UNIVERSIDAD NACIONAL AUTÓNOMA DE NICARAGUA.
UNAN-LEÓN.

FACULTAD DE CIENCIAS.

DEPARTAMENTO DE COMPUTACIÓN.

APLICACIÓN DISTRIBUIDA: CHAT MULTIUSUARIO BAJO ENTORNO LINUX.

MONOGRAFÍA PARA OPTAR AL TÍTULO DE
INGENIERO EN SISTEMAS DE INFORMACIÓN.

Presentado por:

Br. Helder Saúl Flores Tercero.
Br. Valeria Mercedes Medina Rodríguez.

Tutor:

Msc. Aldo René Martínez.

León, Nicaragua, Mayo de 2006.

DEDICATORIA.

Dedico este trabajo monográfico:

A mis padres:
Porfirio Antonio Flores López y Feliciana Tercero por haberme dado la oportunidad de recibir una educación superior, por apoyarme incondicionalmente en las situaciones más difíciles a pesar de las necesidades, y sobre todo, por depositar en mí su fe y confianza, sin dudar en ningún momento que sería capaz de superar los obstáculos para alcanzar mis metas.

A mi compañera de trabajo Valeria Medina y su familia:
A quienes doy mi gratitud y cariño por haberme brindado la hospitalidad de su hogar y por tratarme como un miembro más de su familia.

Helder Saúl Flores Tercero.

DEDICATORIA.

Dedico este trabajo monográfico a:

Dios, por haberme iluminado y guiado siempre por el gozoso sendero de la sabiduría.

Mis padres Santiago Medina y María Jesús Rodríguez, que con su esfuerzo y abnegación, me han apoyado en todo los momentos de mi vida, han iluminado y guiado mis pasos y sobre todo han llenado mi vida de alegría y de amor.

A mi hermano Hermógenes Medina Rodríguez, su esposa Melba Centeno, por su apoyo incondicional y a mi sobrina Carol Melissa Medina, rayito de luz y alegría en nuestras vidas.

A mis hermanos María Magdalena, Salvadora y Leopoldo Medina Rodríguez, quienes me han apoyado y fortalecido en momentos difíciles y a mi sobrinito Geremy Ariel Medina, gotita de amor y de alegría en nuestra familia.

A mi compañero de monografía Helder Saúl Flores, por su empeño, sinceridad y calor fraternal que me ha brindado a lo largo del desarrollo de este trabajo.

A todo ellos muchas gracias por ser parte de este triunfo.

Valeria Mercedes Medina Rodríguez.

AGRADECIMIENTO.

Agradecemos:

A Dios: Por darnos vida, fuerzas, esperanza e iluminarnos en cada momento de nuestra existencia.

A nuestras familias: Por su empeño y abnegación, en ayudarnos a alcanzar uno de nuestros más grandes anhelos, coronar la carrera de Ingeniería en Sistemas de Información.

A Msc. Aldo René Martínez: Por su tiempo y apoyo en la elaboración de este trabajo.

A Lic. Eduardo Santiago Molina Poveda: Por su ayuda incondicional en el transcurso del desarrollo de este trabajo y por habernos transmitido sus conocimientos sobre la programación con Llamadas a Procedimientos Remotos.

A todo el personal docente del Departamento de Computación: Por su abnegada labor, que con mucho empeño supieron transmitirnos sus conocimientos.

Queremos también, dar gracias a todas aquellas personas que de alguna manera nos apoyaron en el período de elaboración de este trabajo.

ÍNDICE.

ÍNDICE DE FIGURAS.

I. INTRODUCCIÓN.

La computación desde sus inicios ha sufrido muchos cambios, comenzando por los grandes ordenadores que permitían realizar tareas en forma limitada y de uso un tanto exclusivo de organizaciones muy selectas, hasta los actuales ordenadores, ya sean personales o portátiles, que tienen las mismas e incluso mayores capacidades que los primeros y que están cada vez más introducidos en el que hacer cotidiano de una persona.

Los mayores cambios se atribuyen principalmente a dos causas, que se dieron desde la década de los setenta:

1. El desarrollo de los microprocesadores, que permitieron reducir en tamaño y costo a los ordenadores y aumentar en gran medida las capacidades de los mismos y su acceso a más personas.
2. El desarrollo de las **redes de área local** y de las comunicaciones, que permitieron conectar ordenadores con posibilidad de transferencia de datos a alta velocidad.

Es en este contexto que aparece el concepto de "Sistemas Distribuidos" popularizándose en la actualidad y que tiene como ámbito de estudio las redes, como por ejemplo: Internet, redes de teléfonos móviles, redes corporativas, redes de empresas, etc.

Por tanto, se decidió diseñar un Chat Multiusuario que evidencie la importancia de los Sistemas Operativos Distribuidos, haciendo énfasis en el proceso de comunicación de los mismos, y más específicamente el modelo de comunicación **Cliente-Servidor** utilizando llamadas a procedimientos remotos (**RPC**).

II. ANTECEDENTES.

En los trabajos monográficos realizados en el Departamento de Computación, no se han realizado aplicaciones sobre Sistemas Operativos Distribuidos y más específicamente la comunicación en este tipo de sistemas.

El presente trabajo tiene como antecedente principal el "**Proyecto de Fin de Carrera: Plan docente para la asignatura de Sistemas Operativos II**", elaborado por el Lic. Eduardo Santiago Molina Poveda, en el año 2002, en la Universidad de Alcalá (Escuela Politécnica). Dicho proyecto es un estudio complementario a la titulación de Licenciatura en Computación de la UNAN – León, donde se desarrolló una Aplicación de Chat Multiusuario con RPC, pero con la limitante de que éste no contempla la transmisión de ficheros de imágenes y audio, y no cuenta con una interfaz gráfica de usuario.

Otro antecedente de Chat Multiusuario, aunque desarrollado en el lenguaje Java y programado con Sockets, es el trabajo monográfico: "**Chat interactivo desarrollado en Java para el Departamento de Computación**", realizado por los bachilleres Javier Enrique Hernández Sandoval, Jairo Isaac Herrera Mora y Edwin Alberto Penado Rodríguez, este software contempla una interfaz gráfica.

El diseño de la interfaz gráfica de usuario con el diseñador de interfaces (Diseñador Qt) y el uso del IDE (Integrated Develop Enviroment) para desarrollo en KDE (KDevelop: KDE/C++), no tiene ningún antecedente en las investigaciones y trabajos monográficos realizados en el Departamento de Computación.

III. JUSTIFICACIÓN.

Debido al auge que actualmente han cobrado las comunicaciones y más específicamente **las redes LAN,** se considera necesario contar con diferentes modos de comunicación en el ambiente informático.

Se ha observado que en el Departamento de Computación no se ha puesto énfasis en el desarrollo de software relacionados con Sistemas Operativos Distribuidos, por lo que es de suma importancia realizar aplicaciones basadas en dichos sistemas, pues actualmente están cobrando gran auge en la industria y en el desarrollo de las telecomunicaciones, lo que servirá como base para futuros trabajos en el Departamento o para desarrollos propios que se pudieran hacer bajo el entorno de los Sistemas Distribuidos usando o no RPC.

Debido a las necesidades de los seres humanos de comunicarse a través de un equipo de Cómputo, se desarrollará un software que permita dicha comunicación, basados en el modelo de comunicación Cliente Servidor vía RPC.

Toda aplicación moderna deberá contar al menos, con una pequeña interfaz gráfica de usuario, que haga que el usuario final de la aplicación se sienta cómodo al utilizar el sistema. En este trabajo hemos decidido usar para la creación de la interfaz gráfica de usuario, **Qt Designer** (Diseñador Qt). Esto, debido a que es una herramienta muy potente que permite diseñar de una forma muy sencilla y rápida, ventanas de diálogos con las **librerías Qt**. Para la codificación de los procedimientos y funciones se utilizará **KDevelop**, ya que está pensado para que la tarea de programar aplicaciones de interfaces gráficas con Qt se simplifique; KDevelop viene con un Editor de Diálogo que se encarga de generar todo el código necesario para mostrar dichos componentes. Tanto Qt Designer como las librerías Qt y KDevelop, se distribuyen bajo licencia **GPL** (General Public License) lo que significa que podemos usarlo y distribuirlo con total libertad y disponemos además de su código fuente.

IV. OBJETIVOS.

1. Objetivos generales.

- Presentar al lector nociones básicas sobre los Sistemas Operativos Distribuidos y los modelos de comunicación empleados en dichos sistemas.

- Aplicar el modelo de comunicación Cliente-Servidor para diseñar un Servidor de Chat Multiusuario.

2. Objetivos específicos.

- Introducir al lector al entorno de los Sistemas Operativos Distribuidos, resaltando las ventajas que estos poseen con relación a los Sistemas Centralizados.

- Mostrar el funcionamiento del modelo de comunicación Cliente-Servidor vía llamadas a procedimientos remotos (RPC).

- Permitir la transferencia de archivos de imágenes en formato .jpeg y .jpg y archivos de audio con formato. mp3.

- Desarrollar una interfaz gráfica de usuario para la aplicación Cliente con KDevelop y Qt Designer, que sea fácil de manejar para el usuario final.

- Utilizar XDR (eXternal Data Representation, estándar para la descripción y representación de datos, que permite la transferencia de los mismos entre máquinas independientes), como Lenguaje de Definición de Interfaces (IDL) para la creación de interfaces de las rutinas de servicio y la representación externa de datos.

V. MARCO TEÓRICO.

V.I. Aspectos generales.

1. Historia de los Sistema Distribuido .

Cuando a finales de los años 50 los ordenadores empezaron a estar comercialmente disponibles, los pocos que había resultaban caros y no se aprovechaban bien. Los **grandes ordenadores centralizados** (años 60 y 70) resolvieron el problema del aprovechamiento. Aún así, pocas empresas podían disponer de ellos. Pero a mediados de los años 80, surgió el desarrollo de potentes microprocesadores y la aparición del PC, **un ordenador personal**. Si bien éste era de precio asequible, el problema de los PC's era la carencia o dificultad para disponer de recursos como: impresoras, scanner, procesadores, sistemas de almacenamiento, etc., pues éstos seguían siendo caros.

La solución vino de la mano de las redes de alta velocidad, que permitieron la interconexión de todo tipo de ordenadores mediante una red de comunicaciones, lo cual permitió compartir y aprovechar recursos. Con la unión de los ordenadores personales y las LAN nacieron los **Sistemas en Red**. Los usuarios de estos sistemas pueden hacer uso de los recursos que les ofrece la red de ordenadores, son conscientes de la multiplicidad de máquinas en la red y necesitan conocerlas, pues para acceder a cada recurso deben saber en qué máquina está ubicado tal recurso, y deben conectarse explícitamente a dicha máquina.

El siguiente paso fue el de los **Sistemas Distribuidos**. Con estos sistemas, los usuarios pueden saber que hay multiplicidad de estaciones y equipos en la red, pero no necesitan conocerlos explícitamente, ellos **solamente saben que hay recursos en la red y conocen su nombre o identificador**, de tal forma que pueden acceder a ellos de igual manera que acceden a los recursos locales (por su nombre), sin tener que conectarse en ningún caso a la máquina propietaria para utilizar el recurso.

Como podemos ver, la diferencia fundamental entre los sistemas en red y los sistemas distribuidos es la **transparencia**. **En el Sistema Distribuido, la composición y estructura de la red le pasa desapercibida al usuario, es decir ni la ve, ni le importa. Solamente le importan los recursos disponibles sin tener en cuenta en que máquina están ubicados**.

1.1. Concepto de Sistema Distribuido.

Aunque el concepto ya tiene bastantes años, todavía está en completo desarrollo, y hay distintas ideas y opiniones sobre lo que debe ser un Sistema Distribuido. Una de las definiciones más importantes es: **Se trata de un conjunto de ordenadores independientes e intercomunicados por una Red y provistos de software distribuido, tal que el usuario lo perciba como si se tratara de un único ordenador**.

El objetivo primordial de los Sistemas Operativos Distribuidos es el compartimiento fácil y eficiente de los recursos entre múltiples usuarios.

1.2. Características de los Sistemas Operativos Distribuidos.

1. **Concurrencia.**
   Esta característica de los Sistemas Distribuidos permite que los recursos disponibles en la red puedan ser utilizados simultáneamente por los usuarios y/o agentes que interactúan en la red.
2. **Carencia de reloj global.**
   Las coordinaciones para la transferencia de mensajes entre los diferentes componentes para la realización de una tarea, no tienen una temporización general, está más bien distribuida a los componentes.
3. **Fallos independientes de los componentes.**
   Cada componente del sistema puede fallar independientemente, con lo cual los demás pueden continuar ejecutando sus acciones. Esto permite el logro de las tareas con mayor efectividad, pues el sistema en su conjunto continua trabajando.

2. Ventajas de los Sistemas Operativos Distribuidos.

2.1. Con respecto a Sistemas Centralizados.

a. **Economía:** Una de las ventajas de los Sistemas Distribuidos es la **economía**, pues es mucho más barato añadir servidores y clientes cuando se requiere aumentar la potencia de procesamiento.
b. **Velocidad:** Relacionado con el punto anterior, la velocidad sumada es muy superior.
c. Tienen una **mayor confiabilidad:** Al estar distribuida la carga de trabajo en muchas máquinas, la falla de una de ellas no afecta a las demás; el sistema sobrevive como un todo. Por lo que soporta mayor tolerancia a fallo.

d. **Escalabilidad:** Capacidad de **crecimiento incremental**. Se puede añadir procesadores al sistema incrementando su potencia en forma gradual según sus necesidades.
e. **Distribución:** Algunas aplicaciones requieren por sí, una distribución física.

2.2. Con respecto a PCs Independientes.

a. **Compartir datos:** Un Sistema Distribuido permite compartir datos más fácilmente que los sistemas aislados, que tendrían que duplicarlos en cada nodo para lograrlo.
b. **Compartir dispositivos:** Un Sistema Distribuido permite acceder a dispositivos desde cualquier nodo en forma transparente, lo cual es imposible con los sistemas aislados. Dichos dispositivos o recursos pueden ser: Impresoras, dispositivos de almacenamiento masivo, etc. Al compartir recursos, satisfacen las necesidades de muchos usuarios a la vez. Ejemplo: Sistemas de reservas de aerolíneas.
c. **Comunicaciones:** La comunicación persona a persona es factible en los Sistemas Distribuidos, en los sistemas aislados no. Ejemplo: el correo electrónico.
d. **Flexibilidad:** La distribución de la carga de trabajo, es factible en el Sistema Distribuido, se puede incrementar el poder de cómputo.

2.3. Desventajas de los Sistemas Distribuidos.

a. El principal problema es el software, el diseño, la implantación y uso del software distribuido, pues presenta numerosos inconvenientes. Los principales interrogantes son los siguientes:
    - ¿Qué tipo de S. O., lenguaje de programación y aplicaciones son adecuados para estos sistemas?
    - ¿Cuánto deben saber los usuarios de la distribución?
    - ¿Qué tanto debe hacer el sistema y qué tanto deben hacer los usuarios?
b. Otro problema tiene que ver con las redes de comunicación. Por ejemplo: pérdida de mensajes, saturación en el tráfico, etc.
c. Un problema que puede surgir al compartir datos es la seguridad de los mismos, pues si se consigue el acceso a red (de forma ilegal), se consigue ya un acceso sencillo a una gran cantidad de información, lo cual requiere establecer unos mecanismos de acceso muy seguro.

En general se considera que las ventajas superan a las desventajas, si estas últimas se administran seriamente.

3. Conceptos de Hardware.

Todos los Sistemas Distribuidos constan de varias CPUs, organizadas de diversas formas, especialmente respecto a: la forma de interconectarlas entre sí y los esquemas de comunicación utilizados.

Existen diversos esquemas de comunicación para los sistemas de cómputos con varias CPUs:
Uno de los más conocidos es la "**Taxonomía de Flynn**" en honor a su autor Michael Flynn, profesor de ingeniería eléctrica en la de la Universidad de Stanford:

- Esta taxonomía es ampliamente conocida y aceptada. Describe las computadoras por cómo los flujos de instrucciones interactúan con los flujos de información.
- La clasificación incluye equipos **SISD**, **SIMD**, **MISD y MIMD**.

Se define como flujo de instrucciones al conjunto de instrucciones secuénciales que son ejecutadas por un único procesador y como flujo de datos, al flujo secuencial de datos requeridos por el flujo de instrucciones.

Tabla 1 Taxonomía de Flynn.

	Número de Flujos de Información.		
Número de Flujos de Instrucciones.		Uno solo	Múltiples
	Uno solo	SISD	SIMD
	Múltiples	MISD	MIMD

3.1. SISD. (Single Instruction Single Data).

Un flujo de instrucciones y un flujo de datos. Se refiere a las computadoras convencionales de Von Neuman (Ver Fig. 1). Ejemplo: PC's. En la categoría SISD están la gran mayoría de las computadoras existentes. Son equipos con un solo procesador que trabaja sobre un solo dato a la vez, es decir, se ejecuta una instrucción detrás de otra, donde, en cualquier momento, sólo se ejecuta una única instrucción.

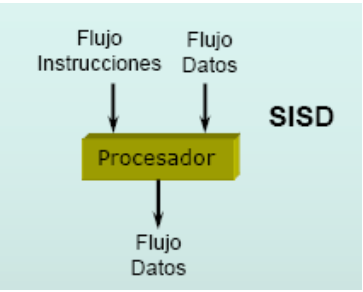


Fig. 1 SISD.

3.2. SIMD. (Single Instruction Multiple Data).

Un flujo de instrucciones y varios flujos de datos. Arreglo de procesadores (Ver Fig. 2). Cada procesador sigue el mismo conjunto de instrucciones; diferentes elementos de información son asignados a cada procesador. Utilizan memoria distribuida. Típicamente tienen miles de procesadores simples. Son utilizadas en redes neuronales.

Las computadoras SIMD tienen una sola unidad de control y múltiples unidades funcionales. La unidad de control se encarga de enviar la misma instrucción a todas las unidades funcionales. **Cada unidad funcional trabaja sobre datos diferentes**. Estos equipos son de propósito específico, es decir, son apropiados para ciertas aplicaciones particulares, como por ejemplo el procesamiento de imágenes.

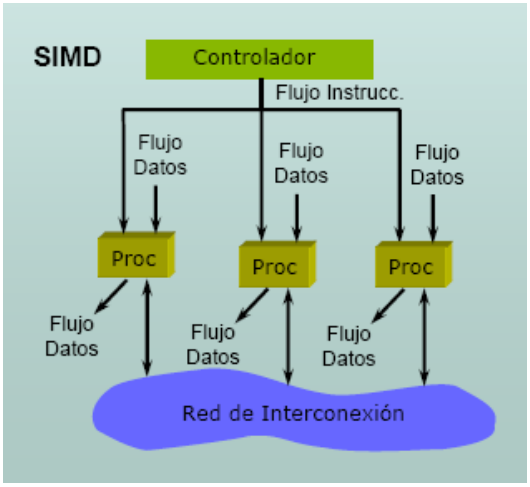


Fig. 2 SIMD.

3.3. MISD. (Multiple Instruction Single Data).

Un flujo de varias instrucciones y un solo flujo de datos (Ver Fig. 3). No son usadas, y no son significativas. Existe controversia acerca de si realmente existen equipos de tipo MISD. Hay quienes argumentan que estos equipos no existen. Otras personas consideran que un grupo de equipos que trabaja sobre un solo dato se puede considerar como un sistema de tipo MISD.

Un ejemplo sería un conjunto de equipos que trata de factorizar un número primo muy grande utilizando diferentes algoritmos.

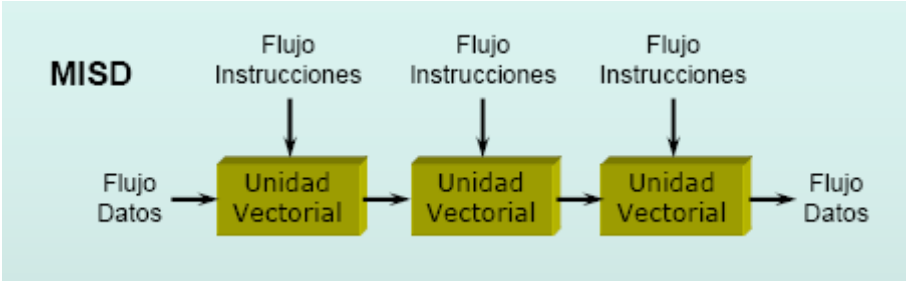


Fig. 3 MISD.

3.4. MIMD. ((Multiple Instruction Multiple Data).

Un grupo de computadoras independientes, cada una con su propio contador del programa y datos. Múltiples computadoras y multiprocesadores (Ver Fig. 4). Los procesadores pueden ejecutar la misma instrucción o diferentes instrucciones. Se puede decir que MIMD es un súper conjunto de SIMD.

Diferentes elementos de información se asignan a diferentes procesadores. Pueden tener memoria distribuida o compartida. Cada procesador MIMD corre casi independientemente de los otros. Las computadoras MIMD pueden ser utilizadas en aplicaciones con información en paralelo o con tareas en paralelo.

En la categoría MIMD están los equipos con varios procesadores completos. Cada procesador tiene su propia unidad de control y su propia unidad funcional. Esta categoría incluye varios subgrupos: equipos de memoria compartida, equipos de memoria distribuida y redes de computadores. Los equipos MIMD son de propósito general. Todos los Sistemas Distribuidos son de este tipo.

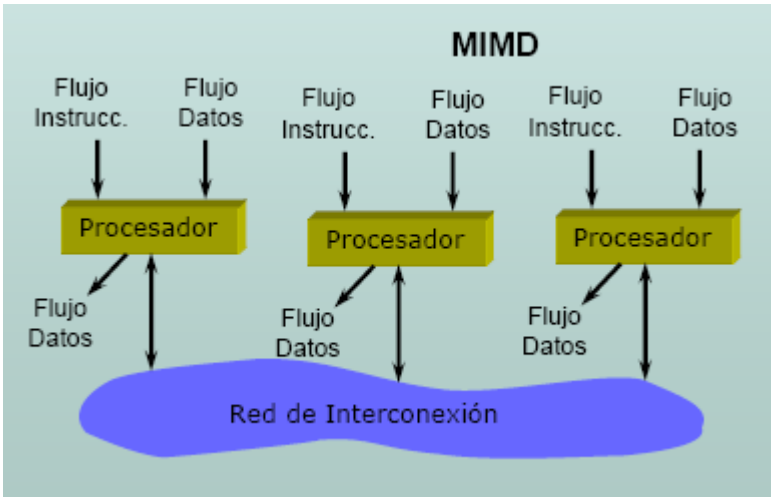


Fig. 4 MIMD.

Un avance sobre la clasificación de Flynn incluye la división de las computadoras MIMD en dos grupos:

a. Multiprocesadores: poseen memoria compartida.

Un multiprocesador puede verse como un computador paralelo (clasificación moderna que hace alusión única y exclusivamente a los sistemas que tienen más de un procesador) compuesto por varios procesadores interconectados que comparten un mismo sistema de memoria.

Los sistemas multiprocesadores son arquitecturas MIMD con memoria compartida. Tienen un único espacio de direcciones para todos los procesadores y los mecanismos de comunicación se basan en el paso de mensajes desde el punto de vista del programador.

Dado que los multiprocesadores comparten diferentes módulos de memoria, pudiendo acceder a un mismo módulo varios procesadores, a los multiprocesadores también se les llama sistemas de memoria compartida.

Los multiprocesadores son sistemas fuertemente acoplados (tightly-coupled), dado el alto grado de compartición de los recursos (hardware o software) y el alto nivel de interacción entre procesadores, lo que hace que un procesador dependa de lo que hace otro.

b. Multicomputadoras: no poseen memoria compartida.

Los sistemas multicomputadores se pueden ver como un computador paralelo en el cual cada procesador tiene su propia memoria local. En estos sistemas la memoria se encuentra distribuida y no compartida como en los sistemas multiprocesador. Los computadores se comunican a través de paso de mensajes, ya que éstos sólo tienen acceso directo a su memoria local y no a las memorias del resto de procesadores.

La transferencia de los datos se realiza a través de la red de interconexión (Ver Fig. 5) que conecta un subconjunto de procesadores con otro subconjunto. La transferencia de unos procesadores a otros se realiza por tanto, por múltiples transferencias entre procesadores conectados dependiendo del establecimiento de dicha red.

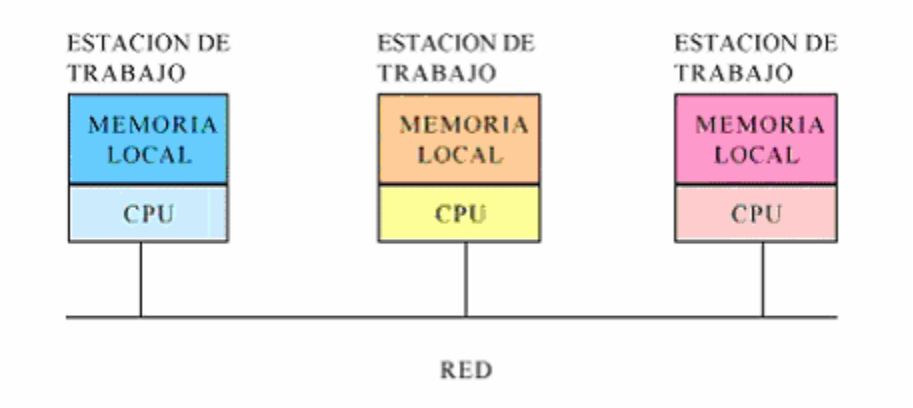


Fig. 5 Multicomputadoras con Base en Bus.

Dado que la memoria está distribuida entre los diferentes elementos de proceso, estos sistemas reciben el nombre de distribuidos.

Por otra parte, estos sistemas son débilmente acoplados, ya que los módulos funcionan de forma casi independiente unos de otros.

Cada una de las categorías indicadas se puede clasificar según la arquitectura de la red de interconexión en:

a. **Esquema de bus:**
   - Existe una sola red, bus, cable u otro medio que conecta todas las máquinas: Ej.: la televisión por cable.

b. **Esquema con conmutador:**
   - No existe una sola columna vertebral de conexión:
     - Hay múltiples conexiones y varios patrones de conexionado.
     - Los mensajes se mueven a través de los medios de conexión.
     - Se decide explícitamente la conmutación en cada etapa para dirigir el mensaje a lo largo de uno de los cables de salida.
     - Ej.: el sistema mundial telefónico público.

Otro aspecto de la clasificación considera el acoplamiento entre los equipos:

a. **Sistemas fuertemente acoplados:**
   - El retraso al enviar un mensaje de una computadora a otra es corto y la tasa de transmisión es alta.
   - Generalmente se les utiliza como sistemas paralelos.

b. Sistemas débilmente acoplados:

- El retraso de los mensajes entre las máquinas es grande y la tasa de transmisión es baja.
- Generalmente se les utiliza como Sistemas Distribuidos.

Generalmente los multiprocesadores están más fuertemente acoplados que las multicomputadoras.

4. Conceptos de Software.

La importancia del software supera frecuentemente a la del hardware. La imagen que un sistema presenta queda determinada en gran medida por el software del S.O. y no por el hardware. Los S.O. no se pueden encasillar fácilmente, como el hardware, pero se los puede clasificar en dos tipos:

- Débilmente acoplados.
- Fuertemente acoplados.

El software débilmente acoplado de un Sistema Distribuido:

- Permite que las máquinas y usuarios sean independientes entre sí en lo fundamental.
- Facilita que interactúen en cierto grado cuando sea necesario. Por ejemplo: Consideremos una red LAN de ordenadores en la cual se comparten recursos (dispositivos e información). Si la red falla por alguna razón, las máquinas individuales continúan su ejecución en cierto grado, aunque se pueda perder cierta funcionalidad (p.e. capacidad de imprimir archivos).

En el otro extremo, podríamos tener el caso de un sistema multiprocesador en el que cada procesador, se encarga de tratar una parte de un programa muy grande. En este caso se habla de software fuertemente acoplado.

5. Principios de Diseño de un Sistema Operativo Distribuido.

Los aspectos claves en el diseño de S. O. distribuidos son:

5.1. Transparencia.

El concepto de transparencia de un Sistema Distribuido, va ligado a la idea de que todo el sistema funcione de forma similar en todos los puntos de la red, independientemente de la posición del usuario. Queda como labor del sistema operativo el establecer los mecanismos que oculten la

naturaleza distribuida del sistema y que permitan trabajar a los usuarios como si de un único equipo se tratara.

En un sistema transparente, las diferentes copias de un archivo deben aparecer al usuario como un único archivo. Queda como labor del sistema operativo el controlar las copias, actualizarlas en caso de modificación y en general, la unicidad de los recursos y el control de la concurrencia.

El que el sistema disponga de varios procesadores, debe lograr un mayor rendimiento del sistema, pero el sistema operativo debe controlar, que tanto los usuarios como los programadores vean el núcleo del Sistema Distribuido como un único procesador. El paralelismo es otro punto clave que debe controlar el sistema operativo, que debe distribuir las tareas entre los distintos procesadores como en un sistema multiprocesador, pero con la dificultad añadida de que ésta tarea hay que realizarla a través de varios ordenadores.

Se pretende ocultar al usuario los detalles de gestión de los recursos distribuidos:

- *De acceso*: El acceso a recursos locales y remotos debe realizarse de la misma forma.
- *De ubicación*: El usuario no tiene que conocer la ubicación exacta del recurso.
- *De migración*: El recurso podría ser llevado de una estación a otra, de forma transparente al usuario.
- *De replicación*: El sistema operativo puede sacar duplicados de un recurso en distintas estaciones, con total transparencia para el usuario. Es decir permite utilizar varios ejemplares de cada recurso.
- *De fallos*: Permite ocultar los fallos, pudiendo así los usuarios o aplicaciones completar sus tareas a pesar de los fallos de componente hardware o software.
- *De concurrencia*: El usuario no debe preocuparse por la compartición de recursos, el sistema operativo se ocupa de los accesos concurrentes.
- *De paralelismo*: El sistema operativo podría sacar provecho del paralelismo para separar la ejecución de un proceso en varios procesos o hilos.
- *De configuración*: Permite la re-configuración dinámica del sistema, para mejorar el rendimiento a medida que varía la carga de trabajo.
- *De escala*: Permite al sistema y a las aplicaciones expandirse en tamaño sin cambiar la estructura del sistema o los algoritmos de aplicación.

5.2. Flexibilidad.

Un proyecto en desarrollo, como es el diseño de un sistema operativo distribuido debe estar abierto a cambios y actualizaciones que mejoren el funcionamiento del sistema. Esta necesidad ha provocado una diferenciación entre las distintas arquitecturas del núcleo del sistema operativo: el núcleo monolítico y el micronúcleo. Las diferencias entre ambos son los servicios que ofrece el núcleo del sistema operativo. Mientras el núcleo monolítico ofrece todas las funciones básicas del sistema integradas en el núcleo, el micronúcleo incorpora solamente las fundamentales, que incluyen únicamente el control de los procesos y la comunicación entre ellos y la memoria. El resto de servicios se cargan dinámicamente a partir de servidores en el nivel de usuario.

- **Núcleo monolítico:**
  Cada máquina debe ejecutar un núcleo tradicional que proporcione la mayoría de los servicios. Como ejemplo de sistema operativo de núcleo monolítico está UNIX. Estos sistemas tienen un núcleo grande y complejo, que engloba todos los servicios del sistema. Está programado de forma no modular, y tiene un rendimiento mayor que un micronúcleo. Sin embargo, cualquier cambio a realizar en cualquier servicio requiere la parada de todo el sistema y la recompilación del núcleo.

- **Micronúcleo (microkernel):**
  La arquitectura de micronúcleo ofrece la alternativa al núcleo monolítico. Se basa en una programación altamente modular, y tiene un tamaño mucho menor que el núcleo monolítico. Como consecuencia, el refinamiento y el control de errores son más rápidos y sencillos. Además, la actualización de los servicios es más sencilla y ágil, ya que sólo es necesaria la recompilación del servicio y no de todo el núcleo. Como contraprestación, el rendimiento se ve afectado negativamente.

  En la actualidad la mayoría de sistemas operativos distribuidos en desarrollo tienden a un diseño de micronúcleo. Los núcleos tienden a contener menos errores y a ser más fáciles de implementar y de corregir. El sistema pierde ligeramente en rendimiento, pero a cambio consigue un gran aumento de la flexibilidad. Contrariamente al núcleo monolítico, el micronúcleo no proporciona el sistema de archivos, el sistema de directorios, toda la administración de procesos o gran parte del manejo de las llamadas al sistema. El objetivo es mantener el micronúcleo pequeño.

Con núcleo monolítico:

- La mayoría de las llamadas al sistema se realizan mediante señalamiento al núcleo:
  El núcleo realiza el trabajo.
  El núcleo regresa el resultado al proceso del usuario.
- La mayoría de las máquinas tienen discos y administran sus propios sistemas locales de archivos.

El micronúcleo es más flexible y proporciona solo cuatro servicios mínimos:

- Un mecanismo de comunicación entre procesos.
- Cierta administración de la memoria.
- Una cantidad limitada de planificación y administración de procesos de bajo nivel.
- Entrada/Salida de bajo nivel.

Todos los demás servicios del S.O. se implementan generalmente como servidores a nivel usuario:

- Para obtener un servicio:
  - El usuario envía un mensaje al servidor apropiado.
  - El servidor realiza el trabajo y regresa el resultado.

Una importante ventaja de este método es su alta modularidad:

- Existe una interfaz bien definida con cada servicio (conjunto de mensajes que comprende el servidor).
- Cada servicio es igual de accesible para todos los clientes, independientemente de la posición.

Es fácil implantar, instalar y depurar nuevos servicios, sin necesidad de detener el sistema totalmente.

5.3. Fiabilidad ó Confiabilidad.

Un importante objetivo de los Sistemas Distribuidos es que si una máquina falla, alguna otra debe encargarse del trabajo.

Una de las ventajas claras que nos ofrece la idea de Sistema Distribuido es que el funcionamiento de todo el sistema no debe estar ligado a ciertas máquinas de la red, sino que cualquier equipo pueda suplir a otro en caso de que uno se estropee o falle.

La forma más evidente de lograr la fiabilidad de todo el sistema está en la redundancia. La información no debe estar almacenada en un solo servidor de archivos, sino en por lo menos dos máquinas. Mediante la redundancia de los principales archivos o de todos, evitamos el caso de que el fallo de un servidor bloquee todo el sistema, al tener una copia idéntica de los archivos en otro equipo.

Otro tipo de redundancia más compleja se refiere a los procesos. Las tareas críticas podrían enviarse a varios procesadores independientes, de forma que el primer procesador realizaría la tarea normalmente, pero ésta pasaría a ejecutarse en otro procesador si el primero hubiera fallado.

Un aspecto de la confiabilidad es la disponibilidad, que se refiere a la fracción de tiempo en que se puede utilizar el sistema.

La disponibilidad se mejora mediante:

- Un diseño que no exija el funcionamiento simultáneo de un número sustancial de componentes críticos.
- La redundancia, es decir la duplicidad de componentes claves del hardware y del software.

Los datos no deben perderse o mezclarse y si los archivos se almacenan de manera redundante en varios servidores*, todas las copias deben ser consistentes*.

Otro aspecto de la confiabilidad general es la *seguridad*, lo que significa que los archivos y otros recursos deben ser protegidos contra el uso no autorizado.

Un aspecto también relacionado con la confiabilidad es la *tolerancia a fallas*, según la cual, las fallas se deben ocultar brindando una *recuperación transparente* para el usuario, aunque haya cierta degradación.

5.4. Rendimiento.

Cuando se ejecuta una aplicación en un Sistema Distribuido no debe parecer peor que su ejecución en un único procesador, pero esto es difícil de lograr.
Algunas métricas del desempeño son:

- Tiempo de respuesta.
- Rendimiento (número de trabajos por hora).
- Uso del sistema y cantidad consumida de la capacidad de la red.

El problema se complica por el hecho de que la comunicación entre equipos es lenta comparada con:

- La velocidad de proceso.
- La velocidad de la comunicación dentro de un mismo procesador.

Se requiere el uso de protocolos de comunicaciones en los extremos (procesadores) que intervienen en la comunicación, con lo que se incrementa el consumo de ciclos del procesador.

Para *optimizar el desempeño* frecuentemente hay que:

- Minimizar el número de mensajes:
  - La dificultad es que la mejor forma de mejorar el desempeño es tener muchas actividades en ejecución paralela en distintos procesadores, pero esto requiere el envío de muchos mensajes.
- Centralizar el trabajo en una sola máquina:
  - Resulta poco apropiado para un Sistema Distribuido.

5.5. Escalabilidad.

Un Sistema Operativo Distribuido debería funcionar tanto para una docena de ordenadores como varios millares. Igualmente, debería no ser determinante el tipo de red utilizada (LAN o WAN) ni las distancias entre los equipos, etc.

Aunque este punto sería muy deseable, puede que las soluciones válidas para unos cuantos ordenadores no sean aplicables para varios miles. Del mismo modo el tipo de red condiciona tremendamente el rendimiento del sistema, y puede que lo que funcione para un tipo de red, para otro requiera un nuevo diseño.

La escalabilidad propone que cualquier ordenador individual ha de ser capaz de trabajar independientemente como un Sistema Distribuido, pero también debe poder hacerlo conectado a muchas otras máquinas.

La tendencia indica que el tamaño de los Sistemas Distribuidos es hacia cientos de miles y aun decenas de millones de usuarios conectado.

Existen *cuellos de botella potenciales* que se debe intentar evitar en los Sistemas Distribuidos de gran escala:

- Componentes centralizados:
  - Ej.: un solo servidor de correo para todos los usuarios.
- Tablas centralizadas:
  - Ej.: un único directorio telefónico en línea.
- Algoritmos centralizados:
  - Ej.: realización de un ruteo con base en la información completa.

Se deben utilizar algoritmos descentralizados con las siguientes características:

- Ninguna máquina tiene la información completa acerca del estado del sistema.
- Las máquinas toman decisiones solo en base a la información disponible de manera local.
- El fallo de una máquina no arruina el algoritmo.

6. Aplicaciones de los Sistemas Operativos Distribuidos.

La influencia de los Sistemas Operativos Distribuidos enseguida se ha manifestado en diversas áreas de aplicación.

a. Sistemas Comerciales. Inicialmente fueron construidos con hardware dedicado y entornos centralizados, son, por sus características de distribución geográfica y necesidad de acceso a sistemas distintos, ideales para implementarse en Sistemas Distribuidos. Requieren ciertas características de fiabilidad, seguridad y protección. Algunos ejemplos son:

- Sistemas de reservas de líneas aéreas.
- Aplicaciones bancarias.
- Caja y gestión de grandes almacenes.

b. Redes WAN. Debido al gran crecimiento de este tipo de redes (Internet), ha tomado gran importancia el intercambio de información a través de la red. Y para esto tenemos los siguientes ejemplos:

- Correo electrónico.
- Servicio de noticias (NEWS).
- Servicio de transferencia de ficheros (FTP).
- Búsqueda de ficheros.
- La World Wide Web (WWW).

c. Aplicaciones Multimedia. Son las últimas incorporaciones a los Sistemas Distribuidos. Estas aplicaciones imponen ciertas necesidades de hardware para poder tener una

velocidad y regularidad de transferencia de una gran cantidad de datos. Los ejemplos de estos sistemas son:

- Videoconferencia.
- Televigilancia.
- Juegos multiusuarios.
- Enseñanza asistida por ordenador.

d. Áreas de la informática aplicadas a los Sistemas Distribuidos. En este punto se tienen en cuenta toda la variedad de aplicaciones de los Sistemas Distribuidos , pues su diseño involucra a muchas áreas, por ejemplo:

- Comunicaciones.
- Base de datos distribuidas.
- Servidores distribuidos de ficheros.
- Lenguajes de programación distribuidos.
- Sistemas de tolerancia de fallos.

7. Desafíos de los Sistemas Operativos Distribuidos.

Heterogeneidad de los componentes. La interconexión, sobre todo cuando se usa Internet, se da sobre una gran variedad de elementos hardware y software, por lo cual necesitan de ciertos estándares que permitan esta comunicación. Los Middleware, son elementos software que permiten una abstracción de la programación y el enmascaramiento de la heterogeneidad subyacente sobre las redes. También el Middleware proporciona un modelo computacional uniforme.

Extensibilidad. Determina si el sistema puede extenderse y ser reimplementado en diversos aspectos (añadir y quitar componentes). La integración de componentes escritos por diferentes programadores es un autentico reto.

Seguridad. Reviste gran importancia por el valor intrínseco para los usuarios. Tiene tres componentes:

- Confidencialidad.- Protección contra individuos no autorizados.
- Integridad.- Protección contra la alteración o corrupción.
- Disponibilidad.- Protección contra la interferencia con los procedimientos de acceso a los recursos.

Escalabilidad. El sistema es escalable si conserva su efectividad al ocurrir un incremento considerable en el número de recursos y en el número de usuarios.

Tratamiento de fallos. La posibilidad que tiene el sistema para seguir funcionando ante fallos de algún componente en forma independiente, pero para esto se tiene que tener alguna alternativa de solución. Técnicas para tratar fallos:

- Detección de fallos. Algunos fallos son detectables, con comprobaciones por ejemplo.
- Enmascaramiento de fallos. Algunos fallos detectados pueden ocultarse o atenuarse.
- Tolerancia de fallos. Sobre todo en Internet se dan muchos fallos y no es muy conveniente ocultarlos, es mejor tolerarlos y continuar.
- Recuperación frente a fallos. Tras un fallo se deberá tener la capacidad de volver a un estado anterior.
- Redundancia. Se puede usar para tolerar ciertos fallos (DNS, BD, etc.).

Concurrencia. Compartir recursos por parte de los clientes a la vez.

8. Componentes de un Sistema Distribuido.

El desarrollo de un Sistema Distribuido complejo requiere el uso de las siguientes funciones y servicios:

8.1. Servicio de Comunicación.

Los componentes de un Sistema Distribuido no solamente están separados lógicamente, sino físicamente, por lo que requieren líneas de comunicaciones para interaccionar.

Tipos de comunicación:

– Punto a punto (unicast).

Este tipo de comunicación se lleva a cabo entre un emisor y un único receptor (Ver Fig. 6).

Fig. 6 Comunicación unicast.

– Multipunto (multicast).

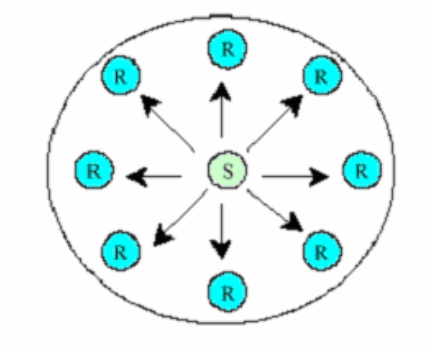


Fig. 7 Comunicación multicast.

El modelo de comunicación **multicast** consiste en el envío de un mismo mensaje desde un origen a un grupo de nodos destinos (Ver Fig. 7). No se debe confundir con **broadcast** (o difusión), que es el envío de un mensaje de forma que pueda ser escuchado por todos los nodos de la red, y que en Sistemas Distribuidos se utiliza con menor frecuencia.

Se trata de una comunicación **uno - muchos** (un emisor, muchos receptores), que se distingue de la comunicación puntual o **punto a punto** (un emisor, un receptor).

El principal modelo de interacción en un Sistema Operativo Distribuido es; **Modelo Cliente/servidor** (Ver Fig. 8).

Está orientado a la provisión de servicios. Una conversación típica consiste en:

- El proceso Cliente transmite una petición al proceso Servidor.
- El proceso Servidor ejecuta la petición solicitada.
- El proceso Servidor transmite la respuesta al Cliente.

Este modelo implica la transmisión de dos mensajes y alguna sincronización entre el Cliente y el Servidor. Es decir el proceso Servidor debe estar pendiente de la llegada de peticiones del Cliente, y a su recepción debe ejecutar el servicio solicitado y comunicar la respuesta. El proceso Cliente por su parte, una vez enviada la petición, debe esperar bloqueado hasta obtener una respuesta.

Se observan claramente roles diferentes en la interacción:

– Cliente: Solicita servicio.

– Servidor: Proporciona servicio.

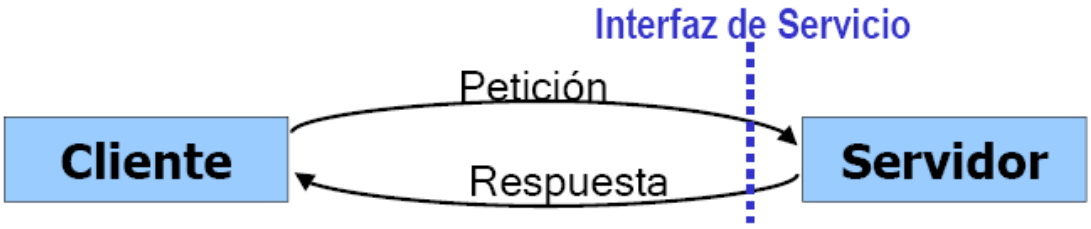


Fig. 8 Modelo Cliente - Servidor.

8.2. Sistema de ficheros distribuido.

Un sistema de ficheros distribuidos es un sistema en el que la ubicación de los ficheros no es relevante ya que pueden ser usados a través de la red. En dicho sistema los ficheros pueden estar en varias máquinas y pueden ser usados por un usuario en una única máquina a través de la red.

Cada persona usa varios ordenadores de manera transparente y todos se mantienen físicamente conectados en red. El sistema de ficheros se mantiene en la red y el uso de sus ficheros no depende de ninguna ubicación dentro de la misma red del sistema. Se usa por tanto ruta lógica no física.

Cuando un usuario pide un fichero para lectura o bien para escritura lo que hace es mandar al kernel la petición y este se encarga de interactuar mediante algún protocolo (Ejemplo: 9P en Plan9 y Styx en Inferno) con el sistema de ficheros distribuido, para localizar el fichero pedido y servírselo al usuario o aplicación que lo pidió. Por ejemplo, el kernel de una máquina recibe la petición de un fichero y este fichero lo tiene el sistema de ficheros de otra máquina que forma parte de su sistema de ficheros distribuido aunque físicamente estén separadas.

Los sistemas de ficheros distribuidos son como se puede ver bastante mas cómodos de manejar para el usuario, aunque tengan problemas como el de usar algún algoritmo especifico para mantener la coherencia entre los ficheros de las distintas máquinas en las que se encuentre el sistema de ficheros distribuido. Compartimos información, ganamos comodidad y disminuyendo el gasto.

Los servidores pueden ofrecer un gran espacio de almacenamiento que sería costoso y poco práctico suministrar a cada cliente. La utilidad de un sistema de ficheros distribuido se ve claramente cuando se considera a un grupo de empleados que tienen que compartir documentos.

Por ejemplo, para compartir programas también es de gran utilidad. En ambos casos la administración del sistema se simplifica.

Un sistema de ficheros distribuido es tan importante para un Sistema Distribuido como para uno tradicional: debe mantener las funciones del sistema de ficheros tradicional (organización, almacenaje, recuperación, protección, nombrado y compartición de ficheros), por lo tanto los programas deben ser capaces de acceder a ficheros remotos sin copiarlos a sus discos duros locales. También proveer acceso a ficheros en los nodos que no tengan disco duro. Normalmente el sistema de ficheros distribuido es una de las primeras funcionalidades que se intentan implementar y es de las más utilizadas por lo que su buena funcionalidad y rendimiento son críticas.

8.3. Servicio de Nombres.

La denominación o gestión de nombres, es la correspondencia entre objetos lógicos y físicos. Por ejemplo, un usuario trata con conjuntos de datos representados por nombres de ficheros, mientras que el sistema gestiona bloques físicos de datos almacenados en pistas de un disco. Normalmente el usuario se refiere a un fichero por un nombre textual, el cual posteriormente se traduce a un identificador numérico que acaba refiriéndose a bloques de un disco. Esta correspondencia entre los dos niveles proporciona a los usuarios una abstracción de cómo y dónde están realmente almacenados los datos.

En un Sistema Distribuido hay que añadir una nueva dimensión a la abstracción: ¿En qué lugar (máquina) de la red está el objeto referenciado?

Capacidad y estructura del esquema de nombres. Veamos ahora cómo puede ser el espacio de nombres en cuanto a su capacidad y estructura.

El Espacio de Nombres puede tener una capacidad Limitada o Infinita. El actual espacio de las direcciones de Internet es un ejemplo de capacidad limitada (ej. 138.100.56.34).

En cuanto a su estructura, puede ser Plana o Jerárquica. La estructura plana de nombres está asociada a espacios de nombres de capacidad finita, mientras que en la estructura jerárquica las direcciones pueden crecer indefinidamente. Cuando se utiliza la estructura jerárquica, se dice que la resolución de nombres se realiza de “acuerdo al contexto”, es decir, traduciendo cada uno de los nombres anteriores al nombre final que indica la jerarquía por la que hay que pasar hasta llegar al

objeto concreto. Ejemplo: dia.eui.upm.es/notas.html hace referencia a un fichero (notas.html) situado en la máquina dia de la eui de la upm. Para resolver tales nombres se va ascendiendo por esta jerarquía de nombres, de tal forma que en cada nivel se es capaz de resolver el nombre y obtener la dirección del siguiente nivel hasta llegar a la máquina de destino, y en ella obtener el objeto con el nombre indicado (notas.html).

No se debe confundir la capacidad de nombres con la capacidad de identificadores de dirección. Así, por ejemplo, aunque el actual sistema de direcciones de Internet es finito (en número de máquinas), el número de algunos recursos referenciables en la red es ilimitado, puesto que cada máquina puede contar con una estructura jerárquica de ficheros potencialmente ilimitada (salvo por la capacidad y limitaciones de tablas).

Podemos ver, entonces, la necesidad de la resolución o traducción de nombres por identificadores de dirección. Una posibilidad sería que cada programa o sistema operativo de un Sistema Distribuido se programara directamente con las direcciones de todos los objetos actuales y futuros de la red completa, pero no resultaría muy práctico, pues no es fácil conocer, a priori, todos los posibles objetos de una red, y sería imposible realizar cualquier cambio de dirección. Parece mucho más razonable ver que esta resolución de nombres no es más que un nuevo servicio que debe ofrecer el sistema a los clientes.

Este nuevo servicio de resolución de nombres se denomina **Servidor de Nombres** (en inglés también se le conoce como binder, ya que a la traducción de nombres se le denomina binding). Así, cuando un cliente necesite conocer la dirección de cualquier servidor, lo único que tiene que hacer es preguntárselo al servidor de nombres. Desde luego, el servidor de nombres residirá en alguna máquina de dirección bien conocida.

Funciones del Servidor de Nombres. Ya hemos visto que la función básica del servidor de nombres es la resolución o traducción de un nombre a un identificador, pero requiere otros servicios adicionales:

- Resolución: La traducción del nombre por el identificador de comunicación.
- Inclusión: Añadir una pareja nombre/identificador al servidor de nombres.
- Borrado: Eliminar una entrada del servidor de nombres.
- Modificación: Modificación del nombre/identificador de una entrada.

Ya hemos comentado que cuando un cliente requiere cualquier servicio del Sistema Distribuido, primero es necesario comunicarse con el servidor de nombres para obtener el identificador de un servidor del servicio requerido. ¿Y si el servidor de nombres falla o se cae? La respuesta es clara: Se pierden todos los accesos a los servicios del sistema.

Dada la importancia del servidor de nombres, cuyo funcionamiento es vital para el resto del sistema, parece que se hace necesario que este servicio especial sea **tolerante a fallos**. Teniendo en cuenta, además, que va a ser un servicio muy requerido, pues todas las utilizaciones de cualquier servicio deben pasar primero por él, para facilitar la tolerancia a fallos y evitar el cuello de botella, suele ser normal que el servicio de nombres esté formado por servidores replicados que ofrezcan este servicio de nombres.

Para acceder a un objeto remoto, el proceso cliente (que sólo conoce el nombre del recurso) debe conseguir el identificador de comunicación del recurso que solicita antes de comunicarse realmente con él. Para ello debe acudir primero a los servidores de nombres intermedios necesarios hasta conseguir dicho identificador de comunicación, con el consiguiente tiempo de demora debido a los tiempos de resolución o traducción de cada uno de los servidores de nombres requeridos.

Cuando se está accediendo a menudo a un objeto remoto, para evitar el tiempo de resolución de los nombres intermedios, el proceso cliente puede mantener una **tabla caché con los identificadores** de dirección de los objetos más recientemente referenciados, y utilizar directamente estos identificadores para acceder a los objetos.

A la hora de diseñar un Sistema de Nombres o de Denominación, se deben perseguir estos dos objetivos:

- Transparencia de ubicación. El nombre de un objeto no debe revelar su ubicación física.
- Independencia de ubicación. El nombre del objeto no debe cambiar cuando cambie su ubicación física. Esto implica transparencia dinámica, ya que un nombre puede asociar el mismo objeto a lugares diferentes en momentos distintos.

