, el servidor maestro, pero puede haber más, los servidores delegados. Cuando un cliente elija participar en una regata, lo hará siempre a través del servidor hospedado en el host más cercano, reduciendo los tiempos de transmisión de los mensajes entre cliente y servidor y permite que, potencialmente, pueda existir más de un servidor por regata, forzando a que exista cierta coherencia entre los estados de cada uno de ellos. Por coherencia se entenderá que posean un estado lo suficientemente parecido como para que no afecte a las simulaciones, no que tengan exactamente el mismo estado entre ellos.

Ninguna de las soluciones propuestas afecta en ningún modo a la funcionalidad que perciben los clientes ni al funcionaminto del applet, aunque sí que pueden ser susceptibles de modificaciones el servidor web, el host y el catálogo. Las modificaciones se centrarán especialmente en los servidores de la regata. Las principales diferencias entre soluciones radican en el nivel de replicación y consistencia entre réplicas que asume cada solución:

- *Solución basada en forwarders:* existe un servidor central, el **servidor maestro**, que se encarga de hacer progresar el estado de la regata.

Existe también un conjunto de **servidores delegados** que actúan como pasarelas y realizan el papel de caché de almacenamiento del último estado comunicado por el servidor maestro.

- *Solución basada en distribución de la carga:* todos los servidores son capaces de hacer evolucionar su estado local. No obstante, cada *servidor delegado* se encarga de calcular el efecto de las operaciones de los clientes que tiene directamente asociados, enviando el diferencial al *servidor maestro*. Por tanto, existe un servidor maestro encargado únicamente de calcular las colisiones y penalizaciones de los barcos que pertenecen a usuarios de distintos servidores, propagando el resultado a los servidores delegados.

- *Solución basada en replicación de servidores:* todos los servidores son capaces de aplicar todas las operaciones y realizar todos los cálculos. Cada servidor delegado calcula las operaciones referentes a los clientes que tiene asociados, difundiendo los diferenciales al resto de servidores. El servidor maestro queda relegado a una función de sincronización para con el resto de servidores. Se sigue cierto criterio a la hora de decidir quién realiza los cálculos referentes a la interacción de barcos de diferentes servidores.

1.5. Organización del resto de la Tesis

La presente tesis describe el proceso seguido para una aplicación real, a la que se denomina BogReg, y cuyo nombre real es TACTICAT, correspondiente a un proyecto de investigación y desarrollo del Intituto Tecnológico de Informática de Valencia, en el grupo de Sistemas Distribuidos, con el propósito de reducir lor retardos en los mensajes desde el servidor a los diferentes clientes y aportar diferentes modelos de distribución a la aplicación.

La presente tesis se organiza de la siguiente forma: en primer lugar, se realiza una introducción general a lo que sería este documento y a la aplicación. Posteriormente, se analizan diferentes conceptos clave a la hora de entender el proceso realizado durante el desarrollo de la tesis. Como siguientes pasos se describen las diferentes aportaciones realizadas, a saber:

- Refactorización del código: donde se muestra un proceso de reorganización de la aplicación, con el fin de tener las entidades que lo componen lo más diferenciadas posibles para facilitar el proceso de distribución.

- J-Sim: Una breve reseña al entorno de simulación empleado, así como al modelo de pruebas del que se ha hecho uso.

- Modelado de la aplicación: En este capítulo se detalla el funcionamiento inicial de la aplicación.
- Modelo basado en repetidores: Se describe el primer modelo de distribución propuesto, basado en unos delegados que actúan a modo de pasarela entre los clientes y el servidor.
- Modelo basado en reparto de carga: Describe el segundo modelo de distribución propuesto, basado en delegados que poseen cierta lógica del simulador.
- Modelo basado en servidores replicados: Describe el último modelo propuesto, basado en servidores iguales que cooperan para ofrecer una imágen única del estado.
- Conclusiones
- Bibliografía.

Capítulo 2

Conceptos Básicos

Antes de entrar en detalle con las soluciones adoptadas en la presente tesis ante los problemas propuestos en la sección 1.3, se procede a presentar una serie de conceptos teóricos necesarios para comprender correctamente las soluciones propuestas. Dichos conceptos no son empleados literalmente, sino más bien aportan una base esencial sobre la cual se ha desarrollado la tesis.

2.1. Sistemas Distribuidos: Definición adoptada

Encontrar una única definición que englobe toda la gama de posibles sistemas distribuidos es realmente complicado: cada definición puede prestar especial atención a un determinado detalle dentro de lo que sería el sistema. Una definición genérica sería la aparecida en [CDK94], en la que dice que " un sistema distribuido es una colección de ordenadores autónomos o interconectados mediante una red y equipados con unos programas que permiten a estos ordenadores coordinar sus actividades y compartir sus recursos". Frente a esta definición encontramos la definición aportada en [TS01], en la que expresa un sistema distribuido como "un conjunto de ordenadores independientes que ofrecen a sus usuarios la imagen de un sistema coherente único", habiendo subrayado en [TR85] el concepto de *trasparencia*.

Definiciones similares se pueden encontrar en [SS94] y [SG98], donde se hace especial hincapié en que para existir un sistema distribuido ha de existir paso de mensajes, frente a [Nut97] que propone englobar sistemas de paso de mensajes con sistemas de memoria compartida bajo el término de *multicomputador*, u otras alternativas como [Jua94] y [Mas98], que proponen un modelo con memoria compartida distribuida.

Uniendo las anteriores definiciones, se toma como definición empleada para sistema distribuido " **un conjunto de ordenadores interconecta-**

dos mediante una red, con un conjunto de capas de software que pueden coordinar sus actividades y/o compartir recursos, que se comunican mediante mensajes y que ofrecen una imagen de sistema coherente único".

Entre los objetivos que trata de cumplir un sistema distribuido tenemos:

1. Facilitar el acceso de los usuarios a ciertos recursos remotos.
2. Proporcionar transparecia de distribución, dentro de la cual se englobará:
   - Transparencia de acceso, que oculta el modo de acceso a los recursos, sean remotos o locales.
   - Transparencia de ubicación, que oculta dónde se ubican los recursos.
   - Transparencia de migración, que oculta el hecho de trasladar un recurso a otro lugar.
   - Transparencia de reubicación, que oculta el traslado de un recurso durante su ejecución.
   - Transparencia de replicación, que oculta si se está accediendo a la copia principal o a una de las réplicas del recurso
   - Transparencia de concurrencia, que oculta si están teniendo lugar varios accesos al recurso.
   - Transparencia de fallos, que oculta los errores del sistema y fallos en los componentes, y su recuperación.
   - Transparencia de persistencia, que oculta si un recurso está en memoria volátil o persistente.
3. Proporcionar escalabilidad al sistema, permitiendo que se pueda ampliar el número de servidores, distribuyendo la carga del sistema y aumentando la cantidad de clientes que pueden ser atendidos.

2.2. El modelo Cliente-Servidor

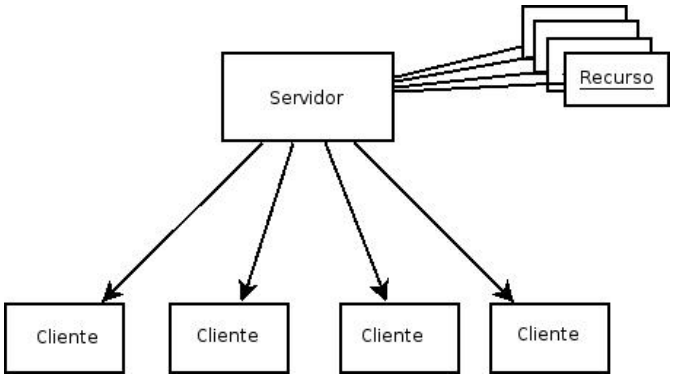
El esquema Cliente-Servidor consiste en un modelo en el que dos o más procesos deben colaborar para el progreso mutuo. De estos procesos, para simplificar la explicación, se asume la existencia de únicamente de dos, uno suele realizar las tareas más pesadas, al cual se denomina servidor, y otro posee la interacción con el usuario, que será el cliente. Este modelo puede variar de modo que el cliente puede poseer cierta capacidad de cómputo, el servidor una interfaz gráfica, etc. Se pueden diferenciar tres componentes esenciales:

1. El Cliente.
2. El Servidor.
3. La infraestructura de comunicaciones.

EL CLIENTE. El proceso cliente es el que generalmente realiza la interacción con el usuario, la mayoría de veces en forma gráfica. Posee procesos auxiliares que se encargan del establecimiento de conexión y de las comunicaciones con el servidor, así como tareas de sincronización.

EL SERVIDOR. El proceso servidor, como su nombre indica, proporciona un servicio al cliente, del cual devuelve los resultados. Puede poseer, aunque no siempre, procesos auxiliares que se encargan de las tareas de comunicación con el cliente. Dado que es el que proporciona los resultados al cliente, la carga computacional asociada a éstos es mayor que la de los clientes.

LAS COMUNICACIONES. Para que los clientes y los servidores puedan comunicarse se requiere una infraestructura de comunicaciones, la cual proporciona los mecanismos básicos de direccionamiento y transporte. Las comunicaciones se pueden realizar por medios tanto orientados a la conexión (TCP) como no orientados a la conexión (UDP). La red debe tener características adecuadas de rendimiento, confiabilidad, transparencia y administración.



Ventajas que ofrece el esquema Cliente-Servidor:

1. La gestión de recursos se centraliza en los procesos servidores, por lo que las aplicaciones son mucho más sencillas, modulares y fácilmente

mantenidas. En caso contrario, sería necesaria una gestión de recursos a nivel global del sistema, que complicaría la lógica de la aplicación y que requeriría un mayor número de rondas de mensajes para evitar condiciones de carrera, como por ejemplo, que varios procesos intentaran acceder a un mismo recurso en forma de escritura al mismo tiempo.

2. Se pueden utilizar componentes, tanto de hardware como de software, de varios fabricantes, lo cual contribuye considerablemente a la reducción de costos y favorece la flexibilidad en la implantación y actualización de soluciones.

3. Facilita la integración entre sistemas diferentes y compartir información.

4. Si se utilizan interfaces gráficas para interactuar con el usuario, no siempre es necesario transmitir información gráfica por la red dado que ésta puede residir en el cliente, lo cual permite aprovechar mejor el ancho de banda de la red.

5. Es más rápido el mantenimiento y el desarrollo de aplicaciones pues se pueden emplear las herramientas existentes (APIs de Java, RPCs, etc.).

6. La estructura modular facilita la integración de nuevas tecnologías y favorece la escalabilidad de las soluciones.

Inconvenientes que ofrece el esquema Cliente-Servidor:

1. Deben existir mecanismos independientes de la plataforma y generales para la interacción entre cliente y servidor (RPC, ROI, etc.).

2. La seguridad en el modelo que nos ocupa es más que recomendable, en especial para evitar usos maliciosos del sistema, bien voluntarios, bien involuntarios. La encriptación de los mensajes es un paso en esta dirección.

3. Pueden presentarse problemas inherentes a la red de comunicaciones tales como congestión en la red, fallos de nodos, bucles en el encaminamiento, etc.

2.3. Tolerancia a fallos

En la presente sección se procede a definir el concepto de **fallo** así como una clasificación de éstos y un conjunto de estrategias para tolerarlos, con el propósito de aumentar la disponibilidad (véase apartado2.4).

2.3.1. Concepto de fallo

Según [Gär99], existen 3 niveles en los fallos: faltas, errores y fallos. Pese a la ambigüedad de la literatura asociada al tema, [Gär99] separa los términos en base a que una falta, situada a un nivel bajo de abstracción, puede producir un error, relacionado con el estado que, a su vez, provoca un fallo en el sistema, en cuanto a su funcionamiento o especificaciones. Por tanto, tomando esta clasificación, y en base a [Nel90], su puede definir *fallo* como la incapacidad de un sistema de presentar la funcionalidad para la que fue diseñado. Así pues, una *falta* sería la condición anómala para que tuviera lugar un fallo.

2.3.2. Clasificación de fallos

Una vez definido el concepto de fallo, se proponen dos clasificaciones recogidas en [TS01], según su persistencia temporal o según el comportamiento.

- En cuanto a la persistencia temporal del fallo, se tiene:
    1. Transitorios: cuando un elemento del sistema tiene un comportamiento anómalo de forma puntual, recobrando inmediatamente su comportamiento lógico.
    2. Intermitentes: un elemento del sistema se comporta de forma anómala de forma intermitente.
    3. Permanentes: cuando se produce el fallo del componente del sistema, éste no se vuelve a recuperar.
- En cuanto al comportamiento del elemento que falla:
    1. Fallos de parada: El elemento que falla deja de funcionar y no interfiere con el resto del sistema una vez ha fallado.
    2. Fallos de omisión: El elemento que falla no realiza cierta parte de su cometido.
    3. Fallos de temporización: Llega a proporcionar servicio, pero el elemento que falla no lo hace en el tiempo previsto.
    4. Fallos de respuesta: El elemento responde incorrectamente a las peticiones que se le realizan. Dentro de este grupo se englobarían los fallos expuestos en [Lam83] Bizantinos, que son respuestas incorrectas indetectables por el sistema.
    5. Fallos arbitrarios: El componente que falla funciona de forma aleatoria y descontrolada.

Existen numerosas especificaciones más, como las expuestas en [Cri91] y [Sch93], aunque se ha optado por la de [TS01] por ser una de las más recientes y relativamente completa.

2.3.3. Fallos en entornos cliente/servidor

El modelo cliente/servidor, que ha sido expuesto en 2.2, puede ser objeto de diversos fallos:

- El cliente no encuentra al servidor: sin solución, únicamente tratado con excepciones.
- La petición del cliente se pierde: no ocurre en entornos TCP. Se retransmite la petición tras cierto tiempo. Se deben detectar las peticiones duplicadas en el servidor para descartarlas.
- El servidor falla:
  1. Al recibir la petición: actúa como el caso anterior.
  2. Al procesar la petición: se proporcionan diferentes semánticas, descritas a continuación.
  3. Al responder: actúa como el caso siguiente.
- La respuesta del servidor se pierde: retrasmitir el mensaje del cliente, que el servidor debe identificar para responderle con los resultados previos.
- Cliente falla antes de recibir respuesta: petición huérfana. Dentro del sistema BogReg pueden existir dos alternativas de gestión de éstas: por un lado, al poder detectarse las caídas de los clientes, sería factible anular todas las peticiones que queden por atender del cliente que haya fallado. No obstante, dado que las peticiones que realiza un cliente en BogReg únicamente afectan al estado de su propio barco, no hay problema en procesar las peticiones huérfanas, dado que una vez detectado el fallo, se eliminará toda la parte del estado referente a ese cliente.

Las semánticas posibles a las que se hacía referencia anteriormente son:

1. Como mínimo una vez: se debe garantizar que el servidor ejecuta el servicio como mínimo una vez.
2. Como máximo una vez: no se reintentan las operaciones fallidas.
3. Exactamente una vez: unión de las dos anteriores.

2.3.4. Tolerancia a fallos

Existe numerosa bibliografía ([CKV01], [BMST92] y [DSS98] por citar algunos ejemplos) que, a pesar de exponer diferentes conceptos, aunque relacionados entre sí, coinciden en la idea de que el mejor mecanismo para aumentar la disponibilidad de un recurso (ver sección 2.4) y mejorar la tolerancia a fallos de un sistema es la replicación, que será tratada en la sección 8.1.

2.4. Alta Disponibilidad

En esta sección se dispone a definir el concepto de Alta Disponibilidad, además de detallar los conceptos previos necesarios para entenderla: fallos (sección 2.3), fiabilidad y disponibilidad. En el entorno BogReg, la disponibilidad es un factor vital, dado que se trata de una aplicación de simulación en tiempo real que debe sincronizarse continuamente, así como realizar ciertos cálculos que únicamente pueden hacerse en el servidor. Si, por el motivo que sea, el servidor deja de prestar su servicio, la aplicación no puede continuar, dado que no puede ni avanzar el estado correctamente ni sincronizarlo con el estado "oficial" del servidor.

2.4.1. Concepto de fiabilidad

Se entenderá por fiabilidad la capacidad (probabilidad) de un sistema de proporcionar un servicio correcto en un instante determinado t sabiendo que el servicio era correcto en un instante inicial t=0, tal como se destila de [Nel90]. La fiabilidad es otro concepto altamente vinculado con la disponibilidad y depende de dos factores: los fallos que puedan tener lugar en el sistema y los mecanismos que se posea para tolerarlos.

2.4.2. Concepto de Disponibilidad

La disponibilidad de un sistema [Nel90] consiste en la probabilidad de que un sistema se encuentre operativo en un determinado instante de tiempo. Dado que el objetivo es obtener una Alta Disponibilidad, será necesario que el sistema provea de una alta fiabilidad y un tiempo mínimo de recuperación ante fallos.

2.4.3. Concepto de Alta Disponibilidad

Se entiende que un sistema es altamente disponible cuando su nivel de disponibilidad supera el 99'999 % de su tiempo de vida, es decir, cuando se minimiza al máximo el tiempo que no ofrece su servicio por causa de un fallo.

Para plasmar mejor los conceptos de disponibilidad y fiabilidad, nos apoyaremos en un ejemplo: supongamos un servidor web de cualquier caja de ahorros o banco. Una falta de disponibilidad se daría en el caso de que se intentase acceder a la página web de la entidad y nos diese un fallo de que no está, valga la redundancia, disponible (Error 404).

Por contra, para entender lo que sería una falta de fiabilidad, supongamos un cliente de cierta aplicación que únicamente realiza una media de 20 peticiones/año y que éstas suelen durar 30 segundos cada una, pero para que cada solicitud tenga éxito, el servidor debe garantizar que el stream con

los resultados no se corte ni que ninguno de los valores retornados pueda ser alterado.

Se tiene dos tipos de servidores, cada uno de ellos 99'3 % disponible. El primero sólo falla un día entero por año, y es incapaz de proporcionar un resultado durante todo ese día, pero funciona perfectamente el resto del año. El segundo falla durante 90 ms en todos los intervalos de 30 segundos, corrompiendo algunos de los bytes transmitidos y sus CRCs asociados, impidiendo que se detecten tales errores. **¿Cuál es más fiable?**

Con el primero, la probabilidad de que alguna de las 20 peticiones anuales para cada cliente falle es bastante baja, obteniendo una alta fiabilidad. En el segundo caso, no se consigue que ninguna de las peticiones efectuadas por los clientes tenga éxito, teniendo una fiabilidad nula.

Entre las técnicas para lograr una Alta Disponibilidad se encuentra la replicación 8.1, así como establecer servicios de pertenencia a grupos para detectar los fallos y actuar en consecuencia.

2.5. Elección de líder

El problema de la elección de líder consiste en elegir un proceso de entre varios, que se ejecutan en diferentes máquinas, en un sistema distribuido, para que actúe a modo de coordinador de los procesos y permita tomar decisiones globales.

Dentro del sistema BogReg, el entorno donde se haría patente la necesidad de un sistema de elección de líder es en el descrito en la sección 8.1: en el caso de que el servidor encargado de coordinar todas las réplicas dejara de estar disponible, se podría elegir un nuevo coordinador el cual continuase normalmente con la simulación, convirtiéndose en el nuevo coordinador de la regata y haciendo de su estado local el estado oficial de ésta. Para ello se requiere cierto algoritmo de elección que, dada la naturaleza del problema de elección de líder, debe ser un algoritmo distribuido, aunque bien se puede simplificar a un algoritmo centralizado que tuviese lugar en un servicio que conociese a todos los servidores participantes en la simulación y si están operativos o no.

Se deben tener en cuenta las siguientes consideraciones, teniendo en cuenta que los algoritmos que se van a emplear son algoritmos distribuidos:

1. Un grupo de procesos en diferentes máquinas requieren de la elección de un coordinador.

2. Comunicación punto a punto: cada proceso puede enviar mensajes a cualquier otro proceso.

3. Se asume que cada proceso tiene un identificador único.

4. Se asume que la prioridad del proceso $\texttt{P}_i$ es `i`

2.5.1. Algoritmo del Dictador

Cualquier proceso $\texttt{P}_i$ envía un mensaje al actual coordinador. Si no hay respuesta en `T` unidades de tiempo, $\texttt{P}_i$ trata de autoproclamarse líder. Pasos a seguir:

En el proceso $\texttt{P}_i$ que ha detectado el fallo del coordinador:

1. *$\boldsymbol{P}_i$ envía un mensaje "ELECCIÓN" a todo proceso con mayor prioridad.*

2. *Si ningún proceso responde, $\boldsymbol{P}_i$ ejecuta el código de coordinador y manda un mensaje a todos los procesos de menor prioridad declarándose como líder "$\boldsymbol{P}_i$ ELEGIDO".*

3. *En caso contrario, $\boldsymbol{P}_i$ espera **T'** unidades de tiempo para recibir noticias del nuevo coordinador. En caso de no obtener respuesta, se reinicia el algoritmo.*

En el resto de procesos (también llamados $\mathbf{P}_i$):

1. Si $\mathbf{P}_i$ no es el coordinador, entonces recibirá los mensajes de $\mathbf{P}_j$.

2. Si $i < j$, $\mathbf{P}_i$ envía "$\mathbf{P}_j$ elegido". $\mathbf{P}_i$ se actualiza para decir que $\mathbf{P}_j$ es el coordinador.

3. Si $i > j$, $\mathbf{P}_j$ envía "ELECCIÓN". $\mathbf{P}_i$ responde a P_j diciendo que está vivo. $\mathbf{P}_i$ inicia elección.

2.5.2. Algoritmo en anillo

Este algoritmo asume que los procesos conforman un anillo: cada uno sólo envía mensajes al siguiente proceso del anillo. Además existe una "Active list" con información de todos los demás procesos activos. También se asume que un mensaje progresa en el anillo aunque uno de los procesos del mismo haya caído.

Contexto: cualquier proceso $\mathbf{P}_i$ envía mensajes al coordinador actual. Si no hay respuesta en T unidades de tiempo, $\mathbf{P}_i$ inicia la elección:

- Inicializa su "Active list" (AL) a vacío.

- Envía un mensaje "ELEGIR(i)" a la derecha y añade i a la AL.

Si un proceso recibe un mensaje "ELEGIR(j)":

1. Es el primer mensaje enviado o visto: inicializar su AL a `[i,j]` y enviar "ELEGIR(i)" y "ELEGIR(j)".

2. Si i != j: añadir i a la AL y redirigir "ELEGIR(j)" a la AL.

3. Si $i == j$, el proceso i ha completado el grupo de procesos activos en su AL. Elige el proceso de Id más elevado y envía "ELEGIDO(x)" a su vecino.

Si un proceso recibe un mensaje "ELEGIDO(x)":

1. Establece su coordinador a x.

2. Reenvía el mensaje "ELEGIDO(x)" al siguiente vecino.

2.6. Pertenencia a Grupos

2.6.1. Motivación

Dada la complejidad que supone el manejo de múltiples nodos dentro de un sistema distribuido, resulta útil a la hora de diseñar los algoritmos necesarios para el funcionamiento del sistema conocer qué nodos o procesos están activos en cada momento de ejecución, no una noción simplista de la composición del sistema, sino una visión global compartida por todos. Con esta finalidad surgen los protocolos o servicios de pertenencia a grupos, que aseguran el acuerdo entre todos los nodos activos del sistema.

Los primeros servicios con esta finalidad fueron descritos en [Cri91] para sistemas distribuidos síncronos. Por otro lado, en [JFR93] se describe el mismo modelo para sistemas asíncronos, aunque más tarde se demostró [CHT96] que este servicio no era implantable en dichos entornos. A pesar de ello, en cualquier sistema actual de comunicaciones de grupos, suele existir un servicio de pertenencia [CKV01].

Los servicios de pertenencia a grupos se apoyan en un resultado teórico sobre la detección de fallos [CT96]: por tanto, no son perfectos, ni lo podrán ser en un entorno asíncrono, aunque sí que son empleables si los algoritmos que los requieran conocen sus limitaciones y actúen en consecuencia.

2.6.2. Ventajas de un servicio de pertenencia

Como se ha explicado anteriormente, los servicios de pertenencia a grupos proporcionan una visión global de los nodos o procesos activos en el sistema. Este hecho proporciona tres ventajas fundamentales a la hora de implantar un sistema distribuido:

- Por un lado, el hecho de conocer los nodos activos en un determinado instante permite que todos los componentes del sistema puedan tomar acciones uniformes ante una caída o recuperación de un determinado proceso o nodo.
- Por otro lado, se simplifican los algoritmos necesarios para tratar las situaciones de fallo o reincoporación, teniendo la garantía de que todos los nodos perciben la misma secuencia de eventos de este tipo.
- Como tercer aspecto, el servicio de pertenencia monitoriza a todos los miembros de un grupo de comunicaciones, por lo que, en caso de que el servidor coordinador del sistema sufriese un fallo, el servicio de pertenencia sería el primero en darse cuenta, procediendo a ejecutar la elección de líder, eligiendo un nuevo coordinardor que prosiguiese con la simulación normalmente.
- Finalmente, proporcionan la base necesaria para implantar un modelo de ejecución con sincronía virtual y permite la realización de difusiones atómicas.

2.7. Modelos de Replicación

Como se ha mencionado en la sección 2.3, la replicación es la principal vía para tolerar fallos, así como un mecanismo para aumentar la disponibilidad dado que, ante un mayor número de réplicas, un mayor número de peticiones se podrán atender (al menos, peticiones de lectura). En la presente sección se describen los principales modelos de replicación. Cada modelo se caracteriza por:

- El número de réplicas que pueden servir directamente las peticiones de los clientes o *réplicas activas*.
- La forma de propagar las peticiones.
- La forma de propagar las actualizaciones.
- La estrategia para soportar caídas de las réplicas.
- Su capacidad para servir operaciones no deterministas.

2.7.1. Modelo Activo

El modelo de replicacion activo, descrito en [Sch93] y [PWS+00], se caracteriza porque todas las réplicas del servicio son activas, es decir, atienden peticiones de los clientes, por lo que las peticiones que llegan deben ser difundidas antes de ejecutarlas, para luego realizar un filtrado de respuestas, que tiene lugar en el stub cliente. Para lograr la consistencia entre las diferentes réplicas se requiere un protocolo de difusión fiable de orden total.