Un servidor de nombres puede interpretar nombres:

- Puros. Son simplemente series de bits sin ninguna interpretación posible (salvo para el servidor de nombres). Por ejemplo: fénix.eu1.upm.es

- Impuros. Incluyen bits que indican directamente una dirección, permisos de acceso o cualquier información sobre el objeto. Estos entran en conflicto con el principio de transparencia. Con un nombre impuro cualquier información implícita que lleve, puede quedarse obsoleta si el recurso al que se refiere cambia su dirección, permisos, etc. Con los nombres puros simplemente hay que preocuparse de mantener actualizadas las bases de datos de los servidores de nombres. Por ejemplo: 138.100.55.fenix

Control de acceso. Para evitar accesos no autorizados a los recursos del sistema, un primer paso consiste en hacer que el identificador de un recurso no se pueda obtener fácilmente a partir de su nombre, si no es a través del servidor de nombres. Y. por supuesto, el servidor de nombres debe ocuparse de comprobar la identidad del cliente que solicita una resolución de nombres antes de darle el identificador solicitado. Los identificadores que se comportan así, se denominan **credenciales**. Por eso se dice que para poder acceder a un recurso o servicio, previamente debe obtenerse la credencial correspondiente.

8.4. Servicios de Sincronización y Coordinación.

La medida del tiempo es una cuestión muy importante en los Sistemas Distribuidos. En los sistemas centralizados también lo es, pero no se le da importancia debido a que no es ningún problema conseguir que todos los procesos tengan una misma referencia: el reloj compartido. En cambio en los Sistemas Distribuidos cada ordenador tiene su propio reloj y, como veremos, no resulta fácil sincronizarlos.

Como hemos dicho, la medida del tiempo es muy importante, pues resulta normal tener que saber a qué hora del día han sucedido los distintos eventos que se producen en un ordenador. La precisión requerida en cada caso varía, siendo los sistemas de tiempo real los que posiblemente requieren una mayor precisión. No obstante, incluso en sistemas de propósito general se hace necesario el disponer de una sincronía horaria.

La herramienta make sirve como ejemplo claro de esta necesidad. Supongamos que tenemos un programa compuesto por múltiples módulos que se están escribiendo y actualizando por diversos programadores desde equipos distintos. En cualquier momento, desde cualquier puesto, puede solicitarse la generación de un programa ejecutable mediante el comando make. Pero ahora estamos en un entorno distribuido con reparto de carga, de tal manera que la compilación de un fichero no tiene por qué realizarse en la misma estación en la que se editó. El comando make comprueba que la hora de un fichero objeto debe ser posterior a la del correspondiente fichero

fuente, pues de no ser así significa que no se ha actualizado el fichero objeto, y se genera la compilación del fichero fuente. Pues bien, supongamos que un fichero compilado genera un objeto con la hora 2144. Poco después el encargado de ese fichero fuente lo edita con su hora local 2143 (obviamente, el equipo donde se edita va retrasado respecto a la estación en la que se compila). Cuando el programador ejecuta el comando make, éste se encuentra con que el fichero fuente se modificó con la hora 2143, mientras que el objeto lo generó una compilación en el momento 2144, lo que le hace suponer que es resultante de la última actualización del fichero fuente. Claramente se observa que el programa ejecutable resultante estará formado a partir de un módulo objeto obsoleto, con lo que su comportamiento no será el esperado, y el programador no entenderá por qué no funciona.

Ya que la medida del tiempo es tan básica para nuestra forma de pensar, y el efecto de no tener los relojes sincronizados puede ser dramático, comenzaremos por ver cómo podría conseguirse la sincronización de los relojes de todas las estaciones del Sistema Distribuido o, lo que es lo mismo, que todos tengan la misma hora simultáneamente. A esto se le conoce como la **sincronización de los relojes físicos**.

Cada ordenador dispone de su propio reloj físico. Este reloj es un dispositivo electrónico basado en un cristal de cuarzo que oscila a una determinada frecuencia, y que además, puede programarse para generar interrupciones a intervalos determinados (cada cierto número de oscilaciones). Sabiendo cada cuanto tiempo se producen estas interrupciones (llamadas **interrupciones de reloj**), pueden aprovecharse para llevar la cuenta del tiempo, y por consiguiente de la fecha y la hora actual. Los sistemas operativos lo hacen mediante un contador que se va incrementando a medida que se producen las interrupciones de reloj.

Esta fecha y hora se suele utilizar para indicar el momento en el que suceden ciertos eventos del sistema, como por ejemplo, la hora de creación o modificación de un fichero de datos. La indicación del momento en que sucede un cierto evento se denomina **marca de tiempo** (time stamp). Para comparar marcas de tiempo producidas por ordenadores que cuentan con un mismo modelo de reloj, parece que bastaría con saber la diferencia de los valores que tienen los contadores de sus respectivos sistemas operativos, sin embargo esta suposición está basada en una premisa falsa, pues es prácticamente imposible que dos relojes “iguales” oscilen exactamente a la misma frecuencia, sean o no del mismo tipo.

Los relojes basados en un cristal están sujetos a una desviación o deriva, es decir que cuentan o miden el tiempo a velocidades distintas, por lo que progresivamente sus contadores van distanciándose. Por muy pequeño que sea el período de oscilación de los dos relojes, la diferencia acumulada después de muchas oscilaciones conduce a una diferencia claramente observable. La velocidad de deriva de un reloj es el cambio por unidad de tiempo en la diferencia de valores entre su contador y el de un reloj perfecto de referencia. Para los relojes de cristal de cuarzo esta velocidad de deriva está alrededor de 10-6, lo que significa una diferencia de un segundo cada 11,6 días.

Los relojes más precisos utilizan osciladores atómicos, cuya precisión es del orden de 1014 (un segundo en 1.400.000 años). Diversos laboratorios (con reloj atómico) de todo el mundo le indican el número de ticks (interrupciones de reloj) periódicamente al Bureau Internationale de l'Heure (BIH), el cual calcula su valor medio, produciendo el **Tiempo Atómico Internacional** (TAI), teniendo en cuenta que el tiempo se empezó a contar el 1 de enero de 1958, una vez que el segundo se definió como el tiempo que necesita un átomo de Cesio-133 en realizar 9.192.631.770 transiciones. No obstante, las unidades de tiempo que utilizamos tienen un origen astronómico (**tiempo universal**), es decir, están definidas en función de los períodos de rotación y traslación de la Tierra alrededor del Sol, y resulta que este período de rotación se está alargando gradualmente, pues nuestro planeta se está frenando, debido, principalmente, a la fricción de las mareas, efectos atmosféricos y a las corrientes de convección en el núcleo de la tierra. Esto quiere decir que el tiempo atómico y el universal tienden a desincronizarse.

La gradual ralentización de la rotación de la Tierra da lugar a que en la actualidad, un día TAI (86.400 segundos TAI) sea 3 milisegundos menor que el día solar. Por esto, cuando la diferencia llega a los 900 ms, el BIH añade un segundo "bisiesto" al contador TAI, dando lugar al **Tiempo Universal Coordinado** (UTC), el cual es la base de todas las medidas políticas, civiles y militares del tiempo.

El tiempo UTC está disponible a través de emisoras de radio que emiten señales horarias, mediante satélites geoestacionarios y GPS cuyas precisiones son 0,1 a 10 ms.; 0,1 ms.; y 1 ms. respectivamente. A pesar de esta relativamente buena precisión, nos encontramos con dos problemas. Por una parte tenemos que un receptor de señales horarias es un dispositivo demasiado caro comparado con el precio de una estación de trabajo y, además, en caso de disponer de él, resulta que la medida de tiempo con una precisión de 10 milisegundos puede

resultar insuficiente para un ordenador de 10 MIPS, en el que esos 10 ms. Pueden ejecutarse 100.000 instrucciones y pueden transmitirse varios mensajes.

Supongamos que un equipo dispone de un receptor de estas señales horarias. El reloj hardware del ordenador puede ir más despacio o más deprisa que el UTC. ¿Qué hacemos para sincronizarlo con el UTC? Si la hora del reloj local es anterior a la hora UTC, simplemente avanzamos la hora local hasta la hora UTC; lo más que puede suceder es que parezca como si hubiera perdido algunos ticks de reloj. Pero ¿qué se puede hacer si la hora local es mayor que la hora UTC? ¿Retrasamos la hora local hasta la hora UTC? Esto nunca debe hacerse, pues se puede confundir a las aplicaciones. Pensemos, en el ejemplo del comando make, si se atrasase el reloj local, podría suceder que en un mismo equipo, la hora de compilación de un programa fuera anterior a la hora de edición, lo cual confundiría al comando make.

La solución para la situación anterior en que la hora local es mayor que la hora UTC no estriba en atrasar la hora, sino ralentizar la velocidad del reloj local hasta que se sincronice con la hora UTC, y entonces volver a ponerlo a su velocidad. De esta manera no se producen paradojas como la de la compilación del fichero.

En los Sistemas Distribuidos resulta incluso más importante la sincronización horaria entre los distintos equipos que el mantener la mínima diferencia con la hora UTC. Si la máxima velocidad de deriva de un reloj hardware de un ordenador es ρ, dos relojes cuya divergencia de la hora UTC sea opuesta, en un momento Δt después de haberse sincronizado pueden mostrar una diferencia horaria de $2\rho\Delta t$. Esto quiere decir que si quiere garantizar que dos ordenadores no difieran más de δ segundos en su hora, deben sincronizarse (su hora hardware) al menos cada $\delta/2\rho$ segundos.

Los algoritmos más utilizados para realizar dicha sincronización son los algoritmos de: **Cristian y Berkeley**.

- **Algoritmo de Cristian.**

Está pensado para entornos en los que un ordenador, al que se llama Servidor de Tiempo, está sincronizado con un ahora UTC, y el resto de las estaciones quieren sincronizarse con él. Periódicamente (cada $\delta/2\rho$ segundos), cada ordenador cliente envía un mensaje al servidor de tiempo preguntándole la hora actual. El servidor de tiempo responde con un mensaje que contiene la hora actual TUTC. Cuando el cliente recibe el mensaje debe actualizar su reloj. Si su hora local es posterior a TUTC debe ralentizar la velocidad del reloj hasta ajustarse a la hora UTC. Si su hora

es anterior a la hora recibida, directamente actualiza su reloj o bien lo acelera hasta igualarlo a la hora UTC.

Ahora hay que considerar el error cometido, pues se ha requerido un tiempo para la transmisión del mensaje con la hora UTC desde el servidor hasta el cliente, y lo que es pero, este retardo es variable dependiendo de la carga de la red.

El algoritmo de Cristian calcula este retraso de la siguiente manera: Mide el tiempo que se tarda en recibir la respuesta desde que se envía el mensaje de petición. Para ello simplemente tiene que anotar la hora de envío TE y de recepción TR, utilizando el reloj local. En ausencia de otra información, el tiempo estimado de propagación del mensaje será (TR - TE)/2. Así cuando el cliente recibe le mensaje con la hora actual, tiene que añadirle el retardo calculado.

La duración estimada del retardo puede mejorarse si se sabe lo que tarda el servidor de tiempo en tratar la interrupción que genera la llegada del mensaje y atender la petición. Si a este tiempo lo llamamos I, el tiempo estimado de propagación del mensaje será (TR - TE-I)/2. (Ver Fig. 9).

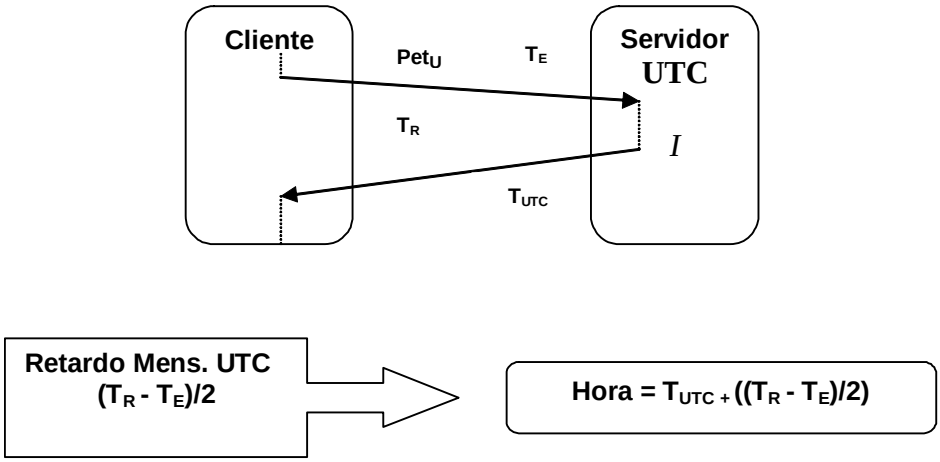


Fig. 9 Algoritmo de Cristian.

Tabla 2 Duración estimada para la transmisión de un mensaje.

Refinando	Hora = $T_{UTC} + ((T_R - T_E - I)/2)$
	Valores medios de varias pruebas.

El problema que se presenta es el mismo de todos los servicios ofrecidos por un único servidor: la posibilidad de fallo. Cristian sugiere que la hora debe suministrarse por varios servidores de tiempo

sincronizados mediante receptores de tiempo UTC; el cliente envía su petición a todos los servidores y toma la primera respuesta recibida.

- **Algoritmo de Berkeley.**

Está pensado para entornos en los que no se dispone de ningún receptor de tiempo UTC, y lo único que se pretende es que todos los ordenadores se mantengan sincronizados con una misma hora.

En este algoritmo, entre todos los equipos del Sistema Distribuido eligen un coordinado para que actúe como maestro o servidor de tiempo. Al contrario que en el algoritmo de Cristian, en el que el servidor era pasivo, en este caso el coordinador elegido pregunta la hora periódicamente, al resto de las estaciones (los esclavos). Cuando los esclavos responden cada con su hora local, el maestro estima los retardos de propagación de los mensajes con cada uno de los esclavos (de manera similar a Cristian) y calcula un tiempo promedio con las horas recibidas y la suya propia.

Queda por actualizar los relojes de los esclavos. Ahora el maestro, en lugar de enviar la hora coordinada a todos los esclavos (lo que introduciría un cierto error debido al tiempo de transmisión), envía a cada uno el desfase que tiene con la hora promedio calculada (Ver Fig. 10). Recibido este desfase, cada equipo actualiza su reloj adelantándolo directamente o ralentizándolo temporalmente en caso de ir adelantado.

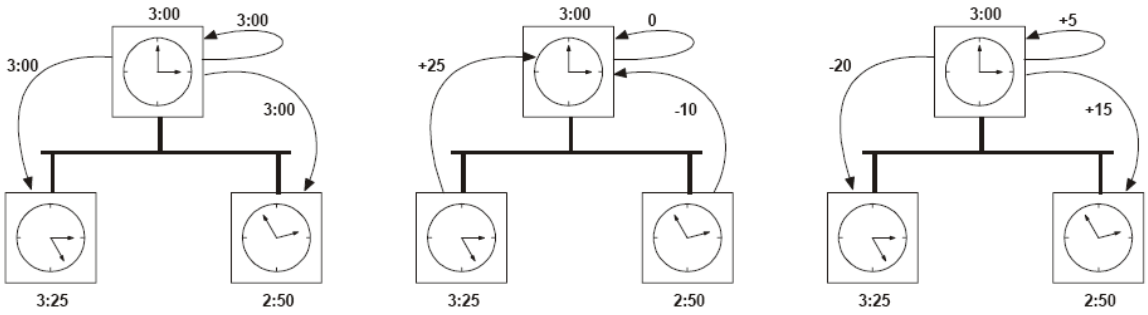


Fig. 10 Algoritmo de Berkeley.

El algoritmo realiza un “promedio tolerante a fallos” con los tiempos recibidos de los esclavos, es decir, que considera solamente el subconjunto de relojes que no difieren entre sí más de una cierta cantidad, considerando que los demás no están funcionando bien.

Si el maestro falla (no pregunta, no envía modificaciones de tiempos, o éstas son “demasiado extrañas”), simplemente se elige otro coordinador.

Relojes Lógicos.

Como se ha visto, resulta difícil sincronizar múltiples relojes distribuidos para que todos ellos mantengan una única hora estándar con la suficiente precisión que no dé lugar a ambigüedades.
Sin embargo, Lamport señaló que la sincronización de los relojes no necesita ser absoluta. Por una parte, si dos procesos no interactúan, no tienen ninguna necesidad de que sus relojes respectivos estén sincronizados, pues la falta de sincronización no será observable y no causará problemas. Por otra parte, lo que normalmente importa no es que todos los procesos estén sincronizados según una misma hora exacta, sino que todos estén de acuerdo sobre el orden en que se suceden los eventos que se producen.

Para algunos entornos en los que no se requiere una sincronización exacta con una hora externa de referencia (tiempo UTC), en lugar de tratar con los relojes físicos, se trabaja con los relojes lógicos, en los que solamente tiene importancia el orden de los eventos, no la medida del momento exacto en el que se producen.

Para sincronizar los relojes lógicos, Lamport definió la relación "**sucedió antes**", según la cual, la expresión a → b quiere decir que "a sucedió antes que b", y significa que todos los procesos coinciden en que primero sucedió el evento a y posteriormente el b. Esta relación se observa directamente en dos situaciones:

- Si dos eventos se producen en el mismo proceso, tienen lugar en el orden que indica su reloj común.
- Cuando dos procesos se comunican mediante un mensaje, el evento de enviarlo se produce siempre antes del evento de recibirlo.

Si dos eventos, x e y, se producen en procesos diferentes que no intercambian mensajes, entonces no es cierto que x → y, ni y → x. En este caso se dice que tales eventos son concurrentes, lo que significa que sencillamente no se sabe nada (ni se necesita saber) sobre cuál de ellos sucedió primero.

Lo que se necesita es una forma de medir el tiempo tal que para todo evento a, se le pueda asignar un valor de tiempo R(a) en el que todos los procesos estén de acuerdo. Veamos el mecanismo de Lamport para asignar tiempos a los eventos.

Lamport inventó un mecanismo denominado **reloj lógico**, el cual es simplemente un contador software monótono creciente, cuyo valor no necesita tener ninguna relación concreta con ningún reloj físico. En cada ordenar hay un reloj lógico R que se utiliza para poner marcas de tiempo a los eventos que se producen, tal que Rp(a) indica la marca de tiempo del evento a en el proceso p. Así, las dos situaciones en las que se observa la relación "sucedió antes" se tratan de la siguiente manera con los relojes lógicos:

1. Rp se incrementa en una unidad antes de que se produzca cada evento en el proceso, es decir Rp = RP + 1.
2. a) Cuando un proceso p envía un mensaje m, se le pega al mensaje el valor t= Rp.
   b) Cuando el proceso q recibe el mensaje (m, t), calcula Rq:= max (Rq, t) y aplica el paso 1 antes de marcar el evento de recepción del mensaje (m, t).

Exclusión mutua.

Aunque el problema de las regiones críticas se produce primordialmente en programas con variables compartidas, en los Sistemas Distribuidos también se producen situaciones en las que hay recursos compartidos que no pueden ser utilizados por más de un proceso al mismo tiempo.

En sistemas con uno o más procesadores que comparten memoria, suelen utilizarse mecanismos como semáforos, monitores, regiones críticas condicionales, etc. Sin embargo, en los Sistemas Distribuidos los procesos ya no comparten la memoria física, por lo que se debe pensar en otros mecanismos o algoritmos que nos proporcionen la exclusión mutua.

Funciones básicas de exclusión mutua:

– **enter():** Acceso a la región critica (bloqueo).
– **operations():** Manipulación de los recursos compartidos.
– **exit():** Liberación del recurso (despierta a procesos en espera).

Los **requisitos básicos** que debe cumplir un algoritmo de exclusión mutua son los siguientes:

Seguridad: Como mucho sólo un proceso puede estar ejecutándose dentro de la región crítica en un momento dado.

Viveza: Un proceso que desea entrar en una región crítica debe poder entrar en algún tiempo. Esto quiere decir que no deben producirse interbloqueos ni inanición.

Orden: La entrada a la región crítica debe realizarse en el orden causal "sucedió antes" definido por Lamport. Se pueden tomar dos enfoques en el diseño de estos algoritmos para proporcionar la exclusión mutua:

- **Algoritmos centralizados.**

El algoritmo más simple (Ver Fig. 11):

– Los clientes solicitan el acceso a un elemento de control que gestiona la cola de peticiones pendientes.

– Tres mensajes: enter, exit y OK.

– No cumple necesariamente la propiedad de ordenación.

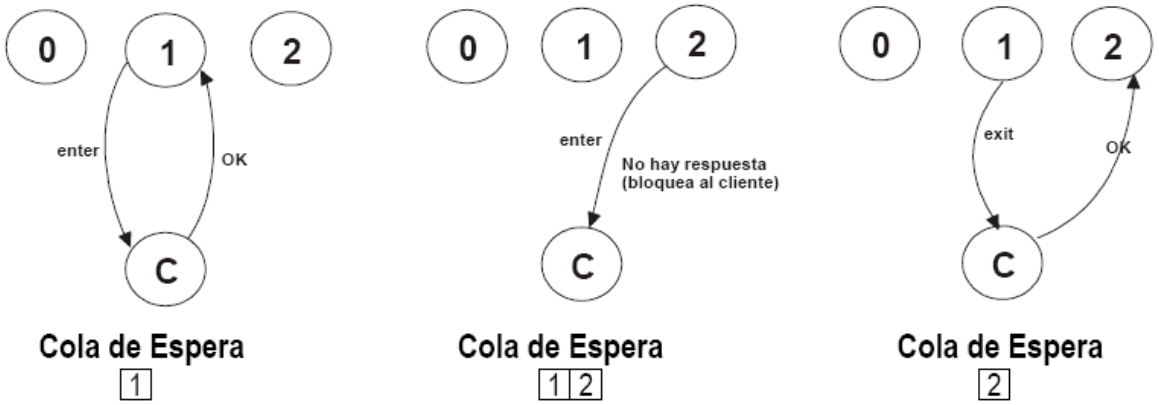


Fig. 11 Algoritmo Centralizado.

- **Algoritmos distribuidos.**

Algoritmos distribuidos de paso de testigo: (Ver Fig. 12)

– Se distribuyen los elementos en un anillo lógico.

– Se circula un token que permite el acceso a la región crítica.

– El token se libera al abandonar la región.

– No cumple la propiedad de ordenación

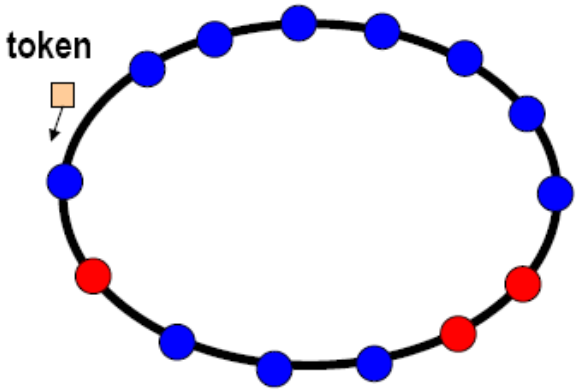


Fig. 12 Algoritmo de paso de testigo.

8.5. Memoria Compartida Distribuida (DSM).

La memoria compartida distribuida o DSM es una abstracción que se propone como alternativa a la comunicación por mensajes. Los procesos acceden a DSM para leer y actualizar, dentro de sus espacios de direcciones, sobre lo que aparenta ser la memoria interna normal asignada a un proceso. Sin embargo, existe un sistema subyacente en tiempo de ejecución que asegura de forma transparente, que procesos diferentes ejecutándose en computadores diferentes observen las actualizaciones realizadas entre ellas. Es como si los procesos accedieran a una única memoria compartida, pero de hecho la memoria física está distribuida. (Ver Fig. 13)

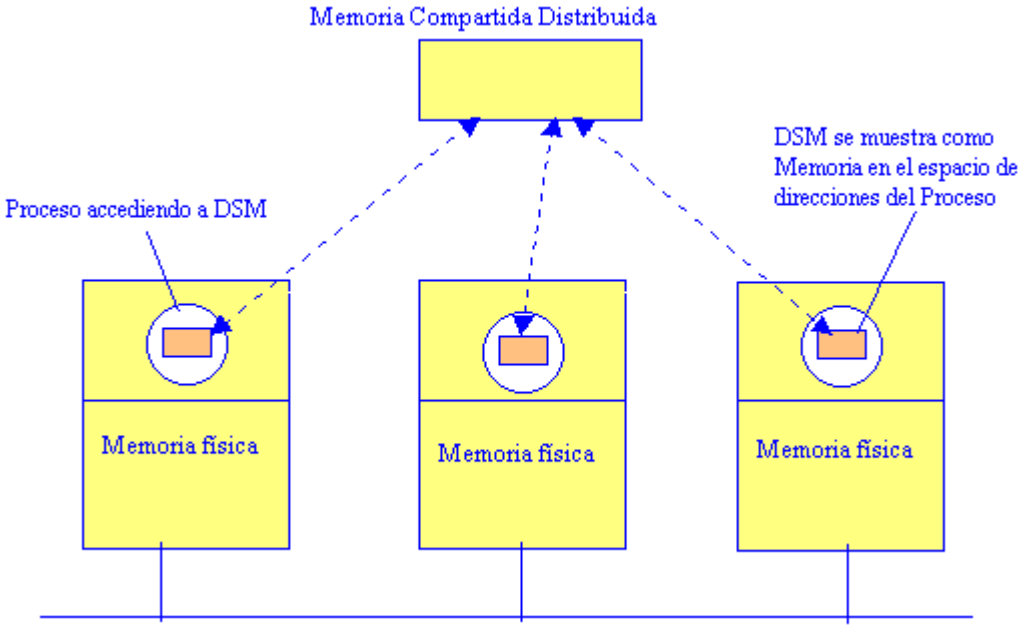


Fig. 13 Abstracción de la Memoria Compartida Distribuida.

La principal característica de DSM es que ahorra al programador todo lo concerniente al paso de mensajes al escribir sus aplicaciones, cuestión que en otro sistema debería tenerse muy presente. DSM es fundamentalmente una herramienta para aplicaciones paralelas o para aplicaciones o grupos de aplicaciones distribuidas en las que se puede acceder directamente a datos individuales que ellas comparten. En general, DSM es menos apropiado para sistemas cliente-servidor, ya que los clientes ven al servidor como un gestor de recursos en forma de datos abstractos que se acceden a través de peticiones (por razones de modularidad y protección). Sin embargo, los servidores pueden proporcionar DSM compartido entre los clientes. Por ejemplo, los archivos plasmados en memoria que son compartidos y sobre los que se gestiona un cierto grado de consistencia son una forma de DSM.

El paso de mensajes no puede ser eliminado completamente en un Sistema Distribuido: en ausencia de memoria compartida físicamente, el soporte en tiempo de ejecución de DSM envía las actualizaciones mediante mensajes entre computadores. Los sistemas DSM gestionan datos replicados: cada computador tiene una copia local de aquellos datos almacenados en DSM que han sido usados recientemente, con el fin de acelerar sus accesos.

Estrategias de implementación:

- **Memoria compartida basada en páginas.**

El esquema de DSM propone un espacio de direcciones de memoria virtual que integre la memoria de todas las computadoras del sistema, y su uso mediante paginación. Las páginas quedan restringidas a estar necesariamente en un único ordenador. Cuando un programa intenta acceder a una posición virtual de memoria, se comprueba si esa página se encuentra de forma local. Si no se encuentra, se provoca un fallo de página, y el sistema operativo solicita la página al resto de computadoras. El sistema funciona de forma análoga al sistema de memoria virtual tradicional, pero en este caso los fallos de página se propagan al resto de ordenadores, hasta que la petición llega al ordenador que tiene la página virtual solicitada en su memoria local. A primera vista este sistema parece más eficiente que el acceso a la memoria virtual en disco, pero en la realidad ha mostrado ser un sistema demasiado lento en ciertas aplicaciones, ya que provoca un tráfico de páginas excesivo.

Una mejora dirigida a mejorar el rendimiento, sugiere dividir el espacio de direcciones en una zona local y privada y una zona de memoria compartida, que se usará únicamente por procesos que necesiten compartir datos. Esta abstracción se acerca a la idea de programación mediante la

declaración explícita de datos públicos y privados, y minimiza el envío de información, ya que sólo se enviarán los datos que realmente vayan a compartirse.

- **Memoria compartida basada en objetos.**

Una alternativa al uso de páginas es tomar el objeto como base de la transferencia de memoria. Aunque el control de la memoria resulta más complejo, el resultado es al mismo tiempo modular y flexible, y la sincronización y el acceso se pueden integrar limpiamente. Otra de las restricciones de este modelo es que todos los accesos a los objetos compartidos han de realizarse mediante llamadas a los métodos de los objetos, con lo que no se admiten programas no modulares y se consideran incompatibles.

8.6. Gestión de Procesos.

Un proceso es un programa en ejecución, es decir, es la actividad que resulta de la ejecución de un algoritmo, con sus datos, sobre un procesador. Decimos que dos procesos son concurrentes, si su ejecución se solapa en el tiempo. Si tales procesos comparten el procesador, se ejecutan en multiprogramación, mientras que si n procesos se ejecutan en n procesadores, entonces existe ejecución paralela. En este caso, si los procesadores comparten una memoria central común, se tiene su sistema multiprocesador, mientras que si los procesadores están conectados mediante una red de comunicaciones, entonces tenemos procesamiento distribuido.

La gestión de procesos en un sistema operativo centralizado se ocupa de los mecanismos y políticas para compartir o repartir un procesador entre diversos procesos de usuario. El objetivo de la gestión de procesos en los sistemas operativos distribuidos, es compartir todos los recursos de proceso entre todos los procesos de usuarios de toda la red del Sistema Distribuido.

Para conseguir esto, es necesario proporcionar mecanismos y políticas para realizar operaciones con los procesos (crear, nombrar, borrar, ejecutar, etc.) tanto locales como remotos, para gestionarlos, comunicarlos y sincronizarlos.

Para mejorar el tiempo de respuesta de los procesos, se va a necesitar la posibilidad de repartir la carga de trabajo de una estación entre otras que estén más descargadas, por lo que se deberá proporcionar la posibilidad de ejecución remota de procesos y de migración de procesos entre estaciones.

En un sistema centralizado, la asignación de recursos disponibles está gestionada por el sistema operativo. Para ejecutar un proceso, se le asigna memoria y se ejecuta sobre el único procesador del sistema. Sin embargo, en un Sistema Distribuido nos encontramos con que hay múltiples máquinas y procesadores, con lo que la decisión de dónde ejecutar un programa, es decir, sobre qué procesador, ya no es tan trivial.

En un Sistema Distribuido un proceso puede ejecutarse:

- Utilizando la propia estación de trabajo.
- Utilizando otras estaciones libres.

La elección del procesador sobre el que se debe ejecutar un programa en un Sistema Distribuido se realiza de acuerdo a una política de **Reparto de Carga de Trabajo**.
El reparto de carga puede realizarse en dos momentos:

- Cuando se van a crear los procesos, eligiendo el procesador de ejecución, y necesitando para ello la posibilidad de ejecución remota de procesos.
- Reasignando los procesadores a los procesos durante su ejecución, esto es, mediante la migración de procesos.

La mejora de las prestaciones globales del sistema depende del objetivo del sistema. Se puede mejorar el rendimiento, el tiempo de respuesta, la justicia, o una combinación de las tres.

El **rendimiento** se define como la cantidad de trabajo desarrollado por unidad de tiempo. Trabajo útil significa programas ejecutados. Así, el rendimiento se maximiza minimizando la sobrecarga, el tiempo de comunicación y el tiempo de espera. Sobrecarga es el tiempo que le dedica el sistema operativo a la planificación y a los cambios de contexto. El tiempo de comunicación es el tiempo que pasan los procesos esperando a recibir datos o eventos, o esperando a que se vacíen buffers de salida de datos para poder continuar su ejecución. El tiempo de espera es el que pasan los procesos en la cola “preparados” esperando a que se haya un procesador disponible.

El **tiempo de respuesta** es el tiempo que pasa desde la entrada de datos hasta que comienza la salida de datos.

La **Justicia** es un concepto más esotérico. Si se mejora el rendimiento o los tiempos de respuesta, es muy posible que algún proceso nunca llegue a ejecutarse, sin embargo, un planificador justo dará a todos los procesos el mismo acceso a los procesadores.

Es fácil ver que maximizar el rendimiento, minimizar el tiempo de respuesta y asegurar justicia absoluta simultáneamente no es posible. El rendimiento es máximo cuando se minimizan los cambios de contexto, por lo que se prima a los trabajos largos sobre los cortos. El tiempo de respuesta, por el contrario, se minimiza ejecutando primero los más cortos. La justicia se obtiene mediante frecuentes cambios de contexto, lo cual no es bueno ni para el rendimiento ni para los tiempos de respuestas.

Ejecución remota y migración de proceso.

El primer paso para llevar a cabo la ejecución remota o la migración de procesos es la asignación de procesadores, por lo que explicaremos las políticas de asignación de procesadores.

Hay muchos algoritmos propuestos para la asignación o reparto de procesadores, para diseñar dichos algoritmos se deben tomar en cuenta los siguientes aspectos:

a. Algoritmos deterministas/heurísticos. Los algoritmos deterministas son apropiados para las situaciones en los que se conocen todos los datos (toda la lista de procesos y todos sus requisitos), o son fácilmente predecible, por ejemplo aplicaciones bancarias o reservas de líneas aéreas. Pero en otros casos, la carga de trabajo puede variar rápidamente y de forma impredecible, pues depende de quién esté trabajando, y de lo que está haciendo. En estas situaciones la asignación del procesador no se puede realizar de una manera matemática y determinista, y es necesario utilizar técnicas heurísticas.

b. Algoritmos centralizados/distribuidos. La concentración de la información en un nodo central permite tomar una mejor decisión, pero es menos robusta y puede generar sobrecarga en el nodo central. Aunque suelen ser preferibles los algoritmos distribuidos, en algunos casos se utilizan algoritmos centralizados por no disponer de variantes distribuidas.

c. Algoritmos óptimos/casi-óptimos. Las soluciones óptimas pueden encontrarse tanto en los algoritmos centralizados como en los distribuidos, pero, obviamente, con mucho más trabajo (requiere mucho más tráfico de información y proceso) que conformándose con una buena solución que no sea la óptima. En la práctica, suele utilizarse algoritmos heurísticos distribuidos y “casi-óptimos”.

d. Políticas de transferencia. Cuando se va a crear un proceso hay que tomar la decisión de si se ejecuta localmente o en un procesador remoto. Si la máquina está demasiado cargada, el nuevo proceso debe transferirse a un procesador remoto. La cuestión es si la decisión

debe tomarse localmente o de una forma global (considerando el estado de carga de todos los procesadores). Los algoritmos locales son más simples y más distantes de la solución óptima. Los algoritmos globales dan una solución solo ligeramente mejor, pero a un costo mucho mayor.

e. Políticas de ubicación. Una vez que la política de transferencia ha decidido que hay que deshacerse del proceso y enviarlo a otro procesador, la política de ubicación debe decidir a cuál. Claramente esta política no puede ser local, pues se necesita información sobre la carga de los demás procesadores para tomar una elección. Esta información puede diseminarse por el sistema mediante dos técnicas. Una posibilidad es que el equipo emisor solicite ayuda a procesadores disponibles (algoritmos iniciados por el cliente). La otra opción es que los procesadores ociosos difundan a priori su condición de disponibles (algoritmos iniciados por el servidor).

Hasta ahora se ha considerado que los algoritmos conocen la carga de su propia máquina, por lo que pueden decidir si está sobrecargado o no y comunicárselo a los demás nodos de la red. Pero establecer el nivel de carga no es tan fácil. Las siguientes son algunas aproximaciones para establecer el nivel de carga de un equipo:

a. Una solución simple puede consistir simplemente en contar el número de procesos de cada máquina y utilizar ese número como índice de carga del sistema. Sin embargo, incluso en una estación ociosa puede haber muchos procesos arrancados, tales como los “demonios” del correo electrónico, de las colas de impresión o los gestores de la ventana. Es decir, procesos que, aunque arrancados, pueden estar bloqueados o en espera de algún evento externo, por lo que no suponen ninguna carga de trabajo.

b. El siguiente paso es considerar solamente los procesos que están en ejecución o preparados. Sin embargo, ya que hay “demonios” que se despiertan periódicamente y se vuelven a dormir, puede suceder que se haga muestreo en uno de estos momentos, con lo que la medida sería engañosa.

c. Una medida más directa para realizar la medición de carga, aunque más costosa, es calcular el porcentaje de tiempo que la CPU está ociosa. Está claro que una máquina con un 20% de utilización de CPU está más cargada que otra con solo un 10%, independientemente del número de programas en ejecución. Una forma de medir esta utilización es programar un temporizador (timer) para que interrumpa periódicamente. En

cada interrupción se comprueba si el proceso que está en ejecución es el proceso ocioso o no. De esta manera se puede calcular la proporción de tiempo que la CPU está ociosa.

Ejecución remota.

Se deben diferenciar dos conceptos:

Servicio remoto: Ejecutar en un equipo remoto un programa residente en el sistema remoto. (Por ejemplo rsh).

Ejecución remota: Ejecutar en un equipo remoto un programa residente en el equipo local.

¿Cuándo es conveniente la ejecución remota de procesos? Hay diversas situaciones en las que un reparto de carga mediante ejecución remota mejora el rendimiento global de un Sistema Distribuido:

- Reparto general de la carga entre los procesadores disponibles para conseguir un menor tiempo global de ejecución.

- Mejorar el tiempo de ejecución de aplicaciones paralelas. Distribuyendo un proceso en subtareas se puede conseguir una disminución del tiempo total de ejecución.

- Ejecución remota de procesos que requieren servicios especiales en las máquinas que mejor proporcionan tales servicios. Por ejemplo, acudiendo a un equipo con procesadores específicos para cálculos numéricos.

- El volumen de los datos de entrada es mayor que el espacio de memoria ocupada por el proceso. Por ejemplo, cuando se realizan análisis estadísticos con una gran cantidad de datos residentes en otro equipo, requiere menos tiempo la comunicación para mover el proceso al otro equipo y ejecutarlo allí, que la comunicación para acceder a todos los datos de entrada.

Requisitos para la ejecución remota. Hay tres puntos que se deben tener en cuenta en el diseño de los mecanismos de ejecución remota:

- Debe haber un mecanismo o protocolo para difundir el estado de carga de una estación. Esto está directamente asociado con los mecanismos de gestión general de recursos de todo el sistema.

- Cuando se selecciona una estación de trabajo para una ejecución remota, el tiempo de respuestas de esa estación para la ejecución de cualquier programa se degrada. Si además el usuario de ese ordenador remoto empieza a trabajar, puede notar unas bajas prestaciones. En esta situación, el proceso cuya ejecución se ha asumido puede verse obligado a abandonar la ejecución para no degradar más las prestaciones del sistema remoto.

- La ejecución remota deberá tener lugar tan fácilmente como si se realizara localmente, sufriendo las menores penalizaciones posibles. Esto quiere decir que la ejecución de un proceso deberá ser independiente de la ubicación.

Este último punto no es fácil de cumplir al cien por cien, y depende de las facilidades que se le ofrezcan a los programas remotos:

- A nivel de programación: Llamadas remotas a procedimientos (RPC).
- A nivel del intérprete de comandos.

Entornos homogéneos/heterogéneos. Una de las principales cuestiones en la ejecución remota es la facilidad con la que los recursos remotos pueden ofrecerse a los clientes. Claramente va a resultar más fácil la ejecución remota en un entorno semejante al propietario del proceso (arquitectura, procesador, etc.), que en entornos heterogéneos. Cuando ambos entornos son homogéneos lo único que hay que hacer es indicar las características del entorno de ejecución necesario (memoria, ficheros, etc.).

El problema principal de la ejecución en un sistema remoto distinto del propietario del proceso, es la descripción del servicio a ejecutar, ya que hay diferencias a dos niveles:

- Sintaxis y semántica de los intérpretes de comandos.
- Cuando las arquitecturas son distintas, se requiere o bien traducción binaria (para código y datos), o bien la ejecución interpretada mediante emuladores.

Migración de los procesos.

Los procesos tienen una naturaleza muy dinámica, de tal manera que en un sistema los procesos se crean y destruyen continuamente, sin poder tener un conocimiento adelantado de qué procesos y cuándo se van a crear o destruir. La ejecución remota de programas mejora esta situación, pero lo que nunca se sabe a priori es la carga que realmente va a generar cada proceso. Por eso, la

asignación o planificación estática de procesos no garantiza un aprovechamiento óptimo del sistema, y se hace necesaria la planificación dinámica que proporciona la migración de procesos.
La idea básica de la migración es la planificación dinámica de procesos, es decir, la posibilidad de mover un proceso, durante su ejecución, de un procesador a otro, de tal forma que continúe la ejecución en otro procesador distinto con acceso completo a todos los recursos que estaba utilizando en el procesador anterior. Esta operación puede arrancarse sin que lo sepa ni el propio proceso, ni ninguno de los procesos con los que está interactuando. En cualquier caso, la migración de procesos debe mantener todos los elementos de cálculo y de comunicación que esté utilizando.

La migración de procesos puede realizarse entre sistemas homogéneos o heterogéneos. Sin embargo la migración entre sistemas homogéneos es más fácil, dada la complejidad adicional que supone migrar procesos a otras arquitecturas distintas en sistemas heterogéneos.

La migración de procesos se desarrolla en dos fases:
1º Planificación (¿cuál?, ¿a dónde?).
2º Transferencia del proceso (¿cuánto? Y ¿cuándo?).

La planificación consiste en tomar una decisión para responder a estas dos preguntas: ¿qué proceso va a ser transferido? Y ¿a qué procesador va a ser transferido?, para contestar estas preguntas, lo primero que se debe hacer en la fase de planificación es obtener datos para poder tomar decisiones.

La transferencia del proceso se refiere a los mecanismos utilizados para realizar la transferencia del proceso a otro procesador. La transferencia de un proceso distribuido se realiza en estas dos etapas:

1. Transferencia del estado del proceso.
2. Transferencia de su espacio de direcciones (la memoria que ocupa).

Las preguntas que deben hacerse en esta fase son ¿qué cantidad del proceso va a transferirse? Y ¿cuándo se va ha reanudar la ejecución del proceso?

8.7. Servicio de Seguridad.

Tanto los sistemas centralizados como los distribuidos tienen problema para ofrecer una buena seguridad.

La Seguridad es un concepto amplio y está dirigida a cuatro requisitos básicos:
–Confidencialidad: La información sólo es accesible por las partes autorizadas.
–Integridad: Los contenidos sólo podrán modificarse por las partes autorizadas.
–Disponibilidad: Los componentes de un sistema informático sólo están disponibles por las partes autorizadas.
–Autenticación: Capacidad de verificar la identidad de los usuarios.

En un Sistema Distribuido, los clientes envían peticiones de acceso a datos administrados por servidores, lo que trae consigo enviar información en los mensajes por la red.
La seguridad no sólo es cuestión de ocultar el contenido de los mensajes, también consiste en conocer con certeza la identidad del usuario u otro agente en nombre del cual se envía el mensaje.
La principal meta de la seguridad es restringir el acceso a la información y los recursos, de modo que sólo tengan acceso aquellos que estén autorizados.

Las amenazas de seguridad se dividen en tres clases:

- **Fuga:** adquisición de información por receptores no autorizados.
- **Alteración:** modificación no autorizada de información.
- **Vandalismo:** interferencia en el modo de operación adecuado de un sistema.

Los ataques en los Sistemas Distribuidos dependen de la obtención de acceso a los canales de comunicación. Los métodos de ataque pueden clasificarse en función del modo en que se abusa del canal:

- **Fisgar:** obtener copias sin autorización.
- **Suplantar:** enviar o recibir mensajes utilizando la identidad de otro sin su autorización.
- **Alterar mensajes:** interceptar mensajes y alterar sus contenidos antes de pasarlos al receptor.
- **Reenviar:** almacenar mensajes interceptados y enviarlos más tarde.
- **Denegación de servicio:** desbordar un canal o recurso para impedir que otros accedan a él.

Pero para sistemas que incluyan programas móviles y sistemas cuya seguridad sea sensible a la fuga de información, hay más ataques.

Ataques desde código móvil. Varios lenguajes de programación, han sido diseñados para permitir la descarga de programas desde servidores remotos y así lanzar procesos que se

ejecutan localmente. En este caso, las interfaces internas y los objetos del interior de un proceso en ejecución pueden quedar expuestos a un ataque por código móvil. Java es el lenguaje de este tipo más conocido.

Fugas de información. Si pudiera observarse la sucesión de mensajes en la comunicación entre dos procesos, sería posible vislumbrar información importante aún de su sola existencia. Hay muchas formas sutiles de fugas de información, algunas maliciosas y otras que son consecuencia de errores inadvertidos. El potencial de las fugas aparece cuando se pueden observar los resultados de un cómputo. La aproximación empleada es la asignación de niveles de seguridad a la información y a los canales, y analizar el flujo de información hacia los canales, con el objetivo de asegurar que la información de alto nivel no fluya hacia los canales de bajo nivel.

La criptografía proporciona la base de la autenticación de mensajes, así como del secreto y la integridad, para los Sistemas Operativos.

La encriptación es el proceso de codificación de un mensaje de forma que quede oculto su contenido.

La criptografía moderna incluye algunos algoritmos seguros de encriptación y desencriptación de mensajes. Todos ellos se basan en el uso de ciertos secretos llamados claves.

Una clave criptográfica es un parámetro empleado en un algoritmo de encriptación de manera que no sea reversible sin el conocimiento de una clave.

Hay dos clases principales de algoritmos de encriptación de uso general:
La primera emplea claves secretas compartidas: donde el emisor y el receptor deben compartir el conocimiento de una clave y ésta no debe ser revelada a ningún otro.

La segunda emplea pares de claves pública/privada: donde el emisor de un mensaje emplea una clave pública, difundida previamente por el receptor, para encriptar el mensaje. El receptor emplea la clave privada correspondiente para desencriptar el mensaje. A pesar de que una multitud de principales pudiera examinar la clave pública, solamente el receptor puede desencriptar el mensaje, gracias a su clave privada. Los algoritmos de encriptación de clave pública requieren usualmente de 100 a 1.000 veces más potencia de procesamiento que los algoritmos de clave

secreta, aunque hay situaciones en las que su conveniencia compensa esta desventaja. Por ejemplo el DES (algoritmo de clave secreta) y el RSA (algoritmo de clave publica).

V.II. Comunicación de procesos en Sistemas Distribuidos.

La diferencia más importante entre un Sistema Distribuido y un sistema de un único procesador es la ***comunicación entre procesos***. En un sistema de un solo procesador la comunicación supone implícitamente la existencia de la memoria compartida, por el contrario en un Sistema Distribuido no existe la memoria compartida, y por ello toda la naturaleza de la comunicación entre procesos debe replantearse.

Los Sistemas Distribuidos requieren líneas de comunicaciones para interaccionar, que permitan a los procesos remotos implicados en un mismo trabajo:

- La transferencia de datos.
- La sincronización de operaciones o acciones.

Uno de los principales problemas a resolver en el diseño de los Sistemas Operativos Distribuidos, es la comunicación entre procesos. Para dar solución a este problema, se puede implementar el **paso de mensajes** como modelo de comunicación. Existen diversas formas de implementar este modelo, entre estas tenemos las "**llamadas a un procedimiento remoto**".

El rendimiento global de un Sistema Distribuido, tiene una dependencia crítica de los mecanismos de los subsistemas de comunicaciones utilizados para la intercomunicación de procesos. Y no depende únicamente de la optimización de los niveles bajos de comunicaciones, sino de de la implementación de la política o **modelos de comunicaciones** utilizados.

1. Modelos de comunicación.

Como se mencionó anteriormente, para dar solución al problema de la comunicación entre procesos, en los Sistemas Distribuidos se deben buscar nuevos métodos de comunicación, entre estos tenemos:

- Protocolos por capas (**OSI**).
- Modelo **multicast**, para comunicación entre grupos de procesos cooperantes.
- Modelo **cliente-servidor**, para comunicación entre parejas de procesos.
- Llamadas a procedimientos remotos.

1.1. Protocolos por capas.

La Organización Internacional de Estándares (**ISO**) diseño el modelo de interconexión de sistemas abiertos (**OSI**), como guía para la elaboración de estándares de dispositivos de computadoras en redes.

Los sistemas abiertos son aquellos preparados para comunicarse con cualquier otro sistema abierto mediante reglas estándar, que constituyen los protocolos y establecen el formato, contenido y significado de los mensajes recibidos y enviados.

A este modelo de red descriptivo, creado por ISO, se le denomina "***modelo de referencia para interconexión de sistemas abiertos***" (ISO OSI o modelo OSI), y proporcionó a los fabricantes un conjunto de estándares que aseguraron una mayor **compatibilidad** e **interoperabilidad** entre los distintos tipos de tecnología de red producidos por las empresas a nivel mundial.

El modelo en sí mismo no puede ser considerado una arquitectura, ya que no especifica el protocolo que debe ser usado en cada capa, sino que suele hablarse de modelo de referencia.

Este modelo está dividido en siete capas. Cada capa proporciona una **interfaz** con la otra capa por encima de ella; la interfaz consiste de un conjunto de operaciones para definir el servicio que la capa está preparada para ofrecer a sus usuarios.

El propósito de cada capa es el de ofrecer ciertos servicios a capas superiores (capa **n+1**) o inferiores (capa **n-1**); de tal forma de que no se tengan que ocupar de detalles inherentes a la capa **n**.

En el campo de las redes, por un estándar de facto, se espera o se pretende que la capa n interactué con la capa **n+1** o **n-1**; pero no con capas mas alejadas (**n+2**, **n-2**, etc.). En los protocolos de redes, este estándar es seguido al pie de la letra.

Estas capas se visualizan generalmente como un montón de bloques apilados o en ingles como un "*stack of blocks*", por lo que a esto se le conoce como el "*OSI Protocol Stack*" y cada protocolo de capa se puede cambiar independientemente de los demás, esto es de fundamental importancia, ya que confiere gran flexibilidad. (Ver Fig. 14).

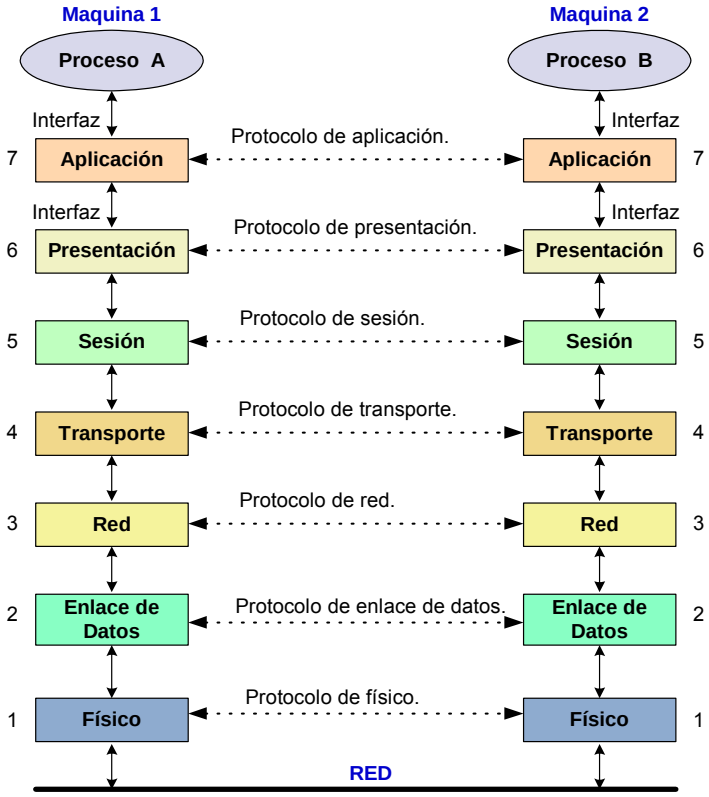


Fig. 14 Modelo OSI.

A continuación se presentará una descripción breve de cada una de estas capas.

Capa Física: Define las características físicas del medio de transmisión; de tipo mecánico, eléctrico y óptico (esto es, el tipo de medio a utilizar, el tamaño o forma de los conectores, el grosor del cable, el tipo de cable, el tipo de aislante, el voltaje de la interfase, la impedancia resistencia - nominal, etc.), además esta la señalización de la interfase (es decir, el como representar la información como un 0 y 1, por ejemplo, un 0 puede representarse como una señal entre 0 y 5 volts, y un 1 en una señal de entre 1 y -5 volts, por ejemplo).

Capa de enlace de datos: La función de esta capa es la de asegurar la transferencia de datos libres de error entre nodos adyacentes (sincronización a nivel de datos), además establece el control de acceso al medio. La capa de enlace de datos está dividida en dos subcapas: el control de acceso al medio (**MAC**) y el control de enlace lógico (**LLC**).

Capa de red: Incluye la capa de Red que se encarga de determinar las rutas adecuadas para llevar la información de un lado a otro (proporciona el enrutamiento); además, su funcionalidad es la de proporcionar una interfase para que la transferencia de datos sea idéntica de la tecnología del enlace de datos.