Entre los inconvenientes que presenta este modelo cabe destacar, en primer lugar, que hay que garantizar que las peticiones lleguen a todas las réplicas, es decir, se requieren protocolos de **difusión fiable**, que además, deben garantizar que las peticiones con interacción entre sí lleguen en el mismo orden a todas las réplicas, es decir, con **orden total**. Por otro lado, resulta difícil atender operaciones no deterministas, dado que cada réplica puede originar respuestas diferentes, dando lugar a inconsistencias. Además, hay una alta dificultad en gestionar las peticiones iniciadas por un objeto replicado, es decir, múltiples peticiones deberían ser identificadas como una sola por el objeto invocado. Finalmente existe la necesidad de un filtrado de peticiones y un filtrado de respuestas.

Entre las ventajas del modelo de replicación activo encontramos, en primer lugar, que todas las réplicas comparten un mismo código y un mismo estado. Por otro lado, la caída de una o más réplicas no requiere un cambio de rol en las réplicas restantes, además de no retardar la entrega de una respuesta a un cliente.

2.7.2. Modelo Pasivo

En el modelo de replicación pasivo, [BMST92] y [PWS+00], únicamente existe una réplica capaz de servir las peticiones, llamada réplica *primaria*, y múltiples réplicas de respaldo llamadas *secundarias*. Todos los clientes mandarán sus peticiones a la réplica primaria, la cual servirá la petición, actualizará su estado, y difundirá los cambios, en caso de haberlos, a las réplicas secundarias, que actuarán a modo de réplicas de respaldo.

La principal desventaja de este modelo radica en el caso de que la réplica primaria caiga, pues deberá iniciarse un protocolo de elección de líder (2.5) que propiciará el cambio de una réplica secundaria a primaria, con el consecuente cambio en los clientes, que deberán modificar a quién dirigen sus peticiones, así como la posibilidad de pérdida de peticiones o de ejecutar más de una vez una petición. A diferencia de otros modelos, en este modelo el rol de cada réplica permanece estático a no ser que tenga lugar un fallo: de ahí surge su segunda desventaja, la existencia de código diferente para

cada rol, dado que primario y secundario se comportan de diferente manera.

Entre las ventajas de este modelo podemos encontrar que no se sobrecarga el sistema, pues sólo existe una réplica que procesa las peticiones. Por otro lado, no se requieren protocolos de difusión en orden, dado que sólo existe un emisor, aunque sí debe garantizarse que un checkpoint ha sido entregado en todas las réplicas secundarias antes de contestar al cliente. Además, soporta sin problemas la ejecución de operaciones no deterministas, dado que al propagar el estado resultante, los secundarios no pueden generar inconsistencias. Finalmente, se facilita la implantación de mecanismos de control de concurrencia, dado que pueden ser locales.

2.7.3. Modelo Semiactivo

Descrito en [Pow94], el modelo semiactivo consiste en un modelo intermedio que comparte algunas características con los dos principales modelos: el activo y el pasivo:

- Las peticiones se envían a todas las réplicas, por lo que se siguen necesitando difusiones fiables, aunque el orden total puede establecerlo el líder.

- Sólo una de las réplicas, la líder, procesará absolutamente todas las peticiones, y generará tanto las respuestas como las peticiones sobre otros servidores.

- El resto de réplicas puede procesar directamente aquellas peticiones que sean deterministas, aunque no podrán generar respuestas, ni iniciar peticiones, y deben esperar el resultado de la ejecución por parte del líder para las operaciones no deterministas.

Entre los inconvenientes del modelo semiactivo podemos encontrar que, primero, resulta necesario un protocolo de difusión fiable para entregar las peticiones, aunque es mucho menos costoso que un protocolo de difusión atómica [HT93]. Además, necesita asegurar el orden total en la ejecución de las operaciones no dirigidas por la réplica líder. Por último, hay que etiquetar de alguna manera las operaciones no deterministas, para que tanto el líder como sus seguidores las ejecuten de manera adecuada.

Entre las ventajas, se tiene que la recuperación ante fallo es rápida, dado que es sencillo elegir otro líder en caso de caída de éste; se pueden ejecutar operaciones no deterministas; y no se requiere filtrar las peticiones ni las respuestas.

2.7.4. Modelo Semipasivo

Según [DSS98] y [PWS+00], el modelo de replicación semipasivo es similar al pasivo, aunque trata de evitar algunos problemas, tales como la dependencia sobre la viveza de la réplica primaria, la reconfiguración rápida en caso de fallo de dicha réplica y reducir los riesgos en cuanto a peticiones en curso. Para ello, sobre el modelo pasivo, se introducen una serie de variaciones:

1. En primer lugar las peticiones realizadas por los clientes llegan a todas las réplicas.
2. También se introdce una fase de consenso que en la práctica se empleará para propagar actualizaciones y confirmar su aplicación.
3. Cuando se sospecha de la caída de la réplica primaria, todos envían la información que poseen en cuanto al procesamiento de la última petición al nuevo primario:
   - Si nadie recibió el resultado de su ejecución, el nuevo primario la vuelve a ejecutar.
   - Por el contrario, si algún participante obtuvo el resultado, el nuevo primario lo difunde a todos los demás normalmente.

En caso de caída del primario, cuando los detectores de fallos locales así lo indiquen, se enviará un mensaje `estimate` al nuevo primario, con el resultado de la última actualización conocida. En base a tal mensaje, el nuevo primario continuará con la última petición interrumpida por el fallo.

Entre las principales desventajas del modelo, cabe destacar que requiere de una difusión fiable en orden total para propagar las peticiones, y que necesita de tres rondas de mensajes para poder aplicar las actualizaciones de estado, lo que hace que el número total de rondas sea igual a cinco, incluyendo el mensaje de petición y el primer mensaje de respuesta.

En el caso de las ventajas, cabe señalar que no se necesita un acuerdo sobre la caída del primario, dado que en caso de sospecha incorrecta, lo que haga el nuevo primario quedará descartado; además de no introducir una sobrecarga excesiva al sistema. Por otro lado, el cliente no tiene que reenviar ninguna petición en caso de fallo del primario. Finalmente, el protocolo seguido para reconfigurar el servicio replicado ya incluye la finalización de la operación que estuvo en curso en el momento de la caída.

Capítulo 3

El proceso de refactorización

3.1. Introducción

En el presente capítulo se procede a describir la primera aportación realizada a la aplicación: un proceso de reestructuración del código original. Inicialmente, la aplicación BogReg, que está implementada en JAVA, sigue un esquema parecido al de programación estructurada, en lugar de seguir el paradigma de la orientación a objetos, aunque sí que define algunas entidades (objetos) dentro de dicho esquema. El principal motivo para presentar el código de la aplicación de esta forma, es la poca familiarización de los primeros desarrolladores de ésta con la orientación a objetos, dado que se trataba de físicos, más familiarizados con lenguajes de programación estructurada, tales como C o Fortran.

BogReg presenta una importante restricción a la hora de desarrollar este capítulo de la tesis: sus detalles de diseño, así como su código, son de licencia propietaria, por lo que es posible que los esquemas que se presenten aquí no sean exactos, ni completamente detallados, ni tengan un nivel de precisión alto. No obstante, se tratará de suplir dichos factores de la mejor forma posible.

3.2. Orientación a Objetos

La orientación a objetos consiste en un paradigma de programación que facilita la creación de software de calidad por sus factores que potencian el mantenimiento, la extensión y la reutilización del código generado bajo este paradigma. Trata de amoldarse al modo de pensar del hombre y no al de la máquina, lo que es posible gracias a la forma racional con la que se manejan las abstracciones que representan las entidades del dominio del problema, y a propiedades como la jerarquía o el encapsulamiento.

El elemento básico de este paradigma no es la función (elemento básico de la programación estructurada), sino un ente denominado objeto. Un objeto es la representación de un concepto para un programa, y contiene toda la información necesaria para abstraer dicho concepto: los datos que describen su estado y las operaciones que pueden modificar dicho estado, y determinan las capacidades del objeto. Java incorpora el uso de la orientación a objetos como uno de los pilares básicos de su lenguaje.

3.2.1. Los objetos

Un objeto se define como la abstracción de un concepto para representarlo dentro de un software, que encapsula un conjunto de datos que conforman su estado y una serie de operaciones, accesibles desde el exterior, que permiten modificarlo. Es por esto por lo que se dice que en la programación orientada a objetos "se unen datos y procesos", y no como en su predecesora, la programación estructurada, en la que estaban separados en forma de variables y funciones.

Un objeto consta de:

- Tiempo de vida: La duración de un objeto en un programa siempre está limitada en el tiempo. La mayoría de los objetos sólo existen durante una parte de la ejecución del programa.

- Estado: Todo objeto posee un estado, definido por sus atributos. Con él se definen las propiedades del objeto, y los valores que adoptan sus atributos en un momento determinado de su existencia.

- Comportamiento: Todo objeto ha de presentar una interfaz, definida por sus métodos, para que el resto de objetos que componen los programas puedan interactuar con él.

El equivalente de un objeto en el paradigma estructurado sería una variable. Asimismo, la instanciación de objetos equivaldría a la declaración de variables, y el tiempo de vida de un objeto al ámbito de una variable. En cambio, dentro del paradigma de orientación a objetos, la implementación de un objeto sería denominada clase, la cual, además de declararla, hay que inicializarla.

3.2.2. Principios de la orientación a objetos

Existen una serie de principios fundamentales para comprender cómo se modeliza la realidad al crear un programa bajo el paradigma de la orientación a objetos. Estos principios son: la abstracción, el encapsulamiento, la modularidad, la jerarquía, el paso de mensajes y el polimorfismo.

- Principio de Abstracción: Mediante la abstracción se busca pasar de entidades reales a modelos simplificados que permitan su implementación.
- Principio de Encapsulamiento: El encapsulamiento permite a los objetos elegir qué información es publicada y qué información es ocultada al resto de los objetos. Para ello los objetos suelen presentar sus métodos como interfaces públicas y sus atributos como datos privados e inaccesibles desde otros objetos.
- Principio de Modularidad: Mediante la modularidad, se propone al programador dividir su aplicación en varios módulos diferentes (ya sea en forma de clases, paquetes o bibliotecas), cada uno de ellos con un sentido propio.Esta fragmentación disminuye el grado de dificultad del problema al que da respuesta el programa, pues se afronta el problema como un conjunto de problemas de menor dificultad, además de facilitar la comprensión del programa.
- Principio de Jerarquía: La mayoría de nosotros ve de manera natural nuestro mundo como objetos que se relacionan entre sí de una manera jerárquica. Del mismo modo, las distintas clases de un programa se organizan mediante la jerarquía. La representación de dicha organización da lugar a los denominados árboles de herencia.
- Principio del Paso de Mensajes: Mediante el denominado paso de mensajes, un objeto puede solicitar de otro objeto que realice una acción determinada o que modifique su estado. El paso de mensajes se suele implementar como llamadas a los métodos de otros objetos.
- Principio de Polimorfismo: Esta propiedad permite que un objeto presente diferentes comportamientos en función del contexto en que se encuentre.

3.2.3. Relaciones entre objetos

Durante la ejecución de un programa, los diversos objetos que lo componen han de interactuar entre sí para lograr una serie de objetivos comunes. Existen varios tipos de relaciones que pueden unir a los diferentes objetos, pero entre ellas destacan las relaciones de: asociación, composición, y especialización.

- Asociación: Se trata de relaciones generales, en las que un objeto realiza llamadas a los servicios (métodos) de otro, interactuando de esta forma con él.
- Composición: Muchas veces una determinada entidad existe como conjunción de otras entidades, como un conglomerado de ellas. La orientación al objeto recoge este tipo de relaciones como dos conceptos;

la agregación y la composición. En este tipo de relaciones un objeto componente se integra en un objeto compuesto.

- Especialización: El ejemplo más extendido de este tipo de relaciones es la herencia, propiedad por la que una clase (clase hija) recoge aquellos métodos y atributos que una segunda clase (clase padre) ha especificado como "heredables".

3.3. Situación inicial

La aplicación se encuentra escrita en el lenguaje de programación JAVA, en el cual el paradigma de la orientación a objetos resulta fundamental, y que permite la organización del código en base a paquetes ordenados bajo criterios de funcionalidad. No obstante, inicialmente la aplicación se muestra de forma monolítica, es decir, existe una única clase, que contiene atributos y métodos necesarios para el funcionamiento de ésta, y que contiene clases internas que representan a los componentes de la regata, el entorno gráfico y la gestión de red, de modo que asemeja más a una aplicación desarrollada en un lenguaje de programación estructurada, como pudiera ser C o FORTRAN, que a una aplicación desarrollada bajo el paradigma de la orientación a objetos.

Entre los principales rasgos que se observan dentro de la estructuración de código, pueden destacarse los siguientes:

- Una superclase que representa la aplicación, y clases internas que representan los objetos que maneja la aplicación. De esta forma, los objetos internos de la aplicación pueden tener acceso de forma directa, es decir, asumen como propios, los atributos de la clase contenerdora, lo cual, en el contexto que nos encontramos, carece de sentido, dado que una boya, por ejemplo, no tiene motivos para necesitar conocer el socket por el que se comunica el cliente con el servidor y, por tanto, se viola el principio de encapsulamiento del paradigma de la orientación a objetos.

- Se integra dentro de la misma aplicación lo que es la aplicación cliente y la aplicación servidora. A pesar de poseer partes comunes, el funcionamiento es radicalmente diferente, así como las interfaces que implementa cada una. Para solventarlo, se realiza una compilación para la aplicación cliente y una diferente para la aplicación servidora, ambas sobre el mismo código, pero con diferentes opciones, es decir, se simula una compilación condicional.

- No existe diferenciación entre la capa de comunicaciones, la capa de gráficos y la capa de manejo de datos de la aplicación, por lo que el

código se vuelve fuertemente acoplado y un ligero cambio en cualquiera de las partes se torna engorroso de manejar.

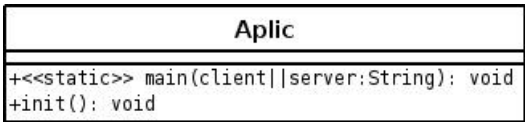


Figura 3.1: Diagrama UML inicial.

3.4. Etapa 1 de la Refactorización: Diferenciación de roles.

Durante la primera etapa del proceso de refactorización, se propone la diferenciación de la clase `Aplic` en tres clases, según su funcionalidad. Así pues, se tendrá una clase con la funcionalidad del cliente, a la que se denominará `AplicClient`, otra clase con la funcionalidad del servidor, llamada `AplicServer`, y una última con las partes que son empleadas tanto por cliente como por servidor, tal como muestra 3.2.

Al haberse separado los roles de cliente y servidor, ya no es necesario realizar la compilación condicional mencionada anteriormente: el cliente hereda de unas clases y el servidor de otras.

Se crean 3 paquetes básicos de software, uno para cada rol:

1. tes.simulator.AplicCommon
2. tes.simulator.AplicClient
3. tes.simulator.AplicServer

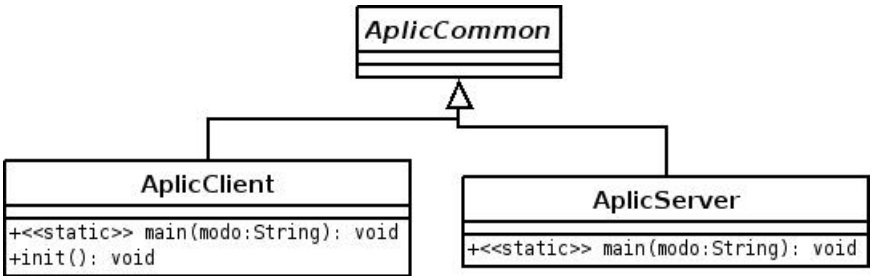


Figura 3.2: Etapa 1: Separación de roles

Existen métodos que poseen funcionalidad diferente en función de un flag que indica si la aplicación es cliente o servidora. En el caso de estos

métodos, quedan como abstractos en la parte común, siendo separados en dos métodos diferentes: uno para el cliente y otro para el servidor. Por otro lado, en la clase AplicCommon queda la funcionalidad común a cliente y servidor referente al avance del estado de la regata.

3.5. Etapa 2 de la Refactorización: Separación de Funcionalidad

En primer lugar, durante esta etapa se extraen las clases internas de la aplicación, que serán ubicadas en función de quién las utilice (server, client o common) y según a la funcionalidad que correspondan, dentro del paquete adecuado:

- tes.simulation.AplicXXXX.net.- Se guardan todas las clases relativas a la gestión de los recursos de red.
- tes.simulation.AplicXXXX.gui.- Se guardan todas las clases relativas al entorno gráfico.
- tes.simulation.AplicXXXX.state.- Se guardan todas las clases relativas a la gestión del estado de la aplicación.

Por otro lado, la funcionalidad del programa cliente contiene un cliente en modo aplicación y otro en modo applet, por lo que se procede a separar ambos. Como en el paso anterior, dichos cambios de funcionalidad se deben a un flag, por lo que, de nuevo, se separa en dos la clase, dando lugar a la clase AplicClientProg, que ejecutará un cliente en modo aplicación, y la clase AplicClientApplet, que hace lo propio con la funcionalidad de applet, quedándose AplicClient como clase abstracta. Dado que AplicClientApplet extiende de AplicClient, no puede extender de JApplet. Por ello, se añade la clase AppletClient, que extiende de JApplet y tiene un AplicClientApplet para realizar la simulación, tal como se muestra en la figura 3.3.

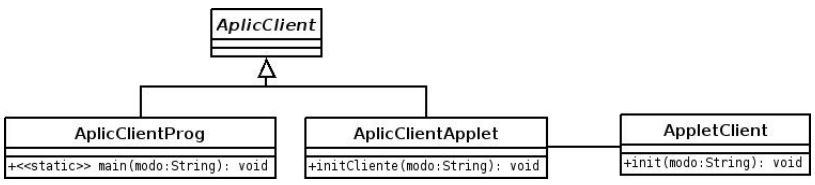


Figura 3.3: Etapa 2: Separar funcionalidad

3.6. Etapa 3 de la Refactorización: Múltiples instancias

Para levantar múltiples instancias de AplicClientApplet en el mismo navegador se hace necesario eliminar la parte estática del cliente. Dado que existe un método de la clase Runnable que es estático, se decide extraerlo a la clase MyRunnable, que lo implementa, tal como se muestra en 3.4

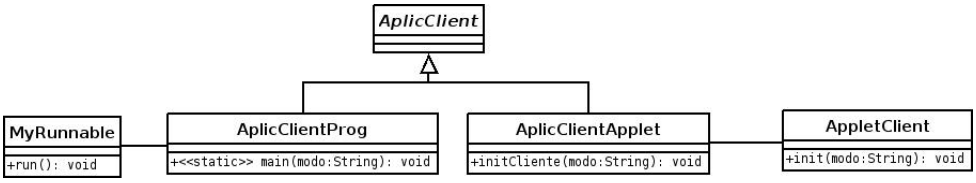


Figura 3.4: Etapa 3: Levantar múltiples instancias del cliente

Por otro lado, se desea tener la posibilidad de levantar múltiples instancias de AplicServer bajo una misma máquina virtual, por lo que se decide eliminar la parte estática del servidor, e implementar un patrón de diseño de Factoría [**?**], que permitirá tanto levantar una instancia de AplicServer de la forma tradicional, como levantar varias instancias mediante la lectura de un fichero de configuración. Así pues, el resultado del proceso de refactorización quedaría como se muestra en la figura 3.5

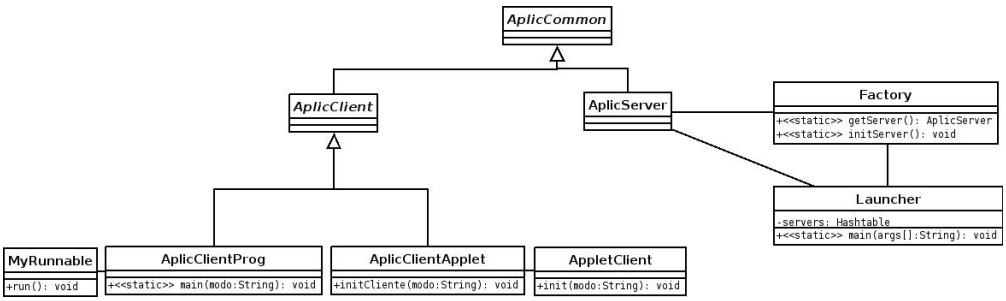


Figura 3.5: Levantar múltiples instancias del servidor.

3.7. Conclusiones

Tras el proceso visto en el presente capítulo de refactorización, se tiene una aplicación de simulación con una parte servidora y otra parte cliente, cada una con su propia funcionalidad y su propio código. Durante este proceso de refactorización se han alcanzado las siguientes metas:

- En primer lugar, el hecho de haber reorganizado el código, separando y extrayendo los objetos que se encontraban internos a la clase del simulador, se obtiene una mayor legibilidad del código, principalmente, así como hace factible localizar código duplicado, modificar las relaciones entre objetos (un simulador ya no **es un** barco más viento, etc.), reutilizar partes del código y localizar zonas que no sean utilizadas.

- Por otro lado, el hecho de separar la aplicación servidora de la aplicación cliente, así como de sus diferentes funcionalidades, elimina la necesidad de simular una compilación condicional, aunque siguen existiendo partes del código que son comunes a ambas. Así pues, la aplicación cliente englobará el paquete común y el paquete cliente, mientras que la aplicación servidora englobará el paquete común y el paquete servidor.

- Dentro de la separación de las aplicaciones, se han formado los paquetes según la funcionalidad, bien sea ésta de gráficos, bien de red o bien de gestión del estado. Esta separación permite tener ubicados los componentes de la aplicación según su funcionalidad, así como tenerlos aislados para que, en caso de que haya que realizar modificaciones, estén todas ubicadas en el mismo paquete.

3.8. Trabajo futuro

En virtud del proceso explicado en el presente capítulo, se exponen dos vías de desarrollo posibles dentro del contexto que sería aislar los componentes de la aplicación, aproximando ésta más al modelo ideal de un sistema distribuido.

En primer lugar, el hecho de cambiar la filosofía de un cliente que realiza peticiones a un servidor a un cliente que **tiene un** servidor y un servidor que **tiene** varios clientes podría aportar cierta transparencia de ubicación. Para ello, una propuesta a realizar es la implementación de una capa de código a modo de middleware que asuma la funcionalidad de un ORB, es decir, que se encargue de registrar objetos remotos y proporcione una interfaz para interactuar con ellos basada en el modelo de proxy-skeleton. De este modo, se pasaría de la orientación de que un cliente **se conecta a** un servidor a la orientación de que un cliente **tiene un** servidor, tomándose la misma orientación en la otra dirección, ocultando al programador la existencia de las comunicaciones y dando la imagen de una aplicación única local.

Capítulo 4

El entorno de simulación

4.1. Introducción

Los modelos que se procede a presentar en los diferentes capítulos de aportaciones están descritos de forma teórica, por lo que necesitan el apoyo en una serie de resultados para demostrar que su aplicación en el entorno elegido realmente consigue los resultados que se esperan de ellos.

Dado que la aplicación directa de dichos modelos, por límites temporales en cuanto a implementación de los modelos, así como de recursos, al no poder disponer de los nodos reales necesarios en las ubicaciones pertinentes, resulta difícilmente abordable, se procede a realizar unos modelos simplificados en un entorno de simulación controlado, con las consecuencias para los resultados obtenidos que ello supone.

4.2. Infraestructura Hardware

El proceso de simulación tiene lugar en un Intel Core 2 Quad con las siguientes características:

- Tecnología de fabricación: 65nm
- Freciencia: 2,40 GHz
- Front Side Bus: 1066 MHz
- Caché Nivel 1: 4x32 kb
- Caché Nivel 2: 2x4 MB
- TDP: 95/105 W
- Socket: LGA 775

Esta máquina posee un sistema operativo Linux Ubuntu Hardy Heron 8.04 64 bits con una versión del núcleo 2.4.24-21.generic. Asímismo, se emplea la versión de Java 1.6.0 y se emplea como editor de código Eclipse versión 3.2.

4.3. El software de simulación: J-SIM

J-Sim es una implementación de un simulador de redes en Java. El principal desafío durante el desarrollo del proyecto fue proporcionar diferentes contextos de ejecución (threads) para que los diferentes componentes del sistema manejasen los datos de entrada. Para ello, JSim introduce un thread de gestión en segundo plano llamada *runtime*.

J-Sim introduce una interfaz para script que soporta lenguajes como Perl, Tcl o Python, así como una implementación de Tcl por Java -Jacl. Las clases están implementadas en Java e 'interconectadas' empleando Tcl.

Entre las características especiales de J-Sim se pueden destacar:

- Modelo de programación basado en componentes, débilmente acoplado.
- Simulación tanto en tiempo real realizada por un proceso como por eventos dictados desde el script.
- Programación en dos lenguajes que permite auto-configuración y monitorización online.
- Implementación de interfaces genéricas para simulaciones que puedan ser traceadas.
- Emulación de aplicaciones de red reales.
- El simulador está enteramente implementado en Java, por lo que es independiente de la plataforma sobre la que se ejecuta. Además el código es abierto y de licencia GPL.
- Scripting basado en Tcl y Jacl encargado de unir los componentes del sistema de simulación y definir cómo deben comportarse.
- REQUIERE ACTUALIZACIÓN AL PARCHE 1.3-4 PARA CORRECTO FUNCIONAMIENTO.
- Posee editor gráfico gEditor v0.6

Dada la estructura modular, la relativa facilidad de uso una vez acostumbrado a sus vicisitudes y que se trata de un soporte de código abierto en el mismo lenguaje de la aplicación, se ha escogido este soporte para realizar las pruebas simuladas de los diferentes modelos del sistema.

4.4. Muestras establecidas

Para la realización de las simulaciones se han hecho mediciones de tres niveles de carga del sistema: bajo, con 10 clientes; medio, con 20 clientes; y alto, con 32 clientes, máximo de la aplicación. Se han realizado 10 rondas de sincronización para obtener el coste de envío de mensajes y se asumen los siguientes aspectos en la simulación:

- En el caso del modelo centralizado y del modelo basado en delegados repetidores, dado que no existe ningún punto intermedio de procesamiento de las peticiones y, por tanto, la lógica de la aplicación se encuentra contenida únicamente en la aplicación central, se asumirá que no existe diferencia entre los retrasos tomados por el envío de una petición "ligera" y los retrasos tomados por el envío de una sincronización. No obstante, en los modelos de servidores delegados basados en distribución de carga y servidores delegados réplicas del maestro, sí que se tendrá en consideración.
- Se obvia el proceso de inicialización del sistema.
- Dado que los resultados mostrados han sido obtenidos de un simulador, se ha introducido una cierta entropía dentro de lo que sería los retrasos de los mensajes, con el fin de que fuese un modelo más realista.

Capítulo 5

La aplicación BogReg

5.1. Introducción

5.1.1. Contexto de la aplicación

Como se ha comentado en anteriores secciones, el modelo inicial del sistema consta de un único servidor de regatas para cada regata, un servidor de catálogo que registra tales regatas, y un servidor web que las aloja.

Existen múltiples nodos que albergan varios servidores, repartidos geográficamente a lo largo del globo, interconectados entre ellos con una red de alta velocidad. No obstante, inicialmente no existe comunicación entre ellos.

El servidor de catálogo registra todas las regatas existentes en los nodos que forman parte del sistema, almacenando el tipo de regata, participantes, ubicación y una breve descripción. Estos datos son facilitados al servidor web para que pueda proporcionar una información en base a la cual los usuarios puedan conectarse a cada una de las regatas.

El servidor web es el portal donde acuden los clientes que desean participar en las regatas y es también el encargado de configurar y facilitar los applets necesarios para que el usuario pueda participar en una regata, así como de proporcionar una vista de la información contenida en el servidor de catálogo.

5.1.2. Introducción al funcionamiento de la aplicación

El funcionamiento de la aplicación es simple: las regatas se encuentran registradas en el servidor de catálogo, el cual muestra las referencias a través del servidor de web. Cuando un usuario desea conectar a una regata, acude al servidor web y la selecciona manualmente, iniciando este último la descar-

ga de un applet configurado listo para conectar con el servidor de regata.

Internamente, cuando se inicia el applet, éste manda un mensaje de inicialización (conexión) al servidor de regata, el cual comprueba los datos y envía el estado al cliente, para que pueda iniciar la simulación. Cuando el cliente realiza una operación, la aplica y la envía al servidor, el cual la aplica en el estado general de la regata y procede a sincronizar al resto de participantes. Adicionalmente, cada cierto tiempo `t` el servidor sincroniza el estado de los participantes con el propio. Al no tratarse de un sistema orientado a la conexión, existe un mecanismo de keep-alive para conocer qué clientes permanecen activos.

Tanto el cliente como el servidor son capaces de hacer evolucionar la regata. La principal diferencia radica en que el servidor posee además la capacidad de calcular las posibles colisiones de cada barco participante en la simulación con otros elementos de la misma, bien sean otros barcos, bien elementos estáticos como las boyas. En caso de existir colisiones, el encargado de introducir las penalizaciones es también el servidor.

5.1.3. Problemas presentados por la aplicación

La solución aquí descrita introduce una serie de problemas, a saber:

- Por un lado, al existir un único servidor de regata, los clientes más alejados geográficamente obtendrán unos mayores retardos en el envío y recepción de mensajes con el servidor.
- Por otro lado, al existir únicamente un servidor que simula la regata, en caso de fallo, la regata se detiene, por lo que no existe una tolerancia a fallos.
- Finalmente, el tercer problema principal encontrado es que, dado que el servidor posee una capacidad de cálculo limitada, la aplicación puede escalar muy poco, por lo que el número de clientes participantes en una misma regata queda bastante limitado.

El hecho de tratarse de un servicio no orientado a conexión y, por tanto, con posibilidad de pérdida de orden de los mensajes o simplemente omisión de éstos no representa un grave problema, dado que este hecho ocurre con una probabilidad p inferior al 0'1 %. Adicionalmente, cada envío de un mensaje de sincronización reescribe el estado completo de los clientes, por lo que si se perdiese uno de estos mensaje, únicamente habría que esperar hasta la próxima actualización para tener un estado consistente de la aplicación. Finalmente, indicar que, por especificación del cliente, el retraso de un mensaje entre el cliente y el servidor nunca deberá exceder los 200 milisegundos.

5.2. Partes constantes durante el proceso de distribución de la aplicación

Dentro de la aplicación existen partes de ésta que a pesar del proceso de distribución permanecerán invariantes, es decir, que no sufrirán variación alguna en su funcionamiento y que, por tanto, se expondrán aquí para así poder referenciarlas fácilmente en lugar de tener que detallarlas constantemente. No obstante, en caso de necesitar una variación para algunos de los algoritmos que serán expuestos en los próximos capítulos, se detallarán dichas variaciones en la sección correspondiente.

5.2.1. El cliente: modelado y funcionamiento

En primer lugar, se presenta la aplicación cliente dentro de las partes que no van a variar a lo largo del desarrollo del presente documento. Esta aplicación cliente es la que interacciona con el usuario y que, por tanto, contiene una funcionalidad reducida respecto al simulador que posee un servidor.

Como primer paso dentro de la descripción de la aplicación cliente, se procede a dar una visión general del mismo para posteriormente ir entrando en detalle, tanto a nivel de descripción como a nivel de formalización. Como nota avisar que las formalizaciones aquí expresadas buscan más dar una visión de la aplicación a modo de pseudocódigo que una corrección matemática del funcionamiento de la aplicación.

La figura 5.1 muestra un algoritmo inicial básico del funcionamiento de un cliente de la aplicación BogReg, que a continuación se pasará a detallar.

En primer lugar, la aplicación es la simulación de una carrera de barcos y, por tanto, a los valores que tenga la aplicación para un determinado instante de tiempo t se le denominará **estado**. De una forma simplificada, el estado de la regata se compone de lo siguiente:

- Un identificador de regata.
- Un recorrido, demarcado por un conjunto de boyas que conforman el mapa de la carrera.
- Una flota, conformada por los barcos capitaneados por usuarios de la aplicación y los barcos capitaneados por la propia aplicación, en caso de que los haya.
- Unas corrientes, generadas a partir de un fichero de datos.
- Un viento, generado a partir de un fichero de datos.

Cliente

Estado
estado = Estado de la regata para el cliente C_i
S_i = Identificador del servidor de la regata
relojAvanza = Reloj encargado de la progresión del estado
relojKeep = Reloj encargado de la gestión de keep-alive
KEEPALIVE = Mensaje de keep-alive
CONNECT = Mensaje de conexión
Acciones
init (host, puerto)
S_i = {host, puerto}
conectar (S_i, CONNECT)
relojKeep = new Reloj (T_KEEP)
relojAvanza = new Reloj (T_AVANZA)
recibir (S_i, E)
lock (estado)
estado = E
unlock (estado)
aplicarOperacion (op)
lock (estado)
estado += op
unlock (estado)
enviar (S_i, op)

Figura 5.1: Cliente de la aplicación de simulación.

Al iniciar la aplicación, se le facilita mediante línea de órdenes los parámetros que definirán el servidor al que se debe conectar, que son el host donde se albergan y el puerto que está a la escucha. Tras ello, se lanza el método inicializar del algoritmo de la figura 5.1.

Inicialización

El método init de la aplicación construye la referencia del servidor y le manda un primer mensaje de conexión, tras el cual recibe un mensaje del cual extrae los parámetros que conforman la parte estática de la regata, como son el viento, las corrientes, el identificador y el recorrido mismo de la regata, tal como muestra la figura 5.2.
Posteriormente, envía un mensaje solicitando la flota existente en la regata, tanto comandada por humanos como por la máquina. Además, inicializa los relojes encargados de la gestión de avance del estado del simulador así como de la gestión del envío de los mensajes de keep-alive.

El reloj de envío de keep-alive inicializa para que se active cada cierto lapso de tiempo T_KEEP y se encargue de enviar un mensaje de keep-alive al servidor para notificar que el cliente, aunque no esté realizando ninguna acción, sigue vivo y participando, como muestra la figura 7.10.

```
Init
    init (host, puerto)
        S_i = < host, puerto >
        conectar (S_i, CONNECT)
        relojKeep = new Reloj (T_KEEP)
        relojAvanza = new Reloj (T_AVANZA)
    conectar (S_i, CONNECT)
        enviar (S_i, CONNECT)
        recibir (mensaje)
        analizar (mensaje)
        enviar (S_i, FLOTA)
        recibir (mensaje2)
        analizar (mensaje2)
    analizar (mensaje)
        if (mensaje.tipo == flota)
            estado.flota = mensaje.flota
        if (mensaje.tipo != flota)
            estado.recorrido = mensaje.recorrido
            estado.corrientes = mensaje.corrientes
            estado.viento = mensaje.viento
            estado.identificador = mensaje.identificador
```

Figura 5.2: Metodo de inicialización.

```
RelojKeep
    enviarKeep (S_i)
        enviar (S_i, KEEPALIVE)
    end
```

Figura 5.3: RelojKeep.

Por último, se inicializa un reloj de avance que ejecute cada cierto tiempo igual a T_AVANZA el método encargado de hacer progresar el estado de la regata, moviendo la flota, tal como muestra la figura 5.4.

```
RelojAvanza
    avanzar ( estado.flota)
    end
```

Figura 5.4: RelojAvanza.

Recepción de sincronizaciones

La aplicación cliente, aunque es capaz de hacer progresar, en cierto modo, el estado de la flota participante de una regata, necesita sincronizarse con el servidor de la misma para funcionar correctamente debido a que, en primer lugar, el cliente es incapaz de realizar ciertos cálculos tales como el cálculo de las colisiones entre barcos, y, por otro lado, necesita tener una visión aproximada a la que tienen el resto de clientes, y coherente con la que hay en el servidor.

```
sincronizar
    recibir (S_i, mensaje)
        if (mensaje.tipo == sincronizacion)
            analizar (mensaje)
    analizar (mensaje)
        estado.flota = mensaje.flota
        estado.colisiones = mensaje.colisiones
    end
```

Figura 5.5: Sincronizar.

Como se puede observar en la figura 5.5, cuando se recibe un mensaje de sincronización, la aplicación toma los nuevos valores de la flota y los introduce en su estado. Además de las nuevas posiciones de los barcos que conforman la flota, se reciben las colisiones existentes, es decir, el análisis de si un barco ha chocado contra una boya o con otro barco, así como el culpable de la colisión y la penalización que se le aplica, datos que es incapaz de calcular la aplicación cliente.

Progreso del estado de la regata

Como se ha comentado en la anterior subsección, el cliente posee un simulador capaz de hacer progresar los barcos de la flota en función de la última posición y dirección conocidas, los vientos para tales variables, así como las

corrientes de éstas, como es reflejado en 5.6, lo que da como resultado una variación de la posición, *delta*, que será aplicada a la posición del barco correspondiente.

El avance del estado se produce cada vez que se ejecuta el reloj encargado de hacer avanzar el estado, llamado aquí RelojAvanza, y corresponde al avance general de toda la flota. El avance de las operaciones que realice un usuario para su barco no progresan de esta forma, sino que van por otro lado de la aplicación, descrito en el siguiente apartado.

```
avanzarEstado
    avanzar ()
        foreach Cata c in estado.flota
            avanzar (c)
    avanzar (cata)
        delta = cata.calcularDelta (cata.posicion, cata.direccion, estado.viento, estado.corrientes)
        cata.posicion = cata.posicion + delta
    end
```

Figura 5.6: Avanzar.

Los detalles del cálculo de *delta* no se reflejan en el algoritmo mostrado en 5.6 dado que forman parte del motor físico de la aplicación, siendo de escaso interés para la meta que nos ocupa.

Operaciones introducidas por el usuario

El avance de un determinado barco, perteneciente al usuario que está ejecutando la aplicación cliente, como se ha comentado en el punto anterior, requiere un tratamiento especial, dado que tanto el servidor como el resto de clientes deben actualizar su estado para conocer las nuevas condiciones del barco en cuestión.

Cuando el programa registra una acción, ésta es aplicada en el estado local del cliente y posteriormente enviada al servidor, que realizará los cálculos pertinentes. En la figura 5.7 se muestra de forma simplificada el funcionamiento de la aplicación cliente para estos casos.

En función de la operación introducida, se calcula la nueva posición y/o la nueva dirección del barco que comanda el usuario, y además se envía la operación al servidor para que tenga en cuenta los cambios a la hora de recalcular el estado.

```
aplicarOperacion
    aplicarOperacion ( op )
        avanzar (estado.miCata, op)
        enviar (S_i, estado.miCata)
    avanzar (c, op)
        c.posicion = c.posicion + op.posicion
        c.direccion = c.direccion + op.direccion
    end
```

Figura 5.7: Aplicar una operación.

5.2.2. La red: soporte de comunicaciones del sistema

El hecho de que BogReg sea una aplicación cliente-servidor implica que debe existir un canal de comunicaciones entre el servidor y los diferentes clientes apuntados a una regata y un protocolo de comunicaciones entre ellos. En el caso abordado en esta aplicación, un cliente se conecta a su servidor mediante la red que conforma internet, empleando el protocolo UDP para tanto los envíos como las recepciones.

La decisión de emplear el protocolo UDP se toma en base a sus tres características básicas: en primer lugar, es un protocolo no orientado a conexión, por lo que se ahorran rondas de mensajes cada vez que un cliente nuevo desee conectarse al servidor, descargando a este último de la carga computacional que ello supone, que aunque es menor, en casos de nodos que contengan múltiples servidores y cada uno de éstos con un número elevado de clientes, puede llegar a representar una ventaja como mínimo significativa. Por otro lado, el tamaño de los paquetes UDP es menor al tamaño de los paquetes TCP, lo que implica un reducción de la fragmentación de éstos en caso de mensajes de gran tamaño. Por último, dada la implementación del propio protocolo UDP, se consigue una reducción de los retrasos en la transmisión de mensajes en comparación con las comunicaciones TCP, lo cual supone una ventaja adicional.

5.3. Partes variables durante el proceso de distribución de la aplicación

5.3.1. El servidor: modelado y funcionamiento

La parte principal que va a variar tanto en diseño como en funcionalidad dentro del desarrollo de esta tesis es la parte servidora de la aplicación BogReg, dado que se estudian diferentes formas de distribución para un servidor centralizado.

En el presente apartado se procede a mostrar una visión detallada del servidor en su estado inicial, centralizado, del mismo modo que ya se hiciera

con el cliente en anteriores apartados. Nuevamente, las explicaciones se van a respaldar sobre algoritmos en pseudocódigo para poder intuir el funcionamiento de cada una de las partes, sin mostrar el código ni el funcionamiento exacto de la aplicación.

Servidor

```
Estado
    estado = estado de la regata
    listaClientes = clientes conectados al servidor
    relojEstado = reloj encargado de la progresión del estado
    relojSincronizar = reloj encargado de sincronizar a los clientes
    T_AVANZA = Lapso temporal entre progresiones del estado
    T_SINCRO = Lapso temporal entre sincronizaciones
Acciones
    init (B_i, V_i, Co_i)
        listaClientes = { - }
        lock (estado)
        cargarEstado (B_i, V_i, Co_i)
        unlock (estado)
        relojAvanza = new Reloj (T_AVANZA)
        relojSincro = new Reloj (T_SINCRO)
    recibir (C_i, mensaje)
        if (mensaje.tipo == CONNECT)
            conectar (C_i)
            actualizaVivo (C_i)
        if (mensaje.tipo == KEEPALIVE)
            actualizaVivo (C_i)
        if (mensaje.tipo == OP)
            actualizaVivo (C_i)
            avanzarOperacion (C_i, OP)
        if (mensaje.tipo == DISCONNECT)
            desconectar (C_i)
        if (mensaje.tipo == 2)
            enviar (C_i, estado.flota)
            actualizaVivo (C_i)
```

Figura 5.8: Servidor de la solución centralizada

Como se muestra en la figura 5.8, el servidor consta de 5 partes primordiales: la inicialización de la regata, donde se cargan los valores iniciales del estado de la regata; la recepción y el posterior tratamiento de los mensajes enviados por los clientes, donde se evalúa el tipo de mensaje y se actúa en función de éste; la progresión del estado, donde se hace evolucionar a toda la flota participante; el avance de barcos en concreto debido a operaciones recibidas del cliente en cuestión; y la sincronización de los clientes.

Inicialización

El proceso de inicialización del servidor de regata es totalmente diferente al del cliente de la regata. El servidor carga los valores estáticos de la regata (viento, recorrido, corrientes) desde un fichero facilitado a la aplicación

durante su arranque. Por otro lado, el identificador de regata es generado automáticamente y se crean los barcos pertenecientes a la inteligencia artificial, aunque su estado permanece a 0, es decir, únicamente se instancia, pero no se inicializan.

Servidor

Inicializar
init ($fichero_{viento}$**,** $fichero_{recorrido}$**,** $fichero_{corrientes}$**, numbots)**
listaClientes = { - }
lock (estado)
cargarEstado ($fichero_{viento}$, $fichero_{recorrido}$, $fichero_{corrientes}$, numbots)
unlock (estado)
relojAvanza = new Reloj (T_AVANZA)
relojSincro = new Reloj (T_SINCRO)
cargarEstado(viento, recorrido, corriente, numbots)
cargarViento (viento)
cargarRecorrido (recorrido)
cargarCorrientes (corriente)
cragarFlotaBot (numbots)
cargarViento (v)
estado.viento = v.datos
cargarRecorrido (r)
estado.recorrido = r.datos
cargarCorrientes (c)
estado.corrientes = c.corrientes
crearFlotaBot (numbots)
for (i=0; i¡numbots; i++)
estado.flota.add (crearBarcoBot ())

Figura 5.9: Inicialización del servidor.

Además del proceso de carga del estado, mostrado en la figura 5.9, el proceso de inicialización del servidor instancia dos relojes, del mismo modo que lo hace el cliente, aunque con funcionalidad algo diferente. El primero de ellos, mostrado en la figura 5.10, será el encargado de hacer progresar el estado en el servidor.

RelojAvanza
avanzar (estado.flota)
end

Figura 5.10: RelojAvanza en el servidor.

El otro reloj que se instancia es un reloj que produce los latidos necesarios para la sincronización de los clientes con el estado del servidor, como muestra la figura 5.11, lo que implica que éstos tomen el estado del servidor como estado oficial de la regata.

```
RelojSincro
    sincronizar (listaClientes)
    end
```

Figura 5.11: RelojSincro.

Avance del estado

En el servidor, como se ha comentado en el punto anterior, existe un reloj encargado de realizar una suerte de latidos que provocan el avance de los movimientos que debe realizar la flota en función de los parámetros de viento y corrientes que existan para un tipo determinado de barco en un punto determinado de recorrido. Además, el método de avance de los barcos calculará si existen colisiones entre los diferentes barcos participantes, sean comandados por bots o por humanos, o entre algún barco y alguna boya del recorrido, penalizando según las reglas de penalizaciones introducidas.

```
avanzarEstado
    avanzar ()
        foreach Cata c in estado.flota
            colision = avanzar (c)
        if (colision)
            calcularPenalizaciones ()
    avanzar (cata)
        colision = false
        delta = cata.calcularDelta (cata.posicion, cata.direccion, estado.viento, estado.corrientes)
        cata.posicion = cata.posicion + delta
        foreach Cata c in estado.flota
            if (c.posicion == cata.posicion)
                colision = true
        return colision
    end
```

Figura 5.12: Avanzar Estado.

Tanto el cálculo del desplazamiento que realiza cada barco, *delta*, como el cálculo de las penalizaciones a aplicar en caso de colisión, no son de un interés vital para el desarrollo del presente documento, además de formar parte del funcionamiento interno de la aplicación, por lo que no se detallará el proceso seguido para dichos cálculos, dejándolos ver en la figura 5.12 únicamente como nombres de métodos y realizando un "acto de fe" acerca de sus correspondientes cálculos.

Recepción de mensajes

La principal funcionalidad de todo servidor consiste en atender las peticiones que le llegan por parte de los clientes. Para ello, el servidor de regatas implementa un sistema a través de cual recibe los mensajes de los clientes,

los analiza y, en función del tipo de mensaje, actúa y responde si es necesario.

```
Recibir Mensajes
    recibir (C_i, mensaje)
        if (mensaje.tipo == CONNECT)
            conectar (C_i)
            actualizaVivo (C_i)
        if (mensaje.tipo == KEEPALIVE)
            actualizaVivo (C_i)
        if (mensaje.tipo == OP)
            actualizaVivo (C_i)
            avanzarOperacion (C_i, OP)
        if (mensaje.tipo == DISCONNECT)
            desconectar (C_i)
        if (mensaje.tipo == FLOTA)
            enviar (C_i, estado.flota)
            actualizaVivo (C_i)
            sincronizar()
    conectar (cliente)
        listaClientes.add (cliente)
        mensaje = estado.recorrido + estado.viento + estado.corrientes + estado.identificador
        if (cliente.tipo in {MASTER, CLIENT})
            estado.flota.add (cliente.barco)
        enviar (cliente, mensaje)
    desconectar (cliente)
        listaClientes.remove (cliente)
        sincronizar ()
    actualizaVivo (cliente)
        cata = estado.flota.get(cliente.id)
        if (T_now - cata.T_keep T_max)
            cata.T_keep = T_now
        else
            desconectar (cliente)
```

Figura 5.13: Recepción de mensajes en el servidor

Como muestra la figura 5.13, los mensajes se pueden catalogar en cinco tipos básicos: de conexión, de petición de flota, de acción, de keep-alive y de desconexión. De estos cinco tipos, los mensajes de acción, es decir, las operaciones, serán tratados en el siguiente apartado.

Cuando el servidor recibe un mensaje de conexión, actualiza su lista de clientes añadiendo al nuevo cliente a ella. Posteriormente, conforma un mensaje que contiene el estado estático de la regata, es decir, su identificador, los vientos y las corrientes existentes, y el recorrido a seguir. Adicionalmente, si el nuevo cliente pertenece al tipo de cliente al que se le puede asignar un barco, éste es asignado. Por otro lado, cuando un cliente desea dejar de participar en una regata, envía un mensaje de desconexión al servidor, el cual lo borrará de su lista de clientes.

Durante la inicialización de un cliente, una vez solicitado el estado estático

de la regata y procesado, se desea conocer la flota participante y los datos relativos a ella, lo cual es solicitado mediante el mensaje FLOTA, cuya respuesta en el servidor es la composición de un mensaje para el cliente en cuestión y la posterior sincronización a todos los clientes que participan en la regata.

El servidor también debe controlar que un cliente, a pesar de no realizar acciones, permanece vivo, por lo que debe gestionar la recepción de los mensajes de keep-alive que éste le envíe. Para ello, se establece un T_{max} transcurrido el cual, si un cliente no ha dado señales de vida, es desconectado. El parámetro que se comprueba con este T_{max} es la diferencia de tiempo entre el último mensaje recibido por el cliente y el anterior. En caso de no superar, se actualizan estos dos factores.

Aplicar operaciones recibidas

En el anterior punto se ha mencionado el hecho de que los mensajes recibidos del tipo OP son tratados de forma particular, lo que es debido a que modifican el estado de únicamente uno de los participantes de la regata, pero puede afectar al resto.

```
aplicarOperacion
    aplicarOperacion (cliente, op )
        avanzar (estado.flota.get(cliente.identificador), op)
        sincronizar ()
    avanzar (c, op)
        c.posicion = c.posicion + op.posicion
        c.direccion = c.direccion + op.direccion
    end
```

Figura 5.14: Aplicar una operación a un cliente.

Cuando se aplica una operación, es porque uno de los clientes ha realizado una acción, por lo que se ha de encontrar el barco asociado al cliente y modificar el estado de dicho barco en función de los parámetros designados en el mensaje de operación. Una vez aplicados los cambios, el resto de barcos, así como el propio cliente que ha enviado el mensaje, deben sincronizar su estado para evitar discrepancias, además de apercibirse de las variaciones sufridas por el barco modificado.

Sincronización de los clientes

El proceso de sincronización se corresponde al envío del estado de la flota participante en una regata a todos los usuarios conectados a la misma, no sólo a los que poseen un barco, lo cual nos da lugar a la aparición de espectadores. Como muestra la figura 5.15, el proceso servidor envía los

datos pertenecientes a los barcos participantes, para obtener una imagen similar en todos los nodos que ejecuten la regata.

```
sincronizar
    sincronizar ()
        foreach barco in estado.flotas
            mensaje += barco.datos
        foreach cliente in listaClientes
            enviar (cliente, mensaje)
    end
```

Figura 5.15: Sincronización de los clientes.

La finalidad de las sincronizaciones no es la de obtener una imagen única de la regata tanto en clientes como en el servidor, sino proporcionar una imagen aproximada a los clientes del estado oficial de la regata, que es el que posee el servidor.

Mediciones de control

En el presente apartado se recogen las mediciones obtenidas de las simulaciones correspondientes al modelo centralizado original de la aplicación, para las diferentes cargas del sistema anteriormente descritas. Estas mediciones pretenden dar un punto de referencia con el que poder comparar los resultados obtenidos de las simulaciones de lo modelos de replicación descritos en los siguientes apartados.

En primer lugar, se procede a analizar el comportamiento de la aplicación para una carga del sistema baja, o lo que es lo mismo, 10 clientes conectados al único servidor del sistema. Tras realizar el proceso de inicialización del sistema y de conexión de los clientes, obviado en el estudio realizado dado que carece de interés, se realiza una serie de 10 rondas de sincronización del servidor, obteniéndose los retardos medios para cada cliente, mostrados en la figura 5.16.

Como se puede observar, el coste medio para cada cliente, según el gráfico, oscila entre los 154 y los 167 milisegundos. No obstante, estos retardos medios puede contener valores anormales, dado que la muestra posee una cierta entropía para dotar de realismo al sistema. Con el objetivo de obtener un coste medio de envío de mensajes, se procede a obtener unos intervalos de confianza para la muestra de retardos obtenida, representado esto por un diagrama Box-Whisker representado en la figura 5.17, en el que se muestran los valores extremos de la muestra, que serán tomados como anormales, pudiendo obtener un retardo medio más ajustado a la realidad.