Capa de transporte: La capa de transporte vincula las capas de ***host*** (sesión, presentación y transporte) con las capas orientadas a la ***red*** (física, de enlace, de red y de transporte); permite la cohesión entre el host y la red, su función es la de asegurar una entrega confiable de la información a través de la red.

Capa de sesión: La capa de sesión tiene la responsabilidad de asegurar la entrega correcta de la información. Esta capa tiene que revisar que la información que recibe esta correcta; para esto, la capa de sesión debe realizar algunas funciones:

- La detección y corrección de errores.
- El controlar los diálogos entre dos entidades que se estén comunicando y definir los mecanismos para hacer las llamadas a procedimientos remotos (Remote Procedure Control - RPC).

Capa de presentación: El objetivo de la capa de presentación es encargarse de la representación de la información, de manera que aunque distintos equipos puedan tener diferentes representaciones internas de caracteres (ASCII, unicode, EBCDIC), números (little-endian tipo intel, big-endian tipo motorola), sonido o imágenes; los datos lleguen de manera reconocible.

Capa de aplicación: Ofrece a las aplicaciones (de usuario o no) la posibilidad de acceder a los servicios de las demás capas y define los protocolos que utilizan las aplicaciones para intercambiar datos, como correo electrónico, gestores de bases de datos y servidor de ficheros.

El modelo OSI distingue entre dos tipos generales de protocolos:

1. Orientados hacia las conexiones:

   En este tipo de protocolo, antes de intercambiar los datos, el emisor y el receptor:

   - Establecen en forma explícita una conexión.
   - Probablemente negocien el protocolo a utilizar.
   - Al finalizar, deben terminar la conexión.

     El teléfono es un sistema de comunicación orientado hacia la conexión.

2. Sin conexión:

   En este tipo de protocolo no es necesaria una configuración de antemano, ya que el emisor transmite el primer mensaje cuando está listo. El depósito de una carta en un buzón es un ejemplo de una comunicación sin conexión.

1.2. Modelo multicast.

El modelo de comunicación **multicast** consiste en el envío de un mismo mensaje desde un origen a un **grupo** de nodos de destino. Un grupo es una colección de procesos que actúan juntos (o de alguna forma determinada por el usuario) en cierto sistema.

Multicast no se debe confundir con **broadcast** (o difusión), que es el envío de un mensaje de forma que pueda ser escuchado por todos los nodos de la red, y que en Sistemas Distribuidos se utiliza con menor frecuencia.

En síntesis, el modelo de comunicación multicast se trata de una comunicación **uno-muchos** (un emisor, muchos receptores), que se distingue de la comunicación puntual o **punto a punto** (un emisor, un receptor). (Ver Fig. 6 y Fig. 7).

Motivos para enviar un mismo mensaje a un grupo de nodos:

- **Búsqueda de un recurso**. Un cliente puede enviar un mensaje a un grupo de procesos servidores, y el que realmente tenga el recurso buscado será el único en contestar.

- **Tolerancia a fallos**. Un cliente solicita un servicio muy importante mediante multicast. Uno o más servidores procesan la petición y contestan. El cliente toma la primera respuesta y deshecha las posteriores. De esta manera, se asegura una respuesta aunque falle un servidor.

- **Actualizaciones**. Cuando se quiere actualizar información común entre varios nodos –como tablas de encaminamiento o la hora– el proceso encargado del mantenimiento se lo envía a los demás mediante multicast.

Hay diversos algoritmos o métodos para el envío de mensajes en modo multicast, la elección del algoritmo de envío utilizado puede repercutir seriamente en el rendimiento general del sistema. También puede ayudar a mejorar la velocidad de envío, el disponer de cierto soporte hardware que, para el algoritmo que le convenga, posibilite el envío paralelo de las múltiples copias del mensaje.

Implementaciones de la comunicación en grupos.

La propiedad fundamental de todos los grupos es que cuando un mensaje se envía al propio grupo, todos los miembros del grupo lo reciben.

La implementación de la comunicación en grupos depende en gran medida del hardware.

- En ciertas redes es posible crear una dirección especial de red a la que pueden escuchar varias máquinas, cuando se envía un mensaje a una de esas direcciones se lo entrega automáticamente a todas las máquinas que escuchan a esa dirección. Esta técnica se denomina **multitransmisión**, y cada grupo debe tener una dirección de multitransmisión distinta.
- Las redes que no soportan multitransmisión operan con transmisión simple: esto significa que los paquetes que tienen cierta dirección se entregan a todas las máquinas. Cada máquina debe verificar, mediante su software, si el paquete va dirigido a ella, en caso negativo se descarta.

  Esta técnica se puede utilizar para implantar los grupos, pero es menos eficiente que la multitransmisión.
- Otra solución, es implantar la comunicación en grupo mediante la transmisión por parte del emisor de paquetes individuales a cada uno de los miembros del grupo, en vez de un paquete se precisan “n” paquetes.

  Esta técnica es una solución valida particularmente con grupos pequeños, pero es menos eficiente que las soluciones anteriores. El envío de un mensaje de un emisor a un único receptor se llama ***unitransmisión***.

Direccionamiento de grupos.

Un grupo debe tener una dirección

- Si el nivel de enlace soporta **multicasting**, se puede implementar una dirección de grupo como una dirección multicasting.
- Si la red soporta difusión, pero no multicasting, el mensaje debe ser recibido por todos los núcleos y extraer de él la dirección del grupo.
- Si ninguno de los procesos de la máquina es miembro del grupo, el mensaje es descartado. En otro caso, es pasado a todos los procesos que pertenecen al grupo.
- Si la red no soporta ni difusión ni multicasting, el núcleo de la máquina emisora tendrá que tener una lista de los procesos que pertenecen al grupo y enviar un mensaje a cada uno de los componentes del grupo.

Para que la comunicación en grupo sea fácil de usar y comprender, requiere de dos propiedades:

- **La atomicidad**: Significa que un mensaje enviado a un grupo debe ser recibido por todos los miembros del grupo o por ninguno de ellos.
- **El orden de los mensajes.**

1.3. Modelo Cliente-Servidor. (C-S).

Comparativa del modelo Cliente-Servidor y del modelo OSI. (Ver Fig. 15).

- Debido a la existencia de los encabezados de paquete en el modelo OSI (agregado por cada capa de nivel), se genera un "costo" adicional de transmisión. Cada envío de un mensaje genera:
  - Proceso en media docena de capas.
  - Preparación y agregado de encabezados en el camino hacia "abajo".
  - Eliminación y examen de encabezados en el camino hacia "arriba".

    Esto produce una carga de procesamiento de los protocolos significativa.
- El "modelo OSI" no dice nada acerca de la forma de estructurar al Sistema Distribuido.
- El "modelo cliente-servidor" tiene como idea fundamental la estructuración del S.O. como:
  - Un grupo de procesos en cooperación, llamados servidores, que ofrecen servicios a los usuarios.
  - Un grupo de procesos usuarios llamados clientes.
- El "modelo cliente-servidor" se basa en un "**protocolo solicitud / respuesta**":
  - Es sencillo y sin conexión.
  - No es complejo como OSI o TCP/IP.
  - El cliente envía un mensaje de solicitud al servidor pidiendo cierto servicio.
  - El servidor:
    - Ejecuta el requerimiento.
    - Regresa los datos solicitados o un código de error si no pudo ejecutarlo correctamente.
- No se tiene que establecer una conexión sino hasta que ésta se utilice.
- La pila del protocolo es más corta y por lo tanto más eficiente.
- Si todas las máquinas fuesen idénticas solo se necesitarían tres niveles de protocolos.

Lo descrito anteriormente se ve mejor en la siguiente imagen.

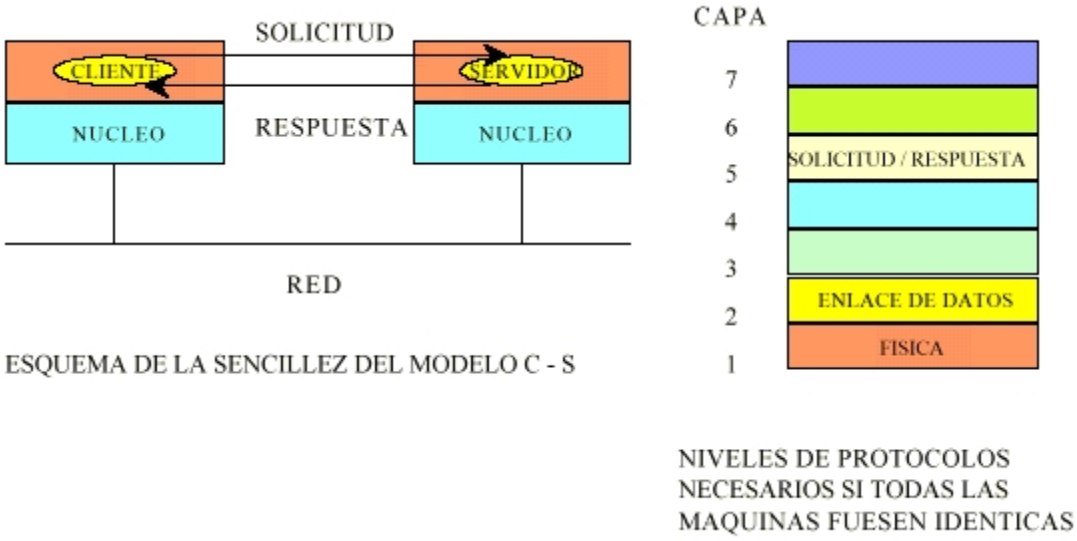


Fig. 15 Comparativa del modelo Cliente-Servidor y del modelo OSI.

En el desarrollo de la investigación, el modelo que se utilizará será el modelo de comunicación Cliente-Servidor. Este modelo de comunicaciones puede implementarse directamente mediante el mecanismo de paso de mensajes (operaciones enviar y recibir) del sistema operativo, pero normalmente se utiliza una construcción del nivel del lenguaje que le **abstrae** al programador de las operaciones de envío-espera-recepción.

Esta construcción, se conoce como **RPC** (Remote Procedure Call) o Llamada a Procedimiento Remoto, y esconde las operaciones de envío y recepción bajo el aspecto de la llamada convencional a una rutina o procedimiento. La implementación del modelo Cliente-Servidor se hará usando RPC's.

1.4. Llamadas a procedimientos remotos (RPC).

El modelo **Cliente-Servidor** es una forma conveniente de estructurar un S.O. distribuido, pero presenta algunas fallas. El concepto de Llamadas a Procedimientos Remotos lo presentaron por primera vez Virrey y Nelson en 1984, para solucionar los problemas del paso de datos entre maquinas y sistemas heterogéneos.

Las llamadas a procedimientos remotos tiene la misma semántica que las llamadas a procedimientos ordinarios, es decir, al realizar las llamadas, el llamante (proceso cliente) le pasa el control al procedimiento llamado (proceso servidor), y lo recupera cuando el procedimiento llamado le devuelve el resultado. Las RPC no son más que operaciones remotas disfrazadas de una interfaz procedural. Normalmente el proceso servidor duerme, esperando la llegada de un mensaje

de requerimiento. El proceso cliente se bloquea cuando envía el mensaje de requerimiento hasta recibir la respuesta. (Ver Fig. 16).

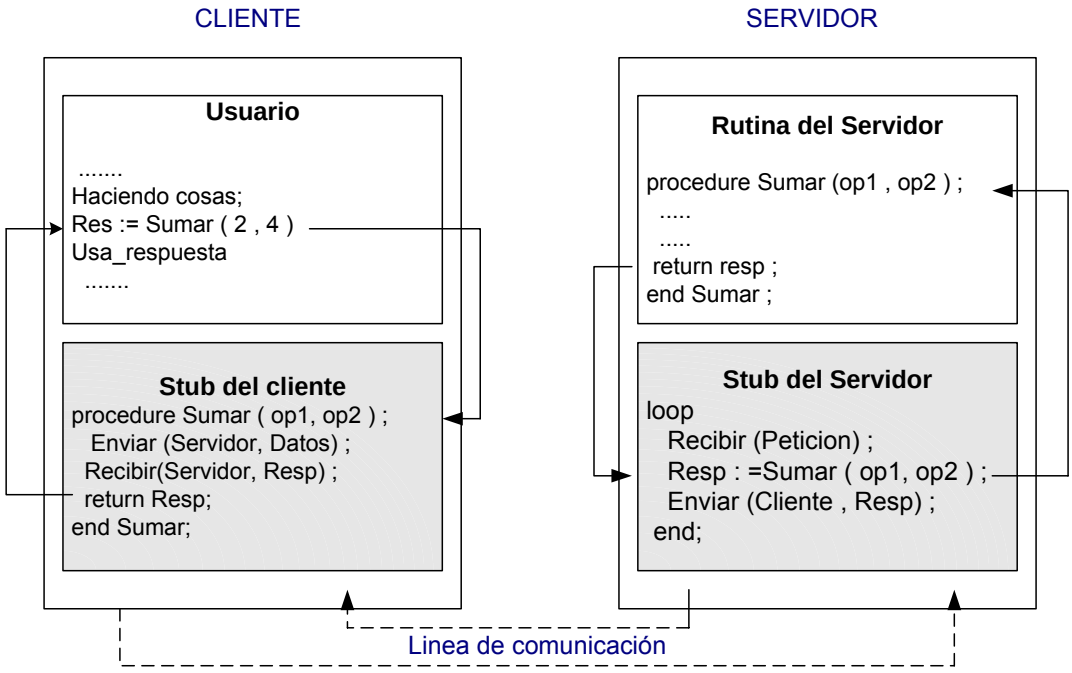


Fig. 16 Llamadas a Procedimientos Remotos.

Componentes del mecanismo de RPC.

Los componentes del mecanismo de las RPC se muestran en la imagen anterior, en ésta podemos ver que la aplicación cliente llama a una subrutina que se ejecuta en otra aplicación (la rutina de servicio en el servidor).

Si la aplicación cliente y servidor formaran parte del mismo proceso, el cliente llamaría directamente a la rutina que ejecuta el servicio, sin embargo, aquí el cliente tiene que llamar a otra subrutina denominada *stub del cliente*.

- **Stub de Cliente:** Subrutina que tiene exactamente la misma interfaz que la rutina de servicio del servidor (la que realiza realmente el servicio requerido por el usuario), pero esta implementada por código que solicita al servidor que ejecute la rutina del servicio, y que a continuación le devuelve al cliente los resultados que recibe del servidor. Primero el stub del cliente copia los parámetros de la pila al mensaje de petición, y a continuación le pide al módulo de comunicaciones que envié el mensaje de petición al servidor.

- **Stub de Servidor:** Recibe los mensajes de petición, toma los parámetros de éste, los pone en la pila y **termina por llamar a la verdadera rutina remota que realiza el servicio solicitado por el cliente**. Cuando esta rutina finaliza y devuelve el resultado, se repite la misma operación, pero en sentido contrario: El stub del servidor pone los parámetros de resultado en un mensaje de respuesta y, mediante primitivas de comunicación, le envía el mensaje al stub del cliente. Este saca el resultado del mensaje y se lo devuelve como parámetro de salida al cliente que lo llamó.
- **Proceso cliente**.
- **Proceso Servidor**.

Tareas de las RPC.

El software que soporta las llamadas a procedimientos remotos debe ocuparse de tres importantes tareas.

- **Búsqueda del Servidor:** Este proceso, conocido como ***binding***, consiste en asignar el servidor apropiado más conveniente para el servicio que requiere el cliente. Esta tarea se abordará con mayor profundidad posteriormente.

- **Gestión de la comunicación:** Esta tarea consiste simplemente en la transmisión y recepción de los mensajes de petición y de respuesta.

- **La interfaz del Servidor:** Es la integración del mecanismo de la RPC con los programas del cliente y del servidor, escrito en lenguajes de programación convencionales. Esto incluye la serialización (**marshalling** y **unmarshalling**) de los parámetros, a través de un lenguaje de representación externa (**XDR**), que se pasan entre el cliente y el servidor, así como el establecimiento, en el servidor, de un mecanismo (**dispatcher**) que reparta las peticiones entre las rutinas de servicio apropiadas. (Ver Fig. 17).

La definición de la interfaz de servicio es la base sobre la que se construye el resto del software que necesita los programas del cliente y del servidor para permitir la llamada a procedimientos remotos.

En el sistema completo va a haber un proceso cliente y un proceso servidor, y cada uno consta de los siguientes elementos:

- **Proceso Cliente:**
  - Programa del usuario.

- Stub del cliente.

- **Proceso Servidor:**
  - Stub del servidor (incluye el repartidor de peticiones o dispatcher).
  - Rutinas de servicio del Servidor (las que realizan la operación solicitada).

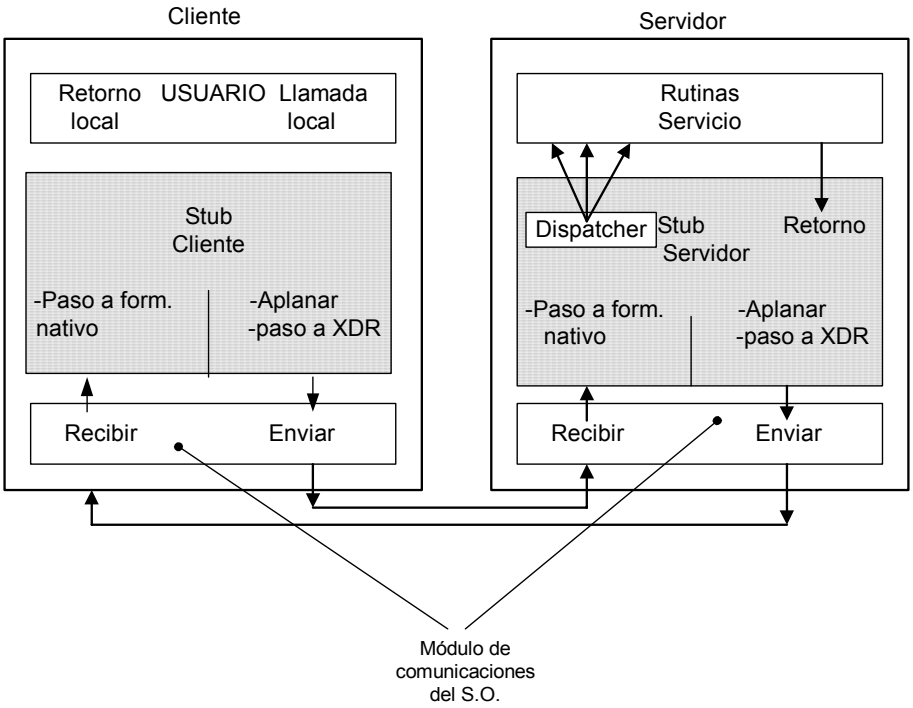


Fig. 17 Dispatcher.

Como se puede ver en el gráfico, ambos stubs, del cliente y del servidor, se apoyan en los servicios de comunicaciones que ofrece el sistema operativo, y uno de sus cometidos básicos es el paso de parámetros entre cliente y servidor (el cual será expuesto posteriormente).

Esta tarea encargada de la interfaz debe preocuparse de la construcción del programa cliente y del programa servidor. Para ello, debe generar el software apropiado y juntar todos los componentes para formar los dos programas.

El programa del usuario y las rutinas de servicio del servidor vienen dadas, así como el modulo de comunicación, que lo proporciona el sistema operativo. Falta por construir los stubs del cliente y del servidor. Posteriormente se explicará la forma en cómo se realiza la **generación de stubs**, también conocida como generación de interfaces.

Paso de parámetros.

Cuando el stub del cliente pasa los parámetros de entrada al stub del servidor, se presenta un problema: Los datos en los programas suelen estar representados mediante estructuras de datos, mientras que en un mensaje la información debe ser una secuencia de bytes, por lo tanto, se debe establecer un mecanismo para "**aplanar**" o **serializar** los parámetros y pasarlos al mensaje. "Aplanar" significa tomar cualquier estructura de datos y reordenarla como una secuencia de bytes.

Aunque el "aplanamiento"(marshalling, en ingles) podría ser una tarea fácil, recordemos que los procesos se ejecutan en maquinas distintas, las cuales utilizan diferentes representaciones de los datos que manipulan. Esto quiere decir, que si un dato se aplana en un proceso, se pone en un mensaje y se envía a otro proceso de la red (en una maquina distinta), es muy posible que el formato en el que se reciben los datos no sea el que espera el receptor, por lo que el contenido del mensaje no podría entenderse o malinterpretarse.

Para evitar este problema de formato de representación de los datos, los procesos comúnmente deben ponerse previamente de acuerdo en el formato en el que se van intercambiar los datos. Existen tres alternativas:

- Antes de la transmisión, los datos se convierten a un formato genérico conocido por todos los procesos (representación externa de datos). En el receptor se convierten al formato local de su arquitectura. Algunos estándares de representación externa de datos son: XDR de Sun, Courier de Xerox y ASN 1.
- Para la comunicación entre dos ordenadores con arquitectura común, el paso anterior puede omitirse. Esto requiere que antes de la transmisión de los parámetros (al establecer la sesión de comunicación), los dos extremos negocien si se requiere pasar los datos a un formato genérico o no.
- Otra posibilidad consiste en transmitir los datos en su formato nativo (el de la arquitectura del transmisor junto con un identificador del tipo de arquitectura subyacente. El receptor, consultando el tipo de arquitectura utilizado, decide si es necesario convertir los datos recibidos o no.

Generación de stubs.

A partir de la interfaz de las rutinas de servicio (o del stub del cliente), puede construirse manualmente la implementación de los stubs del cliente y el servidor, no obstante, puesto que lo que hay que hacer es muy mecánico, parece razonable automatizar el proceso.

Debido a que en los Sistemas Distribuidos los componentes son heterogéneos, se nos presenta un problema; los procesos cliente y servidor, podrían estar programados en lenguajes diferentes, apoyados en sistemas operativos variados y ejecutándose sobre procesadores de distinta arquitectura. Por otra parte el servidor debe ofrecer una interfaz genérica para todos los clientes.

La única forma de solucionar esto es que el servidor no ofrezca la interfaz de sus servicios acorde a un lenguaje de programación específico, si no en un lenguaje genérico al que se le denomina Lenguaje de Definición de Interfaces (**IDL**).

Los lenguajes de definición de interfaces suelen estar derivados de otros lenguajes de programación, pero para construir los stubs añaden cierta información necesaria que no todos los lenguajes de programación proporcionan. Esta información adicional, típicamente es: el sentido de los parámetros (entrada, salida, entrada/salida), discriminantes para registros variantes (o uniones) e información sobre la longitud de los vectores o matrices que se pasan como parámetros.

Ejemplos de IDL's son: NCS (de la OSF), XDR de Sun, Courier (de Xerox), MiG (de Mach).

A partir de una definición de una interfaz genérica (realizada en IDL) se deben generar las interfaces y los cuerpos o implementaciones apropiadas para los lenguajes del servidor y de cada cliente.

Usando una definición de una interfaz genérica en IDL y un compilador de interfaces, se pueden realizar automáticamente las siguientes tareas:
Generacin de los stubs del cliente (y su fichero de interfaz) para todos los perfiles de los procedimientos definidos en la interfaz. El stub se compilará y se montará con el programa del cliente.

Generacin de los stubs del servidor con el repartidor (o dispatcher) para todas las operaciones ofrecidas por el servidor. El stub y el repartidor (incluido en el stub) se montaran con el programa del servidor.

Aprovechando la definición del perfil de los procedimientos de interfaz (que definen los tipos de los parámetros de entrada y salida), se generan las acciones apropiadas para la serialización correspondiente a cada procedimiento de interfaz en los stubs del cliente y del servidor.

Generación del fichero de definición de las operaciones ofrecidas, para el lenguaje correspondiente en el que están implementadas en el servidor. Los cuerpos correspondientes a estas definiciones las proporciona el programador de las operaciones del servidor.

La utilización de una definición común de la interfaz en el proceso de generación de los stubs del cliente y del servidor y el fichero de definición de los servicios del servidor, asegura que los tipos de los argumentos y de los resultados manejados por el cliente se ajustan a los definidos por el servidor.

Binding.

En la definición de una interfaz, los clientes y servidores indican un nombre o identificador textual para referirse al servicio solicitado. Sin embargo los mensajes de petición de los clientes deben dirigirse a un puerto o dirección concreta de un servidor.

El mecanismo de **binding** (ligadura) establece una correspondencia entre el nombre de un objeto y su edentificador de comunicación. Esta correspondencia debe obtenerse cada vez que un programa cliente solicita un servicio remoto. El formato del identificador de comunicación depende del entorno de ejecución; por ejemplo; en un entorno Unix el identificador será una dirección de un socket, compuesto por una dirección internet más un número de puerto. En sistemas basados en microkernel, como Mach, Amoeba o Chorus, será simplemente un identificador de puerto independientemente de la ubicación.

Este mecanismo de ligadura es esencial cuando los identificadores de direcciones del servidor contienen la dirección del ordenador del servidor, pues evita tener que recompilar los programas clientes cada vez que cambian la dirección de un servidor (suponiendo que le sea fácil al usuario conocer la dirección). Si el programa del cliente no incluye la dirección del servidor que necesita, nunca habrá que recompilar por los cambios de dirección del servidor.

En un Sistema Distribuido un binder es un servicio independiente que mantiene una tabla con las correspondencias entre nombres y direcciones de servidores, por lo tanto, no es más que un **servidor de nombres**.

El binder esta pensado para que los servidores pongan su dirección a disposición de sus clientes potenciales. Sus procedimientos típicos podrían ser:

- procedure Registrarse (Servicio: string; Puerto: port; Versión: integer);

El binder anota en sus tablas el nombre, dirección y versión de un servidor.

- procedure Retirarse (Servicio: string; Puerto: port; Versión: integer);
  El binder elimina una entrada su tabla.
- procedure Buscar (Servicio: string; Versión: integer)
  El binder busca el nombre del servicio indicado y devuelve su dirección si coincide la versión.

Cuando un **servidor empieza** a ejecutarse, debe enviar un mensaje al binder para registrar su servicio, versión y dirección. Cuando termina, debe comunicarlo para impedir intentos infructuosos de los clientes.

Cundo un **cliente comienza** su ejecución, debe solicitar al binder la dirección del servidor que se le va a asignar, y utilizar tal dirección hasta que **falle la comunicación**, en cuyo momento debe volver a contactar con el binder para solicitar la nueva dirección.

El propósito del **número de versión** es asegurar que el cliente utiliza la versión apropiada del programa servidor. Si se modifica el programa servidor, alterando el perfil de sus parámetros, con el número de versión se asegura que no se generen mensajes de petición con datos que el servidor podría malinterpretar.

El procedimiento Buscar podría devolver un status para que en caso de no poder proporcionar la dirección del servidor, indique el motivo (por ejemplo, que no hay registrado ningún servicio con ese nombre, o que la versión solicitada esta anticuada).

No debe olvidarse que cuando un **servidor cambie su dirección**, debe informarlo al binder de su nueva dirección.

Transparencia de RPC.

- **Transparencia sintáctica:** una llamada a procedimiento remoto debe tener la misma sintaxis que una llamada local.
- **Transparencia semántica:** la semántica de un RPC es la misma que para una llamada local.

VI. METODOLOGÍA DEL TRABAJO.

El método de desarrollo que se ha seleccionado para la elaboración de la Aplicación: Chat Multiusuario bajo entorno Linux, es el **Método del Ciclo de Vida Clásico** o en **Cascada** (Ver Fig. 18), ya que resulta conveniente en la identificación de las actividades o etapas a seguir en la elaboración de la Aplicación.

Las etapas ha desarrollar durante el proceso investigativo son:

1. Investigación preliminar o Ingeniería del Sistema.
2. Análisis de Requisitos del Software.
3. Diseño del Sistema.
4. Codificación.
5. Prueba.
6. Implementación y evaluación.

Cada una de estas etapas lleva asociada una serie de tareas que deberán realizarse y una serie de documentos que serán las salidas de cada una de estas fases y que servirán de entrada a la siguiente fase, de esta manera se logra progresar a través del análisis, diseño, codificación, prueba e implementación, sin llegar a la etapa de mantenimiento, pues la Aplicación no será instalado en una empresa, para darle mantenimiento posterior.

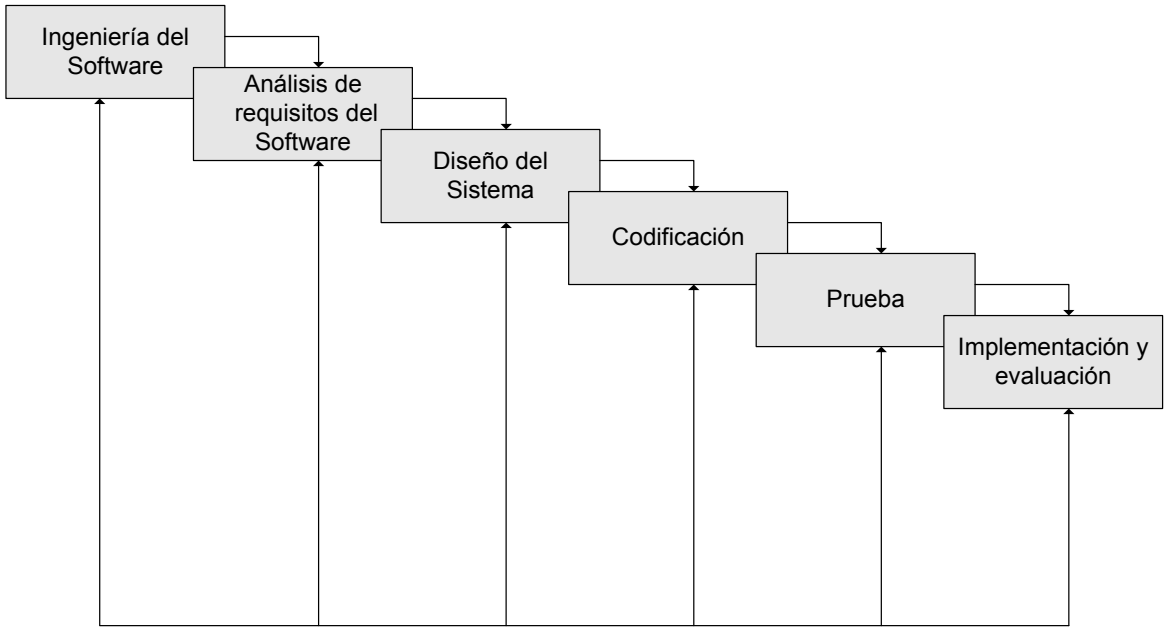


Fig. 18 Metodología del Trabajo.

1. **Investigación preliminar o Ingeniería del Sistema:** En esta fase se realiza el estudio de factibilidad técnica, económica y operacional de la Aplicación.

2. **Análisis de Requisitos del Software:** Se debe comprender la funcionalidad que tendrá la Aplicación, así como su rendimiento y la interfaz requerida.

3. **Diseño del Sistema:** Se diseñan procedimientos precisos para especificar la interfaz que utilizará la Aplicación, tomando como herramienta el Diseñador de interfaces (Diseñador Qt) y el IDE para desarrollo en KDE (KDevelop: KDE/C++), además de permitir la comunicación entre dos ó más individuos, implementando el diseño del Cliente y del Servidor.

4. **Codificación:** El diseño se traduce de forma precisa a la computadora, utilizando un lenguaje de programación de alto nivel, en nuestro caso C/C++. Se debe codificar el programa Cliente y Servidor, además de generar la interfaz gráfica de la Aplicación Cliente y Servidor.

5. **Prueba:** Una vez generado el código respectivo, comienza la fase de prueba de la Aplicación. Confirmando que se puede llevar a cabo una comunicación a través del Chat Multiusuario.

6. **Implementación y evaluación:** Una vez que el código de los programas Cliente y Servidor ha pasado la fase de prueba y se han hecho las correcciones en caso de ser necesario, se procede a la instalación de la Aplicación. Debido a que la Aplicación Chat Multiusuario no está diseña para implementarse en una compañía o empresa, su evaluación consistirá en el uso de la Aplicación por un número determinado de usuarios (egresados de la Carrera de Ingeniería en Sistemas) durante un período predefinido, la cual se llevará a cabo en el Departamento de Computación.

VII. RECURSOS DISPONIBLES Y NECESARIOS.

1. Recursos Hardware.

a. Tres computadoras conectada a la red local del laboratorio y con acceso a Internet, con las características siguientes:
   - Procesador: Intel Pentium IV de 1.8 GHz.
   - Disco Duro; Seagate 20 GB.
   - Memoria Principal: HP 256 MB, DDR.
   - Unidad de CD/ROM, de 56x.
   - 4 puertos USB.

b. Red local Ethernet 10/100 Mbps.

c. Las 3 máquinas están conectadas a un único concentrador 3Com, de 16 puertos, vía interfaz RJ45.

d. Impresora de burbuja HP Deskjet 3535.

2. Recursos Software.

a. Suse Linux 10.0. Linux kernel versión: 2.6.13-15.

   Sistema Operativo.

b. rpcgen

   Herramienta utilizada en la generación código C, para implementar el Protocolo XDR.

c. kwrite.

   Editor de ficheros fuentes para la Aplicación Servidor

d. gcc.

   Colección de compiladores integrados, utilizado para generar el programa ejecutable de la Aplicación Servidor.

e. Star Office (Editores de texto enriquecidos, hojas de cálculo y editores de presentación).

   Herramientas utilizadas para la redacción del informe final.

f. Firefox o Netscape.

   Navegadores Web utilizados para la recopilación de información.

g. Diseñador de interfaces (Diseñador Qt). Versión 3.3.4.

   Herramienta utilizada en el diseño de la interfaz de usuario.

h. IDE para desarrollo en KDE (KDevelop: KDE/C++).Versión 3.2.2.

   Entorno integrado de desarrollo, que facilita la creación de aplicaciones C/C++, KDE/Qt. Utilizado en la Aplicación Cliente.

VIII. FASE DE ANÁLISIS.

1. Introducción.

1.1. Propósito.

Definición del conjunto de especificaciones de los requisitos del software que debe cumplir la Aplicación Chat Multiusuario bajo entorno Linux, que sirva para la comunicación entre usuarios, a través de llamadas a procedimientos remotos (RPC).

1.2. Alcance.

El nombre con el que se conocerá a esta Aplicación será **Chat Multiusuario**.

La Aplicación realizará las siguientes funciones:

a. Intercambio de texto plano entre usuarios.
b. Intercambio de imágenes con extensiones .jpg, .jpeg entre usuarios.
c. Intercambio de archivos de sonido con extensiones .mp3 entre usuarios.

1.3. Definiciones, acrónimos y abreviaturas.

Texto plano: Texto escrito por el usuario a través del teclado.

Nombre de usuario: Nombre con el que se conocerá al cliente en la Sala de Chat.

Id de usuario. Valor único de auto incremento que identifica a cada usuario en el Servidor.

Ruta absoluta: Ubicación del fichero de imagen o sonido a ser transferido.

Buzón: Memoria principal de almacenamiento, implementada mediante tuberías, por cada usuario conectado; que permitirá el intercambio de información en la Sala de Chat.

1.4. Referencias.

Tanenbaum, Andrew S. “Sistemas Operativos Distribuidos”. Prentice Hall., 1996.

Molina, Eduardo Santiago. “Plan docente para la asignatura de Sistemas Operativos II”, 2002.

Página oficial de la API del Diseñador de interfaces (Diseñador Qt).
http://doc.troltech.com

Página oficial de IDE para desarrollo en KDE (KDevelop: KDE/C++).
http://www.kdevelop.org

1.5. Visión general.

Primero se realizará una descripción general de la Aplicación a desarrollar y posteriormente se abordarán cada uno de los requisitos específicos.

2. Descripción general.

2.1. Relaciones de la Aplicación.

La Aplicación Chat Multiusuario se ejecutará como una Aplicación Cliente-Servidor a través de la red del Departamento de Computación. La instalación inicial constará de tres ordenadores, dos clientes y un servidor.

El equipo en el que se desarrollará la Aplicación Cliente posee las siguientes características:

a. Procesador: Intel Pentium IV de 1.8 GHz.
b. Disco Duro; Seagate 20 GB.
c. Memoria Principal: HP 256 MB, DDR.

El equipo en el que se desarrollará la Aplicación Servidor posee las siguientes características:

a. Procesador: Intel Pentium IV de 1.8 GHz.
b. Disco Duro; Seagate 20 GB.
c. Memoria Principal: HP 256 MB, DDR.

2.2. Funciones de la Aplicación Cliente.

La Aplicación Cliente debe contener todas las tareas que realiza el usuario al conectarse al servidor y poder conversar a través de texto e intercambiar imágenes y ficheros de sonido con otros usuarios también conectados al Servidor:

a. Conexión a la Sala de Chat.
b. Conversar a través de texto.
c. Transferir imágenes.
d. Transferir ficheros de sonido.
e. Desconexión de la sala de Chat.

2.3. Funciones de la Aplicación Servidor.

La Aplicación Servidor debe tener todas las tareas necesarias que le permitan recibir una petición, analizar qué procedimiento debe realizar de acuerdo a la petición, invocar dicho

procedimiento y por último recoger los resultados y devolver un mensaje de respuesta al cliente.

a. Iniciar el Servidor.
b. Aceptar la conexión de un cliente.
c. Procesar las peticiones posibles que pudiera realizar un cliente.
d. Cerrar la conexión del cliente.
e. Detener el Servidor.

2.4. Características del usuario.

Los usuarios finales de la Aplicación Chat Multiusuario serán personas con o sin experiencia en el área de la informática.

2.5. Restricciones generales.

a. El lenguaje a utilizar será C, C++, XDR.
b. El máximo número de usuarios conectados será de 10.
c. Los clientes sólo podrán recibir un fichero de imagen o sonido a la vez, esto para simplificar el diseño de la Aplicación Cliente.
d. La aplicación Servidor no constará de una interfaz gráfica.
e. Para simplificar los filtros empleados en la Aplicación Cliente sólo podrán ser transferidos archivos con las siguientes extensiones: .mp3, .jpg, .jpeg.

3. Requisitos específicos.

3.1. Requisitos funcionales de la Aplicación Cliente.

3.1.1. Conexión a la sala de Chat.

3.1.1.1. Especificación.

3.1.1.1.1. Introducción.

Cuando el cliente desee conectarse deberá presionar el botón Conectar, luego se le pedirá su Nombre de Usuario.

3.1.1.1.2. Entradas.

Pulsación del botón Conectar.

Nombre de usuario.

3.1.1.1.3. Proceso.

Se mostrará por pantalla una ventana que contiene una caja de texto donde se deberá introducir el Nombre de Usuario.

3.1.1.1.4. Salida.

Se mostrará un mensaje de bienvenida con el Nombre de Usuario si la conexión se ha realizado con éxito, o un mensaje de error para indicar que se ha producido un error de conexión o del sistema.

3.1.1.2. Interfaces externas.

3.1.1.2.1. Interfaces de usuario.

La captura del Nombre de Usuario.

3.1.1.2.2. Interfaces Hardware.

Se podrá utilizar cualquier computador conectado al Servidor.

3.1.1.2.3. Interfaces Software.

El proceso interactuará directamente con el Servidor.

3.1.1.2.4. Interfaces de comunicación.

Existe una interfaz de comunicación en la Aplicación Cliente para interactuar directamente con el Servidor.

3.1.2. Conversar a través de texto.

3.1.2.1. Especificación.

3.1.2.1.1. Introducción.

Después de establecida la conexión a la Sala de Chat, se podrá compartir texto para el envió de mensajes introducidos por el teclado.

3.1.2.1.2. Entradas.

Mensaje introducido por el teclado.

Pulsación del botón Enviar o pulsación de la tecla Enter.

3.1.2.1.3. Proceso.

Se mostrará por pantalla un área de texto (caja de texto multilínea), donde se podrá introducir los mensajes por teclado para ser enviados y mostrados en la Sala de Chat.

3.1.2.1.4. Salida.

Si todo el proceso se ha realizado correctamente se mostrará el mensaje enviado por el usuario y su nombre de usuario en un área de texto (caja de texto multilínea) o un mensaje de error para indicar que se ha producido un error de conexión o del sistema.

3.1.2.2. Interfaces externas.

3.1.2.2.1. Interfaces de usuario.

La captura del mensaje de texto se hará de forma interactivo.

3.1.2.2.2. Interfaces Hardware.

Se podrá utilizar cualquier computador conectado al Servidor.

3.1.2.2.3. Interfaces Software.

El proceso interactuara directamente con el servidor.

3.1.2.2.4. Interfaces de comunicación.

Existe una interfaz de comunicación en la Aplicación Cliente para interactuar directamente con el servidor.

3.1.3. Transferir imágenes.

3.1.3.1. Especificación

3.1.3.1.1. Introducción.

Después de establecida la conexión a la Sala de Chat, se podrá compartir imágenes con los otros usuarios conectados.

3.1.3.1.2. Entradas.

Por pantalla:

Pulsar sobre uno de los elementos de la lista “Usuarios en Línea”.

Pulsar el botón Enviar Imagen.

Se capturará de forma iterativa la ruta absoluta de la imagen que se desea transferir, a través de un diálogo estándar correspondiente a abrir un fichero.

3.1.3.1.3. Proceso.

Se selecciona de la lista de “Usuarios en Línea” el nombre del usuario al que se debe enviar la imagen y después se pulsa el botón Enviar Imagen, se abre el fichero especificado en la entrada y se envía al usuario correspondiente.

3.1.3.1.4. Salida.

Si todo el proceso se ha realizado correctamente, se mostrará un mensaje de notificación, mediante un diálogo indicando que la imagen ha sido transferida, o un mensaje de error para indicar que se ha producido un error de conexión o del sistema.

3.1.3.2. Interfaces externas.

3.1.3.2.1. Interfaces de usuario.

La especificación de la ruta del fichero de imagen se hará de forma interactiva.

3.1.3.2.2. Interfaces Hardware.

Se podrá utilizar cualquier computador conectado al Servidor.

3.1.3.2.3. Interfaces Software.

El proceso interactuará directamente con el Servidor.

3.1.3.2.4. Interfaces de comunicación.

Existe una interfaz de comunicación en la Aplicación Cliente para interactuar directamente con el Servidor.

3.1.4. Transferir ficheros de sonido.

3.1.4.1. Especificación

3.1.4.1.1. Introducción.

Después de establecida la conexión a la Sala de Chat, se podrá compartir ficheros de sonido con los otros usuarios conectados.

3.1.4.1.2. Entradas.

Por pantalla:

Pulsar sobre uno de los elementos de la lista “Usuarios en Línea”.

Pulsar el botón Enviar Sonido.

Se capturará de forma iterativa la ruta absoluta del fichero de sonido que se desea transferir, a través de un diálogo estándar correspondiente a abrir un fichero.

3.1.4.1.3. Proceso.

Se selecciona de la lista de “Usuarios en Línea” el nombre del usuario al que se debe enviar el fichero de sonido y después se pulsa el botón Enviar Sonido, se abre el fichero especificado en la entrada y se envía al usuario correspondiente.

3.1.4.1.4. Salida.

Si todo el proceso se ha realizado correctamente se mostrará un mensaje de notificación, mediante un diálogo, indicando que el fichero ha sido transferido, o un mensaje de error para indicar que se ha producido un error de conexión o del sistema.

3.1.4.2. Interfaces externas.

3.1.4.2.1. Interfaces de usuario.

La especificación de la ruta del fichero de sonido se hará de forma interactiva.

3.1.4.2.2. Interfaces Hardware.

Se podrá utilizar cualquier computador conectado al Servidor.

3.1.4.2.3. Interfaces Software.

El proceso interactuará directamente con el Servidor.

3.1.4.2.4. Interfaces de comunicación.

Existe una interfaz de comunicación en la Aplicación Cliente para interactuar directamente con el Servidor.

3.1.5. Desconexión de la sala de Chat.

3.1.5.1. Especificación

3.1.5.1.1. Introducción.

Cuando el usuario desee desconectarse de la Sala de Chat deberá pulsar el botón Desconectar. También se podrá desconectar de la Sala de Chat pulsando sobre el botón X, en la parte superior derecha de la ventana, para lo cual deberá aceptar la desconexión.

3.1.5.1.2. Entradas.

Por pantalla:

Presionar el botón de Desconectar o pulsación del botón X.

3.1.5.1.3. Proceso.

El usuario deberá desconectarse de la Sala de Chat presionando el botón Desconectar o pulsando el botón X.

3.1.5.1.4. Salida.

Si todo el proceso se ha realizado correctamente se mostrará mediante un diálogo que se ha desconectado de la Sala de Chat.

3.1.5.2. Interfaces externas.

3.1.5.2.1. Interfaces de usuario.

La desconexión se hará de forma interactiva.

3.1.5.2.2. Interfaces Hardware.

Se podrá utilizar cualquier computador conectado al Servidor.

3.1.5.2.3. Interfaces Software.

El proceso interactuará directamente con el Servidor.

3.1.5.2.4. Interfaces de comunicación.

Existe una interfaz de comunicación en la Aplicación Cliente para interactuar directamente con el Servidor.

3.2. Requisitos funcionales de la Aplicación Servidor.

3.2.1. Iniciar el Servidor.

3.2.1.1. Especificación

3.2.1.1.1. Introducción.

El administrador del Chat primero tendrá que iniciar el Servidor para que los usuarios puedan conectarse.

3.2.1.1.2. Entradas.

Por pantalla: Ejecución del programa desde la línea de orden.

3.2.1.1.3. Proceso.

El administrador tendrá que ejecutar el programa desde la línea de orden, para que los clientes puedan conectarse.

3.2.1.1.4. Salida.

Por pantalla: Ninguna.

3.2.1.2. Interfaces externas.

3.2.1.2.1. Interfaces de usuario.

El programa Servidor interactuará con el programa Cliente.

3.2.1.2.2. Interfaces Hardware.

Se podrá utilizar cualquier computador como Servidor en espera de que un cliente se conecte.

3.2.1.2.3. Interfaces Software.

El proceso interactuará directamente con el programa Cliente.

3.2.1.2.4. Interfaces de comunicación.

Existe una interfaz de comunicación en la Aplicación Servidor para interactuar directamente con el cliente.

3.2.2. Aceptar la conexión de un cliente.

3.2.2.1. Especificación

3.2.2.1.1. Introducción.

Una vez activado el Servidor, podrá recibir solicitudes de conexión de los clientes.

3.2.2.1.2. Entradas.

Solicitud de conexión del cliente, la cual incluye nombre de usuario y el Servidor se encarga de retornarle al usuario su Id. de usuario.

3.2.2.1.3. Proceso.

Se acepta la solicitud de conexión del cliente y se devuelve el Id de usuario correspondiente a dicho cliente, en caso de que la solicitud no sea aceptada se devuelve un mensaje de error.

3.2.2.1.4. Salida.

Por pantalla: Ninguna.

3.2.2.2. Interfaces externas.

3.2.2.2.1. Interfaces de usuario.

El programa Servidor interactuará con el programa Cliente.

3.2.2.2.2. Interfaces Hardware.

Se podrá utilizar cualquier computador como Servidor en espera de que un cliente se conecte.

3.2.2.2.3. Interfaces Software.

El proceso interactuará directamente con el programa Cliente.

3.2.2.2.4. Interfaces de comunicación.

Existe una interfaz de comunicación en la Aplicación Servidor para interactuar directamente con el cliente.

3.2.3. Procesar las peticiones posibles que pudiera realizar un cliente.

3.2.3.1. Especificación

3.2.3.1.1. Introducción.

Una vez activado el Servidor, éste podrá recibir y procesar las diferentes peticiones que puedan hacer los clientes.

3.2.3.1.2. Entradas.

Peticiones de los clientes.

3.2.3.1.3. Proceso.

El Servidor recibe peticiones, verifica si es una petición soportada, y en caso de serlo, procesa dicha petición y retorna un determinado resultado al cliente.

3.2.3.1.4. Salida.

Por pantalla: Ninguna.

3.2.3.2. Interfaces externas.

3.2.3.2.1. Interfaces de usuario.

El programa Servidor interactuará con el programa Cliente.

3.2.3.2.2. Interfaces Hardware.

Se podrá utilizar cualquier computador como Servidor en espera de que un cliente se conecte.

3.2.3.2.3. Interfaces Software.

El proceso interactuará directamente con el programa Cliente.

3.2.3.2.4. Interfaces de comunicación.

Existe una interfaz de comunicación en la Aplicación Servidor para interactuar directamente con el cliente.

3.2.4. Cerrar la conexión del cliente.

3.2.4.1. Especificación

3.2.4.1.1. Introducción.

Una vez que el usuario haya solicitado la desconexión, el Servidor deberá dar de baja a dicho usuario.

3.2.4.1.2. Entradas.

Petición de desconexión del usuario.

3.2.4.1.3. Proceso.

El Servidor acepta la petición de desconexión del cliente y le dará de baja a dicho cliente cerrando el buzón asociado a éste.

3.2.4.1.4. Salida.

Por pantalla: Ninguna.

3.2.4.2. Interfaces externas.

3.2.4.2.1. Interfaces de usuario.

El programa Servidor interactuará con el programa Cliente.

3.2.4.2.2. Interfaces Hardware.

Se podrá utilizar cualquier computador como Servidor en espera de que un cliente se conecte.

3.2.4.2.3. Interfaces Software.

El proceso interactuará directamente con el programa Cliente.

3.2.4.2.4. Interfaces de comunicación.

Existe una interfaz de comunicación en la Aplicación Servidor para interactuar directamente con el cliente.

3.2.5. Detener el Servidor.

3.2.5.1. Especificación

3.2.5.1.1. Introducción.

El administrador del Chat tendrá la opción de no brindar servicios y detener el Servidor.

3.2.5.1.2. Entradas.

Por pantalla: Pulsar las teclas Ctrl. + c.

3.2.5.1.3. Proceso.

Por pantalla el administrador tendrá la oportunidad de detener el Servidor para que no brinde más sus servicios.

3.2.5.1.4. Salida.

Por pantalla: Ninguna.

3.2.5.2. Interfaces externas.

3.2.5.2.1. Interfaces de usuario.

El programa Servidor interactuará con el programa Cliente.

3.2.5.2.2. Interfaces Hardware.

Se podrá utilizar cualquier computador como Servidor en espera de que un cliente se conecte.

3.2.5.2.3. Interfaces Software.

El proceso interactuará directamente con el programa Cliente.

3.2.5.2.4. Interfaces de comunicación.

Existe una interfaz de comunicación en la Aplicación Servidor para interactuar directamente con el cliente.

3.3. Requisitos funcionales.

Requisitos estáticos: el número de terminales o de usuarios trabajando simultáneamente en este sistema será limitado.

3.4. Restricciones de diseño.

3.4.1. Atributo.

3.4.1.1. Seguridad.

Tanto en la Aplicación Cliente como en la Aplicación Servidor, no se brinda ninguna seguridad en la transferencia de datos. Por tanto, se omiten tareas como: integridad de los datos enviados, autentificación de los usuarios, confidencialidad entre usuarios, encriptamiento de datos, interferencia en las señales, caídas en la red.

FASE DE DISEÑO.

IX. FASE DE DISEÑO.

1. Diseño de Datos.

1.1. Diagrama de clases de la Aplicación Cliente.

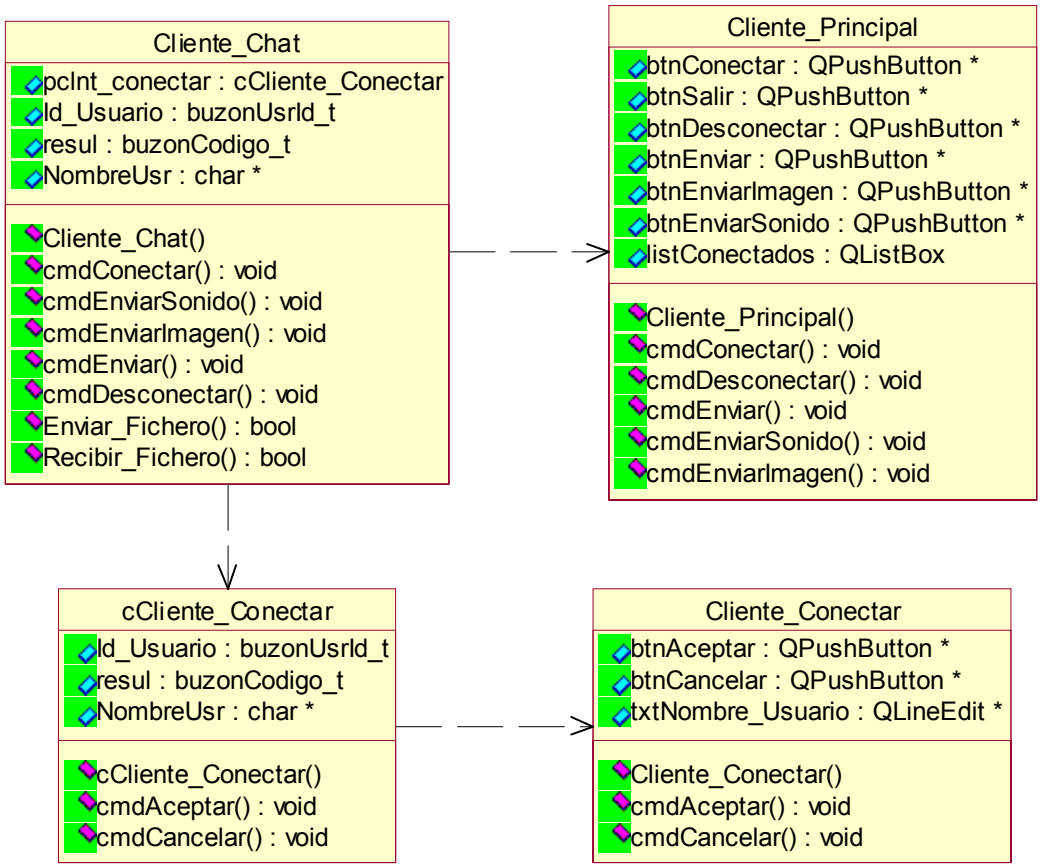


Fig. 19 Diagrama de Clases.