Una vez descartados los resultados anómalos de la muestra, se extrae el coste medio de envío por mensaje de sincronización, siendo éste de 157'70

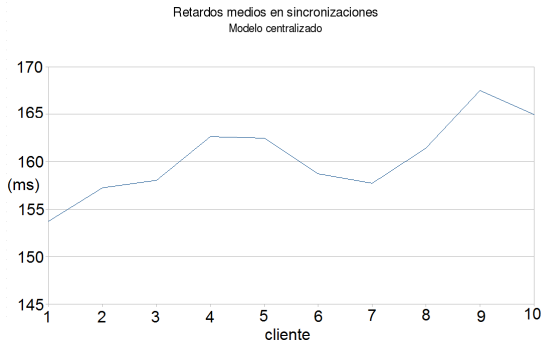


Figura 5.16: 10 clientes en modelo centralizado.

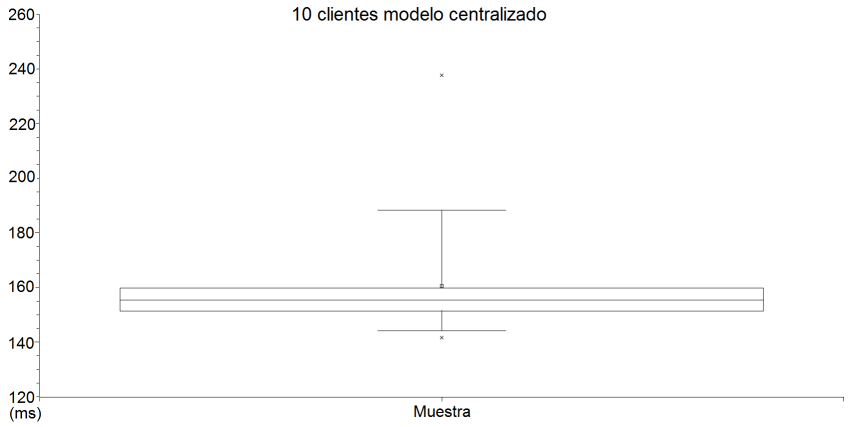


Figura 5.17: Intervalo de confianza de las mediciones obtenidas.

milisegundos, obteniendo una tasa de datos anómalos del 9 %, que será el retardo que se tome como base para realizar las comparaciones de los diferentes modelos de replicación, siendo éste inferior al umbral marcado por los requisitos de 200 milisegundos que se expusieron anteriormente para el funcionamiento de la aplicación.

Por otro lado, se han realizado las mismas mediciones para una carga media del sistema, 20 clientes, conectados a un único servidor. Los tiempos de retraso de lo mensajes obtenidos bajo estas condiciones de carga oscilan, para cada cliente, entre los 156 y los 170 milisegundos, tal como refleja la gráfica 5.18.

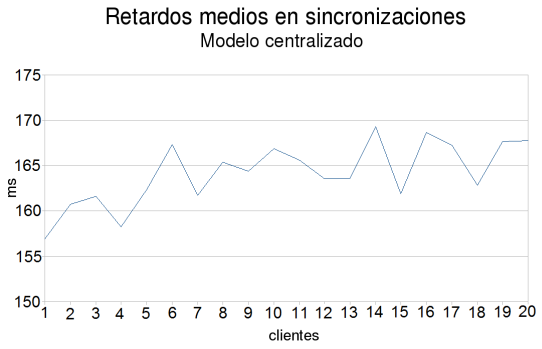


Figura 5.18: 20 clientes en modelo centralizado.

Ya se encuentra un pequeño sobrecoste adicional en las comunicaciones introducido por el aumento de carga del sistema a primera vista, por lo que se procede a realizar un intervalo de confianza para la muestra obtenida, a fin de garantizar que no es producto de una serie de datos anómalos, por lo que se procede a realizar el diagrama Box-Whisker con el fin de hallar el coste medio del mensaje con una muestra fiable, tal como muestra la figura 5.19.

Basándonos en el gráfico obtenido, tras descartar un 2 % de datos anómalos, se obtiene un retraso en las comunicaciones de 163'25 milisegundos, por lo que se introduce un incremento del 3.83 % debido al aumento de carga, que puede venir introducido por un mayor tamaño del mensaje, mayor número de elementos a comprobar en el envío, mayor tiempo de construcción del mensaje, etc...

Por último, se realizan las mismas 10 sincronizaciones para una carga elevada del servidor, esto es, el número máximo de clientes, 32, que puede admitir el servidor central por regata. En primer lugar, se obtiene el retraso medio por cliente en el envío de una sincronización, que oscila entre 156 y

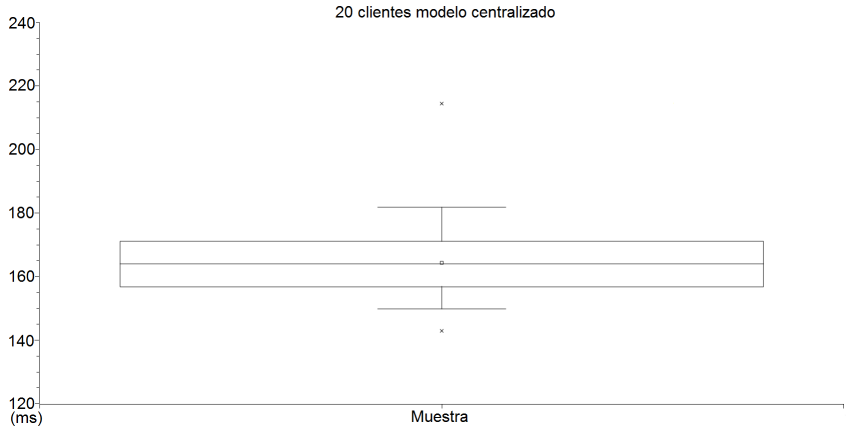


Figura 5.19: Intervalo de confianza de las mediciones obtenidas.

los 171 milisegundos, tal como se desprende de la gráfica 5.20

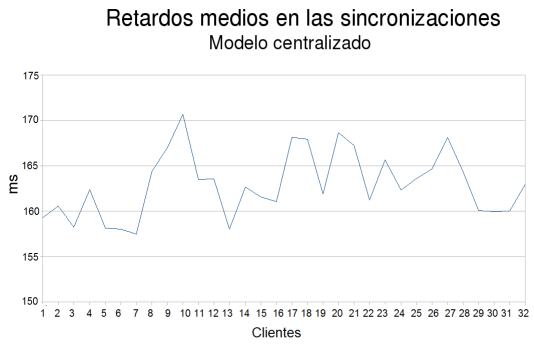


Figura 5.20: 32 clientes en modelo centralizado.

Para obtener un coste medio del envío del mensaje, primero hay que eliminar los datos anómalos de la muestra, que constituyen un 2 % de ésta, mediante la obtención de un intervalo de confianza gracias al diagrama Box-Whisker que se muestra en la gráfica 5.21.

Una vez limpia la muestra de datos anómalos, se obtiene un retardo medio de 162.8 milisegundos, bastante similar al obtenido con carga media, por lo que se asume que el sobrecoste introducido por los clientes adicionales es despreciable en cuanto a parámetros de comunicaciones se refiere.

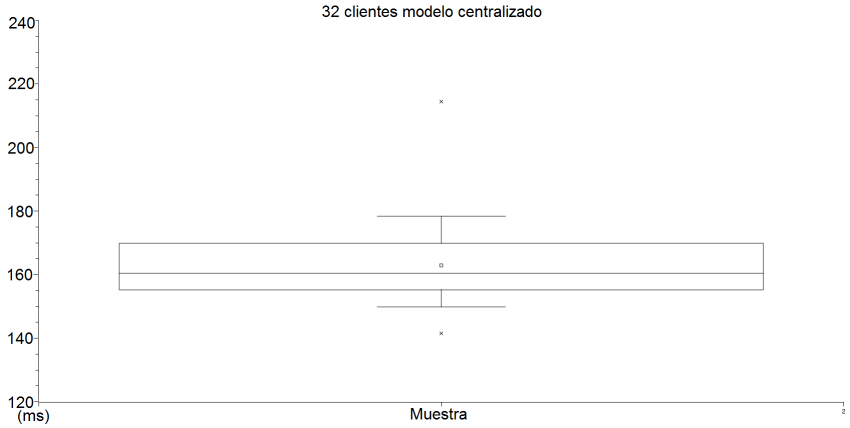


Figura 5.21: Intervalo de confianza de las mediciones obtenidas.

Modelo de consistencia de datos

Como se ha descrito durante la introducción de la aplicación, ésta consta de clientes que envían mensajes a un servidor, y el servidor que sincroniza los estados de los clientes. Exceptuando los mensajes de inicialización y centrándonos en el proceso de simulación, para simplificar el modelo, se puede afirmar que un cliente envía peticiones en base a su estado local (acciones) al servidor, el cual las procesa, independientemente de que lo haga también el cliente, y envía una sincronización a los clientes con las modificaciones pertinentes (respuestas). No obstante, para no saturar la red, en lugar de generar una respuesta por cada petición, se acumulan n peticiones en una sincronización, siendo el ciclo de envío de S/t donde S es el número de sincronizaciones por t unidad de tiempo.

El hecho de realizar envíos y recepciones de datos supone que los clientes pueden, potencialmente, obtener diferentes vistas de un mismo estado. Para evitar este problema, los mensajes de sincronización son numerados mediante un identificador de tal forma que una vez entregado el mensaje de sincronización M_n, el mensaje M_{n-1} sea descartado, no permitiendo entregas fuera de orden, dado que el mensaje M_n contendrá el estado de aplicar sus modificaciones más todas las anteriores y garantizando así un modelo de consistencia de datos FIFO.

No obstante, la consistencia llega a ser causal, pues aunque respeta las garantías FIFO, hay un único emisor de actualizaciones globales, el servidor central, y debido a esto impone también un orden total. Las difusiones FIFO de orden total son realmente difusiones atómicas causales cuando no hay fallos o estos se pueden recuperar. Además, cini sólo hay un único emisor

de actualizaciones globales, realmente la consistencia obtenida es secuencial, algo más estricta que la causal, como se extrae de [CKV01].

Consideraciones del modelo y problemas presentados

El modelo centralizado que presenta la aplicación constituye una primera aproximación al funcionamiento óptimo de ésta, aunque presenta una serie de inconvenientes típicos de una aplicación centralizada, aun basándose en un esquema de cliente-servidor.

El hecho de que exista un único servidor que proporciona la regata establece un cuello de botella en el procesado de las peticiones recibidas por los clientes, dado que el servidor es el único que puede atenderlas y, por tanto, cuanto mayor sea el número de clientes atendidos, mayor carga existirá, pudiendo llegar a colapsar el sistema o provocando la pérdida de peticiones. Para evitar estos hechos se ha establecido una cantidad máxima de clientes que pueden conectarse a una misma regata, sin tener en consideración el tipo de éstos. No obstante, de cara a posibles ampliaciones de la aplicación, esta medida puede perder validez, dado que si se produjese un reparto de carga, se podría aumentar el número de clientes atendidos.

Por otro lado, al existir un único punto de acceso a la regata, el coste temporal de las comunicaciones con los clientes puede variar significativamente dependiendo de la ubicación geográfica de éstos con respecto a la del servidor, aumentando los retrasos producidos en el envío de mensajes no sólo en función de la red, que variará dependiendo de cómo esté interconectada la red que da servicio a los clientes con la red que da servicio al servidor, sino también en función de la distancia que los separe, pudiendo dar lugar a incoherencias en los estados percibidos por los clientes, debidos al propio retraso de las comunicaciones, e incluso aumentando la probabilidad de pérdida de paquetes.

El tercer aspecto a tener en consideración dentro de este diseño de la aplicación es que, también por el hecho de existir un único proceso servidor que puede realizar el avance completo del estado de una regata, en caso de fallo de éste la regata no puede continuar, por lo que la tolerancia a fallos del servidor es nula. Uno de los principales objetivos de los posteriores capítulos será proporcionar tolerancia a fallos al sistema para que, en caso de caída del servidor, la regata pueda continuar.

Los tres aspectos que se acaban de comentar tienen como base fundamental el hecho de la existencia de un único servidor que proporciona la regata, y es ahí donde se centrarán los esfuerzos de la tesis para contrarrestar tales inconvenientes mediante modificaciones en la parte servidora del sistema,

estableciendo vías de desarrollo para la optimización de la aplicación y la mejora de sus características.

5.3.2. El nodo: modelado y funcionamiento

El funcionamiento de los modelos que se proceden a explicar en los siguientes capítulos requiere de un agente que, aunque sí que existe en el modelo centralizado, cobra una especial trascendencia fuera de él: el host. Este elemento pasa de ser un simple contenedor de procesos a un agente activo dentro del sistema.

```
Nodo
    Estado
        listaRegatas = regatas que poseen servidor en el nodo
        threadCreador.start()
    Acciones
        recibir (M_i, mensaje)
            if (mensaje.tipo == START)
```

Figura 5.22: Nodo del sistema

La figura 5.22 muestra el algoritmo que viene a describir el funcionamiento que poseerá el nodo una vez se implementen los modelos de distribución de los servidores de los siguientes capítulos. Básicamente, cuando se requiere un nuevo servidor delegado en el sistema, el servidor maestro envía un mensaje START al nodo que deberá contenerlo, el cual posee una lista de las regatas que tienen un servidor en él y un hilo de ejecución para crear nuevas instancias de los servidores delegados.

El nodo comprueba que no se posee ninguna regata cuyo identificador coincida con el enviado en el mensaje de START. En caso de no poseerlo, el thread generará una nueva instancia de un servidor delegado, a la que le proporcionará el maestro correspondiente y procederá a arrancar la instancia. En caso contrario, se ignorará el mensaje.

5.3.3. Tolerancia a fallos en el contexto actual

La aparición de fallos puede tener lugar en tres partes diferentes dentro del sistema, cada una con su comportamiento frente a éste: el cliente, el servidor y el nodo.

En el caso de fallo de una aplicación cliente, éste deja de participar en la ejecución de la regata, lo cual modificará el estado de ésta, aunque no entorpecerá su desarrollo, a no ser que se trate del único cliente que esté participando en ese momento en la regata, que de ser el caso, ésta detendrá su desarrollo.

Por otro lado, si el fallo tiene lugar en el servidor de la regata, la ejecución de ésta se detiene, dado que no existe ninguna réplica que posea la misma funcionalidad del servidor y que pueda continuar con la ejecución de la regata, adoptando a los clientes que se queden "huérfanos". Por tanto, en caso de fallo del servidor, no existe ningún mecanismo de tolerancia a fallos.

El último componente que puede fallar es el nodo, y en este caso se diferencian dos conjuntos diferentes de nodos: los nodos que albergan clientes y el nodo que alberga al servidor.

- En el caso de que falle un nodo que ejecuta un cliente, éste deja de funcionar o funciona incorrectamente, lo que supone que el proceso cliente del simulador, dejará de funcionar, lo que supondrá que el servicio de la aplicación terminará para ese cliente, aunque el resto de la regata continuará con su funcionamiento normal, tal como se describe en el caso de fallo de la aplicación cliente.

- En el caso de fallo en el nodo que alberga al servidor, éste no podrá continuar con su funcionamiento, por lo que deja de ofrecer servicio para continuar con la regata y, por tanto, la ejecución se detiene.

Como se puede desprender de lo anteriormente descrito, el sistema posee únicamente cierta tolerancia a fallos, desde un punto de vista global de la regata, en los clientes, pero carece de esta característica, bien sea a nivel de servicio o bien a nivel de nodo, en el servidor de la regata.

Capítulo 6

Modelo de Distribución: Delegados Repetidores

6.1. Introducción

6.1.1. Idea seguida en el desarrollo del modelo

El modelo de distribución basado en servidores delegados repetidores se sustenta desde la base de que existe una red de interconexión entre los servidores, y que ésta es de alta capacidad. El funcionamiento propuesto para el modelo es sencillo: sigue existiendo un servidor central, el servidor maestro, que es el que ejecuta la simulación de la regata, y una serie de participantes en la regata, que se correspondería a los usuarios del sistema.

El factor que caracteriza a este modelo es que, además de las dos entidades existentes ya explicadas, servidor maestro y clientes, entra en escena una nueva figura: el servidor delegado repetidor, el cual se encargará de hacer de puente entre el servidor maestro y los clientes conectados a él, retransmitiendo las peticiones que le lleguen por ambos lados. Así pues, cuando un cliente desee realizar una acción, ésta será enviada al servidor delegado, el cual la retransmitirá, y modificará en caso de ser necesario, al servidor maestro, para que sea procesada. Por otro lado, cuando el servidor maestro envíe algún mensaje, éste será recibido por los servidores delegados, los cuales lo retransmitirán al cliente o clientes que deban recibirlo.

Hay que puntualizar que los servidores delegados actúan a modo de pasarela, sin lógica de cálculo relativa a la simulación de la regata. Por otro lado, se considerará durante el desarrollo de este modelo que el servidor maestro de la regata únicamente admite como clientes a los servidores delegados, y éstos a los clientes reales de la aplicación. Además, se considera que el actor que solicita nuevas instancias es el servidor web, y que las

solicita al nodo que debe instanciarlas.

6.1.2. Metas que se abordan en el presente modelo

La finalidad del modelo de servidores delegados repetidores consiste en, por un lado, reducir los retrasos existentes en las comunicaciones, y por otro lado, introducir cierta tolerancia a fallos en caso de caída de uno de los servidores.

La reducción de retrasos en las comunicaciones se consigue por el hecho de que los clientes ahora se conectan al servidor más próximo a ellos que, aunque no tiene porqué ser el servidor más cercano geográficamente, es el que posee un retraso menor en las comunicaciones (un ping). Así, el mensaje es transmitido vía internet el menor trayecto posible, para una vez recibido por el servidor delegado, ser retransmitido hacia el servidor maestro, ésta vez por una red contratada de un ancho de banda mayor.

Así, por ejemplo, un cliente en Valencia que desee jugar en el servidor de Melbourne, con el modelo centralizado tenía un retraso de 150 milisegundos, con éste modelo, suponiendo que existe un delegado en Dubai con un retraso de 80 milisegundos a Valencia, y al servidor maestro por red dedicada de 40 milisegundos, el cliente percibiría un retraso total de 120 milisegundos, menor que en el caso centralizado.

Por otro lado, la introducción de tolencia a fallos en el sistema se produce de forma directa en el caso de los servidores delegados dado que, en caso de fallo de uno de ellos, existen otros servidores delegados que pueden absorver los clientes que estaban conectados al servidor fallido, lo que representa que pueden continuar la regata.

6.2. El servidor maestro o coordinador

6.2.1. Modelado y funcionamiento

El algoritmo del servidor maestro dentro del contexto de un modelo de distribución basado en servidores delegados repetidores es bastante similar al modelo del servidor centralizado, dado que conserva gran parte de su idea inicial: un servidor se encarga de hacer progresar la regata, atender las peticiones que le llegan y difundir las sincronizaciones de estado necesarias para que todos los participantes tengan una visión coherente de la regata.
No obstante, este servidor maestro debe tener en cuenta la variación a la que ha sido sometido el sistema con la introducción de un nuevo elemento, por lo que debe atender a diferentes tipos de peticiones: por un lado, las peticiones recibidas de los servidores delegados y, por otro lado, las peticiones recibidas

de los clientes y transmitidas por los servidores delegados.

Maestro

Estado
estado = estado del servidor
listaClientes = delegados que coordina el master
relojSincroniza = reloj que gestiona las sincronizaciones
relojAvanza = reloj que gestiona el avance de la simulación
T_SINCRO = lapso de tiempo entre sincronizaciones
T_AVANZA = Lapso de tiempo entre progresiones del estado
START = mensaje de arranque de nuevos forwarders.
Acciones
init (B_i**,** V_i**,** Co_i**)**
listaClientes = { - }
lock (estado)
cargarEstado (B_i, V_i, Co_i)
unlock (estado)
relojSincroniza (T_SINCRO)
relojAvanza (T_AVANZA)
recibir (S_i**, {**C_i**, mensaje})**
if (mensaje.tipo == CONNECT)
conectar (C_i)
actualizaVivo (S_i)
if (mensaje.tipo == OP)
aplicarOperacion (C_i, op)
actualizaVivo (S_i)
if (mensaje.tipo == 2)
enviar (S_i, {C_i, estado.flota})
actualizaVivo (S_i)
if (mensaje.tipo == DISCONNECT)
desconectar (C_i)
actualizaVivo (S_i)
recibir (S_i**, mensaje)**
if (mensaje.tipo == CONNECT)
conectar (S_i)
if (mensaje.tipo == KEEPALIVE)
actualizaVivo (S_i)
if (mensaje.tipo == DISCONNECT)
desconectar (S_i)
instanciar ($Nodo_i$**)**
enviar ($Nodo_i$, START)

Figura 6.1: Maestro de la solución basada en delegados repetidores

Además, como se puede ver en la figura 6.1 se introduce un nuevo componente dentro del estado del servidor, el mensaje START, que será el encargado de iniciar nuevas instancias de servidores delegados, siendo enviado a los hosts que deban contenerlos (ver 5.3.2).

Inicialización

El proceso de inicialización del servidor maestro de la regata es similar al proceso de inicialización del servidor centralizado del capítulo anterior. Nue-

vamente, el servidor carga los valores estáticos de la regata (viento, recorrido, corrientes) desde un fichero facilitado a la aplicación durante su arranque. Por otro lado, el identificador de regata es generado automáticamente y se crean los barcos pertenecientes a la inteligencia artificial, aunque su estado permanece a 0, es decir, únicamente se instancia, pero no se inicializan.

Servidor

```
Inicializar
    init (fichero_viento, fichero_recorrido, fichero_corrientes, numbots)
        listaClientes = { - }
        lock (estado)
        cargarEstado (fichero_viento, fichero_recorrido, fichero_corrientes, numbots)
        unlock (estado)
        relojAvanza = new Reloj (T_AVANZA)
        relojSincro = new Reloj (T_SINCRO)
    cargarEstado(viento, recorrido, corriente, numbots)
        cargarViento (viento)
        cargarRecorrido (recorrido)
        cargarCorrientes (corriente)
        cargarFlotaBot (numbots)
    cargarViento (v)
        estado.viento = v.datos
    cargarRecorrido (r)
        estado.recorrido = r.datos
    cargarCorrientes (c)
        estado.corrientes = c.corrientes
    crearFlotaBot (numbots)
        for (i=0; i¡numbots; i++)
            estado.flota.add ( crearBarcoBot () )
```

Figura 6.2: Inicialización del servidor distribuido.

Además del proceso de carga del estado, mostrado en la figura 6.12, el proceso de inicialización del servidor instancia dos relojes. El primero de ellos, mostrado en la figura 6.3, será el encargado de hacer progresar el estado de la regata en el servidor.

```
RelojAvanza
    avanzar ( estado.flota)
    end
```

Figura 6.3: RelojAvanza en el servidor.

El otro reloj que se instancia es un reloj que produce los latidos necesarios para la sincronización de los servidores delegados repetidores del sistema, que se comportan como un cliente frente a este servidor maestro, con el estado del servidor central, como muestra la figura 7.9, lo que implica que éstos tomen el estado del servidor central como estado oficial de la regata.

```
RelojSincro
    sincronizar (listaClientes)
    end
```

Figura 6.4: RelojSincro en el servidor.

Avance del estado

Como se ha mencionado anteriormente, existe un reloj encargado de realizar una suerte de latidos que provocan el avance de los movimientos que debe realizar la flota en función de los parámetros de viento y corrientes que existan para un tipo determinado de barco en un punto determinado de recorrido.Además, el método de avance de los barcos calculará si existen colisiones entre los diferentes barcos participantes, sean comandados por bots o por humanos, o entre algún barco y alguna boya del recorrido, penalizando según las reglas de penalizaciones introducidas.

El progreso del estado de la regata es exactamente igual que en el modelo centralizado, aunque a continuación se vuelve a detallar.

```
avanzarEstado
    avanzar ()
        foreach Cata c in estado.flota
            colision = avanzar (c)
        if (colision)
            calcularPenalizaciones ()
    avanzar (cata)
        colision = false
        delta = cata.calcularDelta (cata.posicion, cata.direccion, estado.viento, estado.corrientes)
        cata.posicion = cata.posicion + delta
        foreach Cata c in estado.flota
            if (c.posicion == cata.posicion)
                colision = true
        return colision
    end
```

Figura 6.5: Avanzar Estado.

Tanto el cálculo del desplazamiento que realiza cada barco, *delta*, como el cálculo de las penalizaciones a aplicar en caso de colisión, no son de un interés vital para el desarrollo del presente documento, además de formar parte del funcionamiento interno de la aplicación, por lo que no se detallará el proceso seguido para dichos cálculos, dejándolos ver en la figura 8.10 únicamente como nombres de métodos y realizando un “acto de fe” acerca de sus correspondientes cálculos.

Peticiones de los delegados

La aparición de un nuevo elemento en el sistema conlleva la aparición de nuevas interacciones. En este apartado se van a contemplar las interacciones que tienen lugar debido al envío de mensajes por parte de un servidor delegado hacia el servidor maestro de la regata.
Dado que el servidor delegado no posee lógica para hacer progresar a la regata, las únicas peticiones que puede realizar como propias son las referentes a la conexión y desconexión al sistema y los mensajes de keep-alive que puede mandar al servidor para notificar que sigue formando parte de la regata aunque no transmita peticiones.

```
Recibir Petición
    recibir (S_i, mensaje)
        if (mensaje.tipo == CONNECT)
            conectar (S_i)
        if (mensaje.tipo == KEEPALIVE)
            actualizaVivo (S_i)
        if (mensaje.tipo == DISCONNECT)
            desconectar (S_i)
    conectar (cliente)
        listaClientes.add (cliente)
    desconectar (cliente)
        listaClientes.remove (cliente)
        sincronizar ()
    actualizaVivo (cliente)
        cliente = listaClientes.get(cliente.id)
        if (T_now - cliente.T_keep T_max)
            cliente.T_keep = T_now
        else
            desconectar (cliente)
    end
```

Figura 6.6: Peticiones con origen en un delegado.