La clase principal de la aplicación Cliente es: **Cliente_Chat**, la cual es derivada de la clase **Cliente_Principal**, la que representa la interfaz de usuario (Ver Fig. 26). La clase Cliente_Chat, controla cada una de las llamadas al fichero de interfaz chat_cif.cpp (Ver Fig. 20), que enlazará nuestra aplicación con el stub del cliente (chat_clnt.cpp), mediante los eventos generados por el cliente.

La clase **Cliente_Chat**, contiene un objeto de la clase **cCliente_Conectar**, la cual es derivada de la clase **Cliente_Conectar**, ésta última representa el cuadro de diálogo que se muestra al pulsar el botón Aceptar. (Ver Fig. 27).

1.2. Diagrama de ficheros.

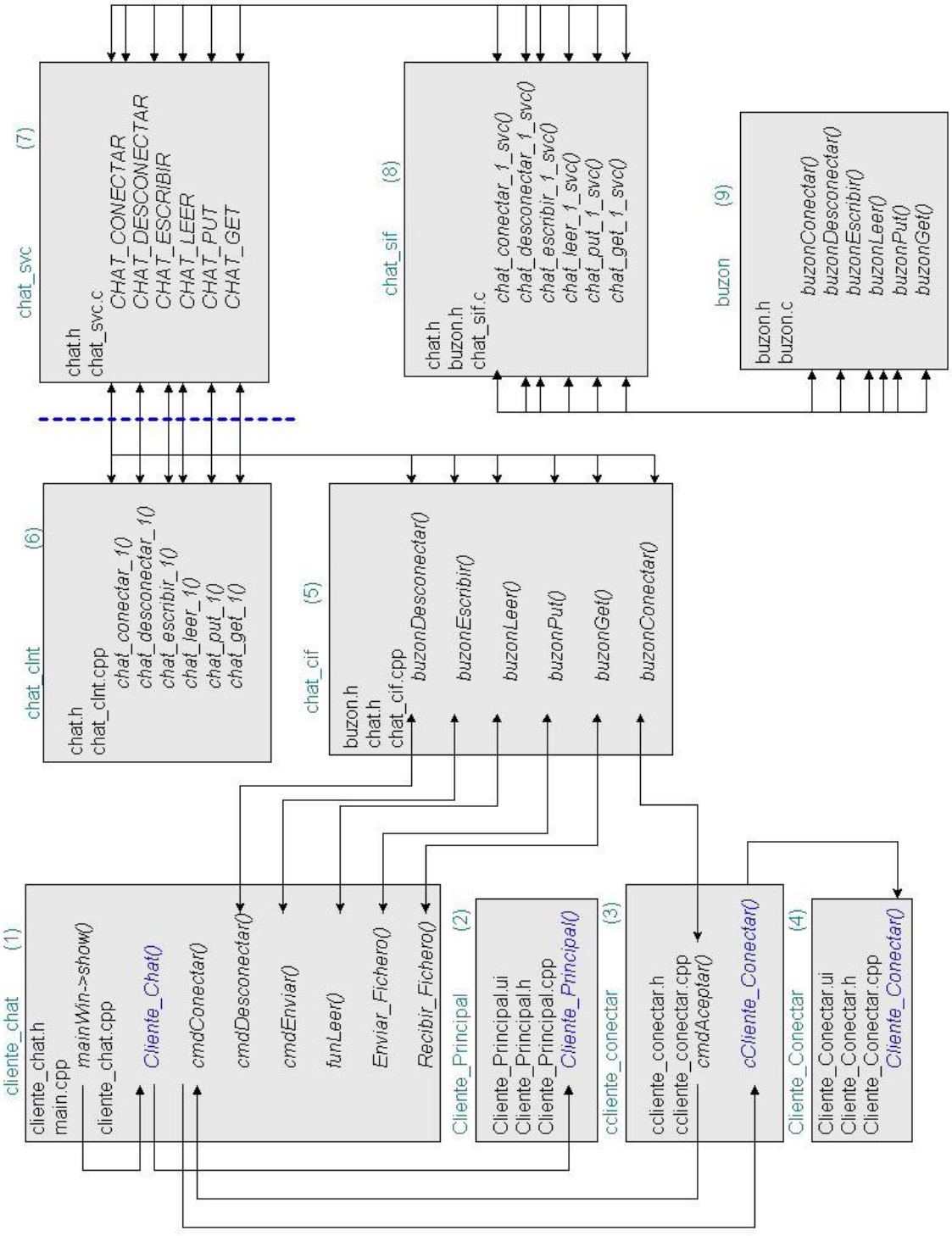


Fig. 20 Diagrama de ficheros.

La ejecución de la Aplicación Cliente comienza por el fichero main.cpp, éste muestra el diálogo que representa la clase **Cliente_Chat** (Ver Fig. 19), la cual visualiza la interfaz gráfica de usuario (Ver Fig. 26), cuando el usuario pulsa el botón Conectar para conectarse a la Sala de Chat, se visualizará el diálogo que representa la clase **cCliente_Conectar** (Ver Fig. 27). Este diálogo, al igual que el diálogo principal de la interfaz de usuario, interactúa con el fichero chat_cif.cpp, el cual se encarga de conectar la aplicación con el stub del cliente (chat_clnt.cpp).

El fichero chat_clnt.cpp, se encarga de enviar las peticiones del cliente al stub del servidor (chat_svc.cpp), éste analiza la petición y se encarga de enviarla a la interfaz del servidor (chta_sif.c), la cual hace la llamada al fichero (buzon.cpp), que se encarga de realizar el procedimiento remoto. Este ciclo se realiza por cada solicitud del cliente, para obtener una respuesta del servidor, éste envía una respuesta al cliente pero recorriendo el camino contrario (Ver Fig. 21).

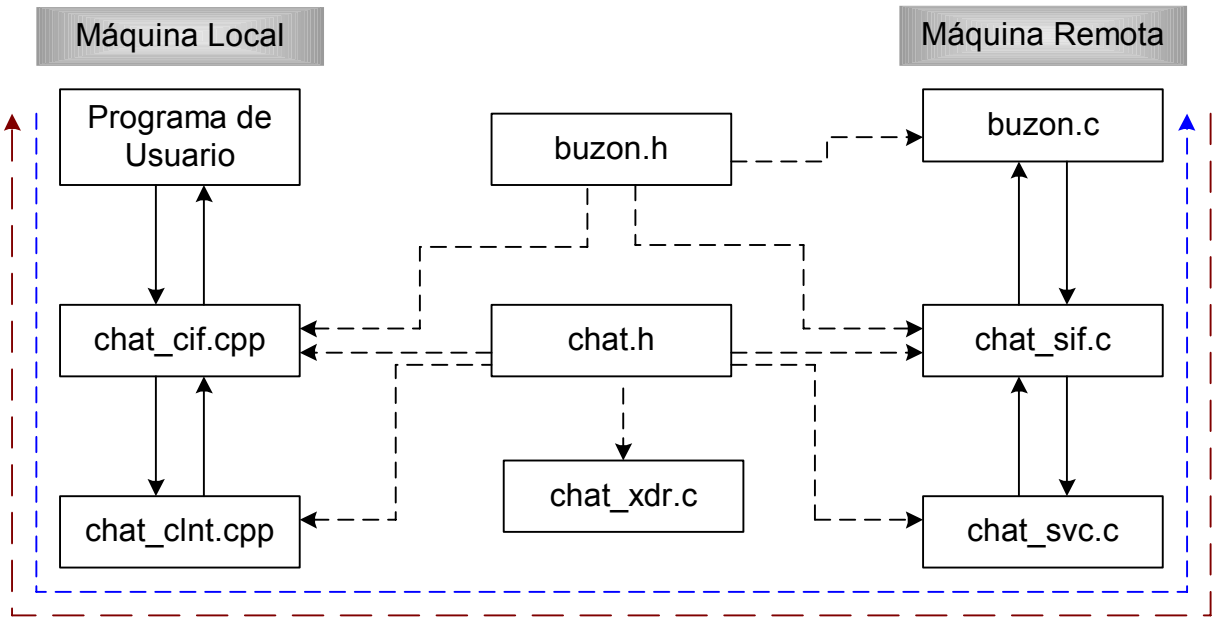


Fig. 21 Ejecución remota.

1.3. Tipos de datos utilizados.

Constantes.

/*Constante que define el número máximo de usuarios que podrán estar conectados a la Sala de Chat.*/

```
const  CHATMAXUSR = 10
```

/*Constante que define el número máximo de caracteres del nombre de un usuario.*/

```
const CHATMAXNOMBRE = 12
```

/*Constante que define el tamaño máximo de los mensajes de texto que podrá enviar o recibir un usuario.*/

```
const CHATMAXLONG = 1000
```

/*Constante que define el bloque máximo de bytes de los ficheros de imagen o sonido a ser transferidos entre dos usuarios.*/

```
const CHATMAXDATOSFICHERO=1024;
```

/*Constante que define el tamaño máximo del nombre de los ficheros a ser transferidos entre los usuarios.*/

```
const CHATMAXNOMBREFICHERO=80;
```

Redefinición de datos.

Redefinición del tipo string.

```
typedef string chatNombreUsr_t < MAXNOMBRE>
typedef string chatMsj_t < MAXLONGMENSAJE >
typedef string type_msj<TIPOMSJ>
typedef string chatNombreFich_t<CHATMAXNOMBREFICHERO>
```

Redefinición de tipo entero.

```
typedef int chatUsrId_t
```

Redefinición de tipo opaque.

/* El compilador rpcgen genera, a partir de este tipo de dato, una estructura que contiene un puntero a char y un entero sin signo. */

```
typedef opaque chatdatos_t<CHATMAXDATOSFICHERO>
typedef struct
{
        _int chatdatos_t_len;
        har *chatdatos_t_val;
} chatdatos_t;
```

Enumerados

/*Tipo de dato utilizado para los posibles mensajes de notificación que se pueden producir durante la comunicación. */

```
enum chatCodigo_t
{
        chatOK,
        chatDemasiadosUsr,
        chatUsrYaDadoAlta,
        chatUsrNoDadoAlta,
        chatNombreUsrNOK,
        chatMsjNull,
        chatNOK,
        chatTuberiaLlena,
        chatTuberiaVacia,
        chatFinFichero,
        chatTuberiaActiva,
        chatTuberiaNoActiva,
        chatTuberiaOcupada
};
```

Estructuras.

/*Tipo de dato utilizado para almacenar temporalmente la información leída (el mensaje, el nombre de usuario y el tipo de mensaje) correspondiente al buzón de un determinado usuario. */

```
        struct chatContestacion_t
        {
                chatNombreUsr_t chatNombreUsr;
                chatMsj_t chatMsj;
                type_msj tipo;
        };
```

/*Tipo de dato que contiene la información (mensaje, identificador y tipo de mensaje) que un determinado usuario enviará al buzón de otro usuario.*/

```
        struct argesc_t
        {
                chatUsrId_t UsrId;
                chatMsj_t chatMsj;
                type_msj tipo;
        };
```

/*Tipo de dato que contiene la información (usuario a quien se desea escribir y datos de un fichero) que un determinado usuario enviará al buzón de otro usuario.*/

```
struct pares_t
{
        chatNombreUsr_t chatNombreUsr;
        chatdatos_t chatContenido;
};typedef struct pares_t pares_t;
```

/*Tipo de dato que contiene la información leída (datos de un fichero) correspondiente al buzón de un determinado usuario.*/

```
struct parleer_t
{
        chatdatos_t chatContenido;
};typedef struct parleer_t parleer_t;
```

2. Diseño Arquitectónico.

2.1. Diseño Arquitectónico de la Aplicación Cliente.

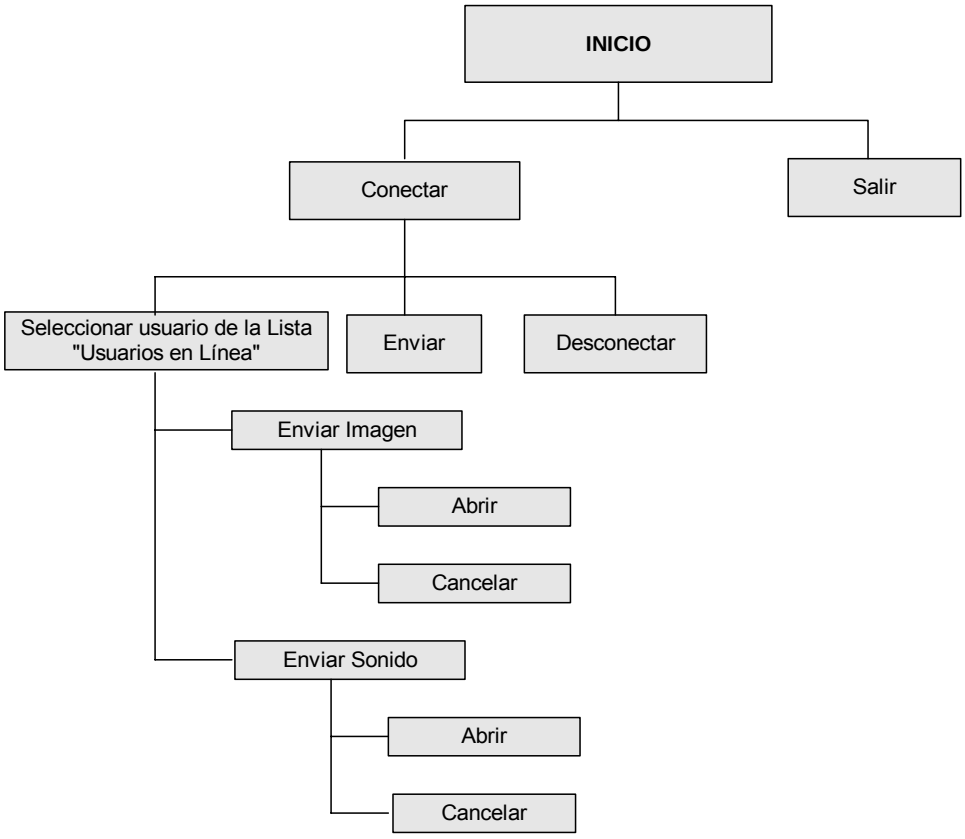


Fig. 22 Diseño Arquitectónico - Aplicación Cliente.

2.2. Diseño Arquitectónico de la Aplicación Servidor.

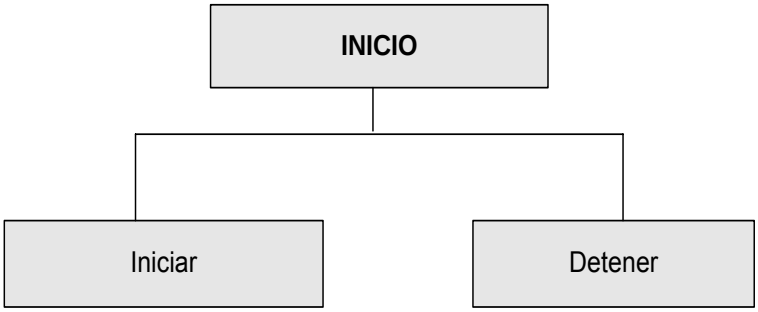
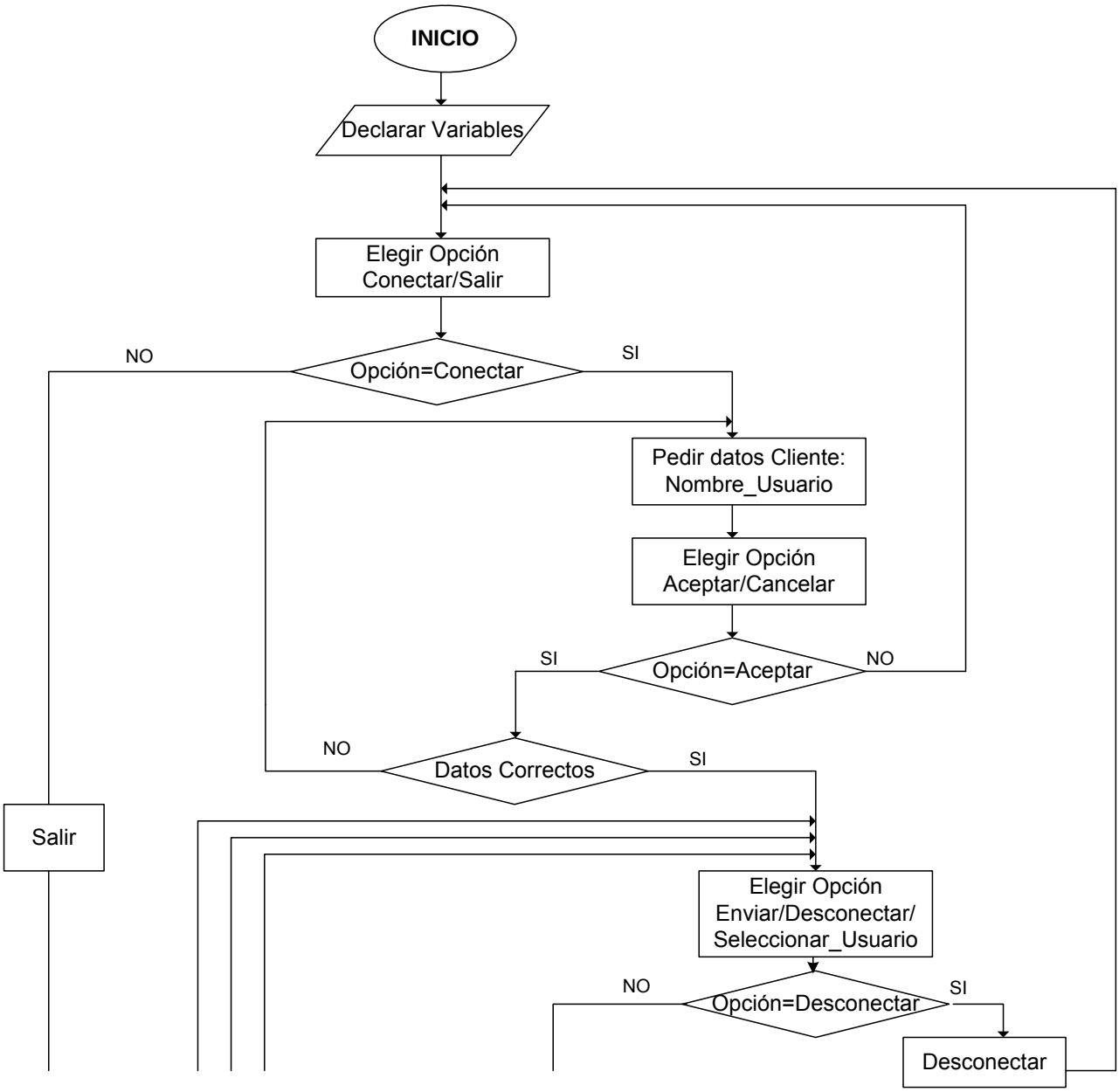


Fig. 23 Diseño Arquitectónico - Aplicación Servidor.

3. Diseño Procedimental.

3.1. Diseño Procedimental de la Aplicación Cliente.



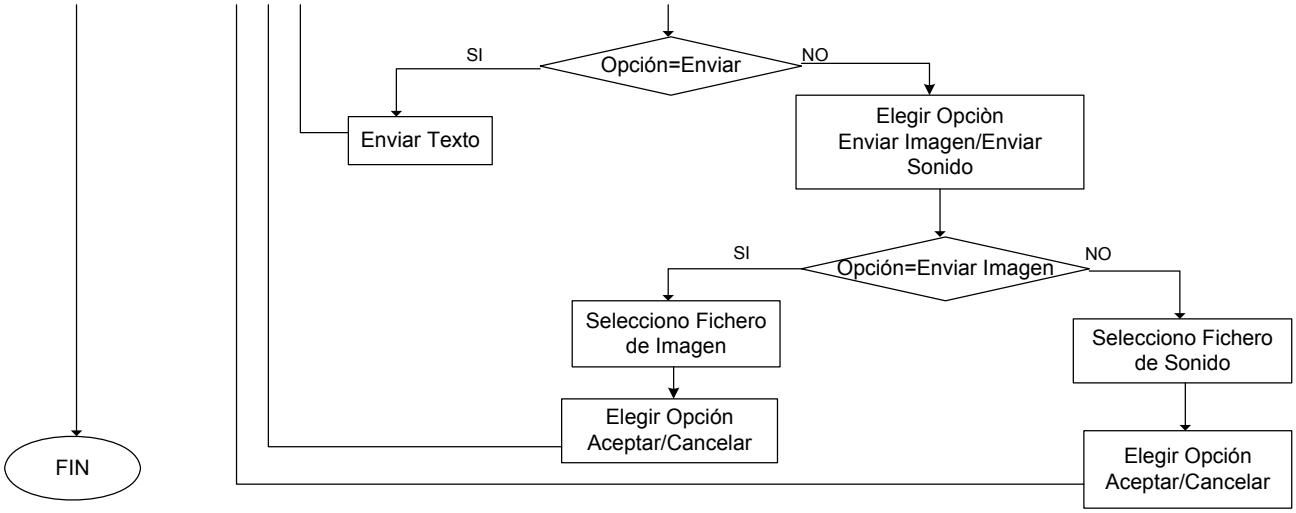


Fig. 24 Diseño Procedimental - Aplicación Cliente.

3.2. Diseño Procedimental de la Aplicación Servidor.

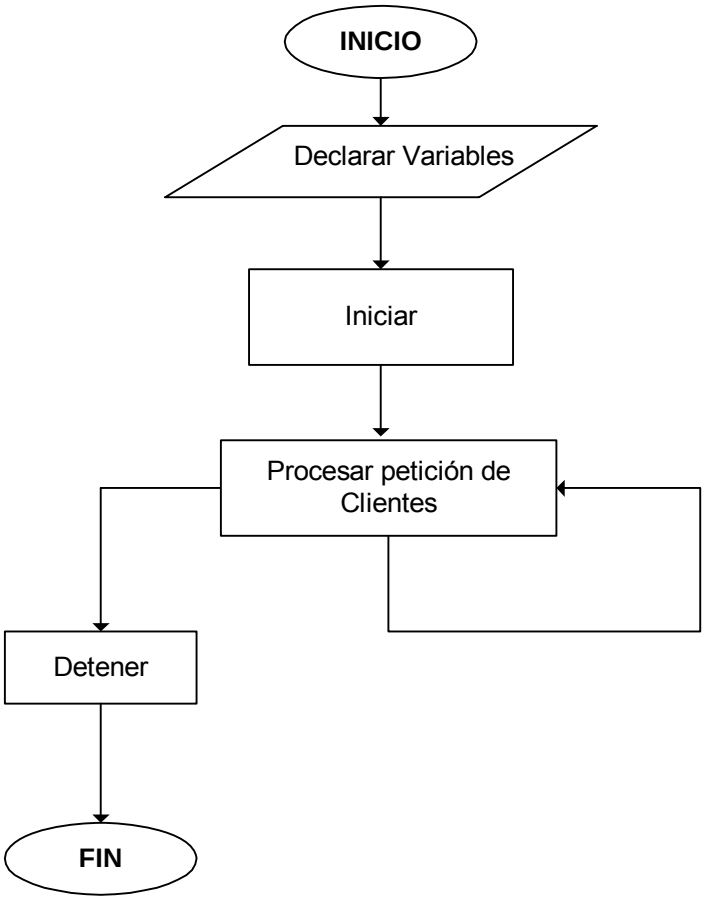


Fig. 25 Diseño Procedimental - Aplicación Servidor.

4. Diseño de la interfaz.

4.1. Interfaz del Cliente.

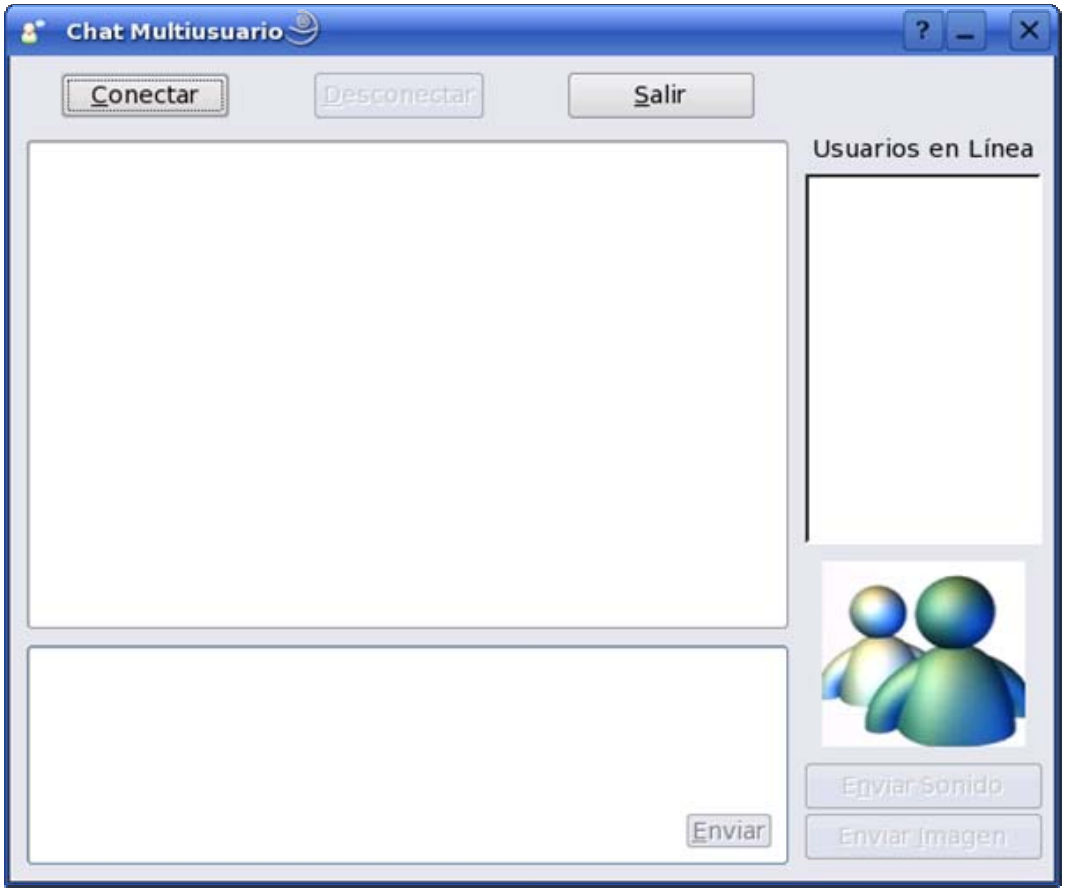


Fig. 26 Interfaz Principal del Cliente.

Diálogo que muestra el botón Conectar.

Fig. 27 Diálogo de conexión del Cliente.

Diálogo que muestra el botón Enviar Imagen.

Fig. 28 Diálogo para enviar Imagen.

Diálogo que muestra el botón Enviar Sonido.

Fig. 29 Diálogo para enviar Sonido.

X. CODIFICACIÓN.

1. Código generado por rpcgen.

La fuente para el desarrollo de los programas Cliente y Servidor, será un fichero de especificación RPC/XDR (al que llamaremos chat.x), que contiene una descripción formal de los procedimientos distribuidos. Una vez escrito nuestro fichero chat.x, hacemos una llamada al comando de unix rpcgen: "**rpcgen -C chat.x**".

Fichero chat.x

```
/*chat.x
Especificación del módulo en lenguaje XDR, que define las estructuras de datos, y los prototipos de las funciones que se publicarán. Definición de la interfaz de servicio. */

/*Constantes.*/
const CHATMAXUSR = 10;                  /*Número máximo de usuarios conectados simultáneamente.*/
const CHATMAXNOMBRE = 12;               /*Tamaño máximo (en caracteres) del nombre de los usuarios.*/
const CHATMAXLONG = 1000;               /*Tamaño máximo (en caracteres) de los mensajes a ser
                                        transferidos.*/
const TIPOMSJ =1;                       /*Delimita en un caracteres el tipo de los mensajes a ser
                                        transferidos.*/
const CHATMAXDATOSFICHERO=1024;         /*Tamaño máximo del bloque de bytes de un fichero a ser
                                        transferido.*/
const CHATMAXNOMBREFICHERO=80;          /*Tamaño máximo (en caracteres) del nombre de un
                                        fichero.*/

/*Definiciones de tipos.*/
typedef string chatNombreUsr_t<CHATMAXNOMBRE>;
typedef string chatMsj_t<CHATMAXLONG>;
typedef int chatUsrId_t;
typedef string type_msj<TIPOMSJ>;

/* Se utiliza para describir datos sin tipo, es decir, secuencias de bytes arbitrarios. Pueden declararse como un array bien de longitud fija o variable. */
typedef opaque chatdatos_t<CHATMAXDATOSFICHERO>;

typedef string chatNombreFich_t<CHATMAXNOMBREFICHERO>;

/*Códigos retornados al cliente.*/
enum chatCodigo_t
{
        chatOK = 1,                     /*Operación realizada correctamente.*/
        chatDemasiadosUsr,              /*Se alcanzó el máximo número de usuarios conectados
                                        simultáneamente.*/
        chatUsrYaDadoAlta,              /*Nombre de usuario duplicado.*/
        chatUsrNoDadoAlta,              /*El nombre de usuario no existe, no está conectado.*/
        chatNombreUsrNOK,               /*Nombre de usuario inválida.*/
        chatMsjNull,                    /*Mensaje enviado vacío.*/
        chatNOK,                        /*Operación no se realizó correctamente.*/
        chatTuberiaLlena,               /*Tubería se encuentra llena (de datos escritos en ella).*/
        chatTuberiaVacia,               /*Tubería se encuentra vacía (no hay datos en ella).*/
        chatFinFichero,         /*Señalización de fin de fichero.*/
        chatTuberiaActiva,              /*La Tubería está activa y utilizándose.*/
```

```
        chatTuberiaNoActiva,        /*La Tubería no se ha creado.*/
        chatTuberiaOcupada          /*La Tubería está siendo usada por otro usuario.*/
};

struct chatContestacion_t           /* Estructura de resultados.*/
{
        chatNombreUsr_t chatNombreUsr;          /*Nombre de quién escribe el mensaje.*/
        chatMsj_t chatMsj;                      /* Cuerpo del mensaje.*/
        type_msj tipo;                          /*Tipo del mensaje.*/
};typedef struct chatContestacion_t chatContestacion_t;

struct argObtenerConectados_t       /* Estructura de resultados.*/
{
        chatNombreUsr_t Usuario_Conectado;      /* Nombre del usuario conectado.*/
};typedef struct argObtenerConectados_t argObtenerConectados_t;

struct argesc_t
{
        chatUsrId_t UsrId;                  /*Id de usuario de quien escribe.*/
        chatMsj_t ChatMsj;                  /*Cuerpo del mensaje.*/
        type_msj tipo;                      /*Tipo del mensaje.*/
}; typedef struct  argesc_t argesc_t;

struct parabr_t         /*Para escribir el nombre del fichero en el destino.*/
{
        chatNombreUsr_t chatNombreUsr;  /*Nombre del usuario a quien se le enviará el fichero.*/
        chatNombreFich_t chatfichero;   /*Nombre del fichero.*/
};typedef struct parabr_t parabr_t;

struct pares_t
{
        chatNombreUsr_t chatNombreUsr;  /*Nombre del usuario a quien se le enviarán los datos.*/
        chatdatos_t chatContenido;      /*Datos del fichero, contenido en bytes y cantidad de
                                        bytes.*/
};typedef struct pares_t pares_t;

struct parleer_t
{
        chatdatos_t chatContenido;  /*Datos del fichero, contenido en bytes y cantidad de bytes.*/
};typedef struct parleer_t parleer_t;

struct chatContestacionparabr_t     /*Estructura de resultados.*/
{
        chatNombreFich_t fichero;   /*Nombre del fichero.*/
};typedef struct chatContestacionparabr_t chatContestacionparabr_t;

union chatLeerRes switch (chatCodigo_t status)
{
        case ChatOK:                /*Operación se realizó correctamente.*/
                chatContestacion_t chatContestacion;
        default:
                void;
};
```

```
/*La siguiente unión se utiliza para diferenciar entre llamadas con éxito y llamadas con error*/
union chatConectadosRes switch (chatCodigo_t status) {
        case chatOK:                    /*Operación se realizó correctamente.*/
                argObtenerConectados_t argObtenerConectados;
        default:
                void;                   /*Con error, no se hace nada*/
};

union chatConectRes switch (chatCodigo_t status)
{
        case chatOK:                    /*Operación se realizó correctamente.*/
                chatUsrId_t UsrId;
        default:
                void;
};

union chatRecibirNombreRes switch(chatCodigo_t status)
{
        case chatOK:                    /*Operación se realizó correctamente.*/
                chatContestacionparabr_t chatContestacionparabr;
        default:
                void;
};

union chatparleerRes switch(chatCodigo_t status)
{
        case chatOK:                    /*Operación se realizó correctamente.*/
                parleer_t parleer;
        default:
                void;
};


/*Sección de especificación de programa y operaciones (RPC).*/
program CHAT_PROGRAM     /*Nombre único del programa.*/
{
        /*Versión del programa.*/
        version CHAT_VERSION
        {
                chatConectRes CHAT_CONECTAR(chatNombreUsr_t)=1;       /*1-> Número de
                                                                      procedimiento .*/
                chatCodigo_t  CHAT_DESCONECTAR(chatUsrId_t)=2;
                chatCodigo_t  CHAT_ESCRIBIR(argesc_t)=3;
                chatLeerRes   CHAT_LEER(chatUsrId_t)=4;
                chatConectadosRes CHAT_CONECTADOS(chatUsrId_t)=5;
                chatCodigo_t CHAT_CREARTUBERIAS(chatUsrId_t)=6;
                chatCodigo_t CHAT_DESTRUIRTUBERIAS(chatUsrId_t)=7;
                chatCodigo_t CHAT_ENVIARNOMBREFICH (parabr_t)=8;
                chatRecibirNombreRes CHAT_RECIBIRNOMBREFICH(chatUsrId_t)=9;
                chatCodigo_t CHAT_PUT(pares_t)=10;
                chatparleerRes CHAT_GET(chatUsrId_t)=11;
                int LOCALIZAR(char *)=12;
        }=1; /*Número que identifica la versión del servidor.*/
}=0x30000000; /* Número que identifica un grupo funcional de procedimientos.*/
```

El compilador rpcgen, genera los siguientes ficheros, con extensión .c, debido al generador de interfaz que se utilizó, a los ficheros: chat_clnt.c y chat_xdr.c, se les asignarán, en el lado del cliente, extensión .cpp, de esta forma se obtiene uniformidad en los ficheros de la Aplicación Cliente.

chat_xdr.c

Como RPC permite llamadas de clientes a servidores que estén en máquinas distintas y, por tanto, puedan tener una arquitectura distinta, es necesario traducir los parámetros y resultados a un "código" universal, independiente de las máquinas. Si los parámetros son de tipos básicos (int, float, char, etc), el sistema unix ya tiene unas funciones de conversión (xdr_int(), xdr_float(), etc). Si los parámetros, como en este caso, son estructuras definidas por nosotros, las funciones de conversión hay que hacerlas. rpcgen genera automáticamente dichas funciones y en nuestro caso, las ha metido en el fichero chat_xdr.c.

chat_svc.c

Este es un ejemplo concreto de servidor. Normalmente nos vale tal cual. Básicamente registra al servidor en el sistema unix para indicarle que atienda a las llamadas y proporciona un "switch-case" para llamar a cada una de las funciones al recibir una petición de un cliente. En la función main se utilizará svcudp_create/svctcp_create para crear la RPC con UDP/TCP. Posteriormente se registrará el programa y su versión asociándoles una función que se ejecutará cuando se reciban peticiones, chat_program_1. El registro se realizará utilizando svc_register. En la función chat_program_1 se analizará el tipo de operación solicitada por el cliente y se ejecutará la función asociada a cada operación (funciona como un repartidor o dispatcher).

chat.h

Aquí están los prototipos de nuestras funciones definidas en chat.x, traducidas al lenguaje de programación en el que vamos a desarrollar la parte cliente y servidor, en nuestro caso C. Cualquier cliente que quiera usarlas, deberá hacer un include de este fichero. El prototipo no es exactamente como esperaríamos. A cada función le añade en el nombre unas "coletillas" para indicar el número de versión. Define también otras constantes como nombre de programa, número de versión, etc, que son útiles a la hora de hacer la conexión con el servidor. Las funciones se generan a partir de la especificación chat.x siguiendo un conjunto de reglas:

- El nombre de la función es en minúsculas, seguido de un guión bajo más el número de versión de la función (ej: _1).

- En el lado del cliente la función tiene 2 parámetros, en el lado servidor tiene el mismo número de parámetros que en la especificación.
- En el lado del cliente las funciones tienen el siguiente formato de parámetros:
    - El primer parámetro es un puntero al parámetro definido en la especificación de la función o void * (NULL).
    - El segundo parámetro es un manejador creado por la función C clnt_create().
- En el lado del servidor, el parámetro es un puntero al parámetro definido en la especificación de la función o void.
- En ambos lados, el valor de retorno de la función es sustituido por un puntero al valor de retorno de la especificación original de la función.

chat_clnt.c

Stub para el cliente. Define las funciones que manejan la llamada remota. Las llamadas a funciones de rpc desde un cliente son más o menos complejas. Se hacen a través de la función "clnt_call()" que lleva la friolera de 7 parámetros. En "chat_clnt.c" rpcgen mete unas funciones de "traducción" para hacernos más sencillas las llamadas desde nuestro cliente. Así, bastará con llamar a "chat_conectar_1 ()" con un par de parámetros simples (nuestros datos y el identificador obtenido con "clnt_create()"), en vez de usar "clnt_call()" con 7 parámetros distintos (identificador de cliente, identificador de la función a llamar, función de conversión del parámetro de entrada, parámetro de salida, etc.).

2. Código común para las Aplicaciones Cliente y Servidor.

Fichero chat.h

```
/*chat.h
Aquí están definidos los prototipos de las funciones. Al prototipo de cada función se le añade en el nombre unas "coletillas" para indicar el número de versión. Define también otras constantes como nombre de programa, número de versión, etc., que son útiles a la hora de hacer la conexión con el servidor.*/

#ifndef _CHAT_H_RPCGEN
#define _CHAT_H_RPCGEN

#include <rpc/rpc.h>            /*Fichero de cabecera, donde están definidas todas las funciones
                                utilizadas por las RPC.*/

#ifdef __cplusplus
extern "C" {
#endif

/*Definición de constantes.*/
#define CHATMAXUSR 10
#define CHATMAXNOMBRE 12
#define CHATMAXLONG 1000
#define TIPOMSJ 1
```

```
#define CHATMAXDATOSFICHERO 1024
#define CHATMAXNOMBREFICHERO 80
/*Redefinición de tipos.*/
typedef char *chatNombreUsr_t;
typedef char *chatMsj_t;
typedef int chatUsrId_t;
typedef char *type_msj;
typedef struct {                        /*Redefinición de tipo opaque.*/
        u_int chatdatos_t_len;          /*Cantidad de bytes a transferir.*/
        char *chatdatos_t_val;          /*Contenido en bytes del fichero a transferir*/
} chatdatos_t;
typedef char *chatNombreFich_t;

enum chatCodigo_t {
        chatOK = 1,
        chatDemasiadosUsr = 1 + 1,
        chatUsrYaDadoAlta = 1 + 2,
        chatUsrNoDadoAlta = 1 + 3,
        chatNombreUsrNOK = 1 + 4,
        chatMsjNull = 1 + 5,
        chatNOK = 1 + 6,
        chatTuberiaLlena = 1 + 7,
        chatTuberiaVacia = 1 + 8,
        chatFinFichero = 1 + 9,
        chatTuberiaActiva = 1 + 10,
        chatTuberiaNoActiva = 1 + 11,
        chatTuberiaOcupada = 1 + 12,
}; typedef enum chatCodigo_t chatCodigo_t;

struct chatContestacion_t {
        chatNombreUsr_t chatNombreUsr;
        chatMsj_t chatMsj;
        type_msj tipo;
};typedef struct chatContestacion_t chatContestacion_t;

struct argObtenerConectados_t {
        chatNombreUsr_t Usuario_Conectado;
};typedef struct argObtenerConectados_t argObtenerConectados_t;

struct argesc_t {
        chatUsrId_t UsrId;
        chatMsj_t chatMsj;
        type_msj tipo;
};typedef struct argesc_t argesc_t;

struct parabr_t {
        chatNombreUsr_t chatNombreUsr;
        chatNombreFich_t chatfichero;
};typedef struct parabr_t parabr_t;

struct pares_t {
        chatNombreUsr_t chatNombreUsr;
        chatdatos_t chatContenido;
};typedef struct pares_t pares_t;

struct parleer_t {
        chatdatos_t chatContenido;
};typedef struct parleer_t parleer_t;
```

```
struct chatContestacionparabr_t {
        chatNombreFich_t fichero;
};typedef struct chatContestacionparabr_t chatContestacionparabr_t;

struct chatLeerRes {
        chatCodigo_t status;
        union {
                chatContestacion_t chatContestacion;
        } chatLeerRes_u;
};typedef struct chatLeerRes chatLeerRes;

struct chatConectadosRes {
        chatCodigo_t status;
        union {
                argObtenerConectados_t argObtenerConectados;
        } chatConectadosRes_u;
};typedef struct chatConectadosRes chatConectadosRes;

struct chatConectRes {
        chatCodigo_t status;
        union {
                chatUsrId_t UsrId;
        } chatConectRes_u;
};typedef struct chatConectRes chatConectRes;

struct chatRecibirNombreRes {
        chatCodigo_t status;
        union {
                chatContestacionparabr_t chatContestacionparabr;
        } chatRecibirNombreRes_u;
};typedef struct chatRecibirNombreRes chatRecibirNombreRes;

struct chatparleerRes {
        chatCodigo_t status;
        union {
                parleer_t parleer;
        } chatparleerRes_u;
};typedef struct chatparleerRes chatparleerRes;

#define CHAT_PROGRAM 0x30000000         /*Número de programa.*/
#define CHAT_VERSION 1                  /*Versión del programa.*/

#if defined(__STDC__) || defined(__cplusplus)
#define CHAT_CONECTAR 1                 /*Procedimiento número 1 */

/*Prototipo del procedimiento 1 (función CHAT_CONECTAR(), a ser implementado por el Cliente.*/
extern  chatConectRes * chat_conectar_1(chatNombreUsr_t *, CLIENT *);
/* Prototipo del procedimiento 1 (función CHAT_CONECTAR(), a ser implementado por el Servidor.*/
extern  chatConectRes * chat_conectar_1_svc(chatNombreUsr_t *, struct svc_req *);

#define CHAT_DESCONECTAR 2
extern  chatCodigo_t * chat_desconectar_1(chatUsrId_t *, CLIENT *);
extern  chatCodigo_t * chat_desconectar_1_svc(chatUsrId_t *, struct svc_req *);
#define CHAT_ESCRIBIR 3
extern  chatCodigo_t * chat_escribir_1(argesc_t *, CLIENT *);
extern  chatCodigo_t * chat_escribir_1_svc(argesc_t *, struct svc_req *);
#define CHAT_LEER 4
extern  chatLeerRes * chat_leer_1(chatUsrId_t *, CLIENT *);
```

```
extern  chatLeerRes * chat_leer_1_svc(chatUsrId_t *, struct svc_req *);
#define CHAT_CONECTADOS 5
extern  chatConectadosRes * chat_conectados_1(chatUsrId_t *, CLIENT *);
extern  chatConectadosRes * chat_conectados_1_svc(chatUsrId_t *, struct svc_req *);
#define CHAT_CREARTUBERIAS 6
extern  chatCodigo_t * chat_creartuberias_1(chatUsrId_t *, CLIENT *);
extern  chatCodigo_t * chat_creartuberias_1_svc(chatUsrId_t *, struct svc_req *);
#define CHAT_DESTRUIRTUBERIAS 7
extern  chatCodigo_t * chat_destruirtuberias_1(chatUsrId_t *, CLIENT *);
extern  chatCodigo_t * chat_destruirtuberias_1_svc(chatUsrId_t *, struct svc_req *);
#define CHAT_ENVIARNOMBREFICH 8
extern  chatCodigo_t * chat_enviarnombrefich_1(parabr_t *, CLIENT *);
extern  chatCodigo_t * chat_enviarnombrefich_1_svc(parabr_t *, struct svc_req *);
#define CHAT_RECIBIRNOMBREFICH 9
extern  chatRecibirNombreRes * chat_recibirnombrefich_1(chatUsrId_t *, CLIENT *);
extern  chatRecibirNombreRes * chat_recibirnombrefich_1_svc(chatUsrId_t *, struct svc_req *);
#define CHAT_PUT 10
extern  chatCodigo_t * chat_put_1(pares_t *, CLIENT *);
extern  chatCodigo_t * chat_put_1_svc(pares_t *, struct svc_req *);
#define CHAT_GET 11
extern  chatparleerRes * chat_get_1(chatUsrId_t *, CLIENT *);
extern  chatparleerRes * chat_get_1_svc(chatUsrId_t *, struct svc_req *);
#define LOCALIZAR 12
extern  int * localizar_1(char *, CLIENT *);
extern  int * localizar_1_svc(char *, struct svc_req *);
extern int chat_program_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);

#else /* K&R C */
#define CHAT_CONECTAR 1
extern  chatConectRes * chat_conectar_1();
extern  chatConectRes * chat_conectar_1_svc();
#define CHAT_DESCONECTAR 2
extern  chatCodigo_t * chat_desconectar_1();
extern  chatCodigo_t * chat_desconectar_1_svc();
#define CHAT_ESCRIBIR 3
extern  chatCodigo_t * chat_escribir_1();
extern  chatCodigo_t * chat_escribir_1_svc();
#define CHAT_LEER 4
extern  chatLeerRes * chat_leer_1();
extern  chatLeerRes * chat_leer_1_svc();
#define CHAT_CONECTADOS 5
extern  chatConectadosRes * chat_conectados_1();
extern  chatConectadosRes * chat_conectados_1_svc();
#define CHAT_CREARTUBERIAS 6
extern  chatCodigo_t * chat_creartuberias_1();
extern  chatCodigo_t * chat_creartuberias_1_svc();
#define CHAT_DESTRUIRTUBERIAS 7
extern  chatCodigo_t * chat_destruirtuberias_1();
extern  chatCodigo_t * chat_destruirtuberias_1_svc();
#define CHAT_ENVIARNOMBREFICH 8
extern  chatCodigo_t * chat_enviarnombrefich_1();
extern  chatCodigo_t * chat_enviarnombrefich_1_svc();
#define CHAT_RECIBIRNOMBREFICH 9
extern  chatRecibirNombreRes * chat_recibirnombrefich_1();
extern  chatRecibirNombreRes * chat_recibirnombrefich_1_svc();
#define CHAT_PUT 10
extern  chatCodigo_t * chat_put_1();
extern  chatCodigo_t * chat_put_1_svc();
```

```
#define CHAT_GET 11
extern  chatparleerRes * chat_get_1();
extern  chatparleerRes * chat_get_1_svc();
#define LOCALIZAR 12
extern  int * localizar_1();
extern  int * localizar_1_svc();
extern int chat_program_1_freeresult ();
#endif /* K&R C */

/*Rutinas para serealizar y deserealizar los datos.*/
#if defined(__STDC__) || defined(__cplusplus)
extern  bool_t xdr_chatNombreUsr_t (XDR *, chatNombreUsr_t*);
extern  bool_t xdr_chatMsj_t (XDR *, chatMsj_t*);
extern  bool_t xdr_chatUsrId_t (XDR *, chatUsrId_t*);
extern  bool_t xdr_type_msj (XDR *, type_msj*);
extern  bool_t xdr_chatdatos_t (XDR *, chatdatos_t*);
extern  bool_t xdr_chatNombreFich_t (XDR *, chatNombreFich_t*);
extern  bool_t xdr_chatCodigo_t (XDR *, chatCodigo_t*);
extern  bool_t xdr_chatContestacion_t (XDR *, chatContestacion_t*);
extern  bool_t xdr_chatContestacion_t (XDR *, chatContestacion_t*);
extern  bool_t xdr_argObtenerConectados_t (XDR *, argObtenerConectados_t*);
extern  bool_t xdr_argObtenerConectados_t (XDR *, argObtenerConectados_t*);
extern  bool_t xdr_argesc_t (XDR *, argesc_t*);
extern  bool_t xdr_argesc_t (XDR *, argesc_t*);
extern  bool_t xdr_parabr_t (XDR *, parabr_t*);
extern  bool_t xdr_parabr_t (XDR *, parabr_t*);
extern  bool_t xdr_pares_t (XDR *, pares_t*);
extern  bool_t xdr_pares_t (XDR *, pares_t*);
extern  bool_t xdr_parleer_t (XDR *, parleer_t*);
extern  bool_t xdr_parleer_t (XDR *, parleer_t*);
extern  bool_t xdr_chatContestacionparabr_t (XDR *, chatContestacionparabr_t*);
extern  bool_t xdr_chatContestacionparabr_t (XDR *, chatContestacionparabr_t*);
extern  bool_t xdr_chatLeerRes (XDR *, chatLeerRes*);
extern  bool_t xdr_chatConectadosRes (XDR *, chatConectadosRes*);
extern  bool_t xdr_chatConectRes (XDR *, chatConectRes*);
extern  bool_t xdr_chatRecibirNombreRes (XDR *, chatRecibirNombreRes*);
extern  bool_t xdr_chatparleerRes (XDR *, chatparleerRes*);

#else /* K&R C */
extern bool_t xdr_chatNombreUsr_t ();
extern bool_t xdr_chatMsj_t ();
extern bool_t xdr_chatUsrId_t ();
extern bool_t xdr_type_msj ();
extern bool_t xdr_chatdatos_t ();
extern bool_t xdr_chatNombreFich_t ();
extern bool_t xdr_chatCodigo_t ();
extern bool_t xdr_chatContestacion_t ();
extern bool_t xdr_chatContestacion_t ();
extern bool_t xdr_argObtenerConectados_t ();
extern bool_t xdr_argObtenerConectados_t ();
extern bool_t xdr_argesc_t ();
extern bool_t xdr_argesc_t ();
extern bool_t xdr_parabr_t ();
extern bool_t xdr_parabr_t ();
extern bool_t xdr_pares_t ();
extern bool_t xdr_pares_t ();
extern bool_t xdr_parleer_t ();
extern bool_t xdr_parleer_t ();
```

```
extern bool_t xdr_chatContestacionparabr_t ();
extern bool_t xdr_chatContestacionparabr_t ();
extern bool_t xdr_chatLeerRes ();
extern bool_t xdr_chatConectadosRes ();
extern bool_t xdr_chatConectRes ();
extern bool_t xdr_chatRecibirNombreRes ();
extern bool_t xdr_chatparleerRes ();

#endif /* K&R C */
#ifdef __cplusplus
}
#endif
#endif /* !_CHAT_H_RPCGEN */
```

Fichero chat_xdr.c

```
/*chat_xdr.c
Estas rutinas se utilizan para convertir estructuras de datos locales al formato XDR y viceversa. Por cada tipo de dato definido en el archivo chat.x, rpcgen genera una rutina con el prefijo xdr_ que se incluye en este archivo.*/
#include "chat.h"              /*Fichero de cabecera, donde están los prototipos de las funciones de
                                aplanamiento de los datos definidos por el programador.*/

bool_t
xdr_chatNombreUsr_t (XDR *xdrs, chatNombreUsr_t *objp)
{
        register int32_t *buf;
        if (!xdr_string (xdrs, objp, CHATMAXNOMBRE))
                return FALSE;
        return TRUE;
}

bool_t
xdr_chatMsj_t (XDR *xdrs, chatMsj_t *objp)
{
        register int32_t *buf;
        if (!xdr_string (xdrs, objp, CHATMAXLONG))
                return FALSE;
        return TRUE;
}

bool_t
xdr_chatUsrId_t (XDR *xdrs, chatUsrId_t *objp)
{
        register int32_t *buf;
        if (!xdr_int (xdrs, objp))
                return FALSE;
        return TRUE;
}

bool_t
xdr_type_msj (XDR *xdrs, type_msj *objp)
{
        register int32_t *buf;
        if (!xdr_string (xdrs, objp, TIPOMSJ))
                return FALSE;
        return TRUE;
}
```

```
bool_t
xdr_chatdatos_t (XDR *xdrs, chatdatos_t *objp)
{
        register int32_t *buf;
        if (!xdr_bytes (xdrs, (char **)&objp->chatdatos_t_val, (u_int *) &objp->chatdatos_t_len,
        CHATMAXDATOSFICHERO))
        return FALSE;
        return TRUE;
}

bool_t
xdr_chatNombreFich_t (XDR *xdrs, chatNombreFich_t *objp)
{
        register int32_t *buf;
        if (!xdr_string (xdrs, objp, CHATMAXNOMBREFICHERO))
                return FALSE;
        return TRUE;
}

bool_t
xdr_chatCodigo_t (XDR *xdrs, chatCodigo_t *objp)
{
        register int32_t *buf;
        if (!xdr_enum (xdrs, (enum_t *) objp))
                return FALSE;
        return TRUE;
}

bool_t
xdr_chatContestacion_t (XDR *xdrs, chatContestacion_t *objp)
{
        register int32_t *buf;
        if (!xdr_chatNombreUsr_t (xdrs, &objp->chatNombreUsr))
                return FALSE;
         if (!xdr_chatMsj_t (xdrs, &objp->chatMsj))
                return FALSE;
         if (!xdr_type_msj (xdrs, &objp->tipo))
                return FALSE;
        return TRUE;
}

bool_t
xdr_argObtenerConectados_t (XDR *xdrs, argObtenerConectados_t *objp)
{
        register int32_t *buf;
        if (!xdr_chatNombreUsr_t (xdrs, &objp->Usuario_Conectado))
                return FALSE;
        return TRUE;
}

bool_t
xdr_argesc_t (XDR *xdrs, argesc_t *objp)
{
        register int32_t *buf;
        if (!xdr_chatUsrId_t (xdrs, &objp->UsrId))
                return FALSE;
         if (!xdr_chatMsj_t (xdrs, &objp->chatMsj))
                return FALSE;
```

```
	if (!xdr_type_msj (xdrs, &objp->tipo))
		return FALSE;
	return TRUE;
}

bool_t
xdr_parabr_t (XDR *xdrs, parabr_t *objp)
{
	register int32_t *buf;
	if (!xdr_chatNombreUsr_t (xdrs, &objp->chatNombreUsr))
		return FALSE;
	if (!xdr_chatNombreFich_t (xdrs, &objp->chatfichero))
		return FALSE;
	return TRUE;
}

bool_t
xdr_pares_t (XDR *xdrs, pares_t *objp)
{
	register int32_t *buf;
	if (!xdr_chatNombreUsr_t (xdrs, &objp->chatNombreUsr))
		return FALSE;
	if (!xdr_chatdatos_t (xdrs, &objp->chatContenido))
		return FALSE;
	return TRUE;
}

bool_t xdr_parleer_t (XDR *xdrs, parleer_t *objp)
{
	register int32_t *buf;
	if (!xdr_chatdatos_t (xdrs, &objp->chatContenido))
		return FALSE;
	return TRUE;
}

bool_t xdr_chatContestacionparabr_t (XDR *xdrs, chatContestacionparabr_t *objp)
{
	register int32_t *buf;
	if (!xdr_chatNombreFich_t (xdrs, &objp->fichero))
		return FALSE;
	return TRUE;
}

bool_t xdr_chatLeerRes (XDR *xdrs, chatLeerRes *objp)
{
	register int32_t *buf;
	if (!xdr_chatCodigo_t (xdrs, &objp->status))
		return FALSE;
	switch (objp->status) {
	case chatOK:
		if (!xdr_chatContestacion_t (xdrs, &objp->chatLeerRes_u.chatContestacion))
			return FALSE;
		break;
	default:
		break;
	}
	return TRUE;
}
```

```
bool_t
xdr_chatConectadosRes (XDR *xdrs, chatConectadosRes *objp)
{
        register int32_t *buf;
         if (!xdr_chatCodigo_t (xdrs, &objp->status))
                 return FALSE;
        switch (objp->status) {
        case chatOK:
                 if(!xdr_argObtenerConectados_t(xdrs,&objp->
                chatConectadosRes_u.argObtenerConectados))
                        return FALSE;
                break;
        default:
                break;
        }
        return TRUE;
}

bool_t
xdr_chatConectRes (XDR *xdrs, chatConectRes *objp)
{
        register int32_t *buf;
         if (!xdr_chatCodigo_t (xdrs, &objp->status))
                 return FALSE;
        switch (objp->status) {
        case chatOK:
                 if (!xdr_chatUsrId_t (xdrs, &objp->chatConectRes_u.UsrId))
                        return FALSE;
                break;
        default:
                break;
        }
        return TRUE;
}

bool_t
xdr_chatRecibirNombreRes (XDR *xdrs, chatRecibirNombreRes *objp)
{
        register int32_t *buf;
         if (!xdr_chatCodigo_t (xdrs, &objp->status))
                 return FALSE;
        switch (objp->status) {
        case chatOK:
                 if(!xdr_chatContestacionparabr_t(xdrs,&objp->
                chatRecibirNombreRes_u.chatContestacionparabr))
                        return FALSE;
                break;
        default:
                break;
        }
        return TRUE;
}

bool_t
xdr_chatparleerRes (XDR *xdrs, chatparleerRes *objp)
{
        register int32_t *buf;
         if (!xdr_chatCodigo_t (xdrs, &objp->status))
```

```
                return FALSE;
        switch (objp->status) {
        case chatOK:
                if (!xdr_parleer_t (xdrs, &objp->chatparleerRes_u.parleer))
                        return FALSE;
                break;
        default:
                break;
        }
        return TRUE;
}
```

Fichero buzon.h

```
/*buzon.h
Especificación del módulo que se encarga de enviar y recibir mensajes entre usuario.
Los mensajes tienen una capacidad limitada de 1000 caracteres.
El número de usuarios simultáneamente conectados no será más de 10 personas.*/

#include <pthread.h>    /*Declara funciones y tipos de datos que nos permiten manejar hilos.*/
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>     /*Definición de llamadas primitivas al sistema.*/
#include <fcntl.h>      /*Control de operaciones sobre ficheros.*/

#ifndef FALSE
#define FALSE   (0)
#endif

#ifndef TRUE
#define  TRUE    (1)
#endif

#define CHAT_MAX_USR 10             /*Número máximo de personas conectadas simultáneamente.*/
#define CHAT_MAX_NOMBRE 12          /*Tamaño máximo (en caracteres) del nombre de usuario.*/
#define CHAT_MAX_LONG 1000          /*Tamaño máximo (en caracteres) de mensajes a enviar/recibir.*/
#define CHAT_MAX_TIPO 1             /*Delimita en uno la cantidad de caracteres utilizados para
                                    identificar el tipo del mensaje a ser transferido.*/
#define CHAT_MAX_NOMBRE_FICHERO 80  /*Tamaña máximo  (en caracteres) del nombre de los
                                      ficheros a transferir.*/
#define CHAT_MAXDATOS_FICHERO 1024          /*Tamaño máximo del bloque de bytes de un fichero
                                              a ser transferido. */
#define TIPO1 "C"                   /*Mensaje de nuevo usuario conectado. */
#define TIPO2 "D"                   /*Mensaje de un usuario desconectado. */
#define TIPO3 "U"                   /*Mensaje de un usuario enviado a toda la Sala de Chat. */
#define TIPO4 "F"                   /*Mensaje de envio de fichero. */
#define TIPO5 "A"                   /*Mensaje de aceptación de fichero. */
#define TIPO6 "N"                   /*Mensaje de no aceptación de fichero. */
#define MSJ1 "Nuevo usuario conectado"
#define MSJ2 "Usuario Desconectados"
#define MSJ3 "Envio de fichero"
#define MSJ4 "Acepto el fichero"
#define MSJ5 "No acepto el fichero"

typedef int buzonUsrId_t;           //Id de usuario devuelto por el Servidor.
typedef unsigned buzonCantidad_t;   //Cantidad de bytes leída o escrita de un fichero.
```

```
typedef enum                    //Código de mensajes retornados por el Servidor.
{
        buzonOK,                //Operación realizada correctamente.
        buzonDemasiadosUsr,     /*Se alcanzó el máximo número de usuarios conectados
                                simultáneamente.*/
        buzonUsrYaDadoAlta,     //Replica en el nombre de usuario.
        buzonUsrNoDadoAlta,     //El nombre de usuario no existe, no está conectado.
        buzonNombreUsrNOK,      //Nombre de usuario no válido.
        buzonMsjNull,           //Mensaje enviado vacío.
        buzonNOK,               //Operación no se llevó acabo correctamente.
        buzonTuberiaLlena,      //Tubería se encuentra llena (de datos escritos en ella).
        buzonTuberiaVacia,      //Tubería se encuentra vacía (no hay datos en ella).
        buzonFinFichero,        //Señalización de fin de fichero.
        buzonTuberiaActiva,     //La Tubería está activa y utilizándose.
        buzonTuberiaNoActvia,   //La Tubería no se ha creado.
        buzonTuberiaOcupada,    /La Tubería está siendo usada por otro usuario.
        buzonServidor           //El servidor no está brindando servicio.
}buzonCodigo_t;
typedef struct descriptor
{
        int activo;                             //El usuario está activo (conectado).
        char usuario[CHAT_MAX_NOMBRE];  //Nombre de usuario.
        int fds[2];                             //Arreglo de descriptores, para las tuberías de
                                                //intercambio de mensajes.
        int fds2[2];                            //Arreglo de descriptores, para las tuberías de
                                                //transferencia de ficheros.
        int TubLlena;                           //Indica que la tubería se encuentra llena.
        int FinFichero;                         //Indica el fin del fichero.
        int Tuberia_Activa;                     //Indica que la tubería se encuentra activa.
        int Tuberia_Ocupada;                    //Indica que la tubería se encuentra ocupada.
        pthread_mutex_t RC;                     //Semáforo binario, para implementar las zonas de
                                                //exclusión mutua en las tuberías.
}descriptor_t;
        buzonCodigo_t buzonConectar(char *usuario, buzonUsrId_t *usrId );
        /*-------------------------------------------------------------------------------------------------------------------
        Se da de alta al usuario, y si todo se realizó correctamente retorna "buzonOK", y un identificador
        de usuario (usrId), para poder hablar en la Sala de Chat. Si corrió algún problema, retorna:
        "buzonDemasiadosUsr", para indicar que ya hay conectados el máximo número de usuarios.
        "buzonUsrYaDadoAlta", para indicar que este usuario ya está dado de alta.
        "buzonNOK", para indicar que se ha producido un error de conexión o del sistema.
        -------------------------------------------------------------------------------------------------------------------*/

        buzonCodigo_t buzonDesconectar(buzonUsrId_t buzonUsrId);
        /*-------------------------------------------------------------------------------------------------------------------
        Permite que el usuario abandone la Sala de Chat. Si había mensajes para él pendientes en el
        buzón se pierden. Si todo se realizó correctamente retorna "buzonOk", de lo contrario:
        "buzonUsrNoDadoAlta", para indicar que este usuario, no está dado de alta en la Sala.
        "buzonNOK", para indicar que se ha producido un error de conexión o del sistema.
        -------------------------------------------------------------------------------------------------------------------*/

        buzonCodigo_t buzonEscribir(buzonUsrId_t buzonUsrId, char *msj,char *tipo);
        /*-------------------------------------------------------------------------------------------------------------------
        Permite escribir en dependencia del tipo de mensaje (tipo), en todos los buzones o en un buzón
        específico. Si todo se realizó correctamente retorna "buzonOk", de lo contrario:
        "buzonUsrNoDadoAlta", para indicar que este usuario, no está dado de alta en la Sala.
        "buzonNOK", para indicar que se ha producido un error de conexión o del sistema.
        -------------------------------------------------------------------------------------------------------------------*/
```

```
buzonCodigo_t buzonLeer(buzonUsrId_t buzonUsrId,char *usuario,char *msj,char *tipo);
/*-----------------------------------------------------------------------------------------------------------------------
Permite leer del buzón, del usuario especificado por buzonUsrId un mensaje. Si todo se realizó
correctamente retorna "buzonOk", y devuelve el mensaje leído, el autor y el tipo de mensaje, de
lo contrario, retorna:
"buzonUsrNoDadoAlta", para indicar que este usuario, no está dado de alta en la Sala.
"buzonNOK", para indicar que se ha producido un error de conexión o del sistema.
-----------------------------------------------------------------------------------------------------------------------*/

buzonCodigo_t buzonConectados(buzonUsrId_t buzonUsrId,char *usuario_conectado);
/*-----------------------------------------------------------------------------------------------------------------------
Permite conocer si el usuario representado por buzonUsrId, está dado de alta en la Sala, si es así
retorna "buzonOk", y el nombre del usuario en usuario_conectado, de lo contrario retorna:
buzonUsrNoDadoAlta, para indicar que ese usuario no está dado de alta en la Sala de Chat.
-----------------------------------------------------------------------------------------------------------------------*/

buzonCodigo_t buzonCrearTuberias(buzonUsrId_t buzonUsrId);
/*-----------------------------------------------------------------------------------------------------------------------
Permite crear una tubería para el usuario (buzonUsrId), si está activa, es decir ya fue creada
retorna "buzonTuberiaActiva", en caso contrario retorna:
"buzonOk", si todo el proceso se realizó correctamente.
"buzonNOK", para indicar que se ha producido un error de conexión o del sistema.
-----------------------------------------------------------------------------------------------------------------------*/

buzonCodigo_t buzonDestruirTuberias(buzonUsrId_t buzonUsrId);
/*-----------------------------------------------------------------------------------------------------------------------
Permite destruir una tubería para el usuario (buzonUsrId). Si todo se realizó correctamente
retorna "buzonOk", de lo contrario:
"buzonNOK", para indicar que se ha producido un error de conexión o del sistema.
-----------------------------------------------------------------------------------------------------------------------*/

buzonCodigo_t buzonEnviarNombreFich(char *chatNombreUsr, char *NombreFich);
/*-----------------------------------------------------------------------------------------------------------------------
Permite escribir en la tubería de un usuario (chatNombreUsr), el nombre de un fichero
(NombreFich), que será transmitido. Si la tubería del usuario no ha sido creada retorna
"buzonTuberiaNoActvia", en caso contrario:
"buzonOK", si todo el proceso se ha realizado correctamente.
"buzonTuberiaOcupada", para indicar que otro usuario está haciendo uso de dicha tubería.
"buzonTuberiaLlena", para indicar que en la tubería existen datos, y que por lo tanto no se puede
escribir en ella o "buzonNOK", para indicar que se ha producido un error de conexión o del sistema.
-----------------------------------------------------------------------------------------------------------------------*/

buzonCodigo_t buzonRecibirNombreFich(buzonUsrId_t buzonUsrId, char *NombreFich);
/*-----------------------------------------------------------------------------------------------------------------------
Permite leer de la tubería del usuario (buzonUsrId), el nombre de un fichero, si en la tubería no hay
datos retorna "buzonTuberiaVacia", en caso contrario retorna:
"buzonOK", si todo el proceso se ha realizado correctamente.
"buzonNOK", para indicar que se ha producido un error de conexión o del sistema.
-----------------------------------------------------------------------------------------------------------------------*/

buzonCodigo_t buzonPut(char *chatNombreUsr, char *contenido, buzonCantidad_t cantidad);
/*-----------------------------------------------------------------------------------------------------------------------
Permite escribir en la tubería de un usuario (chatNombreUsr), la cantidad de bytes (cantidad)
almacenada en contenido, si en la tubería existen datos, es decir se encuentra llena, retorna
buzonTuberiaLlena, de lo contrario retorna:
"buzonOK", si todo el proceso se ha realizado correctamente.
buzonNOK", para indicar que se ha producido un error de conexión o del sistema.
-----------------------------------------------------------------------------------------------------------------------*/
```

buzonCodigo_t buzonGet(buzonUsrId_t buzonUsrId, char *contenido, buzonCantidad_t *cantidad);
/*----------
Permite leer de la tubería de un usuario (buzonUsrId) hasta 1024 bytes, los almacena en contenido y devuelve en cantidad, la cantidad de bytes realmente leídos, si todo el proceso se ha realizado correctamente retorna "buzonOK", en caso contrario retorna:
"buzonTuberiaVacia", para indicar que no hay datos en la tubería y por ende no se puede leer.
"buzonFinFichero", para indicar que ya no hay más datos, es decir se ha llegado al final del fichero.
"buzonNOK", para indicar que se ha producido un error de conexión o del sistema.
----------*/

buzonCodigo_t localizar(char *);
/*----------
Esta función se ejecuta cuando se esta buscando la dirección IP del Servidor. Si todo el proceso se realiza correctamente retorna buzonOK.
----------*/

3. Código de la Aplicación Cliente.

La Aplicación Cliente consta de los siguientes ficheros:

Fichero chat_clnt.cpp

```
/*chat_clnt.cpp
stub del cliente. Procedimientos de "traducción" para hacer sencillas las llamadas a las RPC desde nuestro cliente.*/

#include <memory.h>    /* Fichero de cabecera utilizado por la función memset() */
#include "chat.h"

/* Timeout por defecto, este puede ser modificado por la función clnt_control().*/
static struct timeval TIMEOUT = { 25, 0 };

chatConectRes *
chat_conectar_1(chatNombreUsr_t *argp, CLIENT *clnt){      /*argp -> puntero al nombre introducido
                                                           por el usuario.
                                                           clnt -> puntero a una estructura que
                                                           representa a la entidad cliente.*/
        static chatConectRes clnt_res;                     /*Valor de retorno.*/
        memset((char *)&clnt_res, 0, sizeof(clnt_res));    /*Inicialización del valor retornado.*/
        /*clnt_call -> Esta función lo que en realidad hace es pasar los parámetros al formato de
        codificación externa, aplanar estructuras y solicitar el envío del mensaje por la red. Sus parámetros
        significan lo siguiente:
        clnt -> Entidad cliente.
        CHAT_CONECTAR -> Identificador de procedimiento remoto a invocar.
        xdr_chatNombreUsr_t -> Rutina para serealizar el argumento de entrada del procedimiento
                               CHAT_CONECTAR .
        argp -> Argumrnto de entrada del procedimiento.
        xdr_chatConectRes -> Rutina para deserealizar el resultado.
        clnt_res -> Dirección de la variable, donde se recibirá el resultado de la RPC.
        TIMEOUT -> Tiempo total máximo de espera a la respuesta.*/
        if (clnt_call (clnt, CHAT_CONECTAR,
                (xdrproc_t) xdr_chatNombreUsr_t, (caddr_t) argp,
                (xdrproc_t) xdr_chatConectRes, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);          /*La llamada no se realizó satisfactoriamente, se retorna NULL.*/
        }
        return (&clnt_res);             /*Retornar a chat_cif.cpp*/
}
```

```
chatCodigo_t * chat_desconectar_1(chatUsrId_t *argp, CLIENT *clnt)
{
	static chatCodigo_t clnt_res;
	memset((char *)&clnt_res, 0, sizeof(clnt_res));
	if (clnt_call (clnt, CHAT_DESCONECTAR,
		(xdrproc_t) xdr_chatUsrId_t, (caddr_t) argp,
		(xdrproc_t) xdr_chatCodigo_t, (caddr_t) &clnt_res,
		TIMEOUT) != RPC_SUCCESS) {
		return (NULL);
	}
	return (&clnt_res);
}

chatCodigo_t * chat_escribir_1(argesc_t *argp, CLIENT *clnt)
{
	static chatCodigo_t clnt_res;
	memset((char *)&clnt_res, 0, sizeof(clnt_res));
	if (clnt_call (clnt, CHAT_ESCRIBIR,
		(xdrproc_t) xdr_argesc_t, (caddr_t) argp,
		(xdrproc_t) xdr_chatCodigo_t, (caddr_t) &clnt_res,
		TIMEOUT) != RPC_SUCCESS) {
		return (NULL);
	}
	return (&clnt_res);
}

chatLeerRes * chat_leer_1(chatUsrId_t *argp, CLIENT *clnt)
{
	static chatLeerRes clnt_res;
	memset((char *)&clnt_res, 0, sizeof(clnt_res));
	if (clnt_call (clnt, CHAT_LEER,
		(xdrproc_t) xdr_chatUsrId_t, (caddr_t) argp,
		(xdrproc_t) xdr_chatLeerRes, (caddr_t) &clnt_res,
		TIMEOUT) != RPC_SUCCESS) {
		return (NULL);
	}
	return (&clnt_res);
}

chatConectadosRes * chat_conectados_1(chatUsrId_t *argp, CLIENT *clnt)
{
	static chatConectadosRes clnt_res;
	memset((char *)&clnt_res, 0, sizeof(clnt_res));
	if (clnt_call (clnt, CHAT_CONECTADOS,
		(xdrproc_t) xdr_chatUsrId_t, (caddr_t) argp,
		(xdrproc_t) xdr_chatConectadosRes, (caddr_t) &clnt_res,
		TIMEOUT) != RPC_SUCCESS) {
		return (NULL);
	}
	return (&clnt_res);
}

chatCodigo_t * chat_creartuberias_1(chatUsrId_t *argp, CLIENT *clnt)
{
	static chatCodigo_t clnt_res;
	memset((char *)&clnt_res, 0, sizeof(clnt_res));
	if (clnt_call (clnt, CHAT_CREARTUBERIAS,
		(xdrproc_t) xdr_chatUsrId_t, (caddr_t) argp,
```

```
                (xdrproc_t) xdr_chatCodigo_t, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return (&clnt_res);
}

chatCodigo_t * chat_destruirtuberias_1(chatUsrId_t *argp, CLIENT *clnt)
{
        static chatCodigo_t clnt_res;
        memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call (clnt, CHAT_DESTRUIRTUBERIAS,
                (xdrproc_t) xdr_chatUsrId_t, (caddr_t) argp,
                (xdrproc_t) xdr_chatCodigo_t, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return (&clnt_res);
}

chatCodigo_t * chat_enviarnombrefich_1(parabr_t *argp, CLIENT *clnt)
{
        static chatCodigo_t clnt_res;
        memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call (clnt, CHAT_ENVIARNOMBREFICH,
                (xdrproc_t) xdr_parabr_t, (caddr_t) argp,
                (xdrproc_t) xdr_chatCodigo_t, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return (&clnt_res);
}

chatRecibirNombreRes * chat_recibirnombrefich_1(chatUsrId_t *argp, CLIENT *clnt)
{
        static chatRecibirNombreRes clnt_res;
        memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call (clnt, CHAT_RECIBIRNOMBREFICH,
                (xdrproc_t) xdr_chatUsrId_t, (caddr_t) argp,
                (xdrproc_t) xdr_chatRecibirNombreRes, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return (&clnt_res);
}

chatCodigo_t * chat_put_1(pares_t *argp, CLIENT *clnt)
{
        static chatCodigo_t clnt_res;
        memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call (clnt, CHAT_PUT,
                (xdrproc_t) xdr_pares_t, (caddr_t) argp,
                (xdrproc_t) xdr_chatCodigo_t, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return (&clnt_res);
}
```

```
chatparleerRes * chat_get_1(chatUsrId_t *argp, CLIENT *clnt)
{
	static chatparleerRes clnt_res;
	memset((char *)&clnt_res, 0, sizeof(clnt_res));
	if (clnt_call (clnt, CHAT_GET,
		(xdrproc_t) xdr_chatUsrId_t, (caddr_t) argp,
		(xdrproc_t) xdr_chatparleerRes, (caddr_t) &clnt_res,
		TIMEOUT) != RPC_SUCCESS) {
		return (NULL);
	}
	return (&clnt_res);
}
```

Fichero chat_cif.cpp

```
/*chat_cif.c -> Fichero de interfaz del Cliente.*/

#include "chat.h"
#include "buzon.h"
/*CLIENT * clnt_create (
	char *host,		//Dirección del servidor.
	u_long prognum,		//Número del programa.
	u_long versnum,		//Versión del programa.
	char *protocolo		//Protocolo de transporte usado.
)*/

static CLIENT *clntChat;	/*Entidad Cliente, devulta por la función clnt_create(). Dicha entidad, debe
				incluirse en todas las RPC's dirigidas al servidor.*/
static buzonCodigo_t status;	/*Valor de retorno al Cliente*/
chatCodigo_t result;		/*Valor de retorno de las llamadas a las funciones del Servidor*/

buzonCodigo_t buzonConectar(char *usuario, buzonUsrId_t *usrId)
{
	/*Argumento de la fx. chat_Conectar_1(), contine el nombre del usuario, introducido por el cliente.*/
	chatNombreUsr_t *arg;
	static chatConectRes result_conn;	/*Resultado devuelto por chat_Conectar_1()*/
	arg= &usuario;				/*Referenciar el nombre introducido por el usuario.*/
	char Direccion_Servidor[15];		//Buffer temporal para almacenar la dirección IP del servidor.
	int fd_ser;	/*Descriptor de fichero donde se encuentra almacenana la dirección IP del servidor.*/
	/*Variable que contiene la direccion IP del servidor. Utilizada en el primer parámetro de la función
	clnt_create.*/
	char *Servidor;
	/*i cantidad de bytes realmente leídos del fichero donde se almacena la dirección del servidor.*/
	int i,j;
	memset(Direccion_Servidor,'\0',sizeof(Direccion_Servidor));
	/*Creacion de un programa hijo, que se encarga de localizar la dirección IP del servidor remoto y
	almacenarla en un fichero.*/
	if (fork()==0)
		/*Ejecución del programa que localiza el servidor.*/
		execlp("/usr/local/Cliente_Chat/Recursos/localizador", "localizador",NULL, NULL);
	wait(NULL);	//Sincronización de padre e hijo.
	//Abrir el fichero que contiene la dirección del servidor.
	fd_ser=open("/usr/local/Cliente_Chat/Recursos/Direccion_Servidor",O_RDONLY);
	if(fd_ser==-1)		/*Si el servidor no fue encontrado, el fichero no se pudo crear.*/
		return buzonNOK;
	i=read(fd_ser,Direccion_Servidor,15);		/*Leer del fichero la direccion IP del servidor.*/
	if(i==-1)		/*Error en la lectura del fichero.*/
		return buzonNOK;
```

```
	Servidor=(char *)malloc((strlen(Direccion_Servidor))*sizeof(char));
	memset(Servidor,'\0',sizeof(Servidor));
	for(j=0; j<i; j++)		/*Copiar caracter a caracter la direccion IP del servidor, a la variable Servidor.*/
		Servidor[j]= Direccion_Servidor[j];
	Servidor[j]='\0';		//Indicación de fin de cadena.
	/* clnt_create () -> Rutina para la creación de la entidad cliente.
	RHOST -> Nombre del anfitrión remoto donde se encuentra el servidor
	CHAT_PROGRAM -> Número del programa servidor
	CHAT_VERSION ->Versión del programa servidor
	tcp -> Tipo de protocolo de transporte utilizado*/
	if((clntChat=clnt_create(Servidor,CHAT_PROGRAM,CHAT_VERSION,"tcp")) !=NULL)
	{
		result_conn=*chat_conectar_1(arg,clntChat);	/*Llamar a la función encargada de crear la
								conexión(tubería).*/
		switch(result_conn.status)
		{
		  case chatOK:		//Operación realizada correctamente
			status=buzonOK;
			/*Copiar el valor del Id de usuario devulto por el servidor, a la variable que se ha
			pasado por referncia.*/
			*usrId=result_conn.chatConectRes_u.UsrId;
			break;

		  /*Se ha alcanzado el máximo número de usuarios conectados simultáneamente.*/
		  case chatDemasiadosUsr:
			status=buzonDemasiadosUsr;
			break;

		  case chatUsrYaDadoAlta:	//Nombre de usuario duplicado.
			status=buzonUsrYaDadoAlta;
			break;

		  default:		//La operación no se realizó correctamente.
			status = buzonNOK;
			break;

	}
	else	//Si no se encontró el servidor.
		status = buzonNOK;
	return status;
}

buzonCodigo_t buzonDesconectar( buzonUsrId_t buzonUsrId)
{
	chatUsrId_t *arg;	/*Argumento pasado a la funcion chat_desconectar_1()*/
	/*Copiar el Id de usuario de quien se desconectará al argumento de la función
	chat_desconectar_1().*/
	arg= &buzonUsrId;
	result = *chat_desconectar_1(arg, clntChat);	//Llamar a la función en el stub del cliente.
	switch(result)	{
		case chatOK:	//Operación realizada correctamente.
			status=buzonOK;
			/*Una macro que destruye la asa RPC del cliente. La destrucción usualmente implica
			la liberaciónde estructuras de datos privadas, incluyendo el propio clntChat. El uso
			de clntChat es indefinido tras llamar a clnt_destroy().*/
			clnt_destroy(clntChat);
			unlink("/usr/local/Cliente_Chat/Recursos/Direccion_Servidor");
			break;
```

```
                case chatUsrNoDadoAlta:       /*El usuario que se desea desconectar no está dado de alta.*/
                        status=buzonUsrNoDadoAlta;
                        break;
                default:            /*La operación no se realizó correctamente.
                        status = buzonNOK;
                        break;
        }
        return status;
}

buzonCodigo_t buzonEscribir( buzonUsrId_t buzonUsrId, char *msj,char* tipo)
{
        argesc_t arg;               /*Argumento pasados a la Fx. chat_escribir_1*/
        arg.UsrId= buzonUsrId;            /*Copiar el Id del usuario que escribirá en la Sala.*/
        /*Referenciar a la dirección donde está contenido el mensaje que se escribirá*/
        arg.chatMsj = msj;
        arg.tipo=tipo;    //Referencia la dirección donde se encuentra el tipo del mensaje a escribir.
        if((result=*chat_escribir_1(&arg, clntChat))!=NULL) //Llamar a la función en el stub del cliente.
        {
                switch(result)
                {
                        case chatOK:              //Operación realizada correctamente.
                                status=buzonOK;
                                break;

                        //El usuario que desea escribir en el buzón no está dado de alta
                        case chatUsrNoDadoAlta:
                                status=buzonUsrNoDadoAlta;
                                break;

                        default:                  //La operación no se realizó correctamente.
                                status = buzonNOK;
                                break;
                }
        }
        else
                status=buzonServidor;             //Servidor fuera de servicio.
        return status;
}

buzonCodigo_t buzonLeer(buzonUsrId_t buzonUsrId,char *usuario,char *msj,char *tipo)
{
        chatLeerRes *result_leer;         /*Resultado devuelto por caht_leer_1()*/
        result_leer =(chatLeerRes*)malloc(sizeof(chatLeerRes));
        //Llamar a la función en el stub del cliente.
        if((result_leer =chat_leer_1(&buzonUsrId, clntChat))!=NULL)
        {
                switch(result_leer->status)
                {
                        case chatOK:              //Operación realizada correctamente.
                                status=buzonOK;
                                /*Copiar el nombre del usario que escribió en el buzón, en la variable
                                usuario, pasada por referencia.*/
                                strcpy(usuario,result_leer->chatLeerRes_u.chatContestacion.
                                chatNombreUsr);
                                /*Copiar el mensaje que se leyó del buzón, en la variable msj, pasada por
                                referencia.*/
                                strcpy(msj,result_leer->chatLeerRes_u.chatContestacion.chatMsj);
```

```
                                strcpy(tipo, result_leer->chatLeerRes_u.chatContestacion.tipo);
                                break;

                        /*El usuario que desea leer del buzón no está dado de alta.*/
                        case chatUsrNoDadoAlta:
                                status=buzonUsrNoDadoAlta;
                                break;

                        default:        //La operación no se realizó correctamente.
                                status = buzonNOK;
                                break;
                }
        }

        else
                status =buzonServidor;          //Servidor fuera de servicio.
        return status;
}

buzonCodigo_t buzonConectados (buzonUsrId_t buzonUsrId,char *usuario_conectado)
{
        chatConectadosRes *result_conectados;
        result_conectados = (chatConectadosRes*) malloc (sizeof (chatConectadosRes));
        if((result_conectados = chat_conectados_1(&buzonUsrId,clntChat))!=NULL)
        {
                switch(result_conectados->status)
                {
                        case chatOK:    //Operación realizada correctamente.
                                status=buzonOK;
                                /*Copiar el nombre del usuario conectado, en la variable usuario_conectado,
                                pasada por referencia.*/
                                strcpy(usuario_conectado,result_conectados->chatConectadosRes_u.
                                argObtenerConectados.Usuario_Conectado);
                                break;

                        //El usuario representado por buzonUsrId no está dado de alta.
                        case chatUsrNoDadoAlta:
                                status=buzonUsrNoDadoAlta;
                                break;

                        default:        //La operación no se realizó correctamente.
                                status = buzonNOK;
                                break;
                }
        }
        else
                status = buzonServidor;         //Servidor fuera de servicio.
        return status;
}
buzonCodigo_t buzonCrearTuberias(buzonUsrId_t buzonUsrId)
{
        chatUsrId_t *arg;       //Argumento de la función chat_crartuberias_1().
        arg= &buzonUsrId;       //Referenciar el id del usuario.
        if((result=*chat_creartuberias_1(arg, clntChat))!=NULL)
        {
                switch(result)
                {
                        case chatOK:    //Operación realizada correctamente.
```

```
                        status=buzonOK;
                        break;

                case chatTuberiaActiva:         //La tubería ya ha sido habilitada anteriormente.
                        status = buzonTuberiaActiva;
                        break;

                default:                        //La operación no se realizó correctamente.
                        status = buzonNOK;
                        break;
                }
        }
        else
                status = buzonServidor;         //Servidor fuera de servicio.
        return status;
}

buzonCodigo_t buzonDestruirTuberias(buzonUsrId_t buzonUsrId)
{
        chatUsrId_t *arg;       //Argumento de la función chat_destruirtuberias_1.
        arg= &buzonUsrId;       //Referenciar el Id del usuario.
        if((result=*chat_destruirtuberias_1(arg,clntChat))!=NULL)
        {
                switch(result)
                {
                        case chatOK:    //Operación realizada correctamente.
                                status=buzonOK;
                                break;

                        default:        //La operación no se realizó correctamente.
                                status = buzonNOK;
                                break;
                }
        }
        else
                status = buzonServidor;         //Servidor fuera de servicio.
        return status;
}

buzonCodigo_t buzonEnviarNombreFich(char *chatNombreUsr, char *NombreFich)
{
        /*Estructura utilizada en el primer parámetro de la funció chat_enviarnombrefich_1. Contiene el
        nombre del fichero y el nombre del usuario a quien se le envia el fichero.*/
        parabr_t par_abr;
        //Referenciar el nombre del usuario a quien se le enviará el fichero.
        par_abr.chatNombreUsr=chatNombreUsr;
        par_abr.chatfichero=NombreFich;         //Referenciar el nombre del fichero.
        if((result= *chat_enviarnombrefich_1(&par_abr,clntChat ))!=NULL)
        {
                switch(result)
                {
                        case chatOK:            //Operación realizada correctamente.
                                status=buzonOK;
                                break;

                        case chatTuberiaLlena:  /*El nombre del fichero no se pudo escribir en la tubería*/
                                status=buzonTuberiaLlena;
                                break;
```

```
                        //La tubería del cliente a quien se envia el fichero, aún no ha sido creada.
                        case chatTuberiaNoActiva:
                                status=buzonTuberiaNoActvia;
                                break;

                        /*La tubería de intercambio de fichero esta siendo usada por otro cliente.*/
                        case chatTuberiaOcupada:
                                status= buzonTuberiaOcupada;
                                break;

                        default:        //La operación no se realizó correctamente.
                                status = buzonNOK;
                                break;
                }
        }
        else
                status = buzonServidor;         //Servidor fuera de servicio.
        return status;
}

buzonCodigo_t buzonRecibirNombreFich(buzonUsrId_t buzonUsrId, char *NombreFich)
{
        /*Estructura que almacena el resultado devuelto por la función chat_recibirnombrefich_1*/
        chatRecibirNombreRes *res;
        res =(chatRecibirNombreRes*)malloc(sizeof(chatRecibirNombreRes));
        if((res=chat_recibirnombrefich_1(&buzonUsrId, clntChat))!=NULL)
        {
                switch(res->status)
                {
                        case chatOK:    //Operación realizada correctamente.
                                status=buzonOK;
                                strcpy(NombreFich,res->chatRecibirNombreRes_u.chatContestacionparabr.
                                fichero);       /*Copiar en NombreFich el nombre del fichero a recibir*/
                                break;

                        case chatTuberiaVacia://No se ha escrito en la tubería el nombre del fichero a recibir.
                                status=buzonTuberiaVacia;
                                break;

                        default:        //La operación no se realizó correctamente.
                                status = buzonNOK;
                                break;
                }
        }
        else
                status = buzonServidor;         //Servidor fuera de servicio.
        return status;
}

buzonCodigo_t buzonPut(char *chatNombreUsr, char *contenido, buzonCantidad_t cantidad)
{
        /*Estructura pasada como primer parámetro a la función chat_put_1. Contienen el nombre del
        usuario que está enviando el fichero, así como el contenido y la cantidad en bytes de éste*/
        pares_t par_es;
        par_es.chatContenido.chatdatos_t_val=(char*)malloc(sizeof(char)*cantidad);
        par_es.chatNombreUsr= (char*)malloc(sizeof(char)*CHAT_MAX_NOMBRE);
        strcpy(par_es.chatNombreUsr,chatNombreUsr);         /*Copiar en par_es la información a enviar.*/
        memcpy(par_es.chatContenido.chatdatos_t_val,contenido,cantidad);
```

```
	par_es.chatContenido.chatdatos_t_len=cantidad;
	if((result= *chat_put_1(&par_es,clntChat))!=NULL)
	{
		switch(result)
		{
			case chatOK:	//Operación realizada correctamente.
				free(par_es.chatContenido.chatdatos_t_val);
				free(par_es.chatNombreUsr);
				status=buzonOK;
				break;

			case chatTuberiaLlena: /*La tubería está llena y la información no pudo ser escrita.*/
				free(par_es.chatContenido.chatdatos_t_val);
				free(par_es.chatNombreUsr);
				status=buzonTuberiaLlena;
				break;

			default:		//La operación no se realizó correctamente.
				status = buzonNOK;
				break;

		}
	}
	else
		status = buzonServidor;		//Servidor fuera de servicio.
	return status;
}

buzonCodigo_t buzonGet(buzonUsrId_t buzonUsrId, char *contenido, buzonCantidad_t *cantidad) {
	chatparleerRes *leer_res;	/*Estructura que almacena el resultado devuelto por chat_get_1.*/
	leer_res=(chatparleerRes*)malloc(sizeof(chatparleerRes));
	if((leer_res= chat_get_1(&buzonUsrId, clntChat))!=NULL) {
		switch(leer_res->status) {
			case chatOK:	//Operación realizada correctamente.
				/*Copiar en en la variable referenciada por contenido los bytes devueltos en
				el campo chatdatos_t_val de la estructura leer_res.*/
				memcpy(contenido,leer_res->chatparleerRes_u.parleer.chatContenido.
				chatdatos_t_val,leer_res->chatparleerRes_u.parleer.chatContenido.
				chatdatos_t_len);
				/*Copiar en la variable referenciada por cantidad, el número de bytes
				indicados por el campo chatdatos_t_len de la estructura leer_res.*/
				*cantidad=leer_res->chatparleerRes_u.parleer.chatContenido.
				chatdatos_t_len;
				status= buzonOK;
				break;
			case chatTuberiaVacia:		/*No hay ningún contenido en la tubería.*/
				status=buzonTuberiaVacia;
				break;

			case chatFinFichero:		/*Ya se ha recibido todo el contenido del fichero.*/
				status=buzonFinFichero;
				break;

			default:		//La operación no se realizó correctamente.
				status= buzonNOK;
				break;
		}
	}
```

```
        else
                status = buzonServidor;         //Servidor fuera de servicio.
        return(status);
}
```

Fichero ccliente_conectar.h

```
/* ccliente_conectar.h
Fichero de cabecera, donde se declara la clase cCliente_Conectar, derivada de la clase Cliente_Conectar, generada automáticamente por KDevelop, apartir del fichero Cliente_Conectar.ui y que representa el diálogo mostrado al momento de la conexión de un cliente.*/

#ifndef CCLIENTE_CONECTAR_H
#define CCLIENTE_CONECTAR_H

#include "../debug/src/Cliente_Conectar.h"      /*Ruta donde se crea el fichero de cabera y el de
                                                implementación de la clase Cliente_Conectar.*/
#include <qlineedit.h>
#include <qstring.h>
#include <qmessagebox.h>
#include "buzon.h"

class cCliente_Conectar : public Cliente_Conectar
{
public:
        cCliente_Conectar();            //Constructor.
        ~cCliente_Conectar();           //Destructor.
        QString QNombre_Usuario;        //Nombre introducido por el usuario.
        buzonUsrId_t Id_Usuario;        //Id de usuario, devuelto por el servidor.
        buzonCodigo_t resul;            //Resultado de la función buzonConectar().
        char *NombreUsr;                //Nombre de usuario en formato ASCII.
        bool opcion;                    //Bandera para indicar que se ha conectado el usuario.

public slots:
        void cmdAceptar();              //Función controladora del botón btnAceptar.
        void cmdCancelar();             //Función controladora del botón btnCancelar.
};
#endif
```

Fichero ccliente_conectar.cpp

```
/*ccliente_conectar.cpp
Fichero de implementación de la clase cCliente_Conectar.*/

#include "ccliente_conectar.h"          /*Fichero donde está declarada la clase cCliente_Conectar*/

cCliente_Conectar::cCliente_Conectar(): Cliente_Conectar()      /*Llamar al constructor de la clase
                                                                Cliente_Conectar.*/
{
        NombreUsr=NULL;
        txtNombre_Usuario->setFocus();  //Poner el foco a la caja de texto txtNombre_Usuario.
        opcion=false;
}

cCliente_Conectar::~cCliente_Conectar()
{
}
```

```
void cCliente_Conectar::cmdAceptar()
{
	QNombre_Usuario = txtNombre_Usuario->text();		/*Obtener el nombre de usuario
								introducido por el cliente.*/
	if(QNombre_Usuario.isEmpty())			//El nombre de usuario es una cadena vacía.
		QMessageBox::about( this, "Chat Multiusuario","Error en Nombre de Usuario."
		"\nDebe escribir un Nombre de Usuario.");
	else if(QNombre_Usuario.length() >12)		/*Longitud del Nombre de usuario es mayor de
								12 caracteres.*/
	{
		QMessageBox::about( this, "Chat Multiusuario","Error: Nombre de Usuario demasiado
		largo.");
		txtNombre_Usuario->setText("");
	}
	else		/*El nombre introducido por el usuario no es una cadena vacía y no es mayor de
			caracteres.*/
	{
		NombreUsr=(char*)QNombre_Usuario.ascii();	/*Transformar una cadena de QString a
								representación en ASCII.*/
		resul = buzonConectar(NombreUsr, &Id_Usuario);	/*Llamado a la función encargada
								de realizar la conexión a la Sala
								de Chat.*/
		if(resul==buzonDemasiadosUsr)		/*Se ha alcanzado el máximo número de
								usuarios conectados simultáneamente.*/
		{
			QMessageBox::about( this, "Chat Multiusuario","No se ha podido
			conectar.""\nDemasiados usuarios conectados.");
			txtNombre_Usuario->setText("");
		}
		if(resul==buzonUsrYaDadoAlta)		/*El Nombre de Usuario introducido, está
								duplicado.*/
		{
			QMessageBox::about( this, "Chat Multiusuario","No se ha podido
			conectar.""\nNombre de usuario duplicado.");
			txtNombre_Usuario->setText("");
		}
		if(resul==buzonNOK)			//Error de sistema o de conexión.
		{
			QMessageBox::about( this, "Chat Multiusuario","Error al intentar conectarse.");
			txtNombre_Usuario->setText("");
		}
		if(resul==buzonOK)			//Conexión exitosa
		{
			QMessageBox::about( this, "Chat Multiusuario","Bienvenid@ " +
			QNombre_Usuario + " a la Sala de Chat.");
			opcion=true;		//Señalar la conección
			close();		//Cerrar  el diálogo.
		}
	}
}

void cCliente_Conectar::cmdCancelar()
{
	opcion=false;
	close();			//Cerrar el diálogo.
}
```

Fichero cliente_chat.h

```
/*cliente_chat.h
Fichero de cabecera generado por KDevelop cuando se creó el proyecto.
Aquí se declara la clase Cliente_Chat, derivada de la clase Cliente_Principal, generada por KDevelop, a
partir  del fichero Cliente_principal.ui y que representa la interfaz de usuario.
En este fichero se redefinen las funciones controladoras de los widgets del diálogo de la interfaz del cliente.*/

#ifndef _CLIENTE_CHAT_H_
#define _CLIENTE_CHAT_H_
#define MAX_RUTA_FICH       100

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

#include <kmainwindow.h>
#include "../debug/src/Cliente_Principal.h"      /*Ruta donde se crea el fichero de cabecera y el
                                                   de implementación de la clase Cliente_Principal.*/
#include <qpushbutton.h>
#include <qtextedit.h>
#include <qlabel.h>
#include <qlistbox.h>
#include "ccliente_conectar.h"
#include <qfiledialog.h>
#include <qtimer.h>
#include <qmessagebox.h>
#include <qstring.h>
#include <qfile.h>

class Cliente_Chat : public Cliente_Principal
{
   Q_OBJECT
public slots:
        void cmdConectar();             //Función controladora del botón btnConectar.
        void cmdSalir();                //Función controladora del botón btnSalir.
        void cmdDesconectar();          //Función controladora del botón btnDesconectar.
        void cmdEnviar();               /*Función controladora del botón btnEnviar y de la tecla Enter en
                                          la caja de texto multilínea txtSalida.*/
        void cmdEnviarImagen();         //Función controladora del botón btnEnviarImagen.
        void cmdEnviarSonido();         //Función controladora del botón btnEnviarSonido.
        void funLeer();                 //Función controladora de la lectura en el buzón.
        void cmdTextSalida();           /*Función controladora para la caja multilínea txtSalida. Evita que
                                          se escriben más de mil caracteres en ésta.*/

public:
        QTimer *timer;          //Temporizador para activar la función que permite leer en Sala de Chat.
        cCliente_Conectar pclnt_conectar;   /*Objeto de la clase que representa el diálogo de
                                              conexión del cliente.*/
        buzonUsrId_t Id_Usuario;            //Id de usuario devuelto por el Servidor.
        buzonCodigo_t resul;                //Valor de retornos de las FXs. de chat_sif.cpp.
        buzonCantidad_t cantidad_Enviar;    //Cantidad de bytes del fichero que se está enviando.
        buzonCantidad_t cantidad_Recibir;   //Cantidad de bytes del fichero que se está recibiendo.
        char *NombreUsr;                    //Nombre de usuario del Cliente
        char *mensaje_enviado;              //Mensaje enviado por el cliente.
        char *NuevoUsr;                     //Nombre de los nuevos usuarios conectados.
        char *tipo;                         //Tipo del mensaje leído desde el buzón.
        char *Usr_Escribe;                  //Nombre del autor de los mensajes recibidos.
        char *mensaje_recibido;
```

```
        char *Usr_Seleccionado;                 //Usuario al que se le enviará un fichero.
        char *fich_Enviar;                      //Nombre del fichero a enviar.
        char *fich_Recibir;                     //Nombre del fichero a recibir.
        char *ruta_fich_Enviar;                 //Ruta absoluta del fichero a enviar.
        char *datosfich_Recibir;        /*Buffer para almacenar los bytes del fichero que se está recibiendo.*/
        int item_seleccionado;          //Índice del elemento seleccionado de la lista "Usuarios en Linea".
        int contador_car;       //Contador del número de caracteres escritos en la caja multilínea txtSalida.

protected:
        void closeEvent(QCloseEvent *event);            //Función controladora del evento de cierre.

public:
        /*Default Constructor*/
        Cliente_Chat();
        void  Obtener_Usuarios_Conectados(); /*Función que permite obtener el nombre de los usuarios
                                               que están previamente conectados en la Sala de Chat.*/
        bool Enviar_Fichero();                  //Función encargada de enviar los bytes del fichero.
        bool Recibir_Fichero();                 //Función encargada de recibir los bytes del fichero.
        void Habilitarbtn_Inhabilitarbtn(bool); /*Función que permitirá habilitar o inhabilitar los widgets
                                                  en dependencia del argumento que reciba.*/
        void Servidor_Inactivo();               /*Función para cambiar el estado de los widgets, cuando
                                                  el Servidor está inactivo.*/
        /*Default Destructor*/
        virtual ~Cliente_Chat();
};
#endif // _CLIENTE_CHAT_H_
```