Las peticiones con origen en los servidores delegados, tal como muestra la figura 6.7, no influyen en el estado de la regata, dado que los servidores delegados no pueden poseer un barco para participar, derecho exclusivo de los clientes finales.

Peticiones de los clientes

Como se ha mencionado anteriormente, todas las peticiones llegan al servidor maestro a través de los servidores delegados. No obstante, hay peticiones que tienen como emisor un cliente participante en la regata. Estas peticiones reciben un tratamiento especial dentro del servidor.

En el caso de conexión y desconexión de un cliente a la regata, se comprueba el hecho de que ese cliente sea del tipo al que se le puede asignar un barco. En el caso particular de conexión, y si se le puede asignar barco, se le transmite la parte estática del estado y se le asigna un barco. En el caso de

```
Recibir Petición de cliente
  recibir (S_i, {C_i, mensaje})
      if (mensaje.tipo == CONNECT)
          mensaje2 = conectar (C_i)
          actualizaVivo (S_i)
          enviar (S_i, {C_i, mensaje2}
      if (mensaje.tipo == OP)
          aplicarOperacion (C_i, op)
          actualizaVivo (S_i)
      if (mensaje.tipo == 2)
          enviar (S_i, {C_i, estado.flota})
          actualizaVivo (S_i)
      if (mensaje.tipo == DISCONNECT)
          desconectar (C_i)
          actualizaVivo (S_i)
  conectar (cliente)
      mensaje = estado.recorrido + estado.viento + estado.corrientes + estado.identificador
      if (cliente.tipo in {MASTER, CLIENT})
          estado.flota.add (cliente.barco)
      return mensaje
  desconectar (cliente)
      if (cliente.tipo in {MASTER, CLIENT}
          estado.flota.remove (cliente)
      sincronizar ()
  actualizaVivo (cliente)
      cliente = listaClientes.get(cliente.id)
      if (T_now - cliente.T_keep T_max)
          cliente.T_keep = T_now
      else
          desconectar (cliente)
  end
```

Figura 6.7: Peticiones con origen en un cliente.

desconexión, y bajo las mismas condiciones, se retira su barco de la regata. En ambos casos, posteriormente a estas acciones, se sincroniza al resto de clientes para que se den cuenta de los cambios.
Cuando la petición recibida es una petición de la información de la flota participante, mensaje FLOTA, se envía la información relativa a la flota.
Las peticiones recibidas que consistan en una operación ejecutada por un cliente, se atienden de la misma forma que en el modelo centralizado, tal como muestra la figura 7.12.

```
aplicarOperacion
    aplicarOperacion (cliente, op )
        avanzar (estado.flota.get(cliente.identificador), op)
        sincronizar ()
    avanzar (c, op)
        c.posicion = c.posicion + op.posicion
        c.direccion = c.direccion + op.direccion
    end
```

Figura 6.8: Aplicar una operación a un cliente.

La gestión de los keep-alive de los clientes se ve delegada en los servidores repetidores.

Sincronización de clientes

El proceso de sincronización se corresponde al envío del estado de la flota participante en una regata a los servidores delegados, para que puedan sincronizar a sus clientes. Como muestra la figura 6.9, el proceso servidor envía los datos pertenecientes a los barcos participantes, para obtener una imagen similar en todos los nodos que ejecuten la regata.

```
sincronizar
    sincronizar ()
        foreach barco in estado.flotas
            mensaje += barco.datos
        foreach cliente in listaClientes
            enviar (cliente, mensaje)
    end
```

Figura 6.9: Sincronización de los delegados.

Instanciación de nuevos delegados

El servidor maestro posee la potestad de instanciar nuevos servidores delegados en función de la demanda, tal como se muestra en la figura 6.10. Para ello envía un mensaje START a un nodo del sistema para que éste inice un nuevo delegado.

```
Instanciación
    instanciar (nodo)
        enviar (nodo, START)
    end
```

Figura 6.10: Instanciar delegados.

6.3. El servidor delegado o repetidor

6.3.1. Modelado y funcionamiento

El factor que diferencia al modelo distribuido basado en repetidores del modelo centralizado es la introducción de un componente nuevo, el servidor delegado repetidor, en el sistema. La finalidad de este servidor es la de actuar de pasarela entre los clientes y el servidor que ejecuta la regata, retransmitiendo los mensajes que recibe de uno u otro lado en el sentido correspondiente.

Delegado

```
Estado
    relojKeep = reloj encargado de la gestión del keep-alive
    T_KEEP = Lapso temporal entre envios de keep-alive
    KEEPALIVE = Mensaje de keep-alive
    master = referencia al servidor maestro de la regata
    CONNECT = mensaje de conexión
    listaClientes = clientes conectados al servidor
Acciones
    init (M_i)
        master = M_i
        listaClientes = { - }
        enviar (master, CONNECT)
        relojKeep = new Reloj (T_KEEP)
    recibir ( M_i, mensaje)
        difundir (listaClientes, estado)
    recibir (M_i, {C_i, mensaje})
        enviar (C_i, mensaje)
    recibir (C_i, mensaje)
        if (mensaje.tipo != KEEPALIVE)
            enviar (master, op)
        else
            actualizaVivo(C_i)
```

Figura 6.11: Delegado de la solución basada en delegados repetidores

Inicialización

El proceso de inicialización de un servidor repetidor en el sistema es bastante sencillo, dado que basta con facilitarle la referencia del servidor maestro en el momento de su creación para que pueda notificarle que se

conecta al sistema, así como todas las futuras peticiones que deba tramitar.

```
Inicializar
    init ( Mi)
        listaClientes = { - }
        master = Mi
        enviar ( Mi, CONNECT )
        relojKeep = new Reloj (T_KEEP)
```

Figura 6.12: Inicialización del servidor repetidor.

Dado que el servidor maestro debe conocer cuales de sus clientes, es decir, qué servidores repetidores permanecen activos, el servidor repetidor implementa un reloj que gestiona el envío de mensajes de keep-alive cada tiempo T_KEEP, como nos muestra la figura 6.13.

```
RelojKeep
    enviarKeep ()
        enviar (master, KEEPALIVE)
    end
```

Figura 6.13: RelojKeep.

Mensajes procedentes del servidor maestro

Los servidores delegados deben atender dos tipos de mensajes, los que reciben del servidor maestro y los que reciben del cliente. Los mensajes enviados por el servidor maestro pueden ser un mensaje punto a punto a un cliente (ver figura 6.14) de la regata, o bien una difusión (no se contempla la opción del multicast), como se muestra en la figura 6.15, por lo que se debe extraer el destinatario del mensaje en caso de que lo haya.

```
Recibir
    recibir ( Mi, {cliente, mensaje})
        enviar ( cliente, mensaje)
    end
```

Figura 6.14: Punto a punto a un cliente.

Como se puede observar en las figuras anteriores, en ningún caso se contempla la posibilidad de analizar el contenido del mensaje retransmitido, dado que el servidor repetidor carece de la funcionalidad para atenderlo.

```
Recibir
    recibir ( Mi, mensaje)
        difundir (listaClientes, mensaje)
    difundir (listaClientes, mensaje)
        foreach cliente in listaClientes
            enviar (cliente, mensaje)
    end
```

Figura 6.15: Difusiones a los clientes.

Mensajes procedentes de un cliente

Por su parte, los mensajes enviados por cualquiera de los clientes son reenviados directamente al servidor maestro para que los atienda y genere una respuesta que pueda ser retransmitida, tal como se muestra en la figura 6.16.

```
Recibir
    recibir ( Ci, mensaje)
        enviar (master, {Ci, mensaje})
    end
```

Figura 6.16: Retransmisión de los mensajes de los clientes.

6.4. Tolerancia a fallos.

Uno de los aspectos clave la hora de implantar un modelo de distribución en el sistema es su comportamiento ante un fallo en alguno de los componentes que lo integran. Para ello se procede a la descripción de este comportamiento desde el punto de vista de los procesos y de los nodos, en cada uno de los componentes del sistema: cliente, servidor delegado y servidor maestro.

Desde el punto de vista del proceso, un fallo en el cliente constituye que éste deja de funcionar total o parcialmente, lo que produce que no pueda participar en la simulación. Este hecho implica que el usuario asociado a la aplicación cliente quedará sin servicio hasta que inicie una nueva instancia del cliente, sin que el desarrollo de la regata se detenga, existiendo como única variación que el cliente que ha fallado se elimina del estado, y que la nueva instancia del cliente empieza de cero, esto es, no retoma el estado de la instancia fallida.

Por otro lado, dado que, como se ha mencionado anteriormente, los servidores delegados carecen de la lógica de simulación y se limitan a la mera

retrasmisión de mensajes y a la gestión de los keep-alive de los clientes. En el caso de que falle un proceso servidor delegado, pueden ocurrir dos supuestos:

- Que dicho proceso sea el único existente en el nodo: El servidor maestro se dará cuenta de que dicho proceso ha fallado (mediante gestión de keep-alive, por ejemplo) y mandará al nodo una solicitud para instanciar un nuevo servidor delegado, a ser posible en el mismo puerto. Una vez inicializado, los clientes deberán resincronizarse a través de los mensajes recibidos por el nuevo delegado. Se asume que el maestro conoce a los clientes y al delegado que tienen asociado.

- Que exista más de un proceso servidor delegado: Los clientes podrían reconectar con cualquier otro servidor delegado en tiempo de ejecución, una vez se hubiese detectado.

Finalmente, en el caso de que fuese el servidor maestro el que produjese un fallo, al no estar replicada la lógica de progreso de estado en ningún lugar, la simulación no podría continuar y el sistema se detendría.

6.5. Modelo de consistencia de datos

Del mismo modo que en el modelo centralizado existe una serie de clientes que envían mensajes a un servidor, y el servidor que sincroniza los estados de los clientes. Adicionalmente, se añade un conjunto de servidores delegados cuya funcionalidad será la de hacer las veces de pasarela entre los clientes y el servidor maestro. Exceptuando los mensajes de inicialización y centrándonos en el proceso de simulación, para simplificar el modelo, se puede afirmar que un cliente envía peticiones en base a su estado local (acciones) al servidor delegado repetidor, quien las reenvía al servidor maestro, el cual las procesa, independientemente de que lo haga también el cliente, y envía una sincronización a los delegados que éstos propagarán a los clientes con las modificaciones pertinentes (respuestas). No obstante, para no saturar la red, en lugar de generar una respuesta por cada petición, se acumulan n peticiones en una sincronización, siendo el ciclo de envío de S/t donde S es el número de sincronizaciones por t unidad de tiempo.

Dado que se mantienen los principios de difusión fiable FIFO con un único emisor de actualizaciones, se obtendrá el mismo nivel de consistencia de datos que en el modelo centralizado.

6.6. Resultados de la simulación

Del mismo modo que se ha realizado en el modelo centralizado, se han tomado tiempos de retraso de los mensajes para un sistema con una carga

baja de 10 clientes, una carga media de 20 clientes y una carga alta de 32 clientes. Asímismo se asume que los tiempos tomados no dependen del tipo de mensaje que se envía, dado que toda la lógica de procesado de mensajes sigue hallándose en el servidor maestro, como lo hiciera en el modelo centralizado.

Así pues, se presentarán los resultado obtenidos en las simulaciones en comparación con el modelo original, el centralizado, con el objetivo de poner de manifiesto las variaciones obtenidas de la aplicación del modelo de distribución basado en delegados repetidores con respecto a la situación incial.

En primer lugar, se procede a analizar el comportamiento del sistema con una carga baja. Para ello, se tienen 10 clientes conectados a 2 servidores delegados, los cuales conectan a su vez con el servidor maestro. Como en el modelo anterior, se realizan 10 rondas de sincronización para obtener los costes medios de cada cliente, mostrados en la gráfica 6.17, donde también se muestran los mismos resultados para el modelo centralizado.

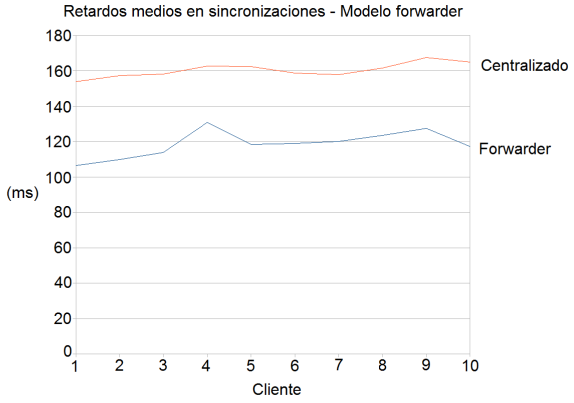


Figura 6.17: 10 clientes en modelo de delegados repetidores.

Como se puede observar en la gráfica, los retrasos medios para 10 clientes oscilan entre los 100 y los 135 milisegundos. Para obtener un retardo medio, se decide obtener el retardo medio del mensaje, obteniendo en primer lugar los intervalos de confianza mediante un diagrama Box-Whisker, mostrado en la figura 6.18, con el fin de eliminar resultados anómalos y obtener un retardo medio acorde a la realidad del sistema.

Una vez depurada la muestra, obtenemos que el retardo medio es de 118'01 milisegundos, con un 3 % de resultados anómalos, obteniendo así un decremento en el retraso de los mensajes del 25'17 %.

Por otro lado, también se realiza el mismo proceso de toma de muestras

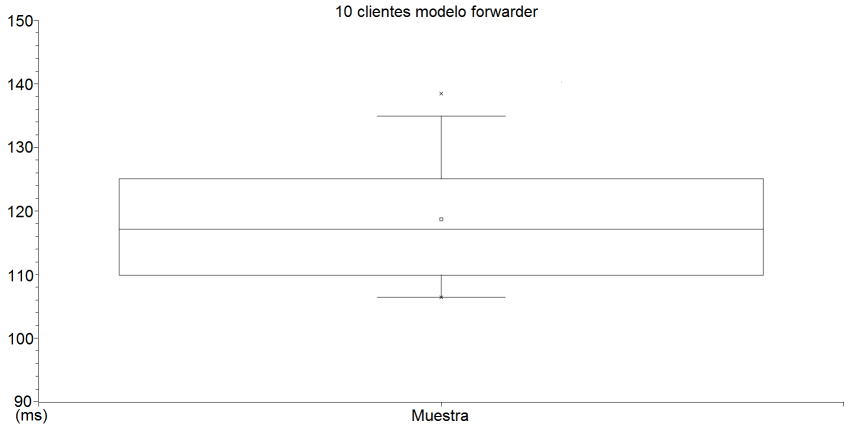


Figura 6.18: Intervalo de confianza de las mediciones obtenidas.

para un sistema con una carga media de 20 clientes conectados a 2 servidores delegados, coordinados por el servidor maestro. Asímismo, se mantienen las 10 rondas de mensajes para obtener los retardos medios por cliente, puestos en contraste con los mismos valores obtenidos en el modelo centralizado en la figura 6.19.

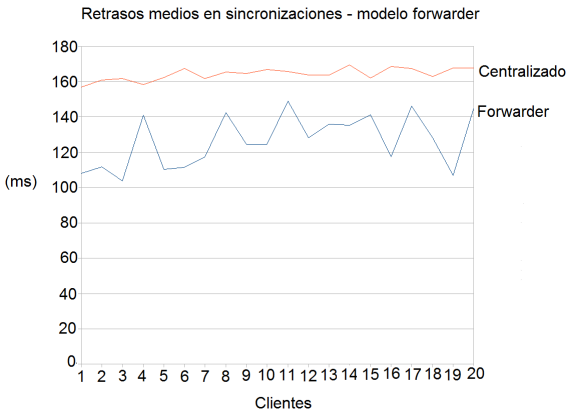


Figura 6.19: 20 clientes en modelo de delegados repetidores.

Revisando la gráfica, observamos que los valores oscilan entre los 105 y 150 milisegundos, siendo, en todo caso, unos costes inferiores a los mostrados en el mismo experimento por el modelo centralizado. No obstante, para mostrar un coste medio del mensaje de una forma fiable, se procede a obtener los intervalos de confianza, mostrados en la figura 6.20, para eliminar de la muestra todos aquellos datos anómalos obtenidos.

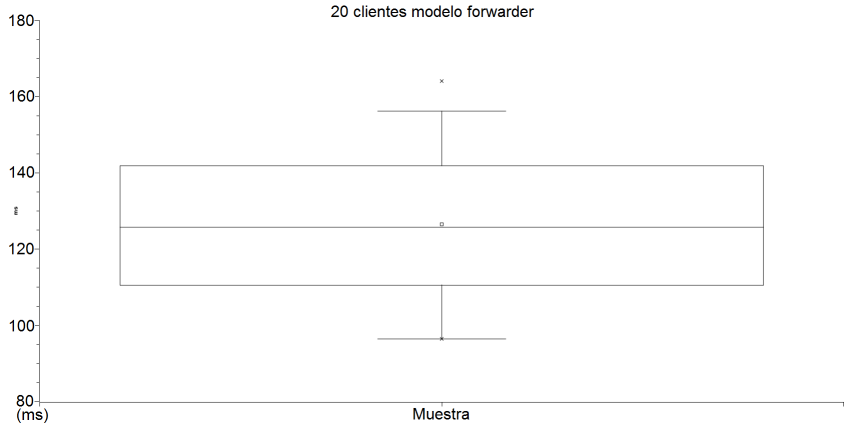


Figura 6.20: Intervalo de confianza de las mediciones obtenidas.

Una vez descartados los datos anómalos de la muestra, un 2 % del total, se obtiene un coste del mensaje de 125'64 milisegundos, que supone una reducción del coste de un 23'03 % con respecto al modelo centralizado.

Por último, se realizan los experimentos de 10 rondas de mensajes para una carga elevada del sistema, siguiendo el mismo esquema expuesto para los dos casos anteriores, pero con un total de 32 clientes en el sistema. En la figura 6.21, se muestran los retardos medios obtenidos durante la ejecución de la simulación del sistema frente a los obtenidos en la correspondiente simulación del modelo centralizado.

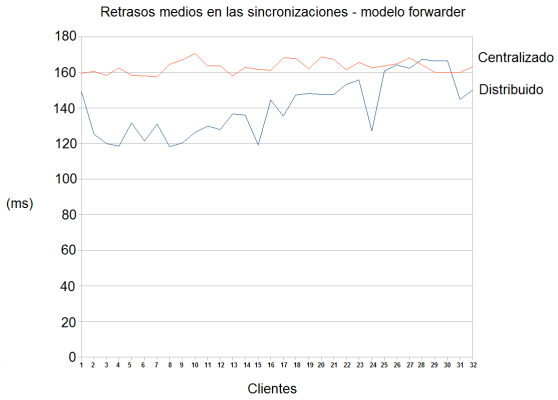


Figura 6.21: 32 clientes en modelo de delegados repetidores.

Como se puede observar, los retardos fluctúan entre los 115 y los 170

milisegundos, aunque para obtener un retardo medio realista, como es habitual, se realiza la obtención de un intervalo de confianza, mostrado en la figura 6.22, que eliminará de la muestra los datos anómalos obtenidos.

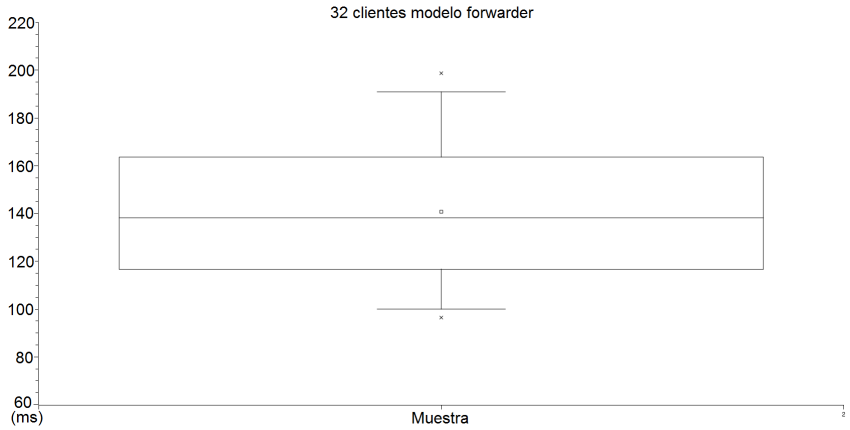


Figura 6.22: Intervalo de confianza de las mediciones obtenidas.

Tras descartar los datos anómalos se obtiene un retardo medio de 140'43 milisegundos por mensaje, lo que supone una reducción del 13'74 % respecto al mismo experimento del modelo centralizado.

6.7. Conclusiones

Al existir una red de mayor ancho de banda entre los diferentes servidores, y estar los diferentes servidores delegados más próximos a los clientes finales de la aplicación, se observa que tanto los tiempos obtenidos para los retrasos en la entrega de sincronizaciones como los retrasos de las peticiones enviadas por los clientes, retrasos de mensajes en general, se ven sustacialmente reducidos, por lo que, al tratarse de una simulación en tiempo real, se obtiene una mejor experiencia en la simulación de la regata.

Por otro lado, la introducción de nuevos elementos en el sistema supone la introducción de nuevos puntos en los que el sistema puede fallar. No obstante, dado que un cliente puede reconectar con otro servidor delegado en tiempo de ejecución, la caída de uno de los servidores delegados únicamente supondría, como peor caso, la obtención de un tiempo de reconexión que se traduciría en un ligero "salto" en la simulación. Una vez superada la reconexión al sistema, los retrasos obtenidos por los clientes mencionados, obtendrían, como máximo, el retraso obtenido en el modelo centralizado, por lo que se afirma que en caso de fallo, en el peor de los casos, se obtiene la misma experiencia de simulación que en el modelo centralizado, siendo,

en el resto de casos, una mejor experiencia de simulación debida a una reducción en los retrasos de los mensajes.

No obstante, dada la naturaleza de la aplicación, el proceso de reconexión al sistema no resulta trivial. Por un lado, en el modo aplicación del cliente, la reconexión a otro servidor delegado se puede realizar en tiempo de ejecución pero, por otro lado, en el caso de un cliente en modo applet, debido a las restricciones propias de los applets, esa reconexión automática no es posible, por lo que debería ser el usuario el que volviese a conectarse a la regata realizando todo el proceso de conexión inicial (busqueda por web, selección de regata, etc).

Asímismo, dado que n clientes participantes están conectados en m servidores delegados, siendo $m << n$, el servidor central deberá enviar un menor número de mensajes de sincronización, por lo que aumenta ligeramente la escalabilidad del sistema.

6.7.1. Problemas que plantea

Al margen de las mejoras que se obtienen con la aplicación de este modelo en el sistema BogReg, también se introducen nuevas características que pueden ocasionar problemas dentro del funcionamiento de la aplicación, o que, al menos, hay que tener controladas.

En primer lugar, sigue existiendo un único punto donde tiene lugar el cálculo del estado del sistema, el servidor maestro, por lo que, en caso de fallo de éste, la aplicación carece de medios para continuar con el desarrollo normal de una regata, dado que los servidores delegados carecen de la lógica del sistema necesaria para hacer progresar el estado de la regata, e incluso carecen de una copia del estado.

Para solventar este problema se pueden plantear dos estrategias: la primera consiste en mantener una réplica del servidor maestro que actúe a modo de réplica secundaria, que permanecería inactiva y que recibiría las actualizaciones desde el servidor maestro, así como mantendría un registro de las peticiones realizadas por los delegados para, en caso de caída del servidor central, aplicarlas al último estado recibido.

La segunda estrategia que se podría seguir es la de permitir que los servidores delegados pudiesen cambiar su rol y promocionar a servidor maestro. Esta estrategia requiere una mayor complejidad a nivel de código, dado que deberían estar implementados ambos servidores y ser capaces de cambiar de delegado a central en tiempo de ejecución. Además, los delegados deberían ejecutar un algoritmo de elección de líder para seleccionar cual de todos pro-

mociona a servidor maestro. Por si fuera poco, debería existir en el sistema un registro de las peticiones recibidas accesible por todos los servidores y los servidores delegados deberían mantener una copia del estado de la regata para cambiar el rol.

El hecho de que el modelo mantenga un único servidor maestro es el principal inconveniente que presenta este modelo, dado que representa un "punto caliente" en el sistema. Además de la falta de tolerancia a fallos existente en ese servidor que se ha comentado anteriormente, el servidor maestro representa un cuello de botella ante un elevado número de peticiones, pudiendo llegar a servirlas incorrectamente e incluso a ignorarlas debido a una elevada carga del sistema.

El servidor maestro, además de representar un cuello de botella ante un gran número de peticiones, es el que limita el número de participantes en una regata en función de los recursos disponibles y de la capacidad de servicio que posea, por lo que este modelo no aporta un alto grado de escalabilidad al sistema.

Como tercera gran cuestión pendiente en este modelo, al introducir un nuevo elemento intermedio en el sistema, el tráfico generado por el sistema se incrementa de forma que, donde inicialmente hacían falta 2 mensajes para una comunicación cliente-servidor (petición y respuesta), ahora se requieren 4 mensajes para la misma comunicación. No obstante, dado que los mensajes circulan por dos redes diferentes, y son diferentes procesos los que los generan y atienden, la red no se vería saturada, dado que en el peor de los casos el servidor atendería la misma cantidad de mensajes que en el modelo centralizado.

6.7.2. Líneas de desarrollo

En este capítulo se ha descrito un modelo de distribución cuya meta principal es la reducción de los retrasos en las comunicaciones debidos a la ubicación geográfica de tanto clientes como servidor, ubicando unos servidores repetidores más cercanos al cliente, que introduzcan los mensajes de éste en la red dedicada de los servidores.

Las mediciones obtenidas para la presentación están realizadas en un simulador que emplea un modelo simplificado del sistema, por lo que un primer paso en la línea de desarrollo del modelo sería la implementación de éste, así como comprobar que los resultados obtenidos se corresponden con los resultados reales.

Por otro lado, otra vía de desarrollo importante es la evolución del mode-

lo haciendo que los servidores delegados posean mayor funcionalidad relativa a la simulación de la regata, como, por ejemplo, el descrito en el siguiente capítulo.

Capítulo 7

Modelo de Distribución: Delegados con Reparto de Carga

7.1. Introducción

7.1.1. Idea seguida en el desarrollo del modelo

Para establecer las bases del modelo que se va a estudiar en este capítulo, conviene tener en cuenta tanto el modelo centralizado, ya que constituirá el punto de partida de este modelo, como el modelo basado en repetidores explicado en el capítulo 6.

Como se pudo ver en el capítulo 6, la idea consistía en introducir unos elementos intermedios que redirigieran las peticiones de los clientes y las respuestas del servidor central. En este capítulo se va un paso más allá, dado que sí que se introduce un paso intermedio entre clientes y servidor central pero, a diferencia del capítulo anterior, en este modelo sí que se cede cierta parte del cálculo relativo a la regata a los servidores delegados, por lo que se produce un reparto de carga.

Cabe puntualizar que si, como se ha citado anteriormente, un cliente no puede hacer progresar toda la regata por sí mismo, en el caso que nos ocupa el servidor central requiere de al menos un servidor delegado para hacer progresar el estado de la regata, así como los servidores delegados requieren de un servidor central.

7.1.2. Metas que se abordan en el presente modelo

En el modelo de distribución basado en delegados con reparto de carga se persiguen dos metas principales: por un lado, como en el caso anterior, la

reducción de los retrasos en las comunicaciones y, por otro lado, un aumento de la escalabilidad del sistema. De forma secundaria, se puede proporcionar cierta tolerancia a fallos haciendo que los delegados existentes absorban los clientes de un delegado que haya fallado.

La reducción de retrasos en las comunicaciones se consigue por el hecho de que los clientes ahora se conectan al servidor más próximo a ellos que, aunque no tiene por qué ser el servidor más cercano geográficamente, es el que posee un retraso menor en las comunicaciones (un ping). Así, el mensaje es transmitido vía internet el menor trayecto posible, para una vez recibido por el servidor delegado, ser retransmitido hacia el servidor central, ésta vez por una red contratada de un ancho de banda mayor.

Por otro lado, al descargar de cálculo al servidor central, éste puede atender un mayor número de peticiones, lo que permitiría que participase un número mayor de clientes y que existiese un mayor número de servidores delegados, es decir, podría escalar en tamaño el sistema, lo que resultaría beneficioso dada la finalidad del programa de simulación (cuantos más participen, mayor diversión, ¿no?).

7.2. El servidor central o coordinador

7.2.1. Modelado y funcionamiento

El servidor central tiene dos funcionalidades principales: por un lado, la inicialización de la regata y por otro lado, atender las peticiones que lleguen desde los servidores delegados, tal y como se muestra en la figura 7.1.

Maestro

Estado
estado = estado de la regata
listaClientes = procesos que son coordinados por el master
START = mensaje de inicio de servidores ligeros.
diferencia = diferencia entre un estado E_i y el siguiente E_{i+1}
alpha = variable para aplicar a posiciones de barcos.
Acciones
init (N_i, B_i, V_i, Co_i)
listaClientes = { - }
cargaEstado (B_i, V_i, Co_i)
enviar (N_i, START)
recibir (S_i, **mensaje**)
analizar (mensaje)

Figura 7.1: Maestro de la solución distribuida

La principal particularidad de este servidor es que no requiere de relojes

internos que mantengan el ritmo de desarrollo de la regata, dado que perfectamente puede atender las peticiones bajo demanda y delegar toda la parte dependiente del tiempo en los servidores delegados.

Inicialización

El servidor central carga los valores estáticos de la regata (viento, recorrido, corrientes) desde un fichero facilitado a la aplicación durante su arranque. Por otro lado, el identificador de regata es generado automáticamente y se crean los barcos pertenecientes a la inteligencia artificial, aunque su estado permanece a 0, es decir, únicamente se instancia, pero no se inicializan, tal como muestra la figura 7.2.

Servidor

Inicializar
init ($fichero_{viento}$**, fichero**$_{recorrido}$**, fichero**$_{corrientes}$**, numbots)**
listaClientes = { - }
lock (estado)
cargarEstado ($fichero_{viento}$, fichero$_{recorrido}$, fichero$_{corrientes}$, numbots)
unlock (estado)
enviar (N_i, START)
cargarEstado(viento, recorrido, corriente, numbots)
cargarViento (viento)
cargarRecorrido (recorrido)
cargarCorrientes (corriente)
cargarFlotaBot (numbots)
cargarViento (v)
estado.viento = v.datos
cargarRecorrido (r)
estado.recorrido = r.datos
cargarCorrientes (c)
estado.corrientes = c.corrientes
crearFlotaBot (numbots)
for (i=0; i¡numbots; i++)
estado.flota.add (crearBarcoBot ())

Figura 7.2: Inicialización del servidor central distribuido.

Por otro lado, se envía un mensaje a un nodo del sistema para que inicialice un servidor delegado para la regata en cuestión.

Recepción de mensajes

En la aproximación a un modelo de aplicación distribuido que nos ocupa, el servidor central de la aplicación en ningún momento conoce la existencia de las peticiones de los clientes, por lo que no requiere recibir los mensajes que éstos envían. Los mensajes que recibe el servidor central de la aplicación son los enviados por los servidores delegados, y son de tres tipos diferentes: conexión y desconexión, keep-alive y actualización del estado.

```
Recibir mensajes
  recibir (S_i, mensaje)
    if (mensaje.tipo == CONNECT)
      listaClientes.add(S_i)
      actualizaVivo (S_i)
      mensaje = { estado.recorrido + estado.viento + estado.corrientes + estado.id_regata}
      enviar (S_i, mensaje)
    if (mensaje.tipo == DELTA)
      actualizaVivo (S_i)
      aplicarDelta (mensaje.datos)
    if (mensaje.tipo == KEEPALIVE)
      actualizaVivo (S_i)
    if (mensaje.tipo == DISCONNECT)
      desconectar (S_i)
    if (mensaje.tipo == FLOTA)
      enviar (S_i, estado.flota)
  desconectar (cliente)
    estado.S_i.remove ()
    sincronizar ()
  actualizaVivo (cliente)
    cliente = listaClientes.get(cliente.id)
    if (T_now - cliente.T_keep T_max)
      cliente.T_keep = T_now
    else
      desconectar (cliente)
```

Figura 7.3: Recepción de mensajes en el servidor central distribuido.

Los mensajes de conexión y desconexión enviados por los servidores delegados se refieren a la inclusión de éstos en el sistema. Así, cuando un servidor delegado envía un mensaje CONNECT, está notificando que se une a la regata, por lo que el servidor central procede a añadirlo a su lista de clientes y a notificarle el estado en que se encuentra la regata. Como contrapunto, cuando se recibe una petición de DISCONNECT por parte de un servidor delegado, se procede a eliminarlo de la lista de difusión que emplea el servidor central, y además se procede a eliminar por completo el estado referente a él.

En este preciso instante se acaba de introducir un nuevo concepto, y es que los servidores delegados poseen una visión completa del estado, pero sin embargo siempre van a tratar con la misma sección de él, y que por tanto, existe un estado diferente asociado a cada servidor delegado. No obstante, esto será explicado con más detalle en la sección 7.5.

Por otro lado, se reciben mensajes de keep-alive que, como en los casos anteriores, sirve para comprobar si los clientes del servidor central continúan activos, aun en períodos de tiempo en los cuales no envían peticiones al servidor central.

Por último, existen los mensajes de tipo DELTA, que se corresponden a mensajes de sincronización parcial del estado, dado que únicamente actual-

izan la parte del estado de la que se encargan de gestionar cada uno de los servidores delegados. Se explicará con más detalle en el siguiente apartado.

Progreso del estado

En el proceso del servidor central del modelo con distribución de carga no existe un avance del estado propiamente dicho. Mientras que son los delegados los ocupados de hacer evolucionar las posiciones de los barcos, éstos envían un mensaje DELTA con las nuevas posiciones de los barcos que gestionan al proceso servidor central, el cual aplica los cambios pertinentes en las posiciones y posteriormente realiza su cálculo.

```
Aplicar Deltas
    aplicarDelta(deltaFlota)
        foreach Cata c in estado.flota
            foreach Cata c2 in deltaFlota
                if (c.id == c2.id)
                    c.datos = c2.datos
        obtenerColisiones ()
    obtenerColisiones ()
        foreach Cata c in estado.flota
          foreach Cata c2 in estado.flota
            if (c.id != c2.id)
                colision1 = c.calcularColision(c2)
                colision2 = c2.calcularColision(c)
                if (colision1 OR colision2)
                  culpable = obtenerCulpable (c, c2)
                separar (culpable)
                penalizar (culpable)
    separar (c)
        lock (estado)
        estado.c.posicion += alpha
        sincronizar ()
        unlock (estado)
```

Figura 7.4: Progreso del estado en servidor central.

Tras recibir el mensaje, se procede a modificar el estado de todos aquellos barcos que estén englobados en el mensaje de tipo DELTA, y se llama a la función que recalcula las posibles colisiones en base al estado recién actualizado. Cada colisión que se calcula se hace en base a dos barcos de la regata, y se comprueba qué barco colisiona con qué otro. En caso de que ambos o uno de ellos haya tenido colisión, se procede a obtener al culpable de la colisión (proceso en el que no se entra en detalle), penalizarlo y separarlo de la víctima, aplicándole a su posición un pequeño desplazamiento denominado ALPHA, tal como se muestra en 7.4, para posteriormente sincronizar a todos los delegados con el nuevo estado, con las nuevas posiciones y las colisiones calculadas.

Sincronizaciones

El proceso de sincronización se corresponde al envío del estado de la flota participante en una regata a los servidores delegados, para que puedan sincronizar a sus clientes. Como muestra la figura 7.5, el proceso servidor envía los datos pertenecientes a los barcos participantes, para obtener una imagen similar en todos los nodos que ejecuten la regata.

```
sincronizar
    sincronizar ()
        foreach barco in estado.flotas
            mensaje += barco.datos
        foreach cliente in listaClientes
            enviar (cliente, mensaje)
    end
```

Figura 7.5: Sincronización de los delegados.

7.3. El servidor delegado con reparto de carga

7.3.1. Modelado y funcionamiento

El segundo componente clave del sistema y el cual lo caracteriza es el servidor delegado. Este componente se establece a modo de pasarela entre el servidor central y los clientes finales de la aplicación pero, al contrario que en el modelo del capítulo 6, en estos servidores delegados sí que existe lógica referente a la simulación de las regatas.

Como se muestra en la figura 7.6, el servidor delegado tiene el comportamiento similar al servidor del modelo centralizado, pero la lógica que lo compone es más próxima a la del cliente que a la del servidor de dicho modelo, ya que está encargado de realizar únicamente los cálculos más ligeros, como puede ser el avance de los barcos, cediendo los cálculos más pesados al servidor central.

Inicialización

El proceso de inicialización del servidor delegado se inicia en el momento que un nodo decide crear una nueva instancia de éste, dándole una referencia al servidor central de la regata. El servidor delegado emplea esta referencia para las comunicaciones con el servidor central, siendo la primera de éstas una petición de CONNECT, cuya respuesta es un mensaje compuesto por la parte estática del estado de la regata. Una vez obtenida y procesada la petición de conexión, el servidor delegado envía una segunda petición FLOTA mediante la cual solicita la flota actual participante en la regata.

Delegado

Estado
master = servidor que inicia la regata
estado = estado de la regata
diferencia = diferencia entre un estado E_i y el siguiente E_{i+1}
listaCliente = clientes asociados al servidor S_i
relojAvanza = reloj encargado del avance de la simulacion
relojSincroniza = reloj que gestiona las sincronizaciones
T_AVANZA = lapso temporal entre avances de estado
T_SINCRO = lapso temporal entre sincronizaciones
relojKeep = reloj encargado de la gestion de keep-alive
T_KEEP = lapso temporal entre keep-alive
CONNECT = peticion para conectar
KEEPALIVE = mensaje de keep-alive
Acciones
init (M_i)
master = M_i
listaClientes = { - }
enviar (master, CONNECT)
estado = recibir (M_i, estado)
relojKeep = new Reloj (T_KEEP)
relojSincroniza = new Reloj(T_SINCRO)
relojAvanza = new Reloj (T_AVANZA)
recibir (M_i, mensaje)
analizar (mensaje)
recibir (C_i, mensaje)
analizar (mensaje)

Figura 7.6: Delegado de la solución distribuida

Inicializar
init (host, puerto)
M_i = < host, puerto >
conectar (M_i, CONNECT)
relojKeep = new Reloj (T_KEEP)
relojAvanza = new Reloj (T_AVANZA)
relojSincro = new Reloj (T_SINCRO)
conectar (M_i, CONNECT)
enviar (M_i, CONNECT)
recibir (mensaje)
analizar (mensaje)
enviar (M_i, FLOTA)
recibir (mensaje2)
analizar (mensaje2)
analizar (mensaje)
if (mensaje.tipo == FLOTA)
estado.flota = mensaje.flota
if (mensaje.tipo != flota)
estado.recorrido = mensaje.recorrido
estado.corrientes = mensaje.corrientes
estado.viento = mensaje.viento
estado.identificador = mensaje.identificador

Figura 7.7: Metodo de inicialización del servidor delegado.

Durante el proceso de inicialización del servidor delegado se crean también tres relojes, que son los que marcarán el ritmo de progreso de la aplicación. En primer lugar, se establece un reloj que haga progresar el estado de la regata, RelojAvanza, que será ejecutado en lapsos de tiempo de T_AVANZA milisegundos.

```
RelojAvanza
    avanzar ( estado.flota)
    end
```

Figura 7.8: RelojAvanza en el servidor delegado.

El segundo reloj que contiene el servidor delegado es el reloj que marca el ritmo de las sincronizaciones de los clientes, con la finalidad de que éstos tengan una visión coherente de la regata. Este reloj se ejecuta cada T_SINCRO milisegundos.

```
RelojSincro
    sincronizar (listaClientes)
    end
```

Figura 7.9: RelojSincro en el servidor delegado.

El tercer reloj que se inicializa es el reloj encargado de los envíos de keep-alive al servidor central, para casos en que la actividad de los clientes es baja y no existen peticiones que mandar al servidor central, seguir informando que el sistema sigue activo. Los mensajes de keep-alive se envían cada T_KEEP milisegundos.

```
RelojKeep
    enviarKeep (M_i)
        enviar (M_i, KEEPALIVE)
    end
```

Figura 7.10: RelojKeep.

Mensajes recibidos

El servidor delegado del sistema actúa a modo de intermediario entre el servidor central y los clientes finales, por lo que tiene dos entradas posibles de mensajes. Los mensajes procedentes del servidor son los empleados por el método de inicialización y, además, los mensajes de sincronización, que serán tomados por el servidor delegado como su estado, y que se encargará éste de notificar el nuevo estado a los diferentes clientes que son administrados por él.

Por otro lado están las peticiones procedentes de los clientes. Estas peticiones pueden ser de CONNECT y DISCONNECT, que añaden y eliminan

```
Recibir Mensajes
  recibir (M_i, mensaje)
      analizar (mensaje)
  recibir (C_i, mensaje)
      if (mensaje.tipo == CONNECT)
          listaCLientes.add (C_i)
          if (C_i.tipo in {MASTER, CLIENT})
              estado.flota.add (C_i.barco)
          enviar (master, estado)
          mensaje2 = { estado.recorrido + estado.viento + estado.corrientes + estado.id_regata}
          enviar (C_i, mensaje2)
      else
          analizar (mensaje)
  analizar (mensaje)
      if (mensaje.tipo == SINCRO)
          estado = mensaje.datos
          sincronizar ()
      if (mensaje.tipo == OP)
          aplicarOperacion (C_i, OP)
      if (mensaje.tipo == FLOTA)
          enviar (C_i, estado.flota)
      if (mensaje.tipo == DISCONNECT)
          desconectar (C_i)
      if (mensaje.tipo == KEEPALIVE)
          actualizaVivo (C_i)
  desconectar (cliente)
      if (cliente.tipo in {MASTER, CLIENT}
          estado.flota.remove (cliente)
      enviar (master, estado)
      sincronizar ()
  actualizaVivo (cliente)
      cliente = listaClientes.get(cliente.id)
      if (T_now - cliente.T_keep T_max)
          cliente.T_keep = T_now
      else
          desconectar (cliente)
```

Figura 7.11: Recepción de mensajes.

respectivamente a un cliente de la lista de clientes perteneciente al servidor delegado y, en caso de que el cliente añadido sea de tipo `master` o `client`, se le otorga un barco. En caso de ser una petición de CONNECT, además, se le envía un mensaje con la parte estática del estado de la regata.Por otro lado están las peticiones de FLOTA, las cuales son atendidas mediante una respuesta que consiste en la flota contenida en el estado de la regata, y que conforma su parte dinámica.
Los mensajes de keep-alive recibidos por el servidor delegado son envíos que realizan los clientes para mostrar que, a pesar de no generar peticiones, siguen activos en la regata. En caso de vencer el tiempo de keep-alive, el servidor procede a eliminar al cliente de la regata.
Por último están las peticiones de operación enviadas por los clientes, que serán detalladas en el siguiente apartado.

Avance del estado

Cuando el servidor delegado recibe una petición de operación por parte de un cliente, pasa a procesarla en su simulador interno. Este simulador, como se ha comentado anteriormente, es un simulador ligero similar al contenido en la aplicación cliente, es decir no realiza los cálculos de mayor complejidad computacional.

```
aplicarOperacion
    aplicarOperacion (cliente, op )
        avanzar (estado.flota.get(cliente.identificador), op)
        enviar (maestro, estado)
    avanzar (c, op)
        c.posicion = c.posicion + op.posicion
        c.direccion = c.direccion + op.direccion
    end
```

Figura 7.12: Aplicar una operación a un cliente.

En el momento de aplicar la operación, el sistema aplica los cambios pertinentes en los valores del estado. La diferencia radica en que tras aplicar estos cambios, se envía un mensaje al proceso servidor central con el nuevo estado, para que éste termine de realizar los cálculos correspondientes a colisiones. Dado que una vez realizado tal cálculo se difunde el nuevo estado con las nuevas colisiones calculadas, no es necesario esperar una respuesta, sino que cuando se reciba la siguiente sincronización, se redifundirá ésta hacia los clientes, con el nuevo estado totalmente calculado.

Sincronización de los clientes

El proceso de sincronización se corresponde al envío del estado de la flota participante en una regata a los clientes finales controlados por los usuarios, para que tengan una visión coherente de la regata. Como muestra la figura

7.13, el proceso servidor envía los datos pertenecientes a los barcos participantes, para obtener una imagen similar en todos los nodos que ejecuten la regata.

```
sincronizar
    sincronizar ()
        foreach barco in estado.flotas
            mensaje += barco.datos
        foreach cliente in listaClientes
            enviar (cliente, mensaje)
    end
```

Figura 7.13: Sincronización de los clientes.

7.4. Tolerancia a fallos.

La tolerancia a fallos en el modelo de servidores delegados con reparto de carga comparte contexto y solución con el modelo basado en delegados repetidores, mostrado en el apartado 6.4. No obstante, el estado global debería ser fácilmente derivable a partir del estado que mantienen los delegados. Bastaría con describir un protocolo de recuperación sencillo basado en la transferencia y fusión de tales estados.

Por otro lado, la "logica de progreso" podría ser fácilmente mantenida por todos los servidores delegados, aunque no se utilizaría hasta que promocionaran tras haber fallado el servidor central.

7.5. Modelo de replicación seguido

En el caso que nos ocupa de servidores delegados con reparto de carga, el modelo de replicación seguido no se ajusta a ninguno de los genéricos, aunque bien se puede explicar gracias a ellos, en concreto, gracias al modelo de replicación pasivo. Realmente, se podría afirmar que este sistema posee un modelo de replicación doblemente pasivo, dado que , para una transacción T enviada por un cliente, la recibe su servidor delegado asociado, el cual la procesa en cualquiera de los casos.

Como se ha indicado en la descripción de los servidores delegados, estos serán los encargados de atender operaciones que llamaremos 'ligeras', quedando el servidor maestro relegado al cálculo de las operaciones 'pesadas'. Para desambiguar estos términos, diremos que una transacción T está compuesta por un conjunto de operaciones ligeras C_l más un conjunto de operaciones persadas C_p, donde se asume que C_l nunca puede ser vacío.

Así pues, una vez procesada la transacción emitida por el cliente en el servidor delegado, éste asume el rol de réplica primaria para con el servidor maestro, enviándole un mensaje con la actualización del estado y, en caso de que proceda, una nueva petición con las operaciones pesadas pendientes de realizar. Aquí se muestra el primer modelo de replicación pasiva.

No obstante, cuando al servidor maestro le llegan las peticiones de operaciones pesadas, se comporta como una réplica primaria, atendiendo dichas operaciones, y posteriormente replicando su estado a los servidores delegados, que se comportan como réplicas secundarias en este caso, empleando de nuevo el modelo de replicación pasivo.

7.6. Modelo de consistencia de datos

Del mismo modo que en el modelo centralizado existe una serie de clientes que envían mensajes a un servidor, y el servidor que sincroniza los estados de los clientes. Adicionalmente, se añade un conjunto de servidores delegados cuya funcionalidad será la de atender ciertas peticiones de los clientes dando cuenta al servidor maestro de lo acontecido, o bien redirigirlas directamente al servidor maestro. Exceptuando los mensajes de inicialización y centrándonos en el proceso de simulación, para simplificar el modelo, se puede afirmar que un cliente envía peticiones en base a su estado local (acciones) al servidor delegado, quien las recibe y, en el caso que corresponda las procesa o bien reenvía al servidor maestro, el cual las procesa, independientemente de que lo haga también el cliente, y envía una sincronización a los delegados que éstos propagarán a los clientes con las modificaciones pertinentes (respuestas). No obstante, para no saturar la red, en lugar de generar una respuesta por cada petición, se acumulan n peticiones en una sincronización, siendo el ciclo de envío de S/t donde S es el número de sincronizaciones por t unidad de tiempo.

El hecho de realizar envíos y recepciones de datos supone que los clientes pueden, potencialmente, obtener diferentes vistas de un mismo estado. Para evitar este problema, los mensajes de sincronización son numerados mediante un identificador de tal forma que una vez entregado el mensaje de sincronización M_n, el mensaje M_{n-1} sea descartado, no permitiendo entregas fuera de orden, dado que el mensaje M_n contendrá el estado de aplicar sus modificaciones más todas las anteriores y garantizando así un modelo de consistencia de datos ligeramente más relajado que el secuencial. Obsérvese que las actualizaciones enviadas por el servidor central proporcionarían tal consistencia secuencial. Sin embargo, parte de esas actualizaciones ya se han realizado en el servidor delegado, sin haberse propagado en dicho momento al resto de nodos. Por tanto, esos servidores delegados obtienen una consis-

tencia similar a la consistencia Procesador [**?**] para sus clientes locales. Sin embargo, los clientes, al recibir únicamente actualizaciones desde el servidor central, siguen percibiendo una consistencia secuencial.

7.7. Resultados de la simulación

En el presente modelo de servidores delegados con reparto de carga se distinguen, como se ha mencionado en el apartado 7.5, dos tipos de operaciones: ligeras y pesadas. El retardo de un mensaje que genere una operación ligera es mínimo, dado que será atendido por el servidor delegado más cercano al cliente y, por tanto, al que estará conectado este último. Por tanto, estos retardos no serán tenidos en cuenta para el análisis del modelo que nos ocupa. Cabe reseñar que el hecho de procesar por separado las operaciones ligeras descarga al servidor maestro de cierta cantidad de mensajes, que depende en gran medida del desarrollo de la simulación, por lo que no es posible estimar.

En cuanto al análisis de los retrasos obtenidos por los mensajes de sincronización (los enviados por el servidor maestro en general), el esquema de pruebas seguido para la obtención de resultados es el mismo empleado en los modelos anteriores: una batería de pruebas para carga baja del servidor (10 clientes), otra para carga media (20 clientes) y otra para alta (32 clientes). En primer lugar, se procede a analizar el comportamiento de la aplicación para una carga del sistema baja, o lo que es lo mismo, 10 clientes conectados a los 2 servidores delegados, los cuales a su vez comunican con el servidor maestro. Tras realizar el proceso de inicialización del sistema y de conexión de los clientes, se realiza una serie de 10 rondas de sincronización del servidor, obteniéndose los retardos medios para cada cliente, mostrados en la figura 7.14 .

Como se puede observar, los retardos obtenidos oscilan entre los 105 y los 115 milisegundos aproximadamente, siempre por debajo de los resultados obtenidos en la misma prueba para el modelo centralizado. No obstante, se procede a eliminar los resultados anómalos mediante la obtención de un intervalo de confianza, mostrado en la figura 7.15, para la obtención de un retardo medio realista.

Tras eliminar los resultados anómalos de la muestra, un 2 % del total, se obtiene un retardo medio de 109'52 milisegundos, por lo que el retardo de estos mensajes frente al modelo centralizado se reduce en un 30'55 %.

Por otro lado, se repite el experimento de 10 rondas de sincronizaciones, esta vez para una carga media del sistema, es decir, 20 clientes conectados a 2 servidores delegados, y éstos al servidor maestro. Tras la realilzación de

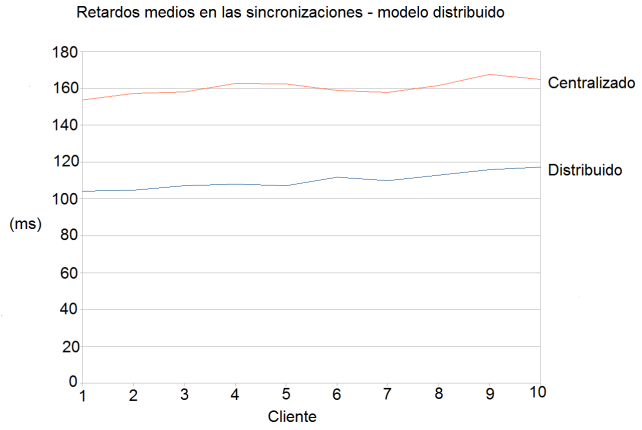


Figura 7.14: 10 clientes en modelo distribuido.