Fichero cliente_chat.cpp

/*cliente_chat.cpp -> Fichero que se encarga de controlar los eventos producidos por el usuario, y de llamar a la función controladora correspondiente.*/

```
#include "cliente_chat.h"
#include <qlabel.h>
#include <kmainwindow.h>
#include <klocale.h>
#include <stdlib.h>

Cliente_Chat::Cliente_Chat():Cliente_Principal( )
{
        Habilitarbtn_Inhabilitarbtn(false);
        btnConectar->setEnabled(true);
        btnSalir->setEnabled(true);
        //Crear un contador de tiempo y asociarle la función manejadora funLeer().
        timer = new QTimer(this);
        connect(timer, SIGNAL(timeout()), this, SLOT(funLeer()));
        //Asignar memoria a las variables.
        NuevoUsr = (char*)malloc(sizeof(char) * CHAT_MAX_NOMBRE);
        Usr_Escribe = (char*)malloc(sizeof(char) * CHAT_MAX_NOMBRE);
        tipo = (char*)malloc(sizeof(char) *  CHAT_MAX_TIPO);
        mensaje_recibido = (char*)malloc(sizeof(char) * CHAT_MAX_LONG);
        Usr_Seleccionado = (char*)malloc(sizeof(char) * CHAT_MAX_NOMBRE);
        fich_Enviar= (char*)malloc(sizeof(char) *CHAT_MAX_NOMBRE_FICHERO);
        ruta_fich_Enviar = (char*)malloc(sizeof(char) *MAX_RUTA_FICH);
        datosfich_Recibir = (char *) malloc (sizeof (char) *CHAT_MAXDATOS_FICHERO);
        fich_Recibir= (char*)malloc(sizeof(char) *CHAT_MAX_NOMBRE_FICHERO);
        contador_car=0;
}
```

```
Cliente_Chat::~Cliente_Chat()
{
	//Liberar la memoria asignada en el constructor.
	free(fich_Recibir);
	free(datosfich_Recibir);
	free(ruta_fich_Enviar);
	free(fich_Enviar);
	free(Usr_Seleccionado);
	free(mensaje_recibido);
	free(tipo);
	free(Usr_Escribe);
	free(NuevoUsr);
	delete(timer);
}

void Cliente_Chat::cmdConectar()
{
	/*Visualiza el diálogo de conexión y detiene la ejecucion de la aplicación principal hasta que el
	diálogo es cerrado.*/
	pclnt_conectar.exec();

	//Si la conexión fue exitosa.
	if(pclnt_conectar.opcion)
	{
		resul= pclnt_conectar.resul;
		if(resul==buzonOK)
		{
			Id_Usuario=pclnt_conectar.Id_Usuario;
			NombreUsr= pclnt_conectar.NombreUsr;
			lblNombre_Usr->setText(NombreUsr);
			Habilitarbtn_Inhabilitarbtn(true);
			btnConectar->setEnabled(false);
			btnSalir->setEnabled(false);
			contador_car=0;
			Obtener_Usuarios_Conectados();	/*Obtener el nombre de los usuarios
								conectados.*/
			timer->start(1000);		//Iniciar el temporizador.
		}
	}
}

void Cliente_Chat::cmdDesconectar()
{
	imer->stop();		//Detener el temporizador.
	resul=buzonDesconectar(Id_Usuario);
	if(resul==buzonOK)
	{
		QMessageBox::about( this, "Chat Multiusuario","Se ha desconectado de la Sala de
		Chat.");
		listConectados->clear();		//Limpiar la lista de "Usuarios en Linea".
		//Limpiar las cajas de texto multilínea.
		lblNombre_Usr->setText("");
		txtEntrada->setText("");
		Habilitarbtn_Inhabilitarbtn(false);	//Inhabilitar los widgets.
		btnConectar->setEnabled(true);
		btnSalir->setEnabled(true);
	}
```

```
        //Si ha ocurrido un problema en la desconección.
        else
        {
                QMessageBox::about( this, "Chat Multiusuario","No se ha podido desconectar de la
                Sala de Chat.");
                timer->start(1000);                 //Iniciar de nuevo el temporizador.
        }
}

/*Función que controla la pulsación del botón cmdSalir.*/
void Cliente_Chat::cmdSalir()
{
        close();
}

void Cliente_Chat::cmdEnviar()
{

        QString QSalida;                /*Variable que almacenará el contenido de la caja de texto
                                        multilínea txtSalida.*/
        QSalida= txtSalida->text();
        if(QSalida.isEmpty())           //Si es un mensaje nulo.
                return;
        else
        {
                mensaje_enviado=(char*)QSalida.ascii();         /*Convertir una cadena de Qstring a
                                                                representación en códgio ASCII.*/
                resul=buzonEscribir(Id_Usuario,mensaje_enviado,TIPO3);      /*Enviar el mensaje a la
                                                                            sala de chat.*/
                if(resul==buzonServidor)    /*Si el servidor no está activo se inhabilitan todos los widget.*/
                        Servidor_Inactivo();
                if(resul==buzonNOK)
                                txtEntrada->setText("El mensaje no se ha enviado a la Sala de Chat.");
                if(resul==buzonOK)
                        txtSalida->setText("");
        }
}

void Cliente_Chat::cmdEnviarImagen()
{

        int i,j=0;
        /*Obtener la ruta absoluta del fichero a enviar a través, de un diálogo estándar.*/
        QString fnImagen = QFileDialog::getOpenFileName( QString::null, "Imagen (*.jpeg *.jpg)",
        this,QString::null, "Abrir fichero de Imagen a enviar." );
        if(fnImagen!=QString::null)     //Si se ha seleccionado un fichero.
        {
                /*Obtner de la lista de "Usuarios en Linea"el nombre del usuario al que se le enviará el
                fichero.*/
                item_seleccionado = listConectados->currentItem();
                if(item_seleccionado==-1)       /*Si no se ha seleccionado un usuario destino.*/
                        QMessageBox::about( this, "Chat Multiusuario","Primero debe seleccionar un
                        Usuario.");
                else
                {
                        strcpy(ruta_fich_Enviar,(char*)fnImagen.ascii());
                        memset(fich_Enviar,'\0',sizeof(fich_Enviar));
```

```
                        /*Extraer de la ruta absoluta el nombre del fichero y almacenarlo en la variable
                        fich_Enviar*/
                        for(i=0;i<(int)strlen(ruta_fich_Enviar);i++)
                        {
                                if(ruta_fich_Enviar[i]!='/')
                                        fich_Enviar[j++]=ruta_fich_Enviar[i];
                                else
                                {
                                        memset(fich_Enviar,'\0',sizeof(fich_Enviar));
                                        j=0;
                                }
                        }
                        fich_Enviar[j]='\0';                    /*Indicar el fin de cadena.*/
                        /*Obtener el nombre del usuario destino.*/
                        Usr_Seleccionado = (char*)listConectados->currentText().ascii();
                        /*Notificar al Usr_Seleccionado el envio del fichero */
                        resul=buzonEscribir(Id_Usuario,Usr_Seleccionado,TIPO4);
                }
        }
}

void Cliente_Chat::cmdEnviarSonido()
{
        int i,j=0;
        QString fnSonido= QFileDialog::getOpenFileName( QString::null, "Sonido (*.mp3)", this,
        QString::null, "Abrir fichero de Sonido a enviar." );
        if(fnSonido!=QString::null)
        {
                item_seleccionado = listConectados->currentItem();
                if(item_seleccionado==-1)
                        QMessageBox::about( this, "Chat Multiusuario","Primero debe seleccionar un
                        Usuario.");
                else
                {
                        strcpy(ruta_fich_Enviar,(char*)fnSonido.ascii());
                        memset(fich_Enviar,'\0',sizeof(fich_Enviar));
                        for(i=0;i<(int)strlen(ruta_fich_Enviar);i++)
                        {
                                if(ruta_fich_Enviar[i]!='/')
                                        fich_Enviar[j++]=ruta_fich_Enviar[i];
                                else
                                {
                                        memset(fich_Enviar,'\0',sizeof(fich_Enviar));
                                        j=0;
                                }
                        }
                        fich_Enviar[j]='\0';
                        Usr_Seleccionado = (char*)listConectados->currentText().ascii();
                        resul=buzonEscribir(Id_Usuario,Usr_Seleccionado,TIPO4);
                }
        }
}

void Cliente_Chat::funLeer()
{
        QString visualizar;                     //Mensaje visualizado en la caja de texto multilínea txtEntrada.
        char *aux;      /*Buffer temporal, en el que se copiarán uno a uno los nombres de los items de la
                        lista "Usuarios en Línea"*/.
```

```
buzonCodigo_t resul_leer;       //Valor de retorno al cliente.
/*Llamada a la función en el fichero de interfaz del cliente.*/
resul_leer=buzonLeer(Id_Usuario,Usr_Escribe,mensaje_recibido,tipo);
if(resul_leer==buzonServidor)                   /*El servidor no está brindando servicios.*/
        Servidor_Inactivo();
if(resul_leer==buzonOK )                        /*Si se pudo leer del buzón.*/
{
        if(strcmp(tipo, TIPO3)==0)              //Mensaje recibido de otro usuario.
        {
                visualizar=QString(QString(Usr_Escribe) + " > " + QString(mensaje_recibido));
                txtEntrada->append(visualizar);
        }
        if(strcmp(tipo, TIPO1)==0)              //Mensaje de un nuevo usuario conectado.
        {
                if(strcmp(NombreUsr,Usr_Escribe)!=0)
                        /*Agregar el nuevo usuario conectado a la lista de "Usuarios en Linea".*/
                        listConectados->insertItem(Usr_Escribe);                }
        if(strcmp(tipo, TIPO2)==0)              //Mensaje de desconexión de un usuario.
        {
                /*Buscar en la lista el nombre del usuario desconectado y removerlo de ésta.*/
                for(int i=0; i<(int)listConectados->count(); i++)
                {
                        aux=(char*)listConectados->text(i).ascii();
                        if(strcmp(Usr_Escribe,aux)==0)
                                listConectados->removeItem(i);
                }
        }
        if(strcmp(tipo, TIPO4)==0) /**/
        {
                int re =QMessageBox ::question( this,"Chat Multiusuario","Aceptar un fichero.",
                QMessageBox::Yes, QMessageBox::No);     /*Consultar al usuario si desea o
                                                        no recibir el fichero.*/
                if(re== 0)      //El fichero fue aceptado por el otro usuario.
                {
                        /*Enviar un mensaje de aceptación del fichero.*/
                        resul=buzonEscribir(Id_Usuario,Usr_Escribe,TIPO5);
                        if(resul==buzonNOK)
                                txtEntrada->setText("El mensaje no se ha enviado a la Sala de
                                Chat.");
                        if(resul==buzonServidor)
                                Servidor_Inactivo();
                        if(resul == buzonOK)
                        {
                                //Crear la tubería por donde se recibirá el fichero
                                resul=buzonCrearTuberias(Id_Usuario);
                                if(resul==buzonServidor)
                                        Servidor_Inactivo();
                                if(resul==buzonNOK)    //No se pudo crear la tubería
                                        txtEntrada->setText("El fichero no ha sido transferido.");
                                if(resul==buzonTuberiaActiva)       /*La tubería ya fue creada.*/
                                        txtEntrada->setText("El fichero no ha sido transferido"
                                        "\nActualmente se recibe otro fichero.");
                                if(resul==buzonOK)
                                {       //Recibir el nombre del fichero
                                        do
                                        {
                                                resul=buzonRecibirNombreFich(Id_Usuario,
                                                fich_Recibir);
```

```
                                        }while(resul==buzonTuberiaVacia);
                                        if(resul==buzonServidor)
                                                Servidor_Inactivo();
                                        if(resul==buzonNOK) //No se pudo crear la tubería
                                                txtEntrada->setText("El fichero no ha sido
                                                transferido.");
                                        if(resul==buzonOK)
                                        {
                                                /*Llamar a la función encargada de recibir el
                                                fichero.*/
                                                if(Recibir_Fichero())
                                                {
                                                        /*Fichero recibido satisfactoriamente.*/
                                                        QMessageBox::about( this, "Chat
                                                        Multiusuario","Fichero transferido.");
                                                }
                                        }
                                }
                        }
                }
                else      //Si no aceptó el fichero
                {
                        /*Enviar un mensaje de no aceptación del fichero.*/
                        resul=buzonEscribir(Id_Usuario,Usr_Escribe,TIPO6);
                        if(resul==buzonNOK)
                                txtEntrada ->setText("El mensaje no se ha enviado a la Sala de
                                Chat.");
                }
        }
        if(strcmp(tipo, TIPO5)==0)        /*Mensaje de tipo aceptación de fichero.*/
        {
                //Enviar el nombre del fichero.
                resul=buzonEnviarNombreFich(Usr_Seleccionado,fich_Enviar);
                if(resul==buzonServidor)
                        Servidor_Inactivo();
                if(resul==buzonNOK)     //No se pudo enviar el nombre del fichero.
                        txtEntrada->setText("El fichero no ha sido transferido");
                /*Otro usuario está usando la tubería para transferir un fichero.*/
                if(resul==buzonTuberiaOcupada)
                        QMessageBox::about( this, "Chat Multiusuario","En este momento no se
                        puede transferir el fichero al usuario deseado.");
                //Aún no existe la tubería de intercambio de fichero.
                if(resul==buzonTuberiaNoActvia)
                {
                        /*Enviar el nombre del fichero mientras no se haya creado la tubería.*/
                        do
                        {
                                resul=buzonEnviarNombreFich(Usr_Seleccionado,fich_Enviar);
                                if(resul==buzonServidor)
                                        Servidor_Inactivo();
                                if(resul==buzonNOK)
                                        txtEntrada->setText("El fichero no ha sido transferido.");
                        }while(resul==buzonTuberiaNoActvia);
                }
                if(resul==buzonOK)      /*Nombre de fichero enviado con éxito.*/
                {
                        /*Llamar a la función encargada de enviar el contenido del fichero.*/
                        if(Enviar_Fichero())
```

```
                                        QMessageBox::about( this, "Chat Multiusuario","Fichero
                                        transferido.")
                        }
                }
                if(strcmp(tipo, TIPO6)==0)      /*Se ha rechazado la transferencia del fichero.*/
                        QMessageBox::about( this, "Chat Multiusuario","El usurio "
                        +QString(Usr_Seleccionado) +" no ha aceptado el fichero.");
        }
}

void Cliente_Chat::cmdTextSalida()
{
        contador_car++;
        /*Controlar el número de caracteres introducidos en la caja de texto multilínea txtSalida.*/
        if(contador_car==CHAT_MAX_LONG)  {
                contador_car=0;
                cmdEnviar();
        }
}

void Cliente_Chat::closeEvent(QCloseEvent *event)
{
        if(btnDesconectar->isEnabled())         /*Si se está saliendo de la aplicación sin desconectarse
                                                previamente.*/
        {
                int re =QMessageBox ::question( this,"Chat Multiusuario","Desea realmente
                desconectarse de la Sala de Chat y salir." ,QMessageBox::Yes, QMessageBox::No);
                if(re==0)
                {
                        timer->stop();
                        resul=buzonDesconectar(Id_Usuario);
                        event->accept();        //Ejecutar el evento de cierre.
                        QMessageBox::about( this, "Chat Multiusuario","Saliendo de la sala de Chat.");
                }
                else
                        event->ignore();        //Ignorar el evento de cierre.
        }
        else
                event->accept();        //Cerrar el widget.
}

void Cliente_Chat:: Obtener_Usuarios_Conectados() {
        int i;
        /*Obtener uno a uno el nombre de los usuarios conectados actualmente.*/
        for(i=0; i<CHAT_MAX_USR;i++) {
                if(i!=Id_Usuario)       /*Si el Id de usuario no es el propio.*/
                {
                        resul=buzonConectados(i,NuevoUsr);
                        if(resul==buzonOK)
                        /*Agregar a la lista de "Usuarios en Línea" el nombre del usuario conectado.*/
                                listConectados->insertItem(NuevoUsr);
                        if(resul==buzonServidor)
                                Servidor_Inactivo();
                }
        }
}
```

```
bool Cliente_Chat::Enviar_Fichero()
{
	/*Buffer para almacenar los datos del fichero a enviar.*/
	char datosfich_Enviar[CHAT_MAXDATOS_FICHERO];
	ulong len;	/*Variable que almacena la cantidad de bytes leídos del fichero.*/

	/*Se asigna a un objeto QFile el nombre del fichero especificado por ruta_fich_Enviar.*/
	QFile qfichero_enviado(ruta_fich_Enviar);
	qfichero_enviado.open(IO_ReadOnly);		/*Abrir el fichero en modo lectura.*/
	memset(datosfich_Enviar,'\0',sizeof(datosfich_Enviar));

	/*Leer el contenido del fichero y almacenarlo en datosfich_Enviar.*/
	len=qfichero_enviado.readBlock(datosfich_Enviar, sizeof(datosfich_Enviar));

	while(len!=0)	/*Enviar el contenido del fichero mientras no se llegue al final de éste.*/
	{
		do /*Mientras la tubería no esté vacía se debe enviar el mismo bloque de bytes al usuario
		especificado por Usr_Seleccionado.*/
		{
			resul=buzonPut(Usr_Seleccionado,datosfich_Enviar,int(len));
			if(resul==buzonServidor)
			{
				qfichero_enviado.close();
				Servidor_Inactivo();
				return false;
			}
		}while(resul==buzonTuberiaLlena);
		memset(datosfich_Enviar,'\0',sizeof(datosfich_Enviar));
		/*Leer del fichero el siguiente bloque de bytes a enviar.*/
		len=qfichero_enviado.readBlock(datosfich_Enviar,sizeof(datosfich_Enviar));
	}
	do
	{
		//Mandar 0 bytes indicando el fin del dichero.
		resul=buzonPut(Usr_Seleccionado,datosfich_Enviar,0);
		if(resul==buzonServidor)
		{
			qfichero_enviado.close();
			Servidor_Inactivo();
			return false;
		}
	}while(resul==buzonTuberiaLlena );
	qfichero_enviado.close();	/*Cerrar el fichero.*/
	return true;	/*Notificar que el fichero se transmitió correctamente.*/
}

bool Cliente_Chat::Recibir_Fichero() {
	QFile qfichero_recibido(fich_Recibir);
	qfichero_recibido.open(IO_WriteOnly|IO_Truncate); /*Abrir el fichero en modo escritura.*/
	memset(datosfich_Recibir,'\0',sizeof(datosfich_Recibir));
	do {	/*Mientras la tubería esté vacía, seguir esperando el siguiente bloque de bytes del fichero.*/
		resul=buzonGet(Id_Usuario,datosfich_Recibir,&cantidad_Recibir);
		if(resul==buzonServidor) {
			remove(fich_Recibir);
			Servidor_Inactivo();
			return false;
		}
	}while(resul==buzonTuberiaVacia);
```

```
        while(resul!=buzonFinFichero) /*Leer de la tubería de intercambio de fichero mientras no se llegue al
        final del fichero.*/
        {
                /*Escribir en el fichero la cantidad de bytes indicados por cantidad_Recibir.*/
                qfichero_recibido.writeBlock(datosfich_Recibir,cantidad_Recibir);
                cantidad_Recibir=0;
                memset(datosfich_Recibir,'\0',sizeof(datosfich_Recibir));
                do
                {       /*Leer de la tubería el siguiente bloque de bytes del fichero.*/
                        resul=buzonGet(Id_Usuario,datosfich_Recibir,&cantidad_Recibir);
                        if(resul==buzonServidor)
                        {
                                remove(fich_Recibir);
                                Servidor_Inactivo();
                                return false;
                        }
                }while(resul==buzonTuberiaVacia);
        }
        qfichero_recibido.close();
        resul=buzonDestruirTuberias(Id_Usuario);        //Cerrar la tubería de intercambio de fichero.
        return true;
}

void Cliente_Chat::Habilitarbtn_Inhabilitarbtn(bool indicador)
{
        /*Habilitar o inhabilitar los widgets en función de la variable indicador. (true o false
        respectivamente.)*/
        btnDesconectar->setEnabled(indicador);
        btnEnviar->setEnabled(indicador);
        btnEnviarImagen->setEnabled(indicador);
        btnEnviarSonido->setEnabled(indicador);
        txtSalida->setEnabled(indicador);
        btnConectar->setEnabled(indicador);
        btnSalir->setEnabled(indicador);
}

void Cliente_Chat::Servidor_Inactivo()
{
        timer->stop();          /*Detener el contador de tiempo.*/
        QMessageBox::about( this, "Chat Multiusuario","Servidor no activo." "\nConectarse en otro
        momento.");
        Habilitarbtn_Inhabilitarbtn(false);
        btnConectar->setEnabled(true);          /*Habilitar el botón btnConectar.*/
        btnSalir->setEnabled(true);             /*Habilitar el botón btnSalir.*/
        listConectados->clear();                //Limpiar la lista de Usuarios en Linea.
        lblNombre_Usr->setText("");
        txtEntrada->setText("");                /*Limpiar la caja de texto multilínea txtEntrada*/
        txtSalida->setText("");                 /*Limpiar la caja de texto multilínea txtSalida.*/
}
#include "cliente_chat.moc"
```

Para obtener el fichero ejecutable del programa Cliente,

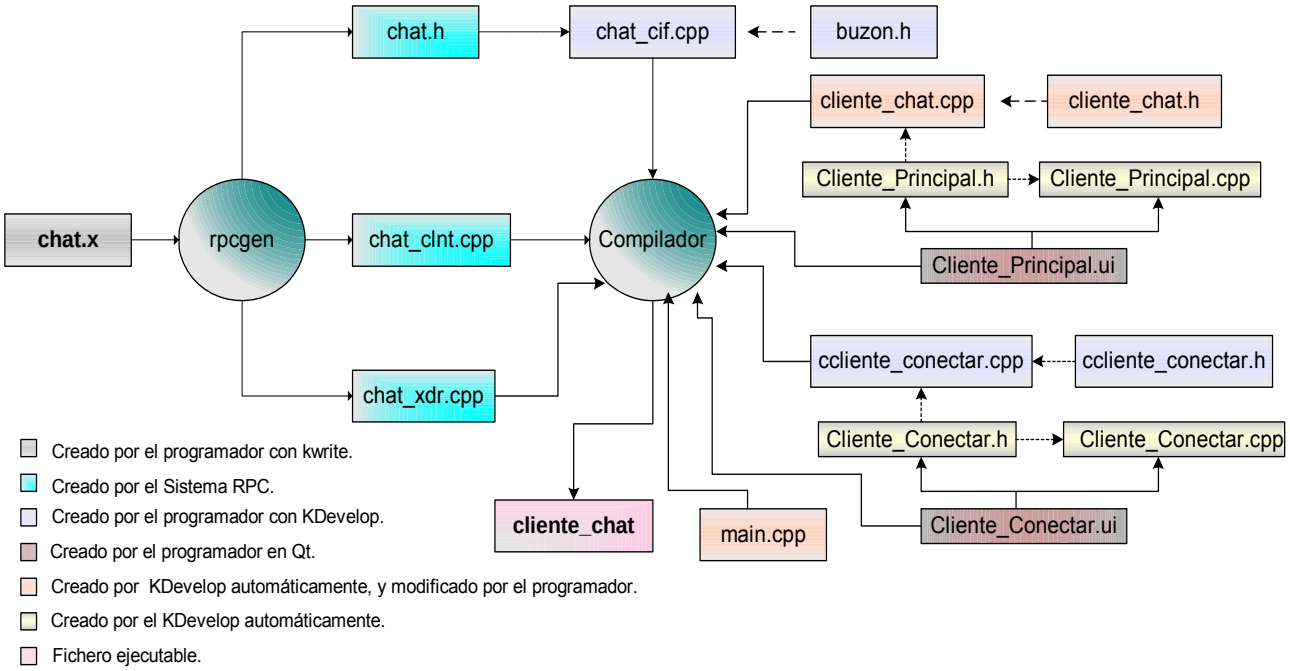


Fig. 30 Generar el fichero ejecutable del Cliente.

4. Código de la Aplicación Servidor.

La Aplicación Servidor consta de los siguientes ficheros:

Fichero chat_svc.c.

```
/*chat_svc.c
Stub del Servidor ->Formado por el programa principal (main) y el dispatcher que recibe el nombre del servidor seguido de un subrayado y la versión (chat_prog_1).
main ->Registra al servidor en el sistema Unix para indicarle que atienda a las llamadas del cliente.
Dispatcher -> Proporciona un "switch-case" para llamar a cada una de las funciones al recibir una petición de un cliente. */

#include "chat.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

/* Dispatcher. -> Contiene las rutinas de serealización y las rutinas de servicio escritas por el programador. Identifica y selecciona el procedimiento que invoca el cliente, devuelve un mensaje de error si el procedimiento no existe. Extrae del paquete que recibe por la red el argumento del paquete (rqstp). Invoca el procedimiento con el parámetro recibido. Recoge el resultado devuelto por el servicio y se lo envía al cliente. Retorna a svc_run().*/
```

```
static void
chat_program_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
	/*rqstp -> puntero a struct svc_req, contiene la credencial del cliente, así como el código de la
	operación solicitada.
	transp -> contiene los parámetros de la operación solicitada en formato externo XDR.*/

	union {
		chatNombreUsr_t chat_conectar_1_arg;
		chatUsrId_t chat_desconectar_1_arg;
		argesc_t chat_escribir_1_arg;
		chatUsrId_t chat_leer_1_arg;
		chatUsrId_t chat_conectados_1_arg;
		chatUsrId_t chat_creartuberias_1_arg;
		chatUsrId_t chat_destruirtuberias_1_arg;
		parabr_t chat_enviarnombrefich_1_arg;
		chatUsrId_t chat_recibirnombrefich_1_arg;
		pares_t chat_put_1_arg;
		chatUsrId_t chat_get_1_arg;
		char localizar_1_arg;
	} argument;	/*Parámetros de entrada de los procedimientos remotos.*/
	char *result;

	/*_xdr_argument -> Para deserializar los parámetros de entrada
	_xdr_result -> Para serealizar el resultado de la llamada a un determinado procedimiento.*/
	xdrproc_t _xdr_argument, _xdr_result;

	char *(*local)(char *, struct svc_req *);

	switch (rqstp->rq_proc)
	{
	/*Procedimiento de diagnóstico que simplemente retorna un cero.*/
	case NULLPROC:
		(void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
		return;

	case CHAT_CONECTAR:
		/*Dirección de la rutina que deserealiza los parámetros de entrada.*/
		_xdr_argument = (xdrproc_t) xdr_chatNombreUsr_t;
		/*Dirección de la rutina que serealiza el resultado de la llamada al procedimiento.*/
		_xdr_result = (xdrproc_t) xdr_chatConectRes;
		/*Dirección de la rutina del servicio correspondiente a la operación solicitada.*/
		local = (char *(*)(char *, struct svc_req *)) chat_conectar_1_svc;
		break;

	case CHAT_DESCONECTAR:
		_xdr_argument = (xdrproc_t) xdr_chatUsrId_t;
		_xdr_result = (xdrproc_t) xdr_chatCodigo_t;
		local = (char *(*)(char *, struct svc_req *)) chat_desconectar_1_svc;
		break;


	case CHAT_ESCRIBIR:
		_xdr_argument = (xdrproc_t) xdr_argesc_t;
		_xdr_result = (xdrproc_t) xdr_chatCodigo_t;
		local = (char *(*)(char *, struct svc_req *)) chat_escribir_1_svc;
		break;
```

```
	case CHAT_LEER:
		_xdr_argument = (xdrproc_t) xdr_chatUsrId_t;
		_xdr_result = (xdrproc_t) xdr_chatLeerRes;
		local = (char *(*)(char *, struct svc_req *)) chat_leer_1_svc;
		break;

	case CHAT_CONECTADOS:
		_xdr_argument = (xdrproc_t) xdr_chatUsrId_t;
		_xdr_result = (xdrproc_t) xdr_chatConectadosRes;
		local = (char *(*)(char *, struct svc_req *)) chat_conectados_1_svc;
		break;

	case CHAT_CREARTUBERIAS:
		_xdr_argument = (xdrproc_t) xdr_chatUsrId_t;
		_xdr_result = (xdrproc_t) xdr_chatCodigo_t;
		local = (char *(*)(char *, struct svc_req *)) chat_creartuberias_1_svc;
		break;

	case CHAT_DESTRUIRTUBERIAS:
		_xdr_argument = (xdrproc_t) xdr_chatUsrId_t;
		_xdr_result = (xdrproc_t) xdr_chatCodigo_t;
		local = (char *(*)(char *, struct svc_req *)) chat_destruirtuberias_1_svc;
		break;

	case CHAT_ENVIARNOMBREFICH:
		_xdr_argument = (xdrproc_t) xdr_parabr_t;
		_xdr_result = (xdrproc_t) xdr_chatCodigo_t;
		local = (char *(*)(char *, struct svc_req *)) chat_enviarnombrefich_1_svc;
		break;

	case CHAT_RECIBIRNOMBREFICH:
		_xdr_argument = (xdrproc_t) xdr_chatUsrId_t;
		_xdr_result = (xdrproc_t) xdr_chatRecibirNombreRes;
		local = (char *(*)(char *, struct svc_req *)) chat_recibirnombrefich_1_svc;
		break;

	case CHAT_PUT:
		_xdr_argument = (xdrproc_t) xdr_pares_t;
		_xdr_result = (xdrproc_t) xdr_chatCodigo_t;
		local = (char *(*)(char *, struct svc_req *)) chat_put_1_svc;
		break;

	case CHAT_GET:
		_xdr_argument = (xdrproc_t) xdr_chatUsrId_t;
		_xdr_result = (xdrproc_t) xdr_chatparleerRes;
		local = (char *(*)(char *, struct svc_req *)) chat_get_1_svc;
		break;

	case LOCALIZAR:
		_xdr_argument = (xdrproc_t) xdr_char;
		_xdr_result = (xdrproc_t) xdr_int;
		local = (char *(*)(char *, struct svc_req *)) localizar_1_svc;
		break;

		default:		/*Código de operación inválida.*/
		svcerr_noproc (transp);
		return;
	}
```

```
	/*Inicializa a cero la variable que recibe los parámetros de entrada.*/
	memset ((char *)&argument, 0, sizeof (argument));

	/*Se pasan los parámetros de entrada a formato nativo, depositándolos en la variable argument.*/
	if (!svc_getargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
		svcerr_decode (transp);
		return;
	}

	/*Ejecuta la rutina de servicio.*/
	result = (*local)((char *)&argument, rqstp);

	/*El resultado se le devuelve al cliente, también serealiza el resultado, en transp, mediante la rutina de sereaización _xdr_result que se le indica como parámetro. */
	if (result != NULL && !svc_sendreply(transp, (xdrproc_t) _xdr_result, result)) {
		svcerr_systemerr (transp);
	}

	/*Una macro que libera cualquier dato reservado por el sistema RPC/XDR cuando decodifica los argumentos a un procedimiento de servicio usando svc_getargs(). Esta rutina devuelve 1 si los resultados se han liberado con éxito y cero en caso contrario. */
	if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
		fprintf (stderr, "%s", "unable to free arguments");
		exit (1);
	}
	return; /* Retorno a svc_run().*/
}

int main (int argc, char **argv) /*Aquí se realiza el registro del servidor en el binder o portmap*/
{
	register SVCXPRT *transp;

	/*Destruye en el servidor portmap todas las correspondencias entre la terna [CHAT_PROGRAM,CHAT_VERSION,*] y los puertos del servidor en el portmap de la máquina.*/
	pmap_unset (CHAT_PROGRAM, CHAT_VERSION);

	/*Crea el socket UDP con el que se van a tender las peticiones del cliente. Devuelve un descriptor de Socket. Al terminar, la variable definida por RPC xprt->xp_sock es el descriptor del conector del medio de transporte y la variable definida por RPC xprt->xp_port es el numero de puerto del medio de transporte.*/
	transp = svcudp_create(RPC_ANYSOCK);
	if (transp == NULL) {
		fprintf (stderr, "%s", "cannot create udp service.");
		exit(1);
	}

	/*Asocia CHAT_PROGRAM y CHAT_VERSION con el procedimiento de atención de servicios chat_program_1, se establece una correspondencia entre la terna [CHAT_PROGRAM, CHAT_VERSION, IPPROTO_UDP] y la variable xprt->xp_port con el servicio portmap local (generalmente el cuarto parámetro de esta función es IPPROTO_UDP o IPPROTO_TCP). La rutina svc_register() devuelve uno en caso de éxito y cero en caso contrario. */
	if (!svc_register(transp, CHAT_PROGRAM, CHAT_VERSION, chat_program_1,
	IPPROTO_UDP)) {
		fprintf (stderr, "%s", "unable to register (CHAT_PROGRAM, CHAT_VERSION, udp).");
		exit(1);
	}
```

```
        /*Esta rutina crea un medio de transporte de servicio RPC basado en TCP/IP devolviendo un
        puntero al mismo. El medio de transporte se asociada con el conector RPC_ANYSOCK, en cuyo
        caso se crea un nuevo conector. Esta rutina devuelve NULL si falla. Al terminar, xprt->xp_sock es el
        descriptor del conector del medio de transporte y xprt->xp_port es el número de puerto del medio de
        transporte.*/
        transp = svctcp_create(RPC_ANYSOCK, 0, 0);
        if (transp == NULL) {
                fprintf (stderr, "%s", "cannot create tcp service.");
                exit(1);
        }

        /* Asocia CHAT_PROGRAM y CHAT_VERSION con el procedimiento de atención de servicios
        chat_program_1, se establece una correspondencia entre la terna [CHAT_PROGRAM,
        CHAT_VERSION, IPPROTO_TCP] y xprt->xp_port con el servicio portmap local (generalmente
        el cuarto parámetro de esta funciónes: IPPROTO_UDP o IPPROTO_TCP). La rutina
        svc_register() devuelve uno en caso de éxito y cero en caso contrario. */
        if (!svc_register(transp, CHAT_PROGRAM, CHAT_VERSION, chat_program_1,
        IPPROTO_TCP)) {
                fprintf (stderr, "%s", "unable to register (CHAT_PROGRAM, CHAT_VERSION, tcp).");
                exit(1);
        }

        /* Esta rutina nunca regresa. Espera la llegada de peticiones RPC por parte del Cliente, en cada
        petición recibida le pasa el control al dispatcher. */
        svc_run ();
        fprintf (stderr, "%s", "svc_run returned");
        exit (1);
}
```

Fichero chat_sif.c

```
/*chat_sif.c
Enlazará nuestra implementación con el stub del Servidor y con la ejecución del los servicios. Aquí se
implementan las rutinas que son invocadas desde el stub, éstas deberán llamar a las rutinas que realizarán
la petición hecha por el Cliente.*/

#include "chat.h"
#include "buzon.h"

static chatRecibirNombreRes nombreres;          /*Valor devuelto por chat_recibirnombrefich_1_svc al stub
                                                del cliente.*/
static chatparleerRes leer_res;                 /*Valor devuelto por chat_get_1_svc al stub del cliente.*/
static chatCodigo_t result;                     /*Valor devuelto al stub del cliente.*/
static chatConectRes result_conn;               /*Valor devuelto por chat_conectar_1_svc() al stub cliente.*/
static chatLeerRes result_leer;                 /*Valor devuelto por chat_leer_1_svc al stub cliente.*/
static chatConectadosRes result_conectados;             /*Valor devuelto por chat_conectados_1_svc al
                                                stub del cliente.*/
buzonCodigo_t status;                           /*Valor devuelto al stub cleinte.*/
int res;                                        //Valor de retorno de la función localizar_1_svc().

chatConectRes * chat_conectar_1_svc(chatNombreUsr_t *usuario, struct svc_req *rqstp)
{
        buzonUsrId_t buzonUsrId;
        char usr[CHATMAXNOMBRE];
        memset(usr,'\0',CHATMAXNOMBRE);
        strcpy(usr, *usuario);
        status = buzonConectar(usr, &buzonUsrId);       /*Llamada al procedimiento en buzon.c.*/
        switch(status)
```

```
	{
		case buzonOK:		/*Operación realizada correctamente.*/
			result_conn.status = chatOK;
			/*Se copia el valor de Id de usuario generado por el servidor (buzonUsrId) a la
			variable result_conn.chatConectRes_u.UsrId, la cual se le retornará al cliente.*/
			result_conn.chatConectRes_u.UsrId = buzonUsrId;
			break;

		case buzonDemasiadosUsr:	/*Se ha alcanzado el máximo número de usuarios
						conectados simultáneamente.*/
			result_conn.status = chatDemasiadosUsr;
			break;

		case buzonUsrYaDadoAlta:	/*Nombre de usuario duplicado.*/
			result_conn.status = chatUsrYaDadoAlta;
			break;

		default:		/*La operación no se realizó correctamente.*/
			result_conn.status = chatNOK;
			break;
	}
	return (&result_conn);
}

extern chatCodigo_t * chat_desconectar_1_svc(chatUsrId_t *chatUsrId, struct svc_req *rqstp)
{
	status= buzonDesconectar(*chatUsrId);		/*Llamada al procedimiento en buzon.c.*/
	switch(status)
	{
		case buzonOK:		/*Operación realizada con éxito.*/
			result=chatOK;
			break;

		case buzonUsrNoDadoAlta :	/* El usuario especificado no existe.*/
			result= chatUsrNoDadoAlta;
			break;

		default:		/*Operación no realizada correctamente.*/
			result = chatNOK;
			break;
	}
	return (&result);
}

extern chatCodigo_t * chat_escribir_1_svc(argesc_t *arg, struct svc_req *rqstp)
{
	status= buzonEscribir(arg->UsrId, arg->chatMsj,arg->tipo); /*Llamada al procedimiento en
	buzon.c, donde se especifica el Id de usr, el mensaje a enviar y el tipo de mensaje.*/
	switch(status)
	{
		case buzonOK:		/*Operación realizada con éxito.*/
			result=chatOK;
			break;

		case buzonUsrNoDadoAlta :	/* El usuario especificado no existe.*/
			result= chatUsrNoDadoAlta;
			break;
```

```
		default:		/*Operación no realizada correctamente.*/
				result = chatNOK;
				break;
	}
	return (&result);
}

extern chatLeerRes * chat_leer_1_svc(chatUsrId_t *chatUsrId, struct svc_req *rqstp)
{
	result_leer.chatLeerRes_u.chatContestacion.chatNombreUsr=(chatNombreUsr_t)malloc(sizeof
	(char)*CHATMAXNOMBRE);
	result_leer.chatLeerRes_u.chatContestacion.chatMsj=chatMsj_t)malloc(sizeof(char)*
	CHATMAXLONG);
	result_leer.chatLeerRes_u.chatContestacion.tipo = (type_msj)malloc(sizeof(char)*TIPOMSJ);

	/*Llamada al procedimiento en buzon.c.*/
	status = buzonLeer (*chatUsrId, result_leer.chatLeerRes_u.chatContestacion.chatNombreUsr,
	result_leer.chatLeerRes_u.chatContestacion.chatMsj,
	result_leer.chatLeerRes_u.chatContestacion.tipo);
	switch(status)
	{
		case buzonOK:		/*Operación realizada con éxito.*/
				result_leer.status=chatOK;
				break;

		case buzonUsrNoDadoAlta :	/*El usuario especificado no existe.*/
				result_leer.status= chatUsrNoDadoAlta;
				break;

		default:		/*Operación no realizada correctamente.*/
				result_leer.status = chatNOK;
				break;
	}

	return (&result_leer);
}

extern chatConectadosRes * chat_conectados_1_svc(chatUsrId_t *chatUsrId, struct svc_req *rqstp) {
	result_conectados.chatConectadosRes_u.argObtenerConectados.Usuario_Conectado=
	(chatNombreUsr_t)malloc(sizeof(char)*CHATMAXNOMBRE);
	/*Llamada al procedimiento en buzon.c.*/
	status=buzonConectados(*chatUsrId,
	result_conectados.chatConectadosRes_u.argObtenerConectados.Usuario_Conectado);
	switch(status)	{
		case buzonOK:		/*Operación realizada con éxito.*/
				result_conectados.status=chatOK;
				break;

		 case buzonUsrNoDadoAlta :		/*El usuario especificado no existe.*/
				result_conectados.status= chatUsrNoDadoAlta;
				break;

		default:		/*Operación no realizada correctamente.*/
				result_conectados.status = chatNOK;
				break;
	}
	return (&result_conectados);
}
```

```
extern  chatCodigo_t * chat_creartuberias_1_svc(chatUsrId_t  *chatUsrId, struct svc_req *rqstp)
{
        /*Llamada al procedimiento en buzon.c.*/
        status = buzonCrearTuberias(*chatUsrId);
        switch(status)
        {
                case buzonOK:           /*Operación realizada con éxito.*/
                        result = chatOK;
                        break;

                case buzonTuberiaActiva:        /*Operación no realizada, debido a que la tubería ya
                                                está creada, por una llamada anterior.*/
                        result = chatTuberiaActiva;
                        break;

                default:            /*Operación no realizada correctamente.*/
                        result = chatNOK;
                        break;
        }
        return (&result);
}

extern  chatCodigo_t * chat_destruirtuberias_1_svc(chatUsrId_t * chatUsrId, struct svc_req * rqstp)
{
        /*Llamada al procedimiento en buzon.c.*/
        status=buzonDestruirTuberias(*chatUsrId);
        switch(status)
        {
                case buzonOK:           /*Operación realizada con éxito.*/
                        result=chatOK;
                        break;

                default:            /*Operación no realizada correctamente.*/
                        result = chatNOK;
                        break;
        }
        return  (&result);
}

extern  chatCodigo_t * chat_enviarnombrefich_1_svc(parabr_t *parabr , struct svc_req *rqstp)
{
        /*Llamada al procedimiento en buzon.c.*/
        status=buzonEnviarNombreFich(parabr->chatNombreUsr, parabr->chatfichero);
        switch(status)
        {
                case buzonOK:           /*Operación realizada con éxito.*/
                        result= chatOK;
                        break;

                case buzonTuberiaLlena:         /*Operación no realizada, por la existencia de datos en la
                                                tubería.*/
                        result=chatTuberiaLlena;
                        break;

                case buzonTuberiaNoActvia:      /*Operación no realizada, porque la tubería no ha sido
                                                creada.*/
                        result=chatTuberiaNoActiva;
                                break;
```

```
		case buzonTuberiaOcupada:	/*Operación no realizada, porque la tubería está siendo
						utilizada por otro usuario.*/
			result=chatTuberiaOcupada;
			break;

		default:		/*Operación no realizada correctamente.*/
			result = chatNOK;
			break;
	}
	return (&result);
}

extern	chatRecibirNombreRes * chat_recibirnombrefich_1_svc(chatUsrId_t * chatUsrId, struct svc_req *
rqstp)
{
	nombreres.chatRecibirNombreRes_u.chatContestacionparabr.fichero=(chatNombreFich_t)malloc
	(sizeof(char)*CHATMAXNOMBREFICHERO);

	/*Llamada al procedimiento en buzon.c.*/
	status=buzonRecibirNombreFich(*chatUsrId,
	nombreres.chatRecibirNombreRes_u.chatContestacionparabr.fichero);
	switch(status)
	{
		case buzonOK:		/*Operación realizada con éxito.*/
			nombreres.status=chatOK;
			break;

		case buzonTuberiaVacia:	/*Operación no relaizada porque en la tubería no hay datos.*/
				nombreres.status=chatTuberiaVacia;
				break;

		default:		/*Operación no realizada correctamente.*/
				nombreres.status = chatNOK;
				break;
	}
	return (&nombreres);
}

extern	chatCodigo_t * chat_put_1_svc(pares_t * pares, struct svc_req * rqstp) {
	/*Llamada al procedimiento en buzon.c.*/
	status=buzonPut(pares->chatNombreUsr,pares->chatContenido.chatdatos_t_val,
	pares->chatContenido.chatdatos_t_len);
	switch(status)	{
		case buzonOK:		/*Operación realizada con éxito.*/
			result=chatOK;
			break;

		case buzonTuberiaLlena:	/*Operación no relaizada por la existenccia de datos en
						la tubería.*/
			result=chatTuberiaLlena;
			break;
		default:		/*Operación no realizada correctamente.*/
			result = chatNOK;
			break;
	}
	return (&result);
}
```

```
extern  chatparleerRes * chat_get_1_svc(chatUsrId_t *chatUsrId, struct svc_req *rqstp)
{
        leer_res.chatparleerRes_u.parleer.chatContenido.chatdatos_t_val=(char*)malloc(sizeof(char)*
        CHATMAXDATOSFICHERO);

        /*Llamada al procedimiento en buzon.c.*/
        status=buzonGet(*chatUsrId,        leer_res.chatparleerRes_u.parleer.chatContenido.chatdatos_t_val,
        &leer_res.chatparleerRes_u.parleer.chatContenido.chatdatos_t_len);
        switch(status)
        {
                case buzonOK:           /*Operación realizada con éxito.*/
                        leer_res.status=chatOK;
                        break;

                case buzonTuberiaVacia: /*Operación no realizada porque no hay datos en la tubería.*/
                        leer_res.status=chatTuberiaVacia;
                        break;

                case buzonFinFichero:           /*Especificación del fin de los datos del fichero.*/
                        leer_res.status=chatFinFichero;
                        break;

                default:            /*Operación no realizada correctamente.*/
                        leer_res.status= chatNOK;
                        break;
        }
        return(&leer_res);
}

extern  int * localizar_1_svc(char *arg,struct svc_req *rqstp)
{
        status=localizar(arg);    //Llamada a la rutina localizar.
        res=1;
        return (&res);
}
```

Fichero buzon.c

```
/*buzon.c
Módulo del programa que esta encargado de manejar el buzón de cada usuario. Maneja la información de
cada uno de los usuarios conectados, agrega y elimina usuarios a la Sala de Chat, lee y escribe información
en los buzones de los usuarios y permite gestionar las tuberías para el intercambio de fichero de cada cliente
conectado a la sala. */

#include "buzon.h"  /*Fichero que declara las funciones prototipos que se implementan aquí.*/

static descriptor_t descriptores[CHAT_MAX_USR];       /*Array para cada uno de los usuarios que se
                                                      podrán conectar.*/

/*¿Hay un descriptor vacío?*/
int hayEntradaLibre(int *slot)
{
        int i;
        for(i=0;(descriptores[i].activo && (i<CHAT_MAX_USR));i++ )
        {}
        *slot=i;
        return (i<CHAT_MAX_USR);
}
```

```
/*Hay un usuario con este nombre.*/
int DadoAlta(char *usuario)
{
	int encontrado = FALSE;
	int i;
	for(i=0; !encontrado && (i<CHAT_MAX_USR);i++)
		encontrado=(strcmp(descriptores[i].usuario,usuario)==0);
	return encontrado;
}

/*Es un Id de usuario valido.*/
int buzonUsrIdValido(buzonUsrId_t buzonUsrId)
{
	if((buzonUsrId<0) || (buzonUsrId > CHAT_MAX_USR))
		return FALSE;
	return (descriptores[buzonUsrId].activo);
}

/*Inicializar el nombre de usuario.*/
void inicializarUsr(int i)
{
	memset(descriptores[i].usuario,'\0',CHAT_MAX_NOMBRE);
}

buzonCodigo_t buzonConectar(char *usuario,buzonUsrId_t *usrId)
{
	int i;
	if(!hayEntradaLibre(&i))
		return (buzonDemasiadosUsr);
	if(usuario==NULL)
		return (buzonNombreUsrNOK);
	if(DadoAlta(usuario))
		return (buzonUsrYaDadoAlta);

	/*Crea un par de descriptores de ficheros, que apuntan a un inodo de una tubería, y los pone en
	el vector de dos elementos apuntado por descriptores[i].fds, descriptores[i].fds[0] es para lectura,
	descriptores[i].fds[1] es para escritura.*/
	if(pipe(descriptores[i].fds))
		return (buzonNOK);

	/*El establecimiento del flag O_NDELAY, hace que la lectura del pipe sea no bloqueante.*/
	if(fcntl(descriptores[i].fds[0],F_SETFL,O_NDELAY)==-1)
		return (buzonNOK);

	/*Hay un hueco libre, no hay nadie con ese nombre y se ha creado el pipe.*/
	descriptores[i].activo=TRUE;
	inicializarUsr(i);
	strcpy(descriptores[i].usuario,usuario);
	*usrId=i;

	/*Notifica a todos los usuarios conectados, que existe un nuevo usuario dado de alta.*/
	buzonEscribir(i,MSJ1,TIPO1);
	return (buzonOK);
}
```

```
buzonCodigo_t buzonDesconectar(buzonUsrId_t buzonUsrId)
{
	/*Notifica a todos los usuarios conectados, que un usuario a abandonado la Sala de Chat.*/
	buzonEscribir(buzonUsrId,MSJ2,TIPO2);
	if(!buzonUsrIdValido(buzonUsrId))
		return (buzonUsrNoDadoAlta);
	descriptores[buzonUsrId].activo=FALSE;
	inicializarUsr(buzonUsrId);
	close(descriptores[buzonUsrId].fds[0]);		/*Cerrar el descriptor de lectura.*/
	close(descriptores[buzonUsrId].fds[1]);		/*Cerrar el descriptor de escritura.*/
	return (buzonOK);
}

buzonCodigo_t buzonEscribir(buzonUsrId_t buzonUsrId,char *msj,char *tipo)
{
	int i, op;
	if(!buzonUsrIdValido(buzonUsrId))		/*El usuario no existe.*/
		return (buzonUsrNoDadoAlta);
	if(msj==NULL)  /*Mensaje vacío.*/
		return (buzonMsjNull);
	if((strcmp(TIPO4, tipo)!=0) && (strcmp(TIPO5, tipo)!=0) && (strcmp(TIPO6, tipo)!=0) )
	{
		/*Mensajes dirigidos a todos los usuarios de la Sala de Chat.*/
		for(i=0;i<CHAT_MAX_USR;i++)
		{
			/*Trabajar en exclusión mutua, para entrar a la Región Crítica.*/
			pthread_mutex_lock(&descriptores[i].RC);
			if(descriptores[i].activo) /*Si el usuario está dado de alta, escribir en la tubería.*/
			{
				if(write(descriptores[i].fds[1],descriptores[buzonUsrId].usuario,
				strlen(descriptores[buzonUsrId].usuario)+1) == -1)	/*Error al escribir
				el nombre de usuario.*/
				{
					/*Salir de la Región Crítica.*/
					pthread_mutex_unlock(&descriptores[i].RC);
					return (buzonNOK);
				}
				if(write(descriptores[i].fds[1],msj,(strlen(msj)+1)) == -1)  /*Error al escribir
				el mensaje.*/
				{
					/*Salir de la Región Crítica.*/
					pthread_mutex_unlock(&descriptores[i].RC);
					return (buzonNOK);
				}
				if(write(descriptores[i].fds[1],tipo,(strlen(tipo)+1)) == -1)  /*Error al escribir
				el tipo del mensaje.*/
				{
					/*Salir de la Región Crítica.*/
					pthread_mutex_unlock(&descriptores[i].RC);
					return (buzonNOK);
				}
			} /*Fin if activo.*/
			pthread_mutex_unlock(&descriptores[i].RC);	/*Salir de la Región Crítica.*/
		}/*Fin del for.*/
	}
```

```
	else
	{
		/*Mensajes dirigidos a un usuario específico de la Sala de Chat.*/
		for(i=0;i<CHAT_MAX_USR;i++)
		{
			if(descriptores[i].activo && (strcmp(descriptores[i].usuario,msj)==0))
			{
				/*Obtener el Id del usuario, al que se le escribirá el mensaje.*/
				op=i;
				break;
			}
		}
		/*Trabajar en exclusión mutua, para entrar a la Región Crítica.*/
		pthread_mutex_lock(&descriptores[op].RC);
		if(write(descriptores[op].fds[1],descriptores[buzonUsrId].usuario,
		strlen(descriptores[buzonUsrId].usuario)+1) == -1)
		{
			pthread_mutex_unlock(&descriptores[op].RC);
			return (buzonNOK);
		}
		if(write(descriptores[op].fds[1],msj,(strlen(msj)+1)) == -1)
		{
			pthread_mutex_unlock(&descriptores[op].RC);
			return (buzonNOK);
		}
		if(write(descriptores[op].fds[1],tipo,(strlen(tipo)+1)) == -1)
		{
			pthread_mutex_unlock(&descriptores[op].RC);
			return (buzonNOK);
		}
		pthread_mutex_unlock(&descriptores[op].RC);		/*Salir de la Región Crítica.*/
	}
	return buzonOK;
}

buzonCodigo_t buzonLeer(buzonUsrId_t buzonUsrId,char *usuario,char *msj,char *tipo)
{
	int i=0,hayCar=TRUE;
	if(!buzonUsrIdValido(buzonUsrId))	/*Usuario inválido.*/
		return (buzonUsrNoDadoAlta);
	memset(usuario,'\0',CHAT_MAX_NOMBRE);
	memset(msj,'\0',CHAT_MAX_LONG);
	memset(tipo,'\0',CHAT_MAX_TIPO);
	/*Trabajar en exclusión mutua, para entrar a la Región Crítica.*/
	pthread_mutex_lock(&descriptores[buzonUsrId].RC);
	while(hayCar)	/*Leer el nombre del autor del mensaje caracter a caracter.*/
	{
		if(read(descriptores[buzonUsrId].fds[0],&(usuario[i]),1)== -1) /*Error al leer el nombre de
		usuario.*/
		{
			pthread_mutex_unlock(&descriptores[buzonUsrId].RC);
			return (buzonNOK);
		}
		hayCar=((usuario[i]) != '\0');	/*¿Es el fin del nombre?*/
		i++;
	} /*while del nombre.*/
	i=0;
	hayCar= TRUE;
```

```
	while(hayCar)   /*Leer el mensaje caracter a caracter.*/
	{
		if(read(descriptores[buzonUsrId].fds[0],&(msj[i]),1)== -1)
		{
			pthread_mutex_unlock(&descriptores[buzonUsrId].RC);
			return (buzonNOK);
		}
		hayCar=((msj[i]) != '\0');		/*¿Es el fin del mensaje?*/
		i++;
	} /*while del msj.*/
	i=0;
	hayCar= TRUE;
	while(hayCar)   /*Leer el tipo del mensaje caracter a caracter.*/
	{
		if(read(descriptores[buzonUsrId].fds[0],&(tipo[i]),1)== -1) {
			pthread_mutex_unlock(&descriptores[buzonUsrId].RC);
			return (buzonNOK);
		}
		hayCar=((tipo[i]) != '\0');		/*¿Es fin del tipo del mensaje?*/
		i++;
	} /*while del tipo de mensaje.*/
	pthread_mutex_unlock(&descriptores[buzonUsrId].RC);		/*Salir de la Región Crítica.*/
	return buzonOK;
}

buzonCodigo_t buzonConectados(buzonUsrId_t buzonUsrId,char *usuario_conectado)
{
	if(descriptores[buzonUsrId].activo){	/*El usuario está activo.*/
		strcpy(usuario_conectado,descriptores[buzonUsrId].usuario); /*Copiar el nombre del usr.*/
		return buzonOK;
	}
	else
		return buzonUsrNoDadoAlta;
}

buzonCodigo_t buzonCrearTuberias(buzonUsrId_t buzonUsrId)
{
	if(!(descriptores[buzonUsrId].Tuberia_Activa))	/*No se ha creado la tubería.*/
	{
		descriptores[buzonUsrId].Tuberia_Activa = TRUE;
			/*Crea un par de descriptores de ficheros, que apuntan a un inodo de una tubería, y
			los pone en el vector de dos elementos apuntado por descriptores[buzonUsrId].fds2,
			descriptores[buzonUsrId].fds2[0] es para lectura, descriptores[buzonUsrId].fds2[1] es
			para escritura.*/
		if(pipe(descriptores[buzonUsrId].fds2)==-1)
		{
			descriptores[buzonUsrId].Tuberia_Activa = FALSE;
			return (buzonNOK);
		}
		/*Establecer el flag O_NDELAY, para que la lectura del pipe sea no bloqueante.*/
		if(fcntl(descriptores[buzonUsrId].fds2[0],F_SETFL,O_NDELAY)==-1)	{
			descriptores[buzonUsrId].Tuberia_Activa = FALSE;
			return (buzonNOK);
		}
		return (buzonOK);
	}
	return buzonTuberiaActiva;
}
```

```
buzonCodigo_t buzonDestruirTuberias(buzonUsrId_t buzonUsrId)
{
	if(close(descriptores[buzonUsrId].fds2[0])==-1)		/*Cerrar el descriptor de lectura.*/
		return (buzonNOK);
	if(close(descriptores[buzonUsrId].fds2[1])==-1)		/*Cerrar el descriptor de escritura.*/
		return (buzonNOK);
	descriptores[buzonUsrId].Tuberia_Activa = FALSE;
	descriptores[buzonUsrId].TubLlena=FALSE;
	descriptores[buzonUsrId].Tuberia_Ocupada=FALSE;
	return (buzonOK);
}

buzonCodigo_t buzonEnviarNombreFich(char *chatNombreUsr, char *NombreFich)
{
	int i,op=0;
	for(i=0;i<CHAT_MAX_USR;i++)
	{
		if(strcmp(descriptores[i].usuario,chatNombreUsr)==0)
		{
			/*Obtener el Id del usuario, al que se le escribirá el nombre del fichero*/
			op=i;
			break;
		}
	}
	if(i==CHAT_MAX_USR)
		return buzonTuberiaNoActvia;
	pthread_mutex_lock(&descriptores[op].RC);
	if(!descriptores[op].Tuberia_Ocupada)		/*La tubería está ocupada por otro usuario.*/
	{
		if(descriptores[op].Tuberia_Activa)
		{
			if(!(descriptores[op].TubLlena))		/*No hay datos en la tubería.*/
			{
				if(write(descriptores[op].fds2[1],NombreFich,(strlen(NombreFich)+1))== -1)
				{
					pthread_mutex_unlock(&descriptores[op].RC);
					return (buzonNOK);
				}
				descriptores[op].TubLlena=TRUE;
				descriptores[op].Tuberia_Ocupada=TRUE;
				pthread_mutex_unlock(&descriptores[op].RC);
				return buzonOK;
			}
			pthread_mutex_unlock(&descriptores[op].RC);
			return buzonTuberiaLlena;
		}
		pthread_mutex_unlock(&descriptores[op].RC);
		return buzonTuberiaNoActvia;
	}
	pthread_mutex_unlock(&descriptores[op].RC);		/*Salir de la Región Crítica.*/
	return buzonTuberiaOcupada;
}

buzonCodigo_t buzonRecibirNombreFich(buzonUsrId_t buzonUsrId, char *NombreFich)
{
	int i=0,hayCar=TRUE;
	pthread_mutex_lock(&descriptores[buzonUsrId].RC);
```

```
	if(descriptores[buzonUsrId].TubLlena)		/*Hay datos en la tubería.*/
	{
		while(hayCar)		/*Leer el nombre del fichero caracter a caracter.*/
		{
			if(read(descriptores[buzonUsrId].fds2[0],&(NombreFich[i]),1)== -1)
			{
				pthread_mutex_unlock(&descriptores[buzonUsrId].RC);
				return (buzonNOK);
			}
			hayCar=((NombreFich[i]) != '\0');		/*Es fin del nombre del fichero.*/
			i++;
		}
		descriptores[buzonUsrId].TubLlena=FALSE;
		pthread_mutex_unlock(&descriptores[buzonUsrId].RC);
		return buzonOK;
	}
	pthread_mutex_unlock(&descriptores[buzonUsrId].RC);		/*Salir de la Región Crítica.*/
	return buzonTuberiaVacia;
}

buzonCodigo_t buzonPut(char *chatNombreUsr, char *contenido, buzonCantidad_t cantidad)
{
	int i,op=0;
	for(i=0;i<CHAT_MAX_USR;i++)
	{
		if(strcmp(descriptores[i].usuario,chatNombreUsr)==0)
		{
			/*Obtener el Id del usuario, al que se le escribirán los datos del fichero.*/
			op=i;
			break;
		}
	}
	if(i==CHAT_MAX_USR)
		return buzonTuberiaNoActvia;
	pthread_mutex_lock(&descriptores[op].RC);
	if(!(descriptores[op].TubLlena))		/*No hay datos en la tubería.*/
	{
		if(cantidad==0)		/*La Cantidad de bytes es cero.*/
		{
			descriptores[op].FinFichero=TRUE;	/*Activar bandera de fin de fichero.*/
			descriptores[op].TubLlena=TRUE;
			pthread_mutex_unlock(&descriptores[op].RC);
			return buzonOK;
		}
		if(write(descriptores[op].fds2[1],contenido,cantidad) == -1)	/*Escribir en la tubería.*/
		{
			pthread_mutex_unlock(&descriptores[op].RC);
			return (buzonNOK);
		}
		descriptores[op].TubLlena=TRUE;
		pthread_mutex_unlock(&descriptores[op].RC);
		return buzonOK;
	}
	pthread_mutex_unlock(&descriptores[op].RC);		/*Salir de la Región Crítica.*/
	return buzonTuberiaLlena;
}
```

```
buzonCodigo_t buzonGet(buzonUsrId_t buzonUsrId, char *contenido, buzonCantidad_t *cantidad)
{
        int i;
        pthread_mutex_lock(&descriptores[buzonUsrId].RC);
        if(descriptores[buzonUsrId].TubLlena)   /*Hay datos en la tubería.*/
        {
                if(!(descriptores[buzonUsrId].FinFichero)) /*La bandera de fin de fichero no se ha activado.*/
                {
                        /*Leer de la tubería y retornar el número de bytes realmente leídos.*/
                        i=read(descriptores[buzonUsrId].fds2[0],
                        contenido,CHAT_MAXDATOS_FICHERO);
                        *cantidad=i;
                        descriptores[buzonUsrId].TubLlena=FALSE;
                        pthread_mutex_unlock(&descriptores[buzonUsrId].RC);
                        return buzonOK;
                }
                else    /*Bandera de fin de fichero está activada.*/
                {
                        *cantidad=0;
                        descriptores[buzonUsrId].FinFichero=FALSE;
                        pthread_mutex_unlock(&descriptores[buzonUsrId].RC);
                        return buzonFinFichero;         /*Valor de retorno que indica el fin de fichero.*/
                }
        }
        pthread_mutex_unlock(&descriptores[buzonUsrId].RC);         /*Salir de la Región Crítica.*/
        return buzonTuberiaVacia;
}

buzonCodigo_t localizar(char *arg)
{
        return buzonOK;         //Operación realizada con éxito.
}
```

Para obtener el fichero ejecutable del programa Servidor,

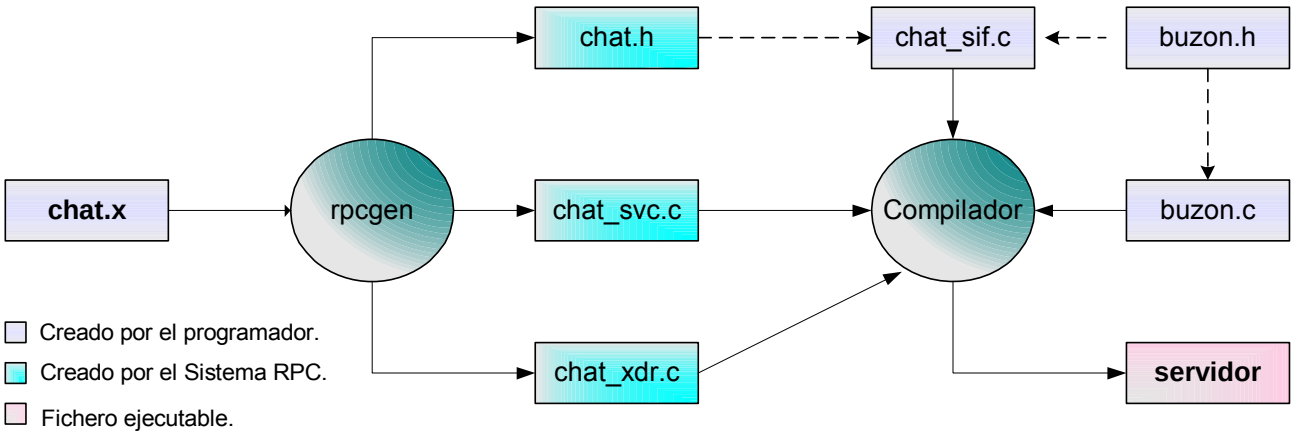


Fig. 31 Generar fichero ejecutable del Servidor.