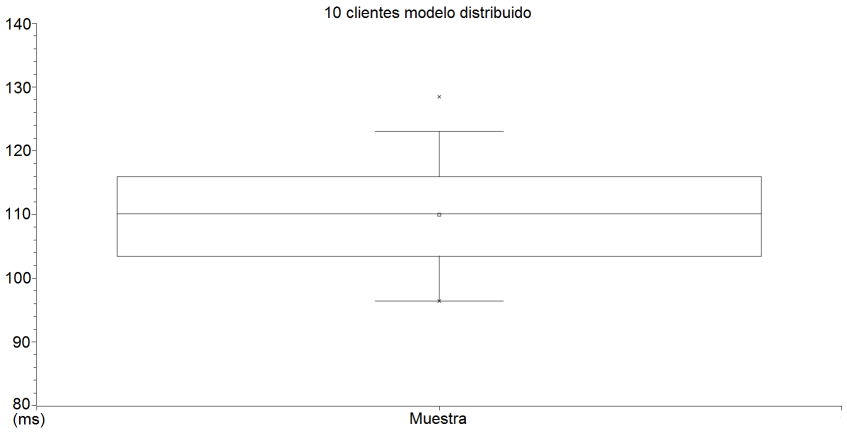


Figura 7.15: 10 clientes en modelo distribuido.

las rondas mencionadas, se obtienen los resultados medios para cada cliente observados en la figura 7.16.

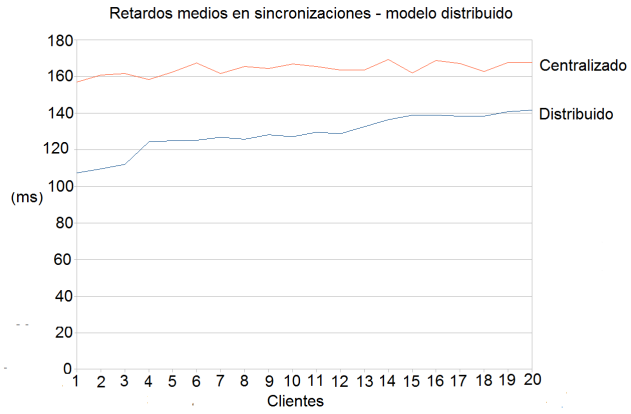


Figura 7.16: 20 clientes en modelo distribuido.

Como se puede observar en la gráfica, los retardos medios por cliente oscilan entre los 108 y los 142 milisegundos de media. Eliminando los datos anómalos de la muestra, mediante el diagrama de la figura 7.17, un 4 % del total, se obtiene un retardo medio de 129'58, un 20'62 % menos que en el modelo centralizado.

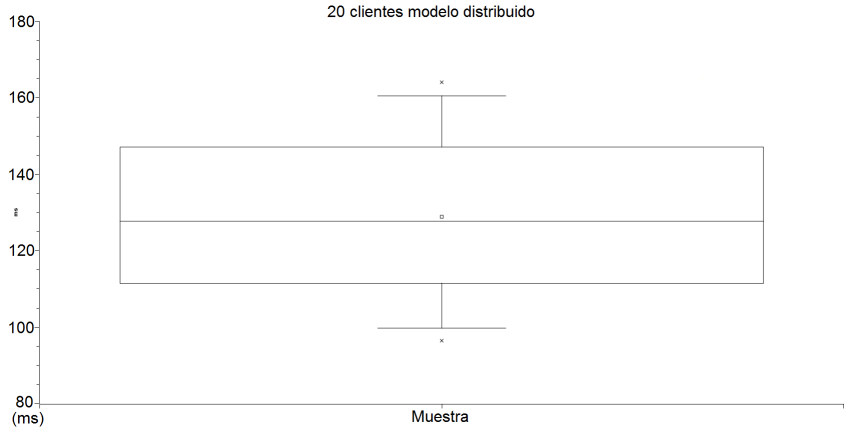


Figura 7.17: 20 clientes en modelo distribuido.

Por último, se realizan los experimentos para una carga elevada del sistema, esto es, 32 clientes conectados a 2 servidores delegados, éstos a su vez conectados a un servidor maestro. Los retardos medios obtenidos tras 10 rondas de mensajes son los mostrados en la figura 7.18, que oscilan entre los

105 y los 180 milisegundos por cliente. No obstante, para obtener un retardo medio confiable, hay que descartar los valores anómalos de la muestra, por lo que se realiza un intervalo de confianza, representado por el diagrama de la figura 7.19.

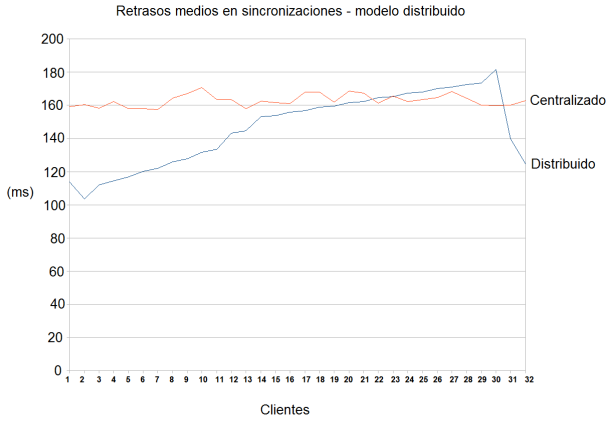


Figura 7.18: 32 clientes en modelo distributed.

Tras limpiar la muestra de resultado anómalos, se extrae un retardo medio por mensaje de 140'69 milisegundos, lo que reduce este retardo un 13'58 % respecto al modelo centralizado.

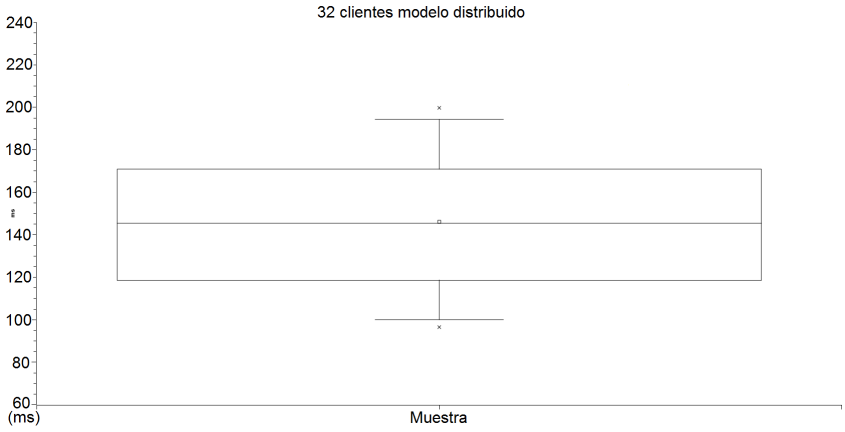


Figura 7.19: 32 clientes en modelo distribuido.

7.8. Conclusiones

7.8.1. Mejoras obtenidas

Por un lado, las transacciones que incluyan únicamente operaciones ligeras, se podrán responder en un tiempo inferior que al resto de casos, dado que no es necesaria la comunicación con el servidor maestro para su respuesta. No obstante, el servidor maestro deberá recibir cuentas de todas las operaciones realizadas por los servidores delegados, siempre y cuando éstas supongan una modificación del estado de la simulación. Por tanto, se descarga de cierto número de peticiones al servidor maestro.

Por otro lado, al existir una red de mayor ancho de banda entre los diferentes servidores, y estar los diferentes servidores delegados más próximos a los clientes finales de la aplicación, se observa que tanto los tiempos obtenidos para los retrasos en la entrega de sincronizaciones como los retrasos de las peticiones enviadas por los clientes, retrasos de mensajes en general, se ven sustacialmente reducidos, por lo que, al tratarse de una simulación en tiempo real, se obtiene una mejor experiencia en la simulación de la regata.

7.8.2. Problemas que plantea

El hecho de que el modelo mantenga un único servidor maestro es el principal inconveniente que presenta este modelo, dado que representa un "punto caliente" en el sistema. Además, el servidor maestro representa un cuello de botella ante un elevado número de peticiones, pudiendo llegar a servirlas incorrectamente e incluso a ignorarlas debido a una elevada carga del sistema.

El servidor maestro, a pesar de representar un cuello de botella ante un gran número de peticiones, lo que puede limitar el número de participantes en una regata en función de los recursos disponibles y de la capacidad de carga que posea, sí que aporta un mayor grado de escalabilidad que el modelo anterior, dado que existe un menor tráfico de peticiones dentro de la red que conecta los servidores.

7.8.3. Líneas de desarrollo

Se ha observado que la reducción de operaciones en el servidor maestro fomenta el hecho que se puedan albergar un mayor número de clientes, dado que el servidor tardará más en sobrecargarse. Por otro lado, se observa también que acercando la lógica de simulación a los clientes se reduce el tiempo de respuesta de sus peticiones, por lo que el paso que se propone

como línea de desarrollo es el de replicar la lógica de un servidor completo en cada uno de los servidores delegados, relegando el papel del servidor maestro a un mero coordinador de estados, tal y como se muestra en el siguiente capítulo.

Capítulo 8

Modelo de Distribución: Delegados Replicados

8.1. Introducción

8.1.1. Idea seguida en el desarrollo del modelo

En el presente capítulo se expone un tercer modelo de distribución de los servidores de la aplicación BogReg, que podría ser considerado como una evolución natural de los modelos explicados en los anteriores capítulos. En éste, todos los servidores poseen exactamente la misma funcionalidad, quedando reducida la funcionalidad especial del servidor maestro a un mero coordinador de estados de los servidores.

Como en los anteriores modelos, se establece un componente intermedio en el sistema, ubicado entre el servidor central y los clientes. No obstante, el hecho que diferencia este modelos de los anteriores es que estos componentes intermedios, los servidores delegados, poseen una réplica exacta de la lógica de simulación de la regata del servidor maestro, es decir, que los servidores delegados poseen un simulador completo de la regata.

Por otro lado, la coordinación de los estados de los diferentes servidores se logrará mediante las actualizaciones por parte de aquel servidor que reciba cualquier cambio de estado al resto de servidores, y adicionalmente, cada cierto tiempo, una actualización por parte del servidor maestro al resto del grupo del estado total de la simulación, dado que se tomará como estado oficial el de este último.

8.1.2. Metas que se abordan en el presente modelo

En el presente modelo, en primer lugar, se persigue una disminución de los retardos en las comunicaciones, estableciendo servidores más cercanos y

que compartan entre sí un ancho de banda dedicado superior al que se puede obtener vía Internet, lo que disminuye la distancia entre cliente y servidor, y por tanto, el tiempo de transmisión de los paquetes en las comunicaciones. Por otro lado, se aborda también un aumento de escalabilidad en el sistema. Al existir un grupo de servidores delegados con la funcionalidad completa del simulador de la regata, cada servidor delegado puede hacer evolucionar el estado completo de los clientes que tiene conectados, lo que se podría comparar con tener múltiples modelos centralizados, coordinados por uno de ellos, escogidos bajo cierto criterio, como puede ser, un algoritmo de elección de líder.

Adicionalmente, el hecho de introducir servidores delegados con la funcionalidad completa del sistema significa que se está introduciendo replicación en el sistema, factor que está directamente relacionado con la tolerancia a fallos en un sistema distribuido. Así, en caso de que falle uno de los servidores, los clientes se pueden reubicar en el sistema. Adicionalmente, si es el servidor central el que falla, bastaría con elegir un nuevo líder en la regata para continuar con el funcionamiento normal de la regata.

8.2. Los servidores en el modelo replicado

8.2.1. Modelado y funcionamiento

En el modelo de distribución basado en servidores replicados tanto éstos como el servidor central poseen exactamente la misma funcionalidad, quedando el servidor central relegado a un mero papel de coordinador, por lo que no se hace necesario separar los modelos de los delegados y del servidor central, tal como muestra la figura 8.1.

En este modelo se establece un flag que indica al proceso si es el coordinador de la regata, y que le permite acceder a la funcionalidad adicional propia del servidor central, tal como coordinación del resto de servidores e inicialización de la regata.

La principal característica que diferencia este modelo de los dos modelos anteriormente explicados es que, en este caso, cualquier servidor puede aceptar clientes y gestionarlos, dado que todos se comportan de una forma similar.

Inicialización

El proceso de inicialización de los servidores es el único que presenta dos facetas, dependiendo de si es el proceso de inicialización del servidor central o de un servidor delegado. Este hecho se debe a que es el servidor central

Estado
regata = identificador de regata
master = coordinador de la regata
diferencia = diferencia entre un estado E_i y el siguiente $E_i + 1$
listaClientes = lista de clientes asociados al servidor S_i
relojAvanza = reloj que gestiona el avance de la simulacion
relojSincroniza = reloj que gestiona las sincronizaciones
T_AVANZA = lapso de tiempo entre progresiones del estado
T_SINCRO = lapso temporal entre sincronizaciones
CONNECT = notificacion de conexion al master
grupoServidores = grupo de comunicaciones al que pertenecen los servidores asociados a una regata.
flag_master = Indica si el servidor es master
Acciones
init (host, puerto)
flag_master = false
listaClientes = { - }
master = M_i
enviar (master, CONNECT)
relojSincroniza = new Reloj(T_SINCRO)
relojAvanza = new Reloj (T_AVANZA)
grupoServidores = master.grupo.join()
init ($fichero_{viento}$**, fichero**$_{recorrido}$**, fichero**$_{corrientes}$**, numbots)**
flag_master = true
listaClientes = { - }
lock (estado)
cargarEstado ($fichero_{viento}$, fichero$_{recorrido}$, fichero$_{corrientes}$, numbots)
unlock (estado)
relojSincroniza = new Reloj (T_SINCRO)
relojAvanza = new Reloj (T_AVANZA)
grupoServidores = this.grupo.create()
recibir (M_i**, mensaje)**
analizar (mensaje)
recibir (S_i**, mensaje)**
analizar (mensaje)
recibir (C_i**, mensaje)**
analizar (mensaje)

Figura 8.1: Servidor de la solución replicada

el primero en inicializarse, cargando su estado en parte desde ficheros y en parte generándola de cero, mientras que los servidores delegados deben solicitar al central el estado de la regata.

El proceso de inicialización del servidor central se basa en la carga de los ficheros que contienen la parte estática del estado de la regata, la creación de la flota comandada por bots, la inicialización de los relojes de avance y sincronización y la creación de un grupo de comunicaciones para los servidores de la regata, como muestra la figura 8.2.

```
Inicializar
    init (fichero_viento, fichero_recorrido, fichero_corrientes, numbots)
        flag_master = true
        listaClientes = { - }
        lock (estado)
        cargarEstado (fichero_viento, fichero_recorrido, fichero_corrientes, numbots)
        unlock (estado)
        relojSincroniza = new Reloj (T_SINCRO)
        relojAvanza = new Reloj (T_AVANZA)
        grupoServidores = this.grupo.create()
    cargarEstado(viento, recorrido, corriente, numbots)
        cargarViento (viento)
        cargarRecorrido (recorrido)
        cargarCorrientes (corriente)
        cargarFlotaBot (numbots)
    cargarViento (v)
        estado.viento = v.datos
    cargarRecorrido (r)
        estado.recorrido = r.datos
    cargarCorrientes (c)
        estado.corrientes = c.corrientes
    crearFlotaBot (numbots)
        for (i=0; i¡numbots; i++)
            estado.flota.add ( crearBarcoBot () )
```

Figura 8.2: Proceso de inicialización del servidor central

Por otro lado, el proceso de inicialización de un servidor delegado, tal como muestra la figura 8.3, comienza por facilitarle un host y un puerto que se corresponderán con la referencia al servidor central, y a donde mandará un mensaje de conexión que, una vez atendido, le facilitará el estado de la regata.

Tras el proceso de carga del estado, punto en el que se diferencian el servidor delegado del servidor central, se intancian dos relojes: el primero de ellos, en la figura 8.4, es el reloj ocupado de hacer progresar el estado de la regata cada cierto lapso de tiempo T_AVANZA.

Por su parte, el segundo reloj que se pone en marcha es el ocupado del proceso de sincronización, como muestra la figura 8.5, tanto para los clientes como entre servidores en caso de ser el servidor central, pero esto será explicado más adelante.

```
Inicializar
    init (host, puerto)
        flag_master = false
        M_i = < host, puerto >
        conectar (M_i, CONNECT)
        relojKeep = new Reloj (T_KEEP)
        relojAvanza = new Reloj (T_AVANZA)
        relojSincro = new Reloj (T_SINCRO)
    conectar (M_i, CONNECT)
        enviar (M_i, CONNECT)
        recibir (M_i, mensaje)
        analizar (mensaje)
        enviar (M_i, FLOTA)
        recibir (mensaje2)
        analizar (mensaje2)
    analizar (mensaje)
        if (mensaje.tipo == FLOTA)
            estado.flota = mensaje.flota
        if (mensaje.tipo != flota)
            estado.recorrido = mensaje.recorrido
            estado.corrientes = mensaje.corrientes
            estado.viento = mensaje.viento
            estado.identificador = mensaje.identificador
```

Figura 8.3: Metodo de inicialización del servidor delegado.

```
RelojAvanza
    avanzar ( estado.flota)
    end
```

Figura 8.4: RelojAvanza en el servidor.

```
RelojSincro
    sincronizar (listaClientes)
    end
```

Figura 8.5: RelojSincro en el servidor.

Recepción de peticiones

Como en los anteriores modelos explicados, el modelo de distribución basado en replicación de servidores también recibe mensajes con peticiones que deben ser atendidas, aunque en este caso todos los servidores son capaces de procesar cualquier tipo de petición.

```
Mensajes recibidos
    recibir (M_i, mensaje
        analizar (mensaje)
    recibir (S_i, mensaje
        analizar (mensaje)
    recibir (C_i, mensaje
        analizar (mensaje)
    analizar (m)
        if (mensaje.tipo == SINCRO)
            if (!flag_master)
                estado = mensaje.datos
        if (mensaje.tipo == OP)
            aplicarOperacion (mensaje.datos)
            actualizaVivo (mensaje.emisor)
            sincronizarDelta ()
        if (mensaje.tipo == DELTA)
            foreach Cata c in estado.flota
                foreach Cata c2 in m.datos.flota
                    if (c.id == c2.id)
                    c.datos = c2.datos
        if (mensaje.tipo == CONNECT)
            conectar (mensaje.emisor)
            sincronizar ()
        if (mensaje.tipo == DISCONNECT)
            desconectar (mensaje.emisor)
            sincronizar ()
        if (mensaje.tipo == KEEP_ALIVE)
            actualizaVivo (mensaje.emisor)
```

Figura 8.6: Recepción de peticiones

Los mensajes recibidos pueden provenir, desde el punto de vista de cualquier servidor S_i sin importar si es maestro (más apropiado en este caso) o delegado, de tres orígenes diferentes: el servidor central o maestro, los servidores delegados y los clientes.

En primer lugar, los mensajes provinientes del servidor maestro nunca serán de tipo CONNECT, dado que se considera que el servidor maestro es el que inicia la regata o, en caso de fallo, uno que ya pertenecía al sistema. Asímismo, tampoco podrá enviar mensajes DISCONNECT, dado que se fomentará que el servidor maestro no cambie salvo cuando haya fallado. En el resto de los casos, si el mensaje CONNECT proviene de un cliente, se añade éste a la lista de clientes gestionada por el servidor y en el caso correspondiente se le otorga barco, o se añade un nuevo servidor a la lista de servidores en caso de provenir de un servidor. En el caso de recibir un mensaje de DISCONNECT, si se trata de un cliente el emisor, se elimina

toda la parte del estado referente a ese cliente. En el caso de provenir desde un servidor, se eliminará todo el estado asociado a dicho servidor.

```
Conectar
    conectar (sender)
        if (sender.tipo instance of Server)
            grupoServidores.add(sender.id)
            if (flag_master)
                enviar (sender, estado)
        if (sender.tipo in {MASTER, CLIENT}
            listaClientes.add (sender)
            estado.flota.add (cliente)
        else
            listaClientes.add (sender)
```

Figura 8.7: Conexión de nuevos elementos al sistema

```
Desconectar
    desconectar (sender)
        if (sender.tipo instance of Server)
            grupoServidores.remove(sender.id)
            estado.remove (sender.estado)
        if (sender.tipo in {MASTER, CLIENT}
            listaClientes.add(sender)
            estado.flota.remove (cliente)
        else estado.remove (sender.id)
```

Figura 8.8: Desconectar componentes del sistema.

Los mensajes de tipo SINCRO no serán atendidos por el servidor maestro, dado que si un mensaje de sincronización es emitido en el instante t y se entrega en el instante $t + n$, todas las peticiones recibidas y procesadas entre ambos instantes serán descartadas, dado que no se garantiza orden de entrega. Si cualquier otro servidor recibe un mensaje de SINCRO, tomará el estado de éste y sustituirá el propio.

Se introduce un nuevo tipo de mensaje, DELTA, cuyo funcionamiento es similar a una sincronización. La particularidad que tiene esta sincronización es que sólo abarca al estado de la regata que gestiona un determinado servidor, ajustandose al modelo de replicación coordinador-cohorte, que será explicado en el apartado correspondiente (ver apartado 8.4).

Los mensajes de KEEP_ALIVE sólo son procedentes de los clientes, dado que el conjunto de servidores posee un monitor de pertenencia a grupos, que notificará cuándo un servidor se une o se separa del sistema. Cuando se recibe un mensaje de este tipo, se actualiza el estado del cliente para que

permanezca vivo aunque no emita peticiones.

```
Actualizar Vivos
    actualizaVivo (cliente)
        cliente = listaClientes.get(cliente.id)
        if (T_now - cliente.T_keep T_max)
            cliente.T_keep = T_now
        else
            desconectar (cliente)
```

Figura 8.9: Gestión de keep-alive.

El último tipo de mensajes que puede recibir un servidor S_i es de tipo OP, es decir, peticiones de operación, que serán tratadas en el siguiente apartado. Despues de ser tratadas por el proceso correspondiente, se genera un mensaje DELTA que será enviado a todos los servidores.

Progreso del estado

El progreso del estado en un servidor S_i, sea éste maestro o delegado, depende de dos factores: la función ejecutada por el reloj de avance de estado y las peticiones peticiones que éste reciba procedentes de los clientes. La función ejecutada por el reloj provoca el avance de los movimientos que debe realizar la flota en función de los parámetros de viento y corrientes que existan para un tipo determinado de barco en un punto determinado de recorrido. Además, el método de avance de los barcos calculará si existen colisiones entre los diferentes barcos participantes, sean comandados por bots o por humanos, o entre algún barco y alguna boya del recorrido, penalizando según las reglas de penalizaciones introducidas. En caso de que las colisiones tengan lugar entre dos barcos del mismo servidor, será éste el que las calcule, mientras que si las colisiones son entre barcos gestionados por diferentes servidores, las ejecutará sólo uno de ellos, elegido bajo cierto criterio como, por ejemplo, un identificador más bajo.

Aplicar una operación es resultado de que uno de los clientes ha realizado una acción, por lo que se ha de encontrar el barco asociado al cliente y modificar el estado de dicho barco en función de los parámetros designados en el mensaje de operación.

Sincronizaciones

El proceso de sincronización se corresponde al envío del estado de la flota participante en una regata a los clientes desde los servidores delegados, para que puedan sincronizar su estado, así como la sincronización de los delegados producida por el servidor maestro. Como muestra la figura 8.12,

```
avanzarEstado
    avanzar ()
        foreach Cata c in estado.flota
            foreach nid in listaClientes
                if (c.id == nid)
                    colision = avanzar (c)
                    if (colision)
                    calcularPenalizaciones ()
    avanzar (cata)
        colision = false
        delta = cata.calcularDelta (cata.posicion, cata.direccion, estado.viento, estado.corrientes)
        cata.posicion = cata.posicion + delta
        foreach Cata c in estado.flota
            if (c.posicion == cata.posicion)
                colision = true
        return colision
    end
```

Figura 8.10: Avanzar Estado.

```
aplicarOperacion
    aplicarOperacion (cliente, op )
        avanzar (estado.flota.get(cliente.identificador), op)
        sincronizar ()
    avanzar (c, op)
        c.posicion = c.posicion + op.posicion
        c.direccion = c.direccion + op.direccion
    end
```

Figura 8.11: Aplicar una operación a un cliente.

el proceso servidor envía los datos pertenecientes a los barcos participantes, para obtener una imagen similar en todos los nodos que ejecuten la regata.

```
sincronizar
    sincronizar ()
        foreach barco in estado.flotas
            mensaje += barco.datos
        foreach cliente in listaClientes
            enviar (cliente, mensaje)
        if (flag_master)
            foreach servidor in grupoServidores
                enviar (servidor, mensaje)
    end
```

Figura 8.12: Sincronización de los delegados y de los clientes.

Adicionalmente, los servidores delegados envían las variaciones en su estado de los clientes que gestionan ellos mismos a todos los servidores, para ofrecer una visión lo más coherente posible, tal como muestra la figura 8.13.

```
sincronizar
    sincronizar ()
        foreach barco in estado.flotas
            mensaje += barco.datos
        foreach cliente in listaClientes
            enviar (cliente, mensaje)
        if (flag_master)
            foreach servidor in grupoServidores
                enviar (servidor, mensaje)
    end
```

Figura 8.13: Envío de deltas a los servidores.

8.3. Tolerancia a fallos.

La introducción de un modelo de servidores con la funcionalidad completamente replicada es un factor determinante a la hora de aportar tolerancia a fallos. Al contrario que ocurría en los modelos descritos en los anteriores capítulos, en el modelo de servidores replicados no existe un servidor con una funcionalidad especial imprescindible para el desarrollo de la simulación, sino que todos comparten dicha funcionalidad, por lo que, en caso de fallo, si un servidor cae, los clientes que tiene conectados pueden reconectarse a cualquiera de los servidores restantes existentes en la regata, por lo que, a no ser que fallen todos los servidores, lo que sin duda es más difícil que el fallo de un único servidor central, la simulación continuará su transcurso normal.

No obstante, aunque los servidores comparten funcionalidad, existe un servidor que actuará a modo de coordinador, sincronizando el estado de los diferentes servidores existentes en el sistema. Este nodo puede ser elegido mediante cualquier criterio de elección de líder y, en caso de fallo del nodo coordinador, al pertenecer los servidores a un mismo grupo de coordinaciones, obtendrían un cambio de vista indicando que el coordinador ha caído, por lo que se reiniciaría el proceso de elección de líder, obteniéndose un nuevo coordinador y, por tanto, prosiguiendo el transcurso normal de la regata.

8.4. Modelo de replicación seguido

La intrducción de múltiples réplicas del servidor con completa funcionalidad favorece el hecho de delegar un máximo número posible de operaciones de los clientes en los diferentes servidores que integran el sistema. De este modo, a la hora de propagar los cambios de estado cualquier servidor, empleará un modelo de coordinador-cohorte en dicha labor, o lo que es lo mismo, cada servidor tratará como réplica activa las operaciones de los clientes que tiene conectados, difundiendo los cambios al resto de servidores, y recibirá los cambios de estado y los aplicará en caso de que provengan de un cliente de cualquier otro servidor. Serán servidores activos para sus clientes y pasivos para el resto.

8.5. Modelo de consistencia de datos

Dado que se mantienen las mismas garantías que en el modelo de consistencia de datos de los servidores con reparto de carga, se sigue el mismo modelo de consistencia que en el capítulo anterior.

8.6. Resultados de la simulación

Como se ha realizado en los anteriores análisis, se han hecho mediciones de pruebas de carga baja, media y alta del modelo propuesto. No obstante, como se ha explicado en la anterior sección, se emplea un modelo cohordinador-cohorte para la aplicación de operaciones y propagación de estados, lo que da fruto a dos diferentes análisis de resultados: uno para los clientes conectados al nodo coordinador y otro para los clientes conectados a los nodos cohorte del modelo.

La distribución que se ha hecho es la siguiente: se han creado 3 servidores replicados, uno de ellos coordinador. Asímismo, se han repartido los clientes de forma equitativa entre los diferentes servidores por lo que, si hay un número N de clientes, cada servidor tendrá $N/3$ clientes.

8.6.1. Análisis de los clientes asociados al nodo coordinador

El análisis de los clientes asociados al nodo coordinador plasma el comportamiento del sistema para los clientes cuyo servidor es el que realiza las operaciones y luego las trasmite al resto de servidores. este análisis se ha realizado para una carga baja de 10 clientes, lo que asocia a dicho servidor 3 clientes; una carga media de 20 clientes, lo que le asocia 6 clientes; y una carga máxima de 32 clientes, con 11 clientes asociados. Para todos los casos se han realizado 10 rondas de mensajes con el fin de obtener los retardos medios de los mensajes desde el servidor a los clientes.

En el modelo de carga baja, como se ha mencionado, se poseen 3 clientes asociados directamente al servidor, y, tal como se desprende de la gráfica 8.14, los retardos oscilan entre los 17 y los 19 milisegundos, es decir, casi inmediatos. Tras realizar un análisis de confianza mostrado en la figura 8.15, se extrae un retardo medio de mensaje de 18 milisegundos, el cual es mucho menor a cualquiera de los retardos obtenidos anteriormente, y representa un 11'41 % del retardo original para este caso en el modelo centralizado.

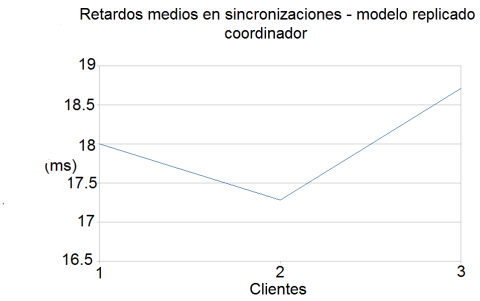


Figura 8.14: 10 clientes en modelo replicado - coordinador.

En las mismas condiciones de simulación, se realiza el experimento para un modelo de carga medio de 20 clientes, lo que asigna al nodo que procesará las operaciones un total de 6 clientes directos, de los cuales se extrae el retardo medio de cada uno de ellos, mostrados en la figura 8.16, que oscilan entre los 18 y 29 milisegundos. Extrayendo resultados anómalos por medio de un análisis de confianza, plasmado en la figura 8.17, se obtiene un retardo medio de 23'34 milisegundos, que supone un 14'29 % del coste original obtenido en el modelo centralizado para el mismo análisis.

Finalmente, se realiza el experimento para la carga máxima original del sistema de 32 clientes, siendo 11 de ellos directamente asociados con el nodo activo del sistema. Los retardos medios obtenidos por cada cliente, plasmados en la figura 8.18, oscilan entre los 22 y los 38 milisegundos. Tras extraer

Figura 8.15: Intervalo de confianza de las mediciones obtenidas.

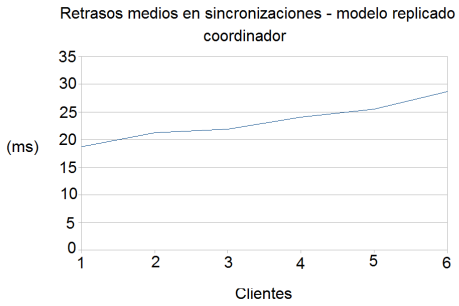


Figura 8.16: 20 clientes en modelo replicado - coordinador.

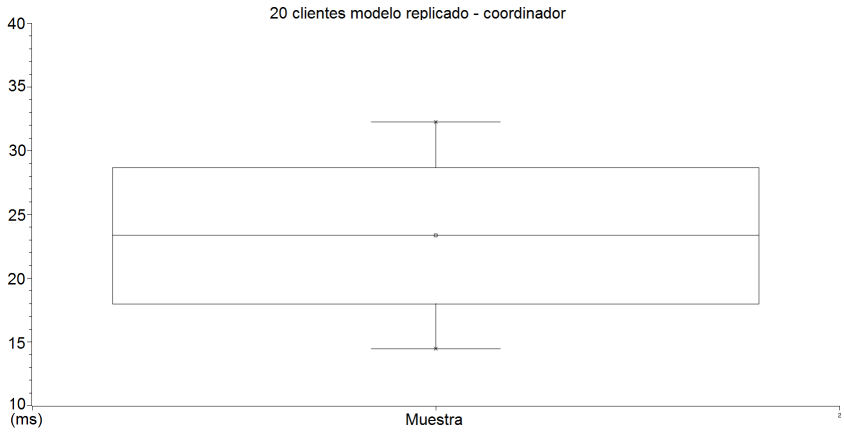


Figura 8.17: Intervalo de confianza de las mediciones obtenidas.

de la muestra los datos que no se corresponden al funcinamiento normal de la aplicación, mediante el diagrama expuesto en la figura 8.19, se obtiene un retardo medio de 32'25 milisegundos, lo que representa un 19'80 % del coste inicial obtenido en el modelo centralizado.

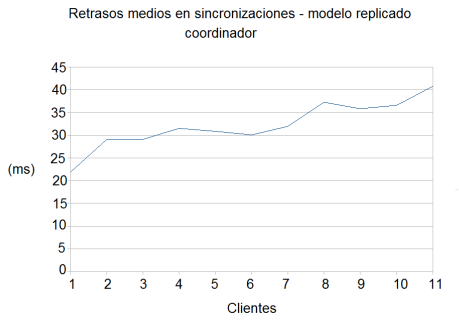


Figura 8.18: 32 clientes en modelo replicado - coordinador.

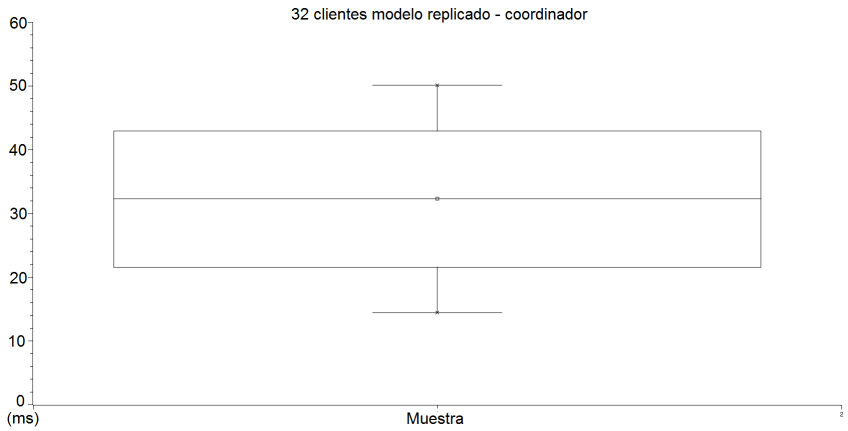


Figura 8.19: Intervalo de confianza de las mediciones obtenidas.

8.6.2. Análisis de los clientes asociados a los nodos cohorte

Una vez analizado el comportamiento del sistema ante los nodos conectados directamente al servidor que procesa las operaciones, queda comprobar el comportamiento bajo las mismas circunstancias que presentan el resto de clientes para cada uno de los anteriores experimentos. Dicho de otro modo, en este apartado se procede a estudiar los retardos de los mensajes para los clientes conectados a la cohorte del sistema. Nótese que estos resultados se desprenden de los mismos experimentos que los presentados en la anterior

sección, solo que bajo otro punto de vista.

Como se ha realizado anteriormente, el primer experimento es para una carga baja del sistema de 10 clientes, de los cuales 7 se corresponden a los asociados a los servidores que conforman la cohorte del sistema. Cabe señalar que las mediciones se han extraído de las mismas 10 rondas de mensajes del caso coordinador. Como se plasma en la figura 8.20, los retrasos en los mensajes oscilan entre los 113 y los 120 milisegundos. Tras extraer posibles datos anómalos mediante los intervalos de confianza mostrados en la figura 8.21, se calcula un retardo medio por mensaje de 117'19 milisegundos, que representa un 74'31 % del coste en el modelo centralizado.

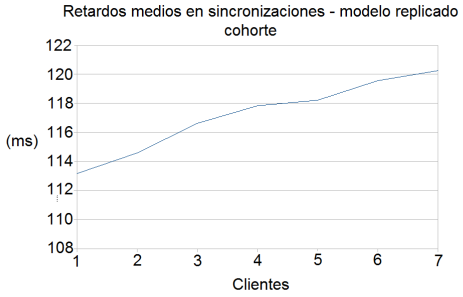


Figura 8.20: 10 clientes en modelo replicado - cohorte.

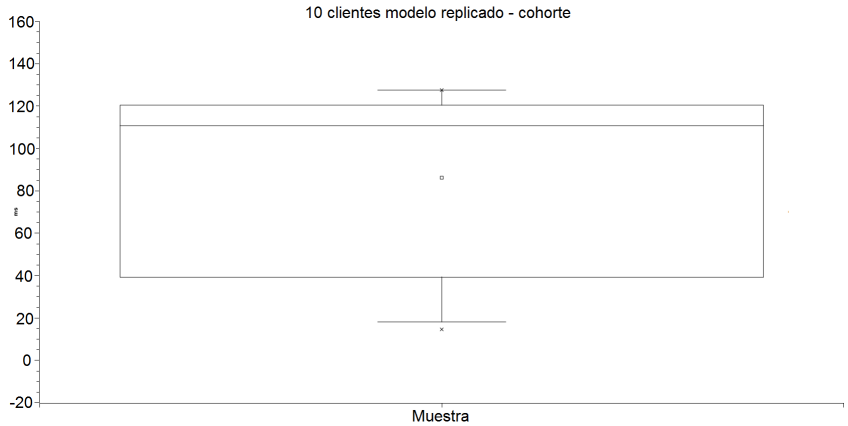


Figura 8.21: Intervalo de confianza de las mediciones obtenidas.

Por otro lado, manteniendo la dinámica seguida durante todos los análisis, se muestra el comportamiento para una carga media, que corresponde a

20 clientes, de los cuales 14 están conectados a nodos cohorte del sistema. Como muestra la figura 8.22, los retardos medios por cliente oscilan entre los 126 y 155 milisegundos lo que, tras realizar el correspondiente análisis de confianza de la muestra, plasmado en la figura 8.23, se extrae un retardo medio de 142'28 milisegundos, lo que representa un 87'15 % del retardo original obtenido en el modelo centralizado para este mismo experimento.

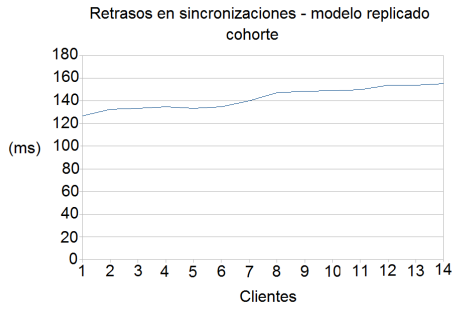


Figura 8.22: 20 clientes en modelo replicado - cohorte.

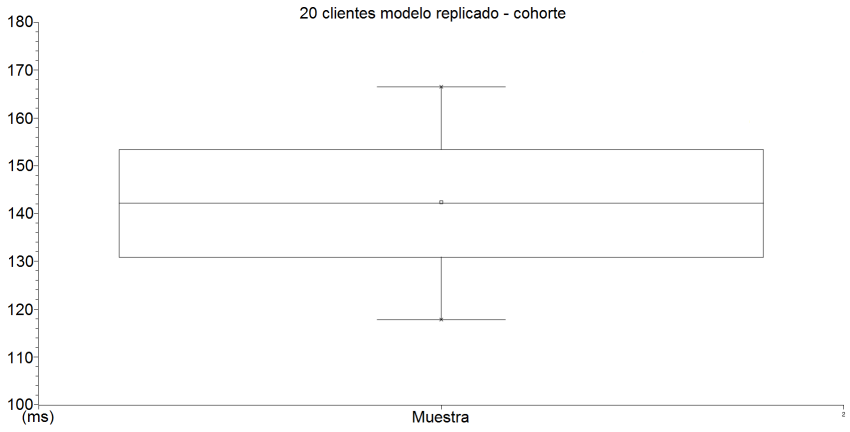


Figura 8.23: Intervalo de confianza de las mediciones obtenidas.

Finalmente, se realiza el correspondiente experimento para una carga máxima de clientes, 32, de los cuales 21 están conectados a los nodos cohorte del sistema. Para estos clientes se ha obtenido un retardo medio que oscila entre los 159 y los 187 milisegundos, tal como muestra la figura 8.24. Realizando el mismo proceso de extracción de datos de la muestra que sean anómalos, mostrado en la figura 8.25, se obtiene un retardo medio por mensaje de 172'92, ligeramente superior al obtenido en el modelo centralizado

(un 6'2 %), que puede venir determinado por la sobrecarga de la red que conecta los servidores.

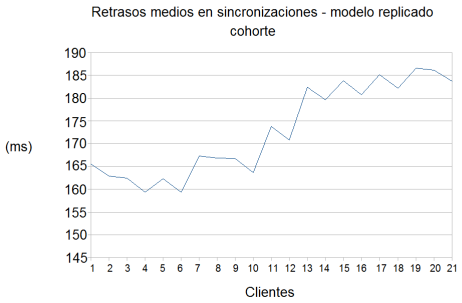


Figura 8.24: 32 clientes en modelo replicado - cohorte.

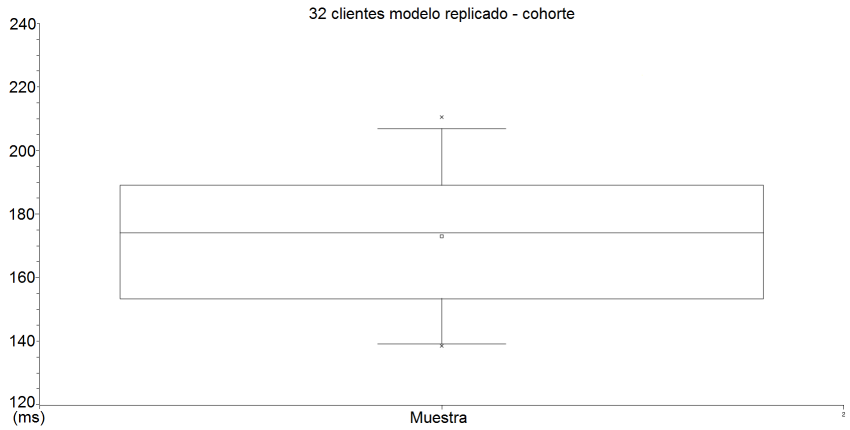


Figura 8.25: Intervalo de confianza de las mediciones obtenidas.

8.7. Conclusiones

8.7.1. Mejoras obtenidas respecto al modelo centralizado

Como se desprende de las simulaciones realizadas, al existir la lógica completa próxima a los clientes, los retardos obtenidos como respuesta de las operaciones que realiza un cliente contra su servidor oscilan entre los 17 y los 25 milisegundos, lo que resulta en una reducción más que considerable de los retardos originales, obteniendo una experiencia de simulación más realista sin 'saltos' producidos por los retrasos en los mensajes.

El hecho de que deje de existir un punto conflictivo en el sistema (un nodo vital para el funcionamiento de éste), favorece una mejor tolerancia a fallos, dado que para que el sistema deje de funcionar totalmente, se requiere que fallen todos los servidores existentes en el mismo.

8.7.2. Problemas que plantea

El hecho de tener que difundir cada cambio de estado a las réplicas que actúan como cohorte en el sistema por parte del servidor que actúa activamente, multiplica la cantidad de mensajes que se deben transmitir a través de la red que comunica los servidores, lo cual es un factor que limita la escalabilidad del sistema.

Asimismo, es necesario añadir cierta lógica adicional al sistema para controlar el grupo de comunicaciones al que pertenece un conjunto de servidores y para la ejecución de los algoritmos de elección de líder que se desee implantar, con el sobrecoste en número de mensajes correspondiente.

Además, el hecho de que haya que propagar las actualizaciones a todos los servidores va a ser casi tan costoso como en la versión centralizada, por lo que la escalabilidad difícilmente mejorará.

8.7.3. Líneas de desarrollo

Como trabajo futuro se plantea la optimización del sistema para que la cantidad de mensajes no crezca en tal magnitud que limite la escalabilidad del sistema, o al menos, que no la limite en la misma medida que lo realiza en el modelo estudiado. Asímismo, se propone la implantación de los modelos aquí estudiados en un entorno real, así como la implementación de los mismos para la aplicación para la que fueron diseñados y su estudio de prestaciones.

Capítulo 9

Conclusiones

La presente tesis describe el proceso seguido para una aplicación real, a la que se denomina BogReg, y cuyo nombre real es **TACTICAT**, correspondiente a un proyecto de investigación y desarrollo del Intituto Tecnológico de Informática de Valencia, en el grupo de Sistemas Distribuidos, con el propósito de reducir lor retardos en los mensajes desde el servidor a los diferentes clientes y aportar diferentes modelos de distribución a la aplicación. No obstante, tras los resultados obtenidos durante el proceso de investigación realizado para la confección de la tesis, se han abierto diversas vías de investigación para el departamento, de las cuales se extraerán publicaciones.

Tras el estudio de la aplicación así como de los diferentes modelos de distribución propuestos para la misma se presenta este capítulo, a modo de aglutinar las conclusiones a las que se ha llegado tras el proceso realizado durante la presente tesis.

Del correspondiente proceso que se ha realizado de refactorización de código en la aplicación, se puede extraer tres factores determinantes:

- En primer lugar, el hecho de haber reorganizado el código, separando y extrayendo los objetos que se encontraban internos a la clase del simulador, se obtiene una mayor legibilidad del código, principalmente, así como hace factible localizar código duplicado, modificar las relaciones entre objetos (un simulador ya no **es un** barco más viento, etc...), reutilizar partes del código y localizar zonas que no sean utilizadas.

- Por otro lado, el hecho de separar la aplicación servidora de la aplicación cliente, así como de sus diferentes funcionalidades, elimina la necesidad de simular una compilación condicional, aunque siguen existiendo partes del código que son comunes a ambas. Así pues, la

aplicación cliente englobará el paquete común y el paquete cliente, mientras que la aplicación servidora englobará el paquete común y el paquete servidor.

- Dentro de la separación de las aplicaciones, se han formado los paquetes según la funcionalidad, bien sea ésta de gráficos, bien de red o bien de gestión del estado. Esta separación permite tener ubicados los componentes de la aplicación según su funcionalidad, así como tenerlos aislados para que, en caso de que haya que realizar modificaciones, estén todas ubicadas en el mismo paquete.

Por otro lado, se han propuesto tres diferentes modelos de distribución, cada uno de ellos con sus correspondientes ventajas. Así pues, se tiene que del modelo basado en servidores delegados pasarela o repetidores:

- Al existir una red de mayor ancho de banda entre los diferentes servidores, y estar los diferentes servidores delegados más próximos a los clientes finales de la aplicación, se observa que tanto los tiempos obtenidos para los retrasos en la entrega de sincronizaciones como los retrasos de las peticiones enviadas por los clientes, retrasos de mensajes en general, se ven sustacialmente reducidos, por lo que, al tratarse de una simulación en tiempo real, se obtiene una mejor experiencia en la simulación de la regata. Esto es debido a la jerarquización de las difusiones.

- Por otro lado, la introducción de nuevos elementos en el sistema supone la introducción de nuevos puntos en los que el sistema puede fallar. No obstante, dado que un cliente puede reconectar con otro servidor delegado en tiempo de ejecución, la caída de uno de los servidores delegados únicamente supondría, como peor caso, la obtención de un tiempo de reconexión que se traduciría en un ligero "salto" en la simulación. Una vez superada la reconexión al sistema, los retrasos obtenidos por los clientes mencionados, obtendrían, como máximo, el retraso obtenido en el modelo centralizado, por lo que se afirma que en caso de fallo, en el peor de los casos, se obtiene la misma experiencia de simulación que en el modelo centralizado, siendo, en el resto de casos, una mejor experiencia de simulación debida a una reducción en los retrasos de los mensajes.

- No obstante, dada la naturaleza de la aplicación, el proceso de reconexión al sistema no resulta trivial. Por un lado, en el modo aplicación del cliente, la reconexión a otro servidor delegado se puede realizar en tiempo de ejecución pero, por otro lado, en el caso de un cliente en modo applet, debido a las restricciones propias de los applets, esa reconexión automática no es posible, por lo que debería ser

el usuario el que volviese a conectarse a la regata realizando todo el proceso de conexión inicial (busqueda por web, selección de regata, etc...)

Del modelo basado en delegados con reparto de carga también se extraen diferentes conclusiones:

- Por un lado, las transacciones que incluyan únicamente operaciones ligeras, se podrán responder en un tiempo inferior que al resto de casos, dado que no es necesaria la comunicación con el servidor maestro para su respuesta. No obstante, el servidor maestro deberá recibir cuentas de todas las operaciones realizadas por los servidores delegados, siempre y cuendo éstas supongan una modificación del estado de la simulación. Por tanto, se descarga de cierto número de peticiones al servidor maestro.

- Por otro lado, al existir una red de mayor ancho de banda entre los diferentes servidores, y estar los diferentes servidores delegados más próximos a los clientes finales de la aplicación, se observa que tanto los tiempos obtenidos para los retrasos en la entrega de sincronizaciones como los retrasos de las peticiones enviadas por los clientes, retrasos de mensajes en general, se ven sustacialmente reducidos, por lo que, al tratarse de una simulación en tiempo real, se obtiene una mejor experiencia en la simulación de la regata.

Y, finalmente, se muestran las conclusiones a las que se llega tras realizar el análisis del modelo basado en delegados replicas del servidor completo:

- Al existir la lógica completa próxima a los clientes, los retardos obtenidos como respuesta de las operaciones que realiza un cliente contra su servidor oscilan entre los 17 y los 25 milisegundos, lo que resulta en una reducción más que considerable de los retardos originales, obteniendo una experiencia de simulación más realista sin 'saltos' producidos por los retrasos en los mensajes.

- El hecho de que deje de existir un punto conflictivo en el sistema (un nodo vital para el funcionamiento de éste), favorece una mejor tolerancia a fallos, dado que para que el sistema deje de funcionar totalmente, se requiere que fallen todos los servidores existentes en el mismo.

Bibliografía

[BMST92] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Optimal primary-backup protocols. In *WDAG*, pages 362–378, 1992.

[CDK94] George Coulouris, Jean Dollmore, and Tim Kindberg. *Distributed Systems: Concepts and Design*. 1994.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. On the impossibility of group membership. In *PODC*, pages 322–330, 1996.

[CKV01] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[Cri91] Flaviu Cristian. Reaching agreement on processor-group memebership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[DSS98] Xavier Défago, André Schiper, and Nicole Sergent. Semi-passive replication. In *Symposium on Reliable Distributed Systems*, pages 43–50, 1998.

[Gär99] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, 1999.

[HT93] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In *Addison - Wesley*, pages 97–145, 1993.

[JFR93] Farnam Jahanian, Sameh A. Fakhouri, and Ragunathan Rajkumar. Processor group membership protocols: Specification, design, and implementation. In *Symposium on Reliable Distributed Systems*, pages 2–11, 1993.

[Jua94] Vicente Cholvi Juan. *Formalización de modelos de memoria.* 1994.

[Lam83] L. Lamport. The weak byzantine generals problem. *J. ACM*, 30(3):668–676, 1983.

[Mas98] Jordi Bataller Mascarell. *Aplicacions distribuïdes sobre memòria compartida: suport i anàlisi formal.* 1998.

[Nel90] Victor P. Nelson. Fault tolerant computing: Fundamental concepts. *IEEE Computer*, 1990.

[Nut97] Gary J. Nutt. *Operating Systems: A modern Perspective.* 1997.

[Pow94] David Powell. Distributed fault tolerance: Lessons from delta-4. In *IEEE Micreo*, pages 36–47, 1994.

[PWS+00] Fernando Pedone, Matthias Wiesmann, André Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In *ICDCS*, pages 464–474, 2000.

[Sch93] Fred B. Schneider. Replication management using the state-machine approach. *ACM Press, Addison-Wesley*, pages 166–197, 1993.

[SG98] Abraham Silberchatz and Peter Baer Galvin. *Operating System Concepts.* 1998.

[SS94] Mukesh Singhal and Niranjan G. Shivaratri. *Advanced Concepts in Operating Sustems: Distributed, Datbase and Multiprocessor Operating Systems.* 1994.

[TR85] Andrew S. Tanenbaum and Robbert Van Renesse. Distributed operating systems. *ACM Comput. Surv.*, 17(4):419–470, 1985.

[TS01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.