XI. CONCLUSIÓN.

Al finalizar con este trabajo monográfico, consideramos que hemos logrado cumplir con los objetivos propuestos, al proporcionar al lector un documento que le brinde nociones básicas sobre los Sistemas Distribuidos y que sirva como base para el desarrollo de futuros trabajos bajo el entorno de los Sistemas Distribuidos.

Al llevar a cabo la fase de diseño y haber utilizado Qt Disegner, en conjunto con KDevelop, concluimos que es una herramienta muy potente que permite crear de forma sencilla y rápida interfaces de usuario, empleando las librerías Qt.

El empleo de las RPC, constituye uno de los principales mecanismos de comunicación en Sistemas Operativos Distribuidos, ya que proporciona una de las principales características de estos sistemas: la transparencia en el uso de los recursos software y hardware proporcionados en toda la red LAN.

Gracias al empleo de las herramientas antes mencionadas (RPC's, Qt Disegner y KDevelop), se concluyó con el desarrollo de la aplicación Chat Multiusuario, propuesta como objetivo fundamental de este trabajo monográfico.

XII. RECOMENDACIÓN.

Debido a que el Servidor remoto, solamente es capaz de atender una a una, las peticiones recibidas, se recomienda aplicar una política de servicio concurrente, que le permita al Servidor atender simultáneamente las peticiones de los diferentes Clientes. Por ejemplo, la creación de procesos hijos con la llamada a fork() en el repartidor, esto, permitirá que el proceso hijo pueda ejecutar el servicio solicitado y el repartidor pueda retornar a svc_run(), para atender las otras peticiones de los clientes.

En nuestro programa, un cliente, sólo puede recibir un fichero a la vez, por lo que se recomienda implementar un mecanismo de múltiple recepción de ficheros – por ejemplo, la implementación de una cola FIFO-, por parte del cliente que está recibiéndolos.

Hasta ahora no se está garantizando la integridad de los datos transferidos a través de la red, por lo que recomendamos implementar un protocolo de transferencia de datos, que garantiza la integridad de los mismos.

XIII. BIBLIOGRAFÍA.

- Tanenbaum, Andrew S. “Sistemas Operativos Distribuidos”. Prentice Hall., 1996.

- Molina, Eduardo Santiago. “Plan docente para la asignatura de Sistemas Operativos II”, 2002.

Referencias a Internet.

- Muñoz Estaban Juan Jesús. Curso de Sistema Operativo. Universidad Carlos III de Madrid Departamento de informática. 1998-1999.
  http://arcos.inf.uc3m.es

- Magister David Luis la Red Martínez. Comunicación en los Sistemas Distribuidos. Departamento de informática. Facultad de Ciencias Exactas y Naturales Y Agrimensura. Universidad Nacional del Nordeste-Argentina. 01/08/05.
  http://exa.unne.edu.ar/depar/areas/informatica/SistemasOperativos

- Meza Múgica Myriam, Baca Trapero Raúl Iván, Nieto Negrete Sergio Alfonso. Modelo de Construcción. Tutorial de Sistemas Distribuidos I. Instituto tecnológico de la Paz .México.
  http://www.itlp.edu.mx

- Macías José, Rico Mariano. Antonio Guía de prácticas para la asignatura de Sistemas Operativos II. Escuela politécnica superior. Madrid España. (2002/2003)
  http://www.ii.uam.es

- I.I. Gijón. Prácticas sobre SOD. Ingeniaría en Informática Área de Arquitectura y Tecnología de Computadores. Departamento de Informática de la Universidad de Oviedo Asturias-España. Curso 2004/2005
  http://www.atc.uniovi.es

- Página oficial de la API del Diseñador de interfaces (Diseñador Qt).
  http://doc.troltech.com

- Página oficial de IDE para desarrollo en KDE (KDevelop: KDE/C++).
  http://www.kdevelop.org

XIV. ANEXOS.

1. Instalación del Programa Cliente.

La instalación del programa Cliente, llamado **cliente_chat**, se deberá hacer como root, para esto se debe copiar al directorio root el paquete cliente_chat proporcionado. Luego desde la línea de orden se debe cambiar al directorio copiado previamente (# cd /root/cliente_chat) y finalmente ejecutar el scrip llamado **instalador** (# ./instalador), el cual contiene las siguientes instrucciones:

```
#Especificar la ruta de instalación del programa.
./configure --prefix=/usr/local/Cliente_Chat
#Iniciar la instalación del programa.
make install
#Informar que el programa fue instalado con éxito.
echo PROGRAMA INSTALADO
#Crear el directorio Recursos. Este contendrá el ejecutable encargado de localizar el servidor
#remoto.
mkdir /usr/local/Cliente_Chat/Recursos
#Modificar los permisos del directorio, de forma que el usuario puede crear y eliminiar, en tiempo de
#ejecución, el fichero donde se guarda la dirección IP del servidor.
chmod 777 /usr/local/Cliente_Chat/Recursos
#Acceder a la carpeta src.
cd src
#Compilar el programa encargado de localizar el servidor remoto.
cc -o localizador localizador.c
#Copiar el ejecutable localizador en el directorio Recursos.
cp  localizador /usr/local/Cliente_Chat/Recursos
#Cambiar permisos de ejecución al fichero localizador de la siguiente forma.
chmod 755 /usr/local/Cliente_Chat/Recursos/localizador
#Volver al directorio raiz.
cd /
```

2. Fichero localizador.c

```
/*localizador.c
Fichero fuente, encargado de localizar al Servidor Remoto. Este es invocado desde el fichero de interfaz
chat_cif.cpp.*/

#include <netdb.h>
#include "chat.h"
#include <stdlib.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <unistd.h>          //Definición de llamadas primitivas al sistema.
#include <sys/types.h>       //Definición de tipos de datos primitivos del sistema.
#include <sys/stat.h>        //Establecimiento de características de ficheros.
#include <fcntl.h>           //Control de operaciones sobre ficheros.

enum clnt_stat estado;       //Tipo de variable retornado por la función clnt_broadcast().
char *Direccion_Servidor;    /*Puntero a char, donde se almacenará la dirección IP del Servidor remoto.*/
```

```
/*Función invocada por clnt_broadcast, cada vez que éste recibe una respuesta.
dato -> Es el mismo parámetro pasado a clnt_broadcast.
dir_servidor-> Apunta a la dirección de la máquina que ha devuelto el resultado.*/
static bool_t respuesta(int *dato, struct sockaddr_in *dir_servidor)
{
	if(dir_servidor==NULL)
		return(FALSE);
	/*Guardar en la variable Direccion_Servidor la dirección de la máquina que ha devuelto el
	resultado.*/
	Direccion_Servidor=(char *)inet_ntoa(dir_servidor->sin_addr);
		return(TRUE);
}

main(int argc, char *argv[])
{
	int dato=0;	//Tipo de retorno del procedimiento LOCALIZAR.
	int fd;		//Descriptor de fichero.


	/*clnt_broadcast -> Se encarga de hacer una multidifusión a todos los servidores que provean el
	servicio LOCALIZAR en la red LAN.
	CHAT_PROGRAM -> Número del programa.
	CHAT_VERSION -> Versión del programa.
	LOCALIZAR -> Procedimiento remoto a localizar. (Ver fichero chat.x, chat.h, chat_svc.c)
	xdr_void -> Codifica los parámetros de entrada del procedimiento LOCALIZAR.
	NULL -> Dirección del argumento de entrada del procedimiento LOCALIZAR.
	xdr_int -> Codifica los parámetros de salida del procedimiento LOCALIZAR.
	dato -> Dirección donde se deposita el resultado del procedimiento LOCALIZAR.
	respuesta -> Procedimiento invocado, por cada respuesta del clnt_broadcast().*/

	estado= clnt_broadcast(CHAT_PROGRAM,CHAT_VERSION,LOCALIZAR,xdr_void,(char *) NULL,
	xdr_int, (char *) &dato,respuesta);

	//Si la operación no se ralizó correctamente y no se ha agotado el TIMEOUT.
	if ( (estado != RPC_SUCCESS) && (estado != RPC_TIMEDOUT))
	{
		clnt_perrno(estado);	/*Informar del tipo de error ocurrido.*/
		exit(1);		//Salir del programa y retornar a chat_cif.cpp.
	}

	if(estado == RPC_SUCCESS)		//Operación realizada satisfactoriamente.
	{
		/*Crear el fichero donde se almacenará temporalmente la dirección del Servidor remoto.*/
		fd=open("/usr/local/Cliente_Chat/Recursos/Direccion_Servidor",O_WRONLY|O_CREAT|O_T
		RUNC,00777);
		if(fd==-1)
			exit(0);	//Si hubo un error de apertura del fichero, salir del programa.

		else	//De lo contrario, escribir en el fichero la dirección IP del Servidor.
		{
			if( write(fd,Direccion_Servidor,strlen(Direccion_Servidor))!=-1)
				exit(0);
			else
				close(fd);
		}
	}
}
```

3. Desarrollo de aplicaciones con KDevelop y Qt Designer.

KDevelop es un entorno de programación hecho en Linux para crear aplicaciones que corran en **KDE**, lo que no quiere decir que no puedan correr en Gnome, pero Gnome usa las librerías GTK+ y KDE usa Qt, lo que hace que las aplicaciones de Qt en KDE anden mas rápido por no tener que cargar las librerías gráficas.

Qt Designer es una aplicación para crear la interfaz de usuario de nuestros programas usando las librerías Qt, asignarle los nombres y crear los eventos y las funciones que queremos que realice, para luego codificarlos usando un Lenguaje de **POO** (**P**rogramación **O**rientada a **O**bjetos) como es C++.

A continuación vamos a describir una serie de pasos para crear algunas aplicaciones sencillas con KDevelop.

3.1. Proyecto: HolaMundo

a. Iniciar el **IDE** (Entorno de Desarrollo Integrado) para desarrollo en KDE (KDevelop:KDE/C++).

b. Crear un nuevo proyecto. (Ver Fig. 32).

- Click en el menú Proyecto->Nuevo Proyecto

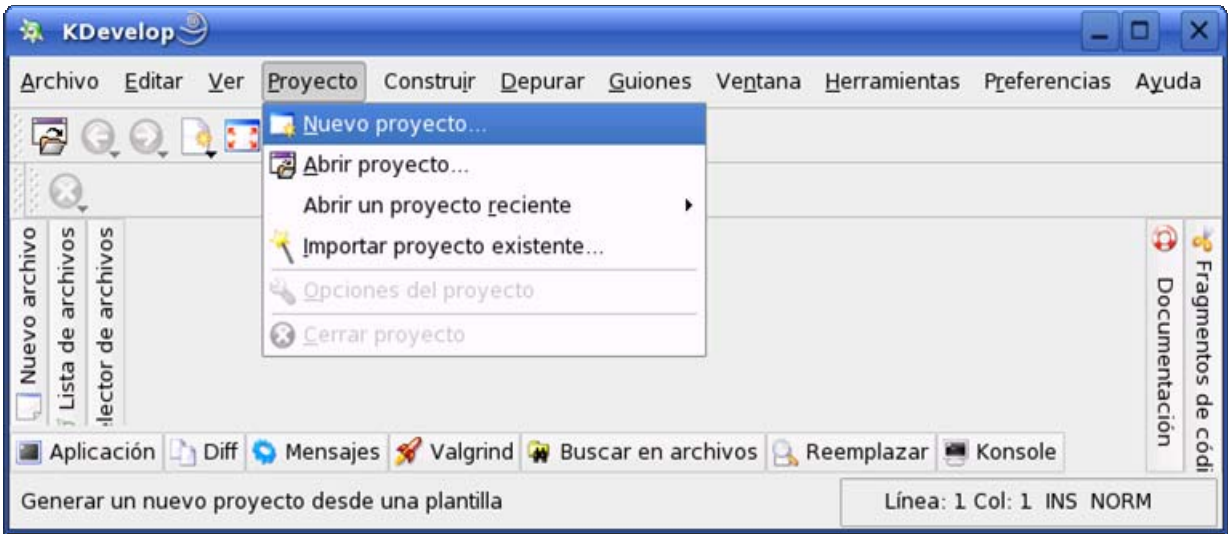


Fig. 32 Nuevo Proyecto.

- Seleccionar como tipo de proyecto: "Aplicación de KDE sencilla". (Ver Fig. 33).
- Nombre de aplicación ->HolaMundo.
- Escoger la ruta donde se creará el proyecto.

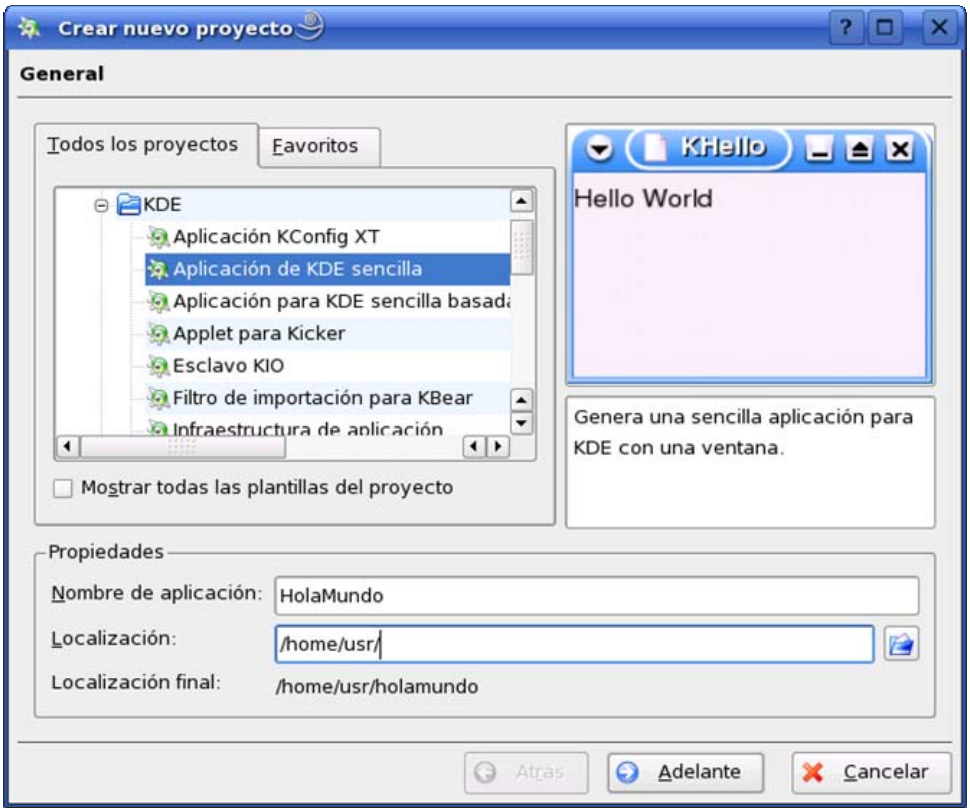


Fig. 33 Tipo de proyecto.

- Click en Adelante.
- Completar la información sobre el autor del proyecto. (Ver Fig. 34).

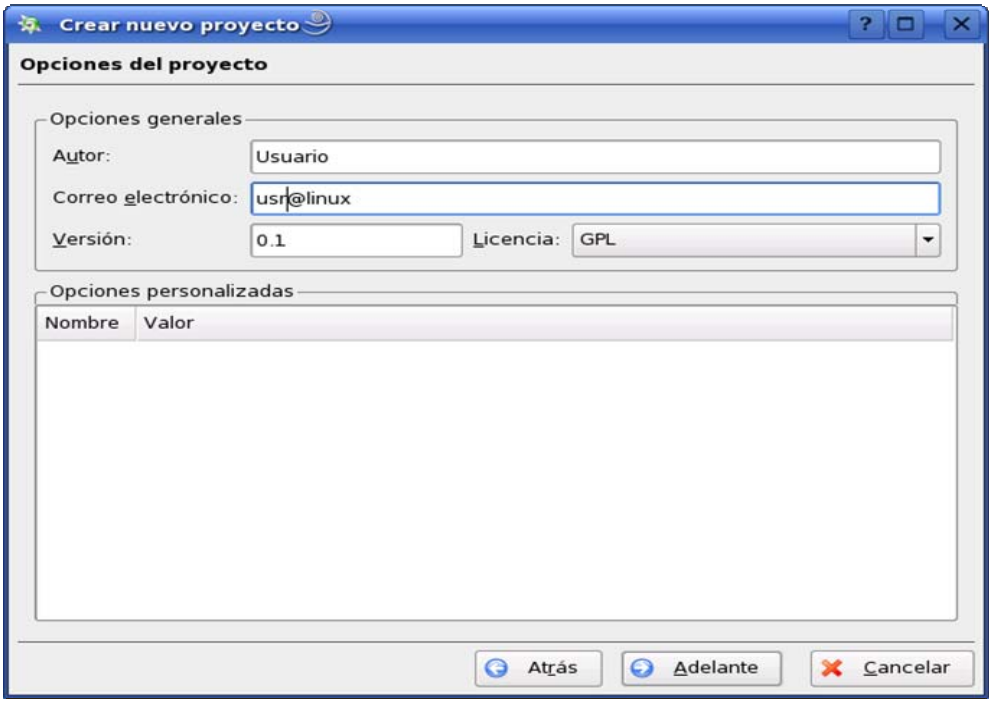


Fig. 34 Autor del proyecto.

- Click en Adelante.
- Dejar seleccionado el Sistema de control de versiones por defecto. (Ver Fig. 35)

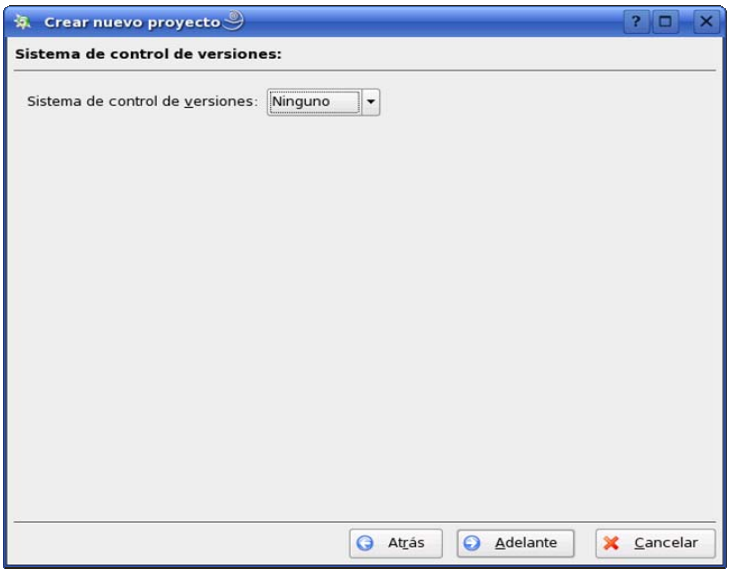


Fig. 35 Sistema de control de versiones.

- Click en Adelante.
- Click en Adelante.
- Click en Finalizar.

Con esto queda creado el esqueleto del Proyecto.

c. Click en el Menú Construir->Ejecutar Programa->Si. (Ver Fig. 36).

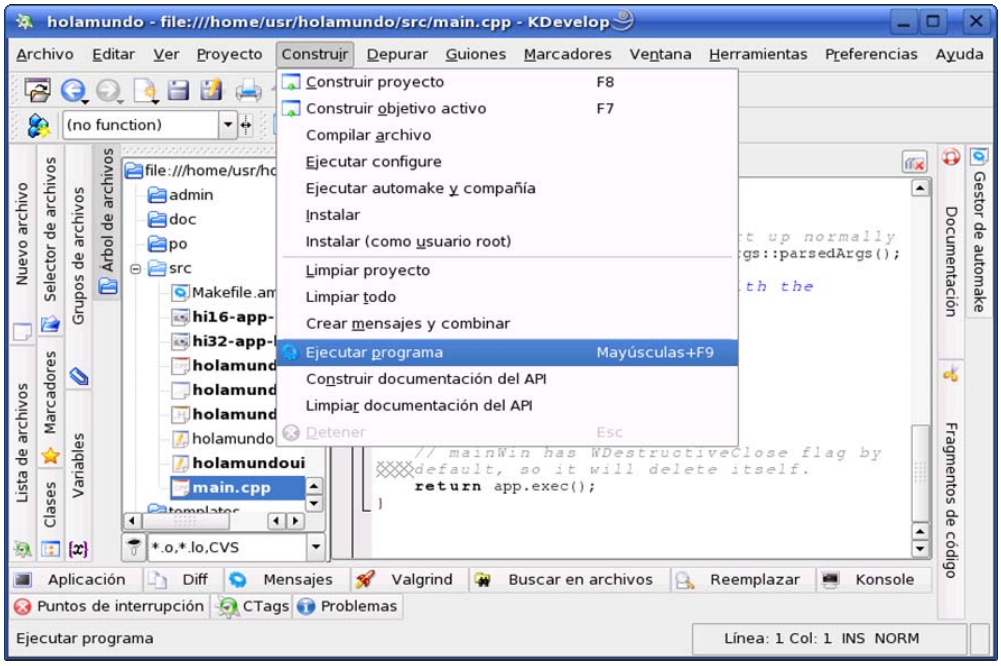


Fig. 36 Ejecutar programa.

Esperar a que se compilen los archivos creados en el proyecto, si todo va bien aparecerá una ventana. (Ver Fig. 37)

Fig. 37 Proyecto HolaMundo en ejecución.

Con esto se han creado una serie de ficheros, pero solamente interesarán los siguientes:

holamundo.h. Fichero generado durante el proceso de creación del proyecto en el directorio ../src, en el se declara la clase principal HolaMundo.

holamundo.ccp. Fichero generado durante el proceso de creación del proyecto en el directorio ../src, en el se realiza la implementación de la clase principal HolaMundo.

main.cpp Fichero generado durante el proceso de creación del proyecto en el directorio ../src, este representa el punto de inicio en la ejecución del proyecto.

d. Personalizar la interfaz de usuario. (Ver Fig. 38).
   - Click en el Menú Archivo ->Nuevo.
   - Nombre del Archivo -> “dlgHolaMundo”.
   - Tipo de archivo ->Dialog(.ui).
   - Click en Aceptar.
   - Click en Aceptar.

Fig. 38 Crear un fichero .ui (dlgHolamundo.ui).

Con esto se crea en la carpeta ../src del proyecto un fichero llamado HolaMundo.ui (que representa la interfaz de usuario que se desea personalizar).

- Click derecho sobre el archivo dlgHolaMundo.ui.
- Click en abrir con -> Diseñador Qt. (Ver Fig. 39).

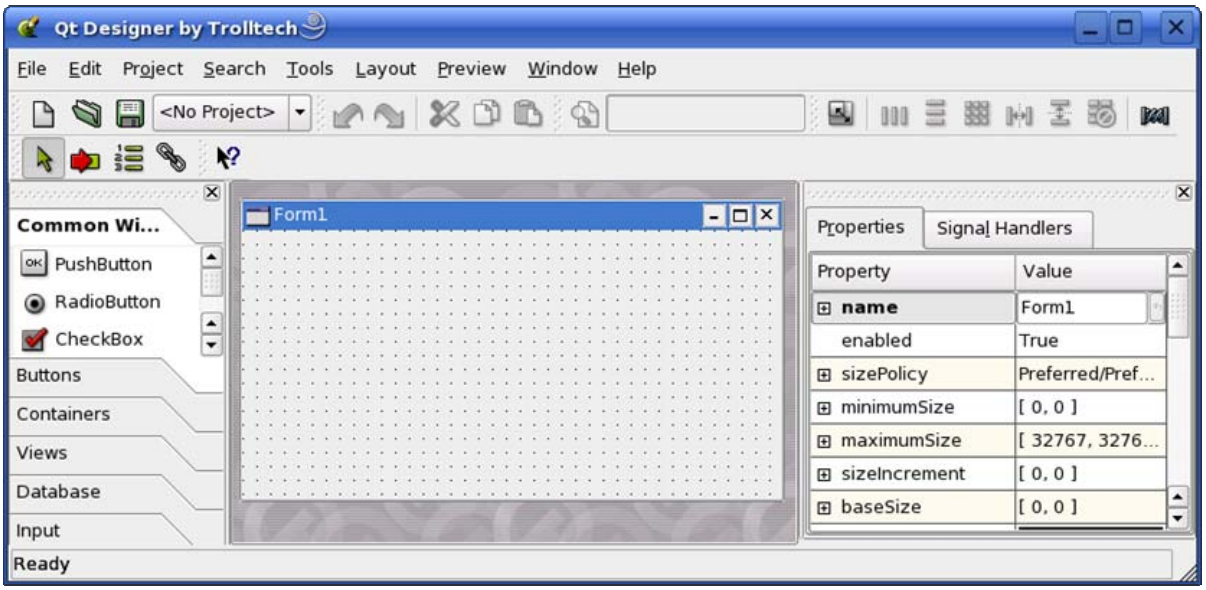


Fig. 39 Interfaz del Diseñador Qt.

Añadir una etiqueta (QLabel -> lblHolaMundo) y un botón (QPushButton-> btnClick). En el panel de propiedades cambiar el text de la etiqueta a una cadena vacía, el text del botón a la cadena "Haga click aquí", el name del formulario (Form1) a frmHolaMundo y el caption del formulario a la cadena “Hola Mundo”. (Ver Fig. 40).

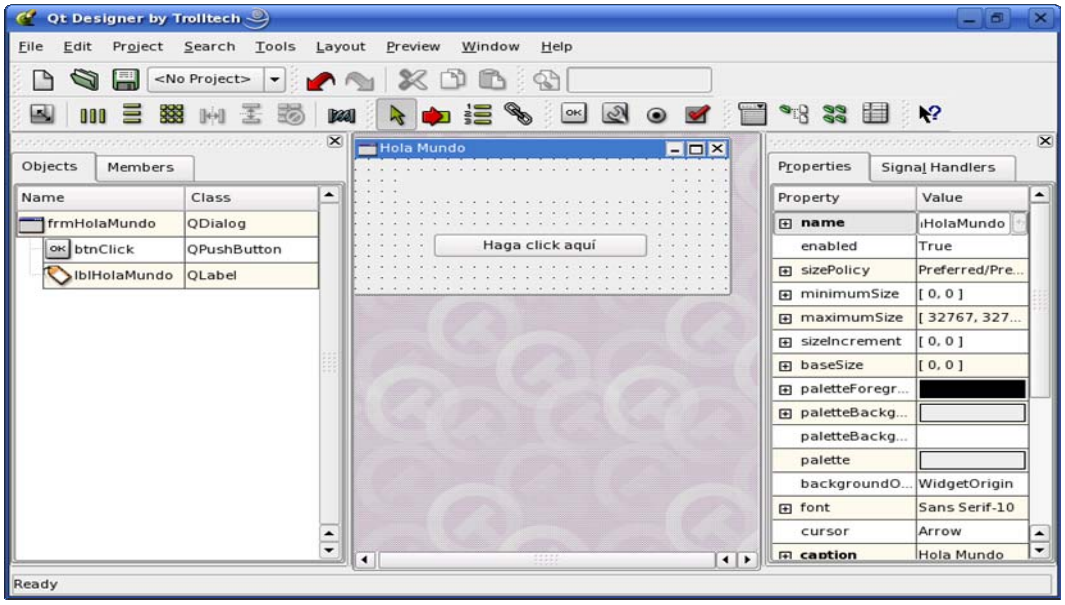


Fig. 40 Añadir controles al formulario.

En el menú Preview del Qt Designer se puede ver como quedaría el formulario con distintos aspectos según el tema de KDE seleccionado. Aún tendrá un problema antes de dejar concluida la interfaz. Si se da cuenta, en el preview, al maximizar, no queda bien la disposición de los componentes. Se pueden realizar dos cosas: Impedir que se pueda maximizar (sería lo mas lógico en esta aplicación) o bien hacer que se redimensione todo. A continuación se hará esto último para ilustrar el uso del los layouts.

Añadir unos widgets que se llaman espaciadores (spacers) a ambos lados de la etiqueta y el botón (Ver Fig. 41). En el formulario aparecerán como una especie de muelles. A continuación se seleccionan los dos espaciadores y la etiqueta y pulsar el botón derecho del ratón. En el menú que sale seleccionar layout horizontal. Hacer lo mismo con el botón. Al final seleccionar todo y elegir en el menú layout en rejilla (grid). El resultado es como el siguiente:

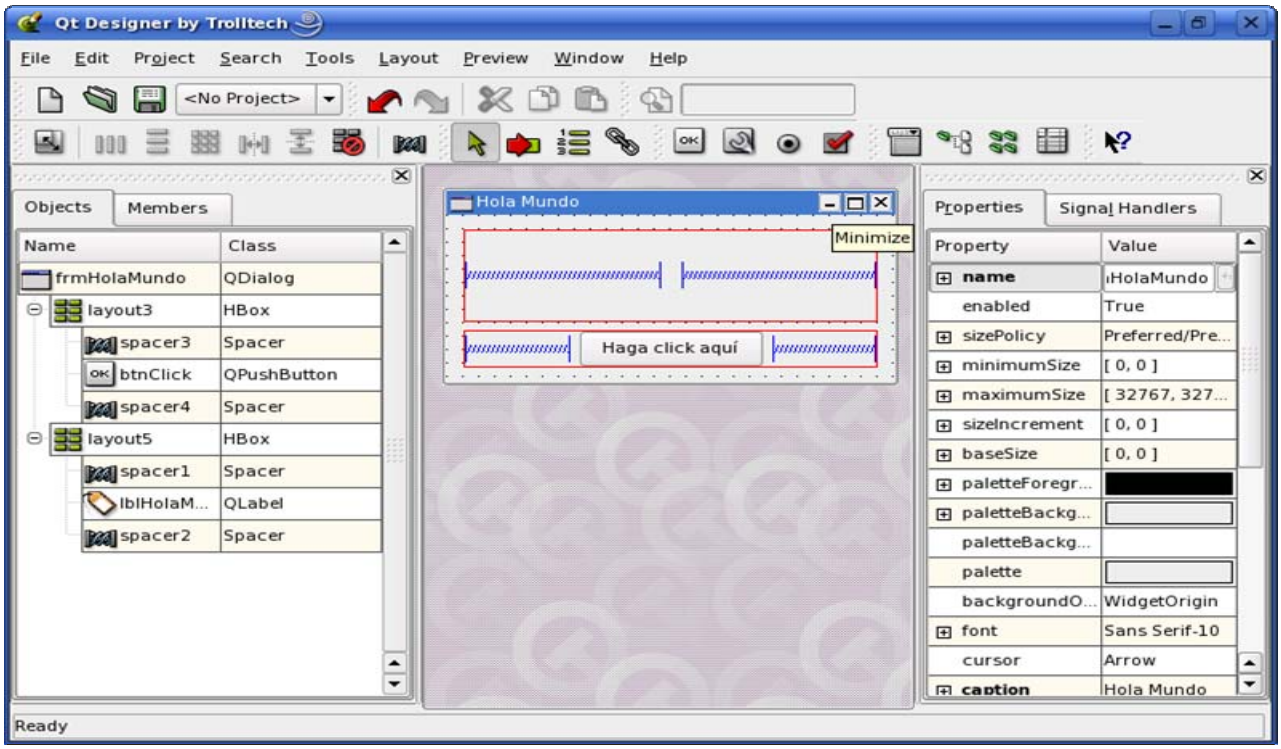


Fig. 41 Añadir spacers al formulario.

Ya sólo queda añadir las señales y los slots. Este es el mecanismo que tiene Qt para hacer que un widget se comunique con otro al capturarse eventos de usuario. Click derecho sobre el formulario, seleccionar en el menú conexiones (connections). Crear una nueva conexión para el botón (btnClick). Como señal escoger clicked() (Ver Fig. 42), que se dispara cuando el usuario haga click en el botón. Se le asigna al frmHolaMundo.

Fig. 42 Crear conexión para el botón btnClick.

Dar click a Edit Slots y después a New Function para crear un nuevo slot llamado diHola().

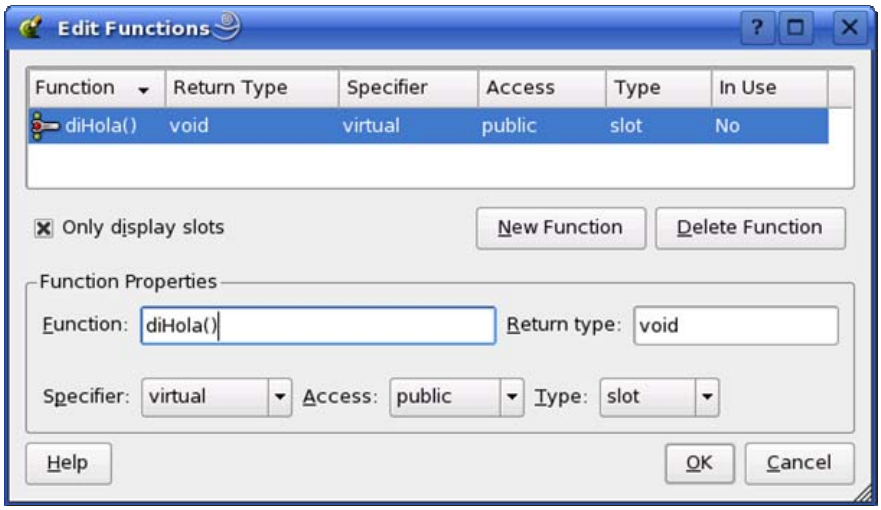


Fig. 43 Crear un slot diHola().

Luego asigne el nuevo slot al botón btnClick.

Fig. 44 Asignar el slot diHola() al botón btnClick.

Guardar y volver a KDevelop.

e. Editar los ficheros del proyecto.

Sólo queda derivar la ventana inicial HolaMundo de frmHolaMundo, para que herede de ella el método diHola().

En el fichero **holamundo.h** habrá que poner:

```
#include "../debug/src/dlgHolaMundo.h"
```

KDevelop crea en esta ruta el archivo de cabecera en el que se declara la clase (frmHolaMundo) que representa la interfaz de usuario, también crea el archivo de implementación de dicha clase. Cabe destacar que el nombre del formulario se corresponde con el nombre de la clase creada (frmHolaMundo) y el nombre que se le asigne al fichero .ui se corresponde con el nombre de los ficheros donde se declara e implementa dicha clase (dlgHolaMundo.h, dlgHolaMundo.cpp).

Se debe modificar la clase base de la cual se deriva HolaMundo, de forma que la nueva clase base sea frmHolaMundo.

```
class  HolaMundo : public KMainWindow
class  HolaMundo : public frmHolaMundo
```

Redefinir la función virtual (diHola) de la clase frmHolaMundo.

```
public slots:
    void diHola();
```

En el fichero **holamundo.cpp** se deberá hacer las siguientes modificaciones:

Se debe cambiar la implementación del constructor, aquí se deberá de llamar al constructor de la clase frmHolaMundo.

La implementación debe quedar de la siguiente forma:

```
HolaMundo::HolaMundo(): frmHolaMundo()
{
}

HolaMundo::~HolaMundo()
{
}
```

En este fichero se agrega el código que define la función diHola.

```
void HolaMundo::diHola()
{
        lblHolaMundo->setText("HOLA MUNDO");
}
```

En el fichero **main.cpp**, habrá que comentar la siguiente línea de código:

```
//RESTORE(HolaMundo);
```

Luego vuelva a ejecutar el proyecto. (Ver Fig. 45).

Fig. 45 Proyecto HolaMundo con eventos en ejecución.

3.2. Proyecto: Saludar.

Este ejemplo se basará en la creación de un formulario en el diseñador Qt, con el cual se ingresará un nombre, y al presionar un botón aparecerá en una etiqueta el nombre que se ingresó previamente.

a. Crear un nuevo proyecto de nombre “Saludar”.

Proyecto->Nuevo Proyecto-> "Aplicación de KDE sencilla”. Seguir las instrucciones del asistente para Crear nuevo proyecto. (Ver Fig. 46).

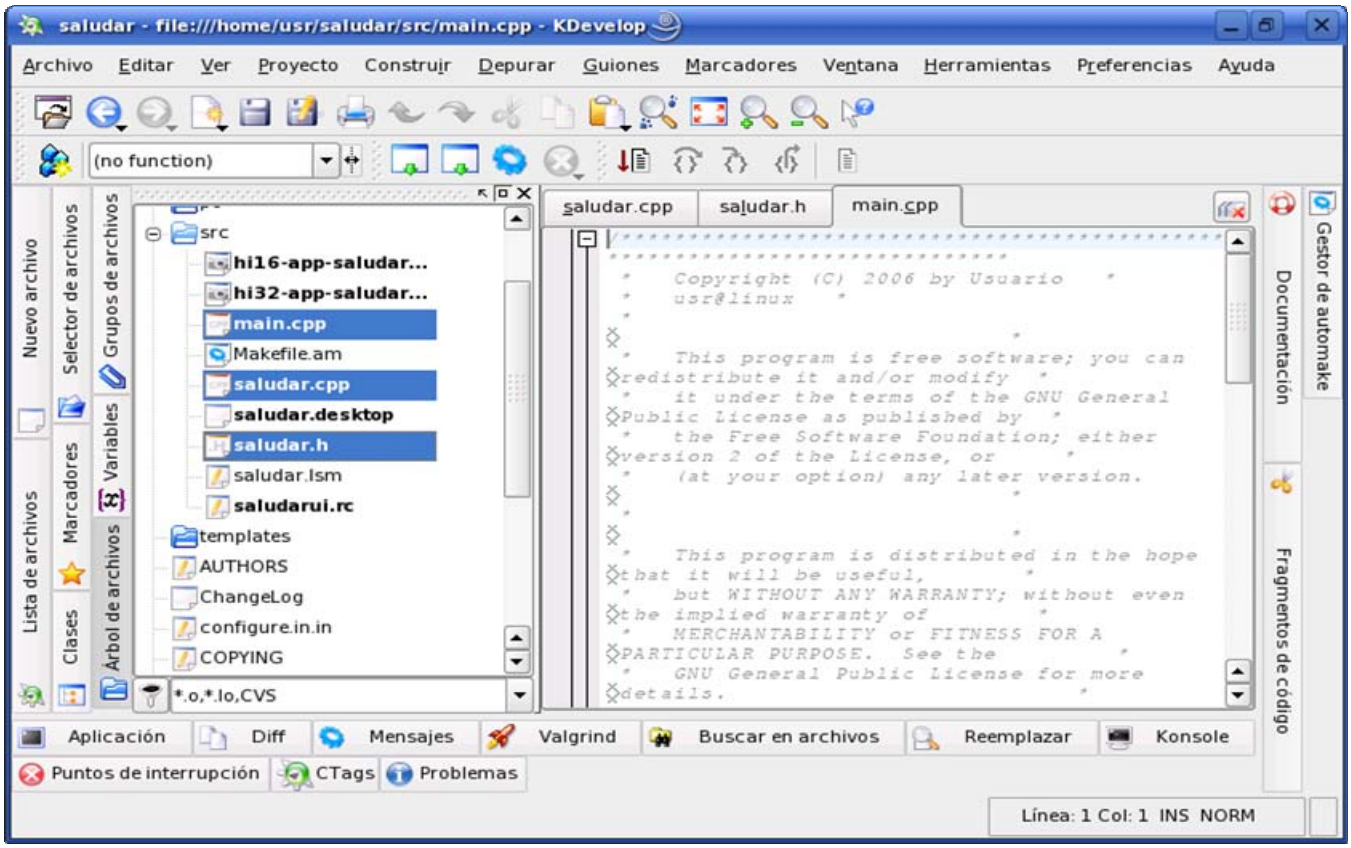


Fig. 46 Ficheros generados por KDevelop.

b. Añadir un fichero .ui. (Ver Fig. 47).
   - Click en el Menú Archivo ->Nuevo.
   - Nombre del Archivo -> dlgSaludar.
   - Tipo de archivo ->Dialog(.ui).
   - Click en Aceptar.
   - Click en Aceptar.

Fig. 47 Añadir un fichero .ui (dlgSaludar.ui).

c. Click derecho en el archivo dlgSaludar.ui->Abrir con->Diseñador Qt.

Y añadir los siguientes componentes al formulario:

- QLabel Label1.
- QLineEdit txtEntrada.
- QLabel Label2.
- QLabel lblSaludar.
- QPushButton btnClick.

Para cambiar el color de fondo del formulario, del botón y de las etiquetas, debe modificar la propiedad paletteBackground al color deseado. En el panel de propiedades cambiar el text de la etiqueta Label1 a la cadena “Entrada”, el text de la etiqueta Label2 a la cadena “Saludar a”, el text del botón btnClick a la cadena "Saludar", el text de la etiqueta lblSaludar a una cadena vacía, el name del formulario a frmSaludar y el caption a la cadena “Saludar”. (Ver Fig. 48).

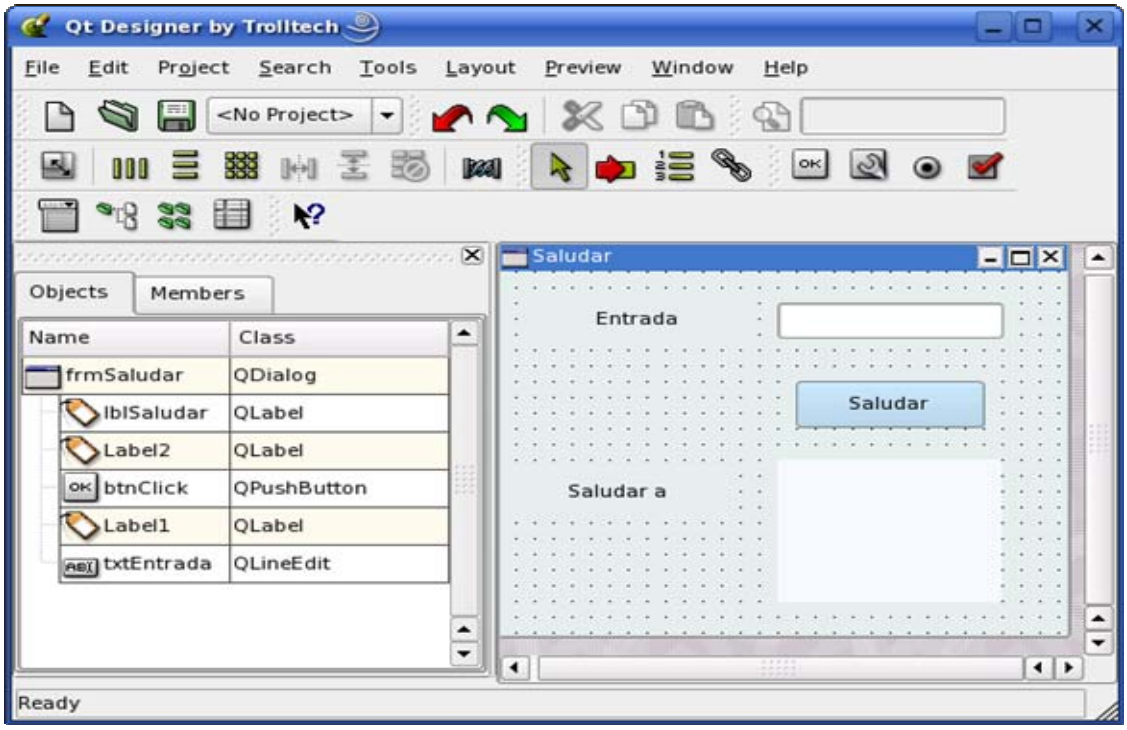


Fig. 48 Añadir widgets al formulario frmSaludar.

d. Crear las conexiones con el botón usando la signal tool de Qt, dando click derecho sobre el botón btnClick->Connections, aparecerá una pantalla, click en New e introducir los siguientes datos: (Ver Fig. 49).

- Click Sender->btnClick.
- Click Signal->clicked().
- Click Receiver->frmSaludar.

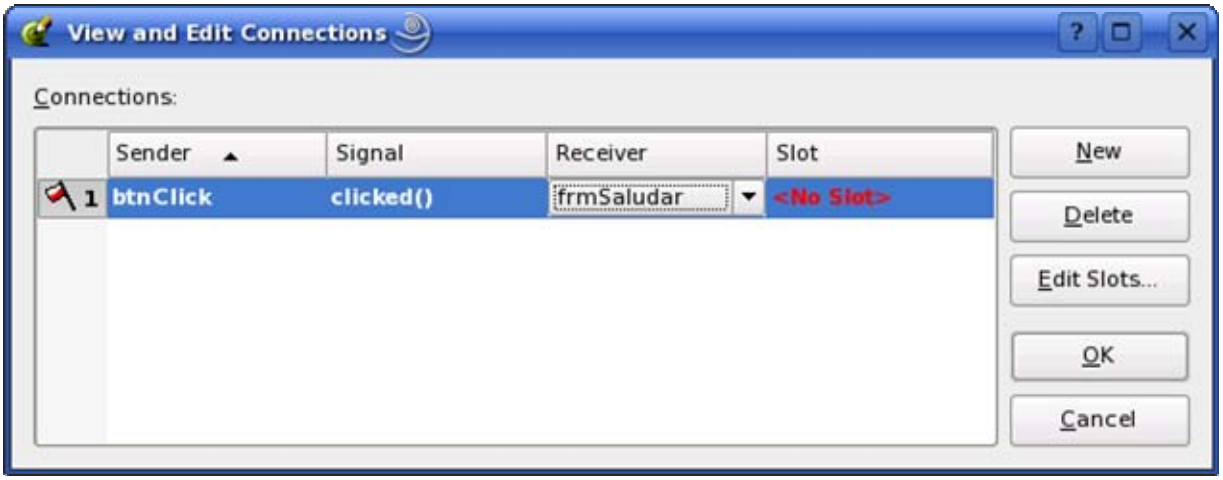


Fig. 49 Crear conexión para el botón btnClick.

Una vez seleccionados las opciones que aparecen en la pantalla (Ver Fig. 49), damos click en el botón “Edit Slots...” (Ver Fig. 50).

- Click en New Function
- Colocar el nombre de la función controladora (cmdSaludar()).

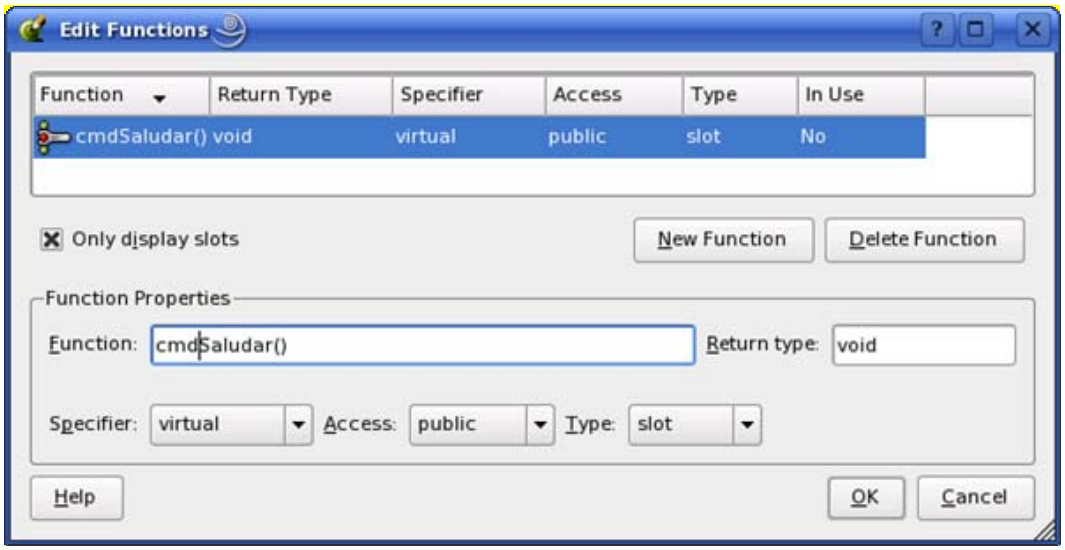


Fig. 50 Crear el slot cmdSaludar.

- Click en OK.
- Luego Click en Slot->cmdSaludar(). (Ver Fig. 51).
- Click en OK.

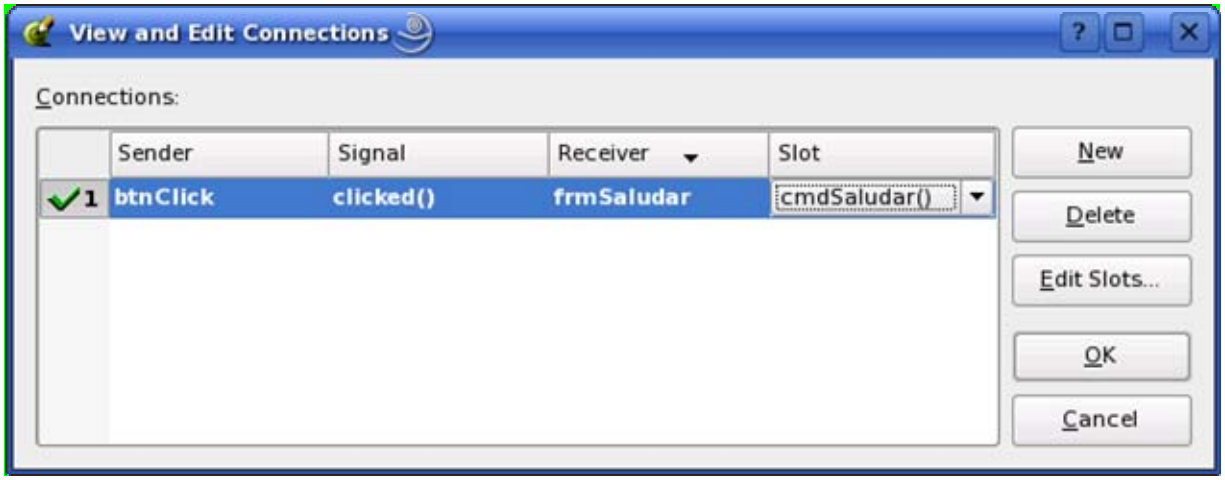


Fig. 51 Asignar el slot cmdSaludar al botón btnClick.

Guardamos el archivo modificado en Qt y regresamos a KDevelop. Luego se procede a la codificación de los eventos y a la compilación del programa. Se debe derivar la ventana inicial Saludar de frmSaludar. A continuación se deben añadir las siguientes líneas de código a los ficheros:

En saluadar.h

Para que soporte las funcionalidades de la librería Qt, se debe incluir los siguientes ficheros:

```
#include "../debug/src/dlgSaludar.h"
#include <qlineedit.h>
#include <qlabel.h>
```

Derivar la clase Saludar de la clase frmSaludar.

```
class  Saludar : public frmSaludar
```

Agregar la declaración de la función controladora.

```
public slots:
        void cmdSaludar();
```

En saluadar.cpp

Llamar al constructor de la clase frmSaludar.

```
Saludar::Saludar():frmSaludar()
{       }
```

Escribir el código de la implementación de la función cmdSaludar.

```
void Saludar::cmdSaludar()
{
        lblSaludar->setText(txtEntrada->text());
}
```

Explicación del código

Se asignó el contenido de la caja de texto a la etiqueta. Las etiquetas y cajas texto tienen dos propiedades, **setText(),** que sirve para asignar un nuevo texto al objeto, eliminando el texto que contenía, y la propiedad **text()**, con la cual se lee el texto que contiene el objeto, ambas propiedades aceptan solo variables del tipo string (en Qt se llama Qstring). También se puede asignar texto mediante lblSaludar->setText("Martín");

En el fichero **main.cpp** habrá que comentar la siguiente línea de código:

//RESTORE(Saludar);

Ahora solo queda ejecutar el proyecto, para esto se da click en el menú Depurar->Ejecutar Programa->Si ó dar click en el icono  (Ver Fig. 52).

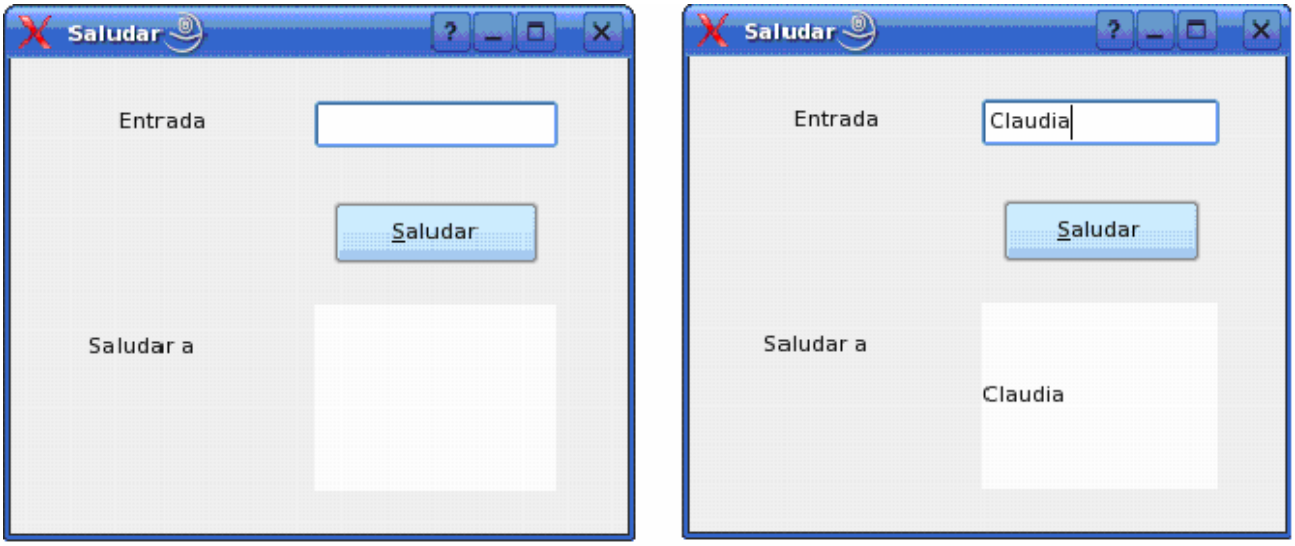


Fig. 52 Proyecto Saludar en ejecución.

3.3. Uso de varios formularios en una aplicación. (Proyecto: Multi_Formularios).

En el caso de que una aplicación requiera el uso de más de una ventana, se deberá crear la interfaz en Qt Designer, utilizando los mismos procedimientos descritos anteriormente (creación de objetos, darle los nombres, crear las conexiones con sus respectivas funciones, etc.), guardar la interfaz creada y regresar a KDevelop, para hacer las correspondientes derivaciones de las clases creadas a partir del fichero .ui.

En el siguiente ejemplo se empezaran a utilizar las tablas.

a. En KDevelop crear un nuevo proyecto llamado Multi_Formularios.
b. Añadir un nuevo fichero llamado dlgMulti_Formularios.ui (será el formulario que se presentará al ejecutar la aplicación).
c. Abrir el fichero dlgMulti_Formularios y editarlo con Diseñador Qt.
   - Cambiar las siguientes propiedades al formulario: name frmMulti_Formularios y de Caption “Llamar a otro formulario”, si desea cambie la propiedad paletteBackground al color deseado.
   - Añadir un botón, cambiar la propiedad text a “Cargar Formulario” y la propiedad name con btnCargar_Form. (Ver Fig. 53).

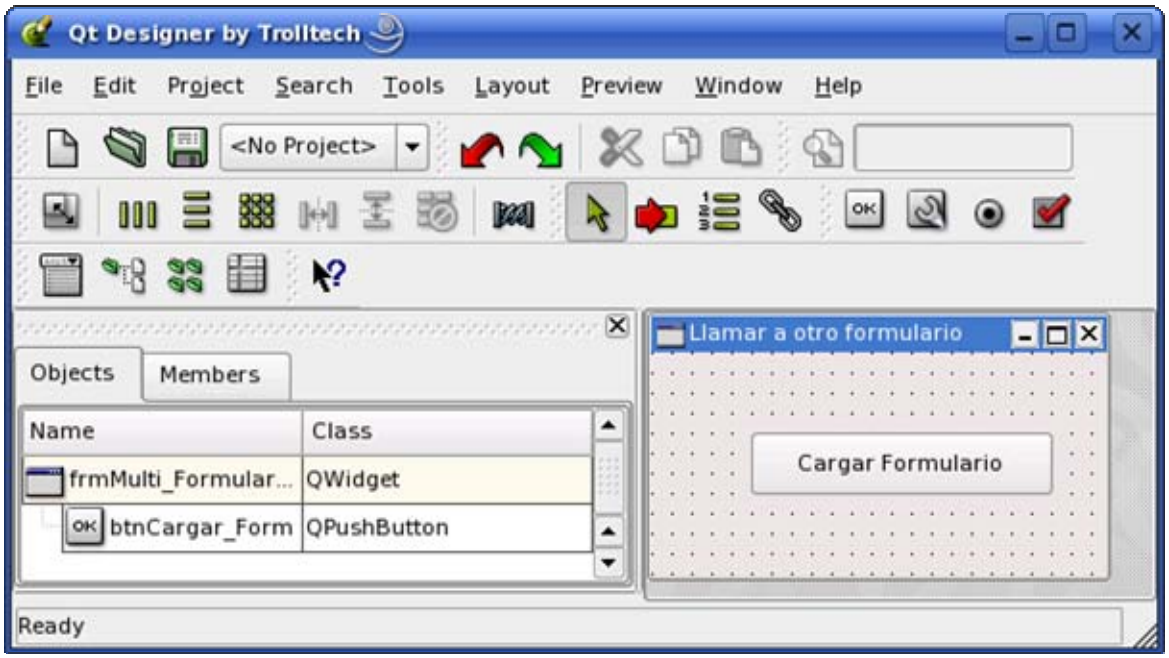


Fig. 53 Editar el formulario frmMulti_Formulario.

d. Crear la conexión con el botón, dando click derecho sobre el botón btnCargar_Form, clic en Connections, aparecerá una pantalla (Ver Fig. 54), Click en New e introducir los siguientes datos:
   - Click Sender-> btnCargar_Form
   - Click Signal->clicked()
   - Click Receiver-> frmMulti_Formularios

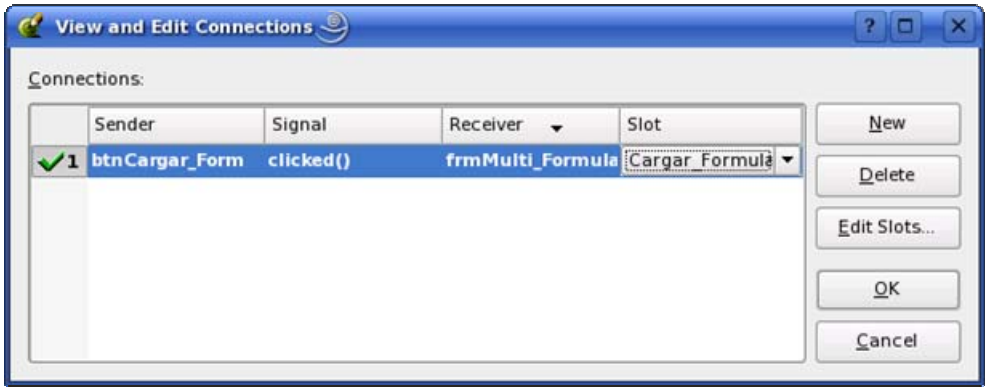


Fig. 54 Editar los spot del formulario frmMulti_Formulario.

- Click en el botón “Edit Slots...”
- Click en New Function
- Colocar el nombre de la función controladora (Cargar_Form ()). (Ver Fig. 55).
- Click en OK.

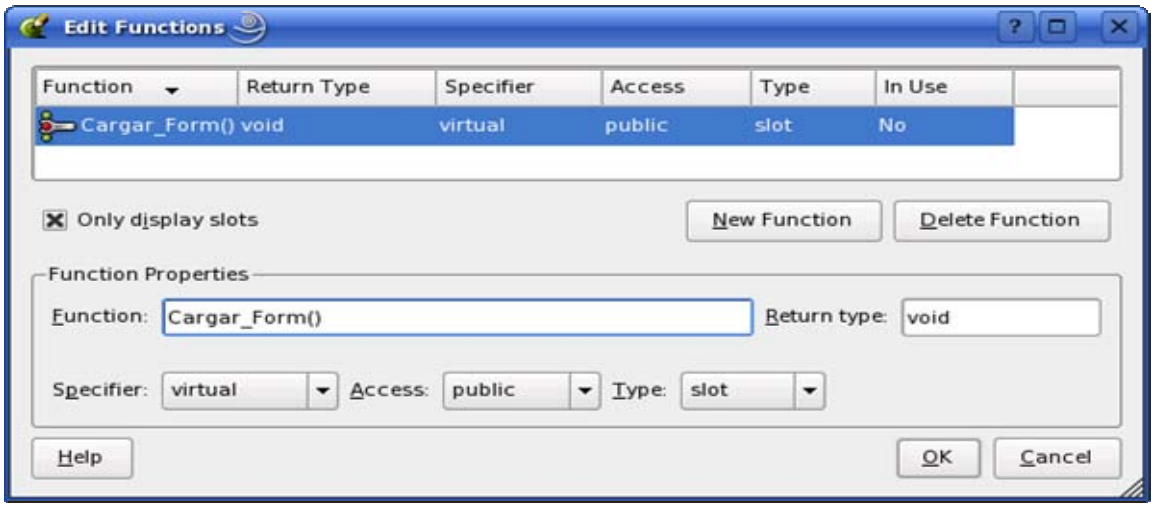


Fig. 55 Crear el slot Cargar_Form()

- Seleccionamos la nueva función creada para el campo Slot. (Ver Fig. 56).

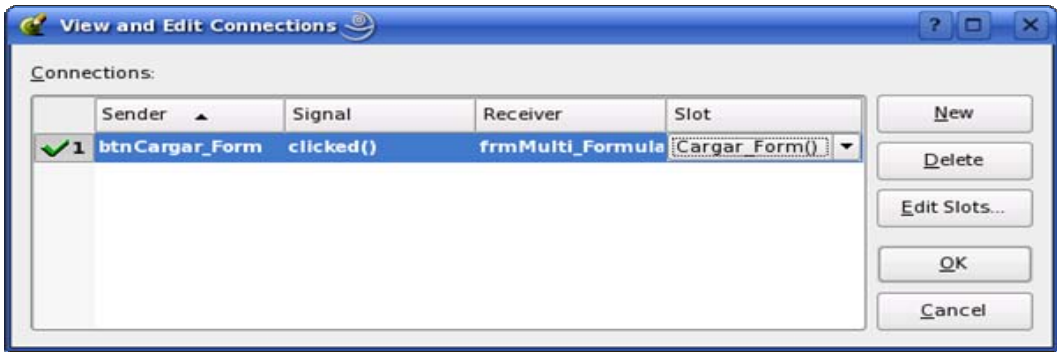


Fig. 56 Asignar el slot Cargar_Form() al botón btnCargar_Form.

- Click en OK.

Guardar los cambios y regresar a KDevelop.

e. Modificación de los ficheros multi_formularios.h, multi_formularios.cpp y main.cpp.

En **multi_formularios.h** incluir el fichero siguiente

```
#include "../debug/src/dlgMulti_Formularios.h"
```

Derivar la clase Multi_Formularios de frmMulti_Formularios

```
class  Multi_Formularios : public frmMulti_Formularios
```

Agregar la declaración de la función controladora.

```
public slots:
        void Cargar_Form();
```

En el fichero **multi_formularios.cpp** se debe agregar la definición de la función Cargar_Form, hasta este momento sin cuerpo.

```
void Multi_Formularios::Cargar_Form()
{
}
```

En el fichero **main.cpp** comentar la siguiente línea de código.

```
// RESTORE(Saludar);
```

Ejecutar el proyecto. (Ver Fig. 57).

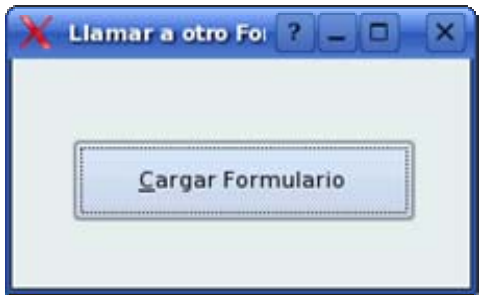


Fig. 57 Proyecto Multi_Formularios en ejecución.

El propósito de este ejemplo es que se visualice otro formulario al dar click en el botón Cargar Formulario. Por tanto se necesita diseñar dicho formulario, para esto, se debe agregar al proyecto un nuevo archivo .ui al que se llamará dlgTabla.ui. Modificar el name del formulario (Form1) a frmTabla y el caption a la cadena “Pasar Datos a Tabala”.

f. Creación del segundo formulario. (Ver Fig. 58).

La nueva interfaz contendrá los siguientes componentes:

- Tres etiquetas.
- Tres cajas de textos
- Un botón de pulsación.
- Una tabla.

Del lado izquierdo de la siguiente imagen se puede ver el tipo de objeto y el nombre de cada uno de estos componentes.

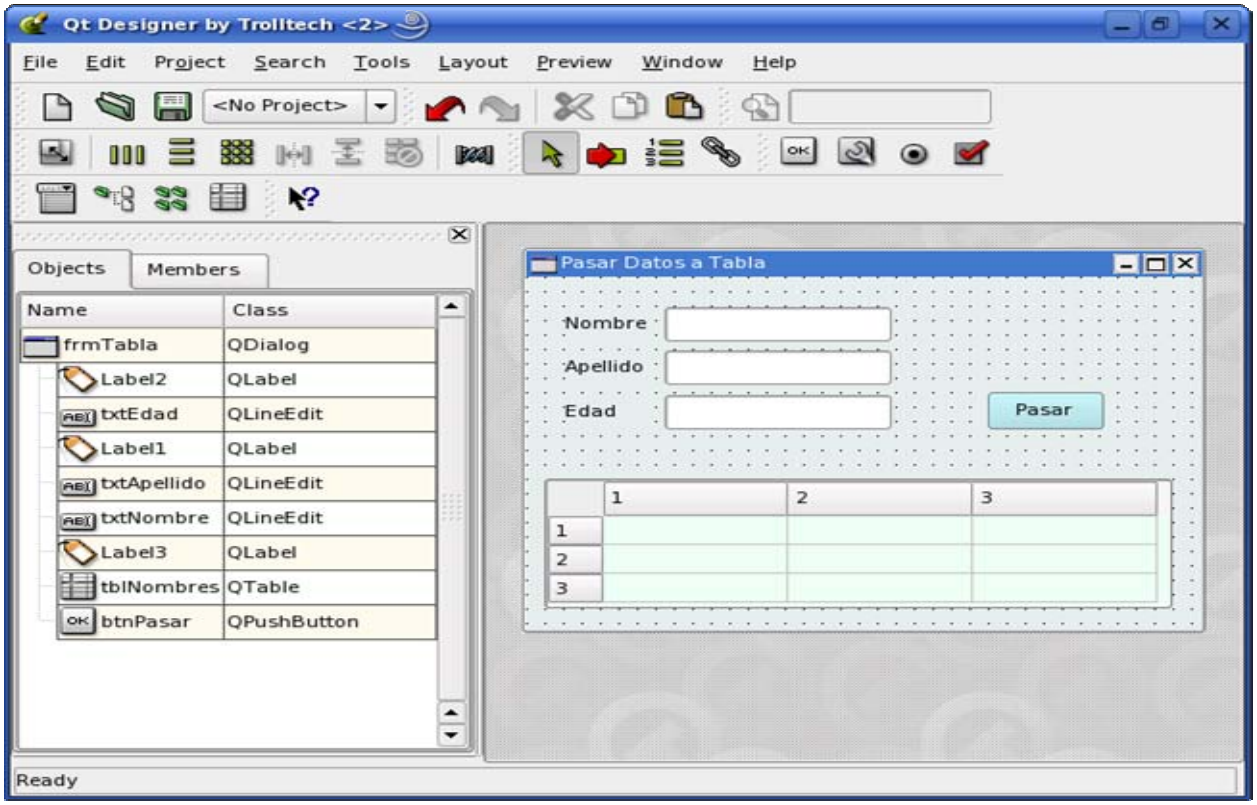


Fig. 58 Añadir un QTable al formulario.

A continuación se crea la conexión del botón btnPasar, para que al pulsar sobre él se pasen los datos ingresados a la tabla, lo mismo debe suceder cuando se pulse Enter sobre la caja de edición txtEdad. Para esto de debe crear una función llamada cmdPasar(). (Ver Fig. 59).

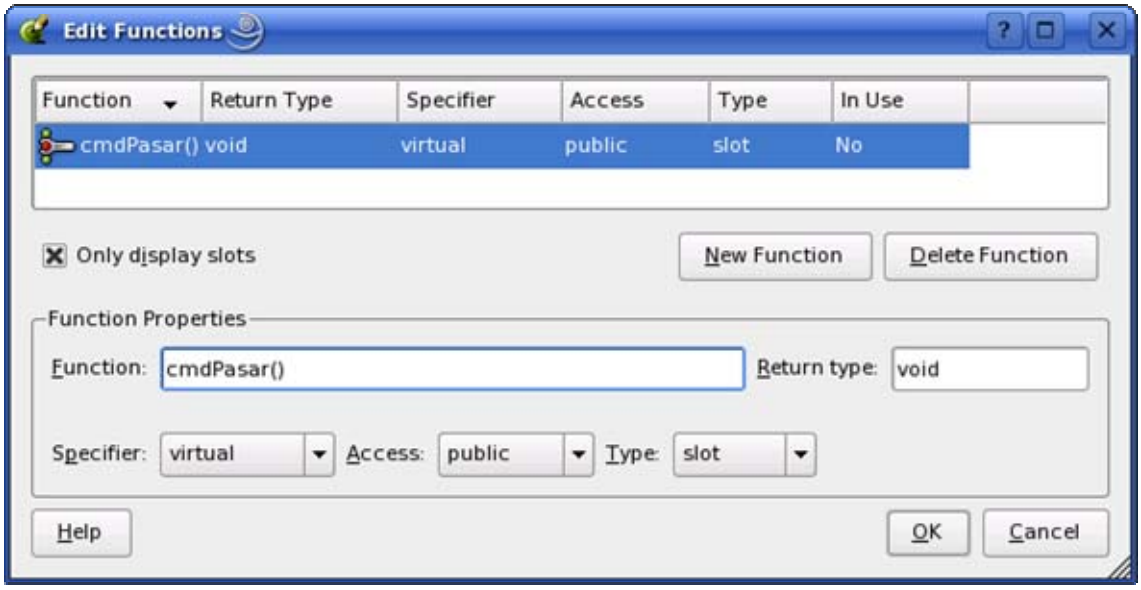


Fig. 59 Crear el slot cmdPasar().

Quedando las conexiones de la siguiente manera: (Ver Fig. 60).

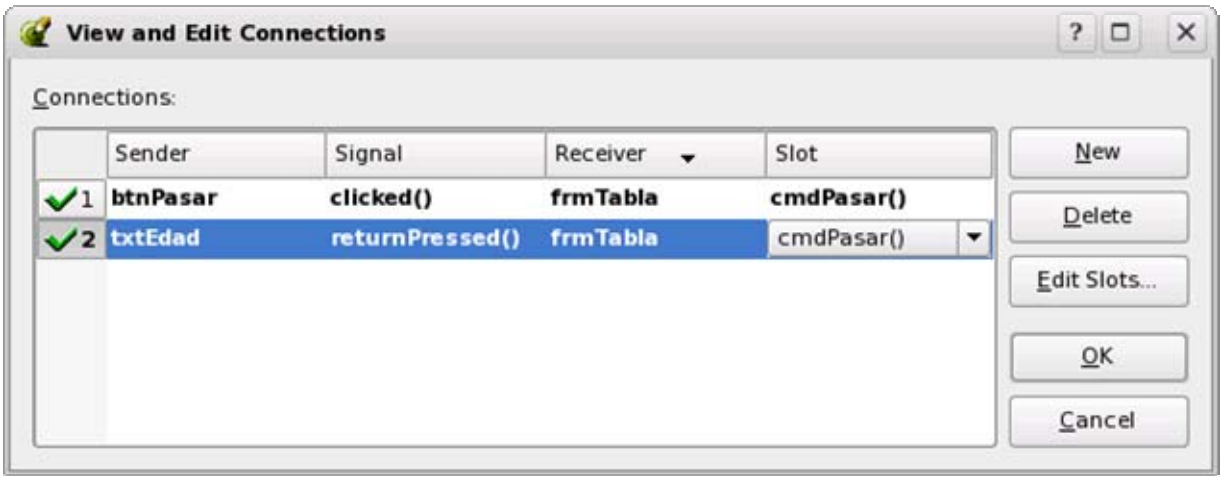


Fig. 60 Conexiones del formulario frmTabla.

Guardar el fichero y regresar a KDevelop.

Crear una nueva clase llamada CTabla la que será derivada de la clase frmTabla (clase que corresponde al formulario anterior).

- Click en el Menú Proyecto ->Nueva Clase. (Ver Fig. 61).

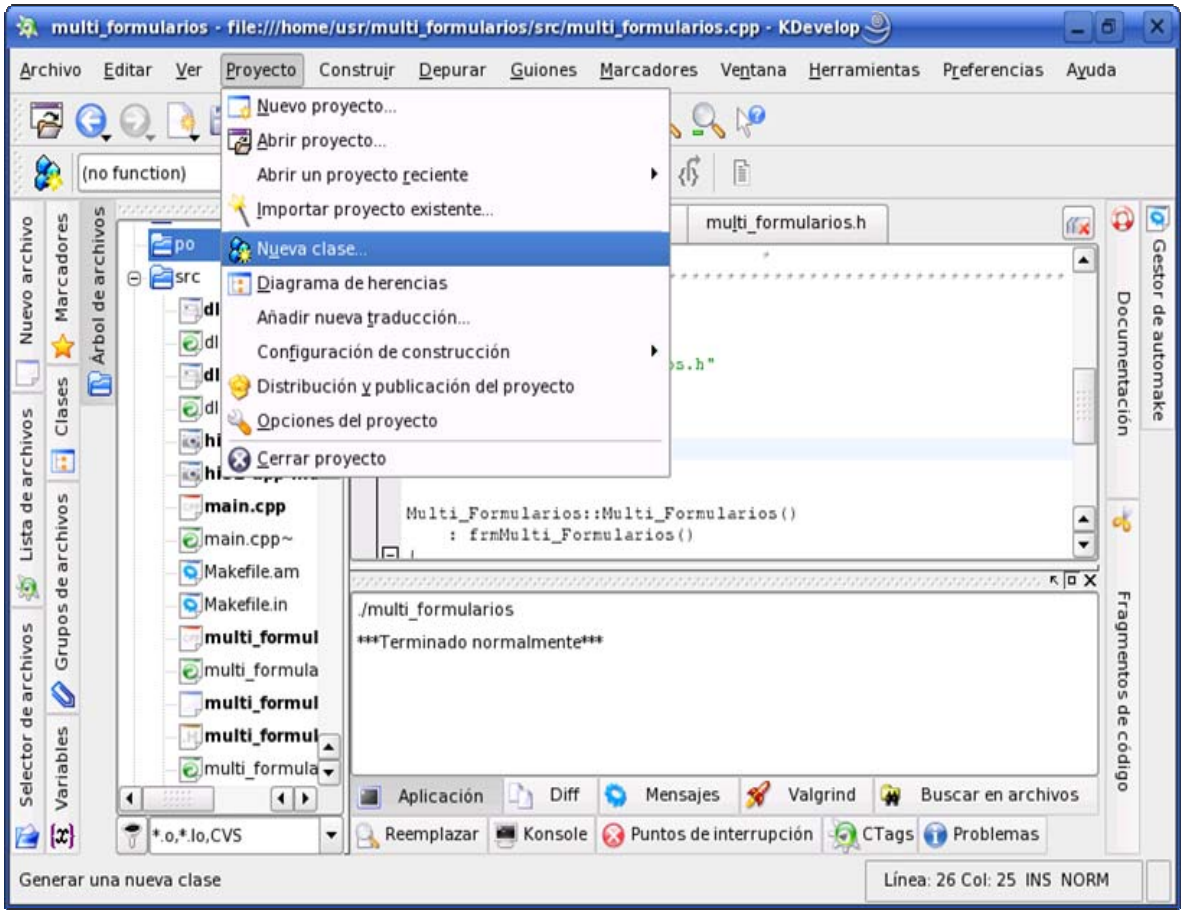


Fig. 61 Crear una clase.

- Nombre -> CTabla
- Clase base -> frmTabla (Ver Fig. 62).
- Pulsar Enter.

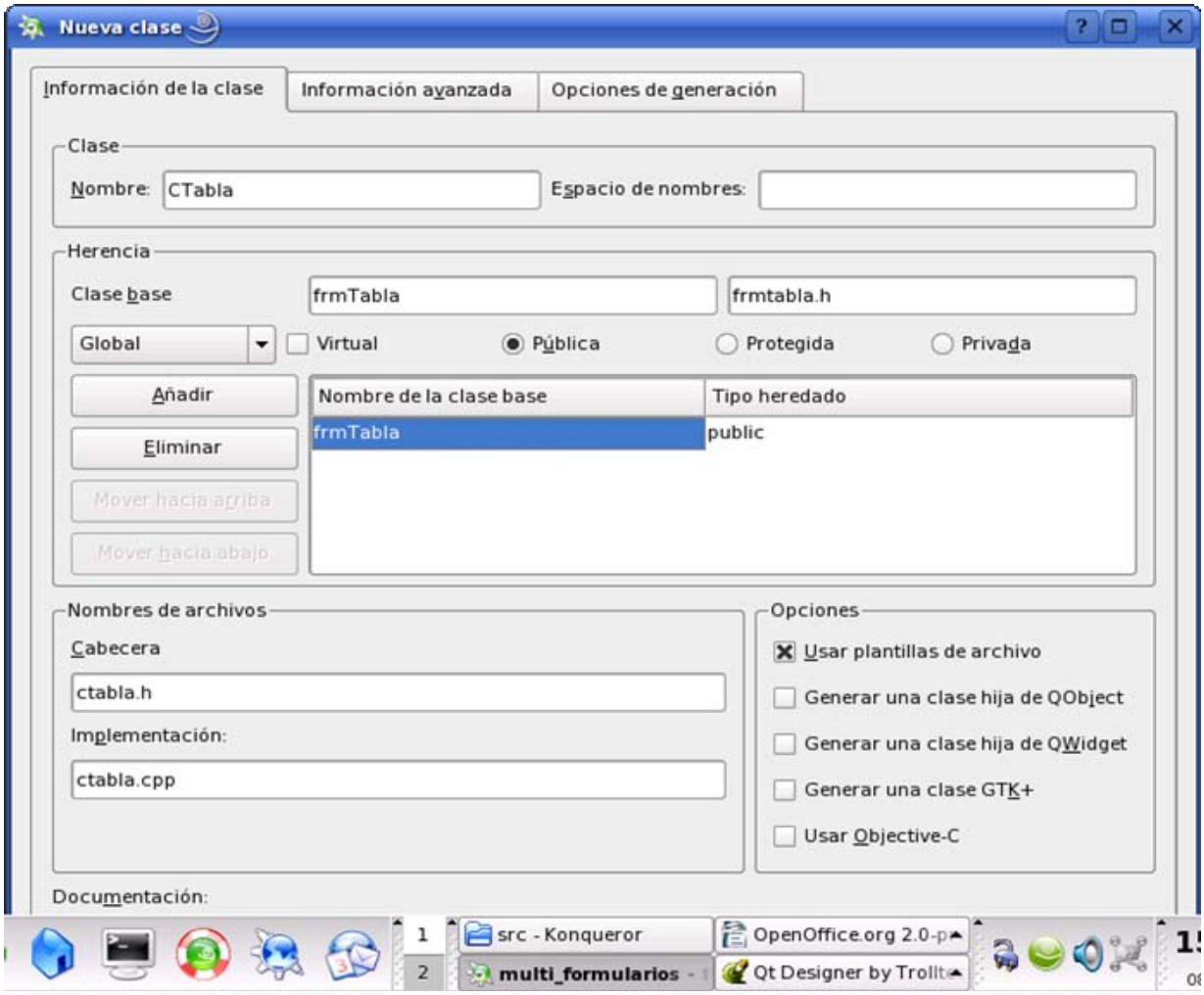


Fig. 62 Derivar una clase.

- Click en Aceptar.

Editar los ficheros ctabla.h y ctabla.cpp

En el fichero ctabla.h

Agregar los siguientes archivos de cabeceras:

```
#include "../debug/src/dlgTabla.h"
#include <qlineedit.h>
#include <qlabel.h>
#include <qtable.h>
```

Añadir el siguiente dato miembro:

```
private:
        int fila;
```

Redefinir la función cmdPasar(), heredada de clase frmTabla.

```
public slots:
        void cmdPasar();
```

En el fichero ctabla.cpp

Editar el constructor de la siguiente manera, de tal forma que al cargar el formulario, tengan títulos las columnas de la tabla.

```
CTabla::CTabla():frmTabla() {
  fila=0;
  QHeader *Titulos = tblNombres->horizontalHeader();
  Titulos->setLabel( 0, ("Nombre"));
  Titulos->setLabel( 1, ("Apellido"));
  Titulos->setLabel( 2, ("Edad"));
  Titulos->setMovingEnabled(TRUE);
  tblNombres->setReadOnly(TRUE);
}
```

Editar la función cmdPasar(), de tal forma que pase el contenido de las cajas de texto (txtNombre, txtApellido, txtEdad) a la fila y columna de la tabla, especificada por los dos primeros parámetros de la función setText.

```
void CTabla::cmdPasar() {
        if ( fila >= tblNombres->numRows() )
        {
          tblNombres->insertRows ( tblNombres->numRows() );
        }
        tblNombres->setText ( fila , 0 , txtNombre->text() );
        tblNombres->setText ( fila , 1 , txtApellido->text() );
        tblNombres->setText ( fila , 2 , txtEdad->text() );
        txtNombre->clear();
        txtApellido->clear();
        xtEdad->clear();
        fila++;
}
```

Para que al pulsar sobre el botón btnCargar_Form de la ventana principal, se pueda visualizar el formulario frmTabla, se debe agregar el código a la función controladora Cargar_Form().

En el fichero multiformulario.cpp.

```
#include "ctabla.h"
```

```
void Multi_Formularios::Cargar_Form()
{
CTabla *objTabla = new CTabla();
        objTabla ->show();
}
```

Ahora solo ejecute el programa(Ver Fig. 63 y Fig. 64).

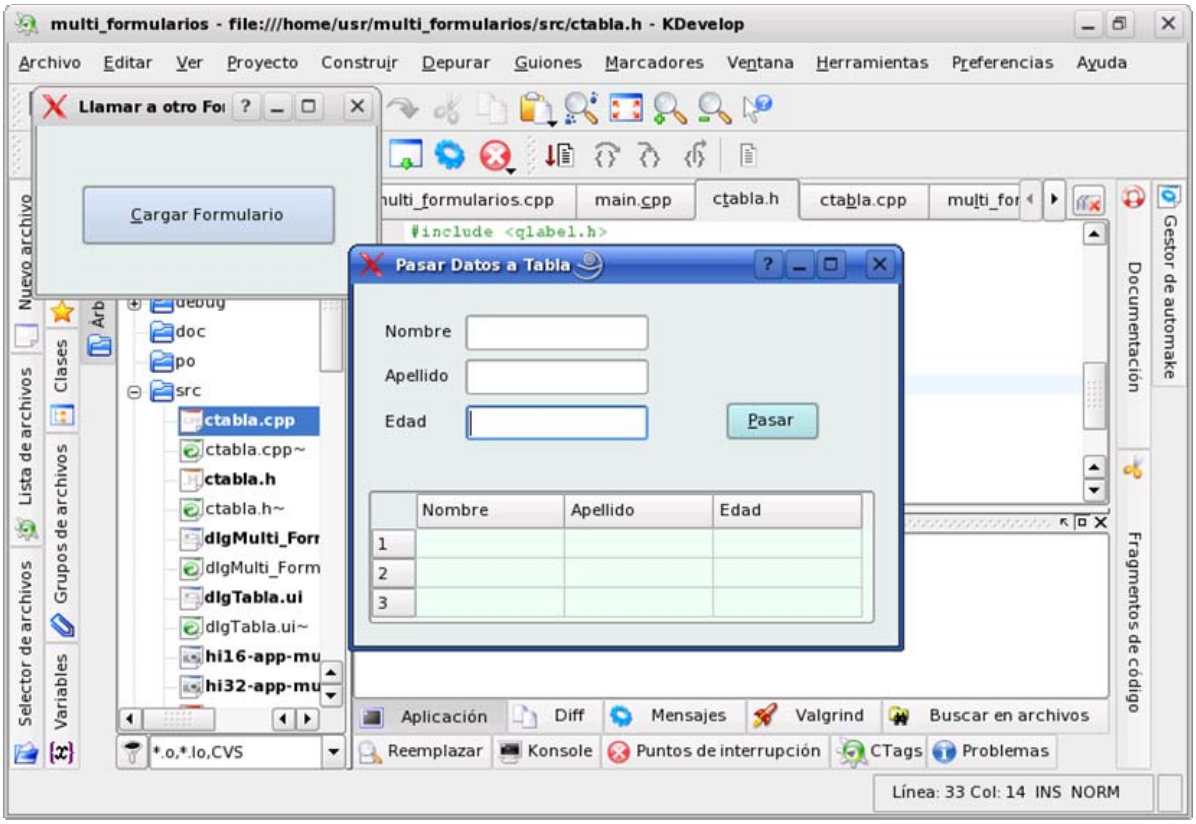


Fig. 63 Proyecto Multi_Formulario completo en ejecución.

Fig. 64 Formulario frmTabla.

4. Glosario de Términos.

Chat.

Chat (español: charla) es un anglicismo que usualmente se refiere a una comunicación escrita a través de Internet entre dos o más personas que se realiza instantáneamente. Es común que estas personas escriban bajo pseudónimos o alias llamados **nick**.

DNS.

El **Domain Name System** (DNS) es una base de datos distribuida y jerárquica que almacena información asociada a nombres de dominio en redes como Internet. Aunque como base de datos el DNS es capaz de asociar distintos tipos de información a cada nombre, los usos más comunes son la asignación de nombres de dominio a direcciones IP y la localización de los servidores de correo electrónico de cada dominio.

Emuladores.

En informática, un **emulador** es un software que permite ejecutar programas de ordenador, en una plataforma (arquitectura hardware o sistema operativo) diferente de la cual fueron escritos originalmente. A diferencia de un simulador, que sólo trata de reproducir el comportamiento del programa, un emulador trata de modelar de forma precisa el dispositivo que se está emulando.

Gcc.

GCC (GNU Compiler Collection, antes GNU C Compiler) compilador integrado del proyecto GNU para C, C++, Objective C y Fortran; es capaz de recibir un programa fuente en cualquiera de estos lenguajes y generar un programa ejecutable binario en el lenguaje de la máquina donde ha de correr.

GNU.

GNU/Linux es la denominación creada por Richard Stallman y otros investigadores, para el sistema operativo que utiliza el **kernel Linux** en conjunto con las aplicaciones de sistema creadas por el proyecto GNU. Comúnmente este sistema operativo es denominado simplemente Linux. Desde 1984, Richard Stallman y voluntarios están intentando crear un sistema operativo libre con un funcionamiento similar al UNIX, recreando todos los componentes necesarios para tener un sistema operativo funcional que se convertiría en el sistema operativo GNU.

GPL.

La GNU General Public License (Licencia General Pública de GNU) es la norma por la que se rigen muchas aplicaciones de libre distribución, entre las que se encuentran la mayoría de los trabajos en torno al sistema GNU/Linux. La licencia fue creada por la Fundación para el Software Libre (Free Software Foundation) a mediados de los 80 y está orientada principalmente a los términos de distribución, modificación y uso de software.

GPS.

El **Global Positioning System** (GPS) o **Sistema de Posicionamiento Global** (aunque se le suele conocer más con las siglas GPS su nombre más correcto es **NAVSTAR GPS**) es un Sistema Global de Navegación por Satélite (**GNSS**), el cual permite determinar en todo el mundo la posición de una persona, un vehículo o una nave, con una precisión de entre cuatro metros y quince metros. El GPS funciona mediante una red de satélites que se encuentran orbitando alrededor de la tierra. Cuando se desea determinar la posición, el aparato que se utiliza para ello localiza automáticamente como mínimo cuatro satélites de la red, de los que recibe unas señales indicando la posición y el reloj de cada uno de ellos. En base a estas señales, el aparato sincroniza el reloj del GPS y calcula el retraso de las señales, es decir, la distancia al satélite. Por "triangulación" calcula la posición en que éste se encuentra. La triangulación en el caso del GPS, se basa en determinar la distancia de cada satélite respecto al punto de medición. Conocidas las distancias, se determina fácilmente la propia posición relativa respecto a los tres satélites. Conociendo además las coordenadas o posición de cada uno de ellos por la señal que emiten, se obtiene la posición absoluta o coordenada reales del punto de medición. También se consigue una exactitud extrema en el reloj del GPS, similar a la de los relojes atómicos que desde tierra sincronizan a los satélites.

IDE.

Un entorno (o ambiente) integrado de desarrollo o en inglés Integrated Development Environment (IDE) es un programa compuesto por un conjunto de herramientas para un programador.

IDL.

Interface Definition Language (IDL) es, como su propio nombre indica un lenguaje de especificación de interfaces que se usa como parte de la tecnología CORBA. Ofrece la sintaxis necesaria para definir los métodos que queremos invocar remotamente. Una vez tengamos esta interfaz creada, deberemos pasarla por un compilador de interfaces que generará el *proxy* o *stub* cliente y el *skeleton* o *stub* servidor.

KDE.

KDE (K Desktop Environment) es un entorno de escritorio gráfico e infraestructura de desarrollo para sistemas Unix y en particular Linux. La 'K' originariamente representaba la palabra "*Kool*", pero su significado fue abandonado más tarde. Actualmente significa simplemente 'K', la letra inmediatamente anterior a la 'L' (inicial de Linux) en el alfabeto. Actualmente KDE es distribuido junto a muchas distribuciones Linux.

Middleware.

El Middleware es un software de conectividad que permite ofrecer un conjunto de servicios que hacen posible el funcionamiento de aplicaciones distribuidas sobre plataformas heterogéneas. Funciona como una capa de abstracción de software distribuida que se sitúa entre las capas de aplicaciones y las capas inferiores (sistema operativo y red). El Middleware nos abstrae de la complejidad y heterogeneidad de las redes de comunicaciones subyacentes, así como de los sistemas operativos y lenguajes de programación, proporcionando una API para la fácil programación y manejo de aplicaciones distribuidas. Dependiendo del problema a resolver y de las funciones necesarias serán útiles diferentes tipo de servicios de middleware.

MIPS.

Acrónimo de "millones de instrucciones por segundo". Es una forma de medir la potencia de los procesadores. Sin embargo, esta medida sólo es útil para comparar procesadores con el mismo juego de instrucciones, porque la misma tarea puede necesitar un número de instrucciones diferentes, si los juegos de instrucciones también lo son. En las comparativas, usualmente se representan los valores de pico, por lo que la medida no es del todo realista. La forma en que funciona la memoria que usa el procesador también es un factor clave para la potencia de un procesador, algo que no suele considerarse en los cálculos con MIPS. Debido a estos problemas, los investigadores han creado pruebas estandardizadas tales como SpecInt para medir el funcionamiento real, y las MIPS han caído en desuso.

Modularidad.

Modularidad es el atributo del software que permite a un programa ser manejable intelectualmente.

Multiprocesadores.

Se denomina multiprocesador a un ordenador que cuenta con dos o más microprocesadores (CPU’s). Gracias a esto, el multiprocesador puede ejecutar simultáneamente varios hilos pertenecientes a un mismo Proceso o bien a procesos diferentes. Los ordenadores

multiprocesador presentan problemas de diseño que no se encuentran en ordenadores monoprocesador. Estos problemas derivan del hecho de que dos programas pueden ejecutarse simultáneamente y, potencialmente, pueden interferirse entre sí. Concretamente, en lo que se refiere a las lecturas y escrituras en memoria. Existen dos arquitecturas que resuelven estos problemas:

- La arquitectura **NUMA**, donde cada procesador tiene acceso y control exclusivo a una parte de la memoria.
- La arquitectura **SMP**, donde todos los procesadores comparten toda la memoria.

Multiprogramación.

Es la técnica que permite que dos o más programas ocupen la misma unidad de memoria principal y que sean ejecutados al mismo tiempo. Así por ejemplo, mientras se ejecutan operaciones de entrada y salida de un programa, la unidad central de proceso puede ocuparse en realizar operaciones distintas de las de entrada y salida pertenecientes a otros programas. La multiprogramación se refiere a dos o más programas corriendo o procesándose al mismo tiempo. La multiprogramación se controla a través del sistema operativo, el cual observa los programas y los vigila hasta que estén concluidos. El número de programas que pueden multiprogramarse en forma efectiva, depende de una combinación de la cantidad de memoria, de la velocidad de la CPU y del número y velocidad de los recursos periféricos que tenga conectados, así como de la eficiencia del sistema operativo.

NFS

El **Network File System** (Sistema de archivos de red), es un sistema de archivos distribuido para un entorno de red de área local. Posibilita que distintos sistemas conectados a una misma red accedan a ficheros remotos como si se tratara de locales. Originalmente desarrollado por Sun Microsystems.

Programación paralela.

La **programación paralela** es una técnica de programación basada en la ejecución simultánea, bien sea en un mismo ordenador (con uno o varios procesadores) o en un cluster de ordenadores, en cuyo caso se denomina computación distribuida. Al contrario que en la programación concurrente, esta técnica enfatiza la verdadera simultaneidad en el tiempo de la ejecución de las tareas.

RPC.

El RPC (del inglés Remote Procedure Call, Llamada a Procedimiento Remoto) es un *protocolo* que permite a un programa de ordenador ejecutar código en otra máquina remota, sin tener que preocuparse por las comunicaciones entre ambos. El protocolo fue propuesto inicialmente por Sun Microsystems como un gran avance sobre los *sockets* usados hasta el momento. De esta manera el programador no tenía que estar pendiente de las comunicaciones, estando éstas encapsuladas dentro de las RPC.

Rpcgen.

Rpcgen es una herramienta que genera código C para implementar el protocolo RPC. La entrada a rpcgen es un fichero, en lenguaje similar a C, en el que se especifican las funciones que se van a implementar de forma remota, los parámetros que se le van a pasar, los parámetros que devuelve y los números de programa y versión de que dispondrá. La salida de este programa serán los ficheros en código C que implementan los **stub** de cliente y servidor. El uso de esta herramienta nos permitirá generar Llamadas a procedimientos remotos con unos conocimientos mínimos de los protocolos RPC y XDR.

Taxonomía.

En su sentido más general, la taxonomía (del griego, taxis, "ordenamiento", y nomos, "norma" o "regla") es la ciencia y el arte de la clasificación. Por lo general se emplea el término para designar la taxonomía biológica, esto es, la clasificación de los seres vivos en (taxa) o taxones que describen jerárquicamente las relaciones de parentesco, y similitud, entre organismos.

UTC.

Tiempo Universal Coordinado, o UTC, también conocido como tiempo civil, es la zona horaria de referencia respecto a la cual se calculan todas las otras zonas del mundo. Es el sucesor del GMT (Greenwich Mean Time: tiempo promedio del observatorio de Greenwich, en Londres) aunque todavía coloquialmente algunas veces se le denomina así. La nueva denominación fue acuñada para eliminar la inclusión de una localización específica en un estándar internacional, así como para basar la medida del tiempo en los estándares atómicos, más que en los celestes.

XDR

eXternal Data Representation Standard. Es un protocolo que permite la transferencia de datos entre máquinas independiente. Trabaja al nivel de ordenamiento de byte, códigos de caracteres y

sintaxis de estructura de datos muy similar a la de C para servir a este propósito. XDR se utiliza para el intercambio de datos en NFS y para los RPC (Remote Procedure Calls).

5. Glosario de Clases.

QcloseEvent.

La clase QCloseEvent contiene parámetros que describen un evento de cierre. El evento Close que envían los widgets, se produce cuando el usuario quiere cerrar un widget, normalmente escogiendo "Cierre" del menú de la ventana, o pulsando sobre el botón **X**. También se envía cuando se llama a *QWidget::close ()* para cerrar un widget programáticamente. Los eventos de cierre contienen una bandera que indica si el receptor quiere cerrar el widget o no. Cuando un widget acepta el evento close, se oculta. Si se niega a aceptar el evento de cierre nada pasa.

La función controladora *QWidget::closeEvent ()* recibe los eventos de cierre. Si no se desea esconder un widget, o se desea algo especial, se debe reimplementar esta función. Si el receptor del evento ha estado de acuerdo en cerrar el widget; llama a *accept()* y llamará a *ignore()* si no quiere cerrar el widget.

void QCloseEvent::accept ()

Acitva la bandera *accept* (aceptar) del objeto que produce el evento close. Activando la bandera o flag accept se indica que el receptor de este evento está de acuerdo en cerrar el widget. La bandera *accetpt* no es por defecto fija.

QDialog.

La clase QDialog es la clase base de las ventanas de diálogo. Una ventana de diálogo es una ventana de nivel superior usada sobre todo para tareas a corto plazo y breves comunicaciones con el usuario. QDialog puede ser modal o no modal. Puede tener botones por defecto. Un diálogo es siempre un widget a nivel superior, pero si tiene un padre (*QWidget*).

Diálogos modales.

Un diálogo modal, es un diálogo que bloquea la entrada a otras ventanas visibles en la misma aplicación. Los usuarios deben cerrarlo antes de que puedan tener acceso a cualquier otra ventana en uso. Los diálogos que se utilizan para solicitar un nombre de archivo, nombre de usuario son generalmente modales. La manera más común de mostrar un diálogo modal es llamar a la función *exec()*. Cuando el usuario cierra el diálogo, *exec()* proporcionará un valor de retorno útil.

Diálogos no modales.

Un diálogo no modal, es un diálogo que funciona independientemente de otras ventanas de la misma aplicación. Estos permiten que el usuario trabaje recíprocamente con la ventana principal de la aplicación y con el diálogo. Se muestran los diálogos no modales usando la función *show()*, que vuelve el control al llamador inmediatamente.

Valor de retorno de los diálogos modales.

Los diálogos modales a menudo se utilizan en situaciones donde se requiere un valor de retorno, por ej. para indicar si el usuario presionó el botón “OK” o "Cancel". Un diálogo puede ser cerrado llamando a la función *accept()* o el slot *reject()*, y la función *exec()* devolverá Accepted o Rejected respectivamente. La llamada a *exec()* devuelve el resultado del diálogo. El resultado está también disponible con la función *result()* si el diálogo no se ha destruido.

int QDialog::exec () [slot]

Muestra el diálogo como un diálogo modal, bloqueando hasta que el usuario lo cierre. La función retorna un resultado de *DialogCode*. (Valor devuelto por un diálogo modal, *QDialog::Accepted* o *QDialog::Rejected*).

Los usuarios no pueden actuar recíprocamente con cualquier otra ventana en la misma aplicación hasta que ellos cierren el diálogo.

void QDialog::setModal (bool modal)

Muestra el diálogo como modal, si el valor **modal** es *TRUE*, también se puede especificar la propiedad *MODAL*.

QFile.

QFile es un dispositivo de I/O para lectura y escritura de archivos binarios y de texto. El nombre del archivo normalmente se pasa en el constructor pero puede establecerse o cambiarse con la función *setName().* Se puede verificar la existencia de un archivo con *exists()* y eliminar un archivo con *remove()*.Los datos normalmente se leen y escriben usando *QDataStream* o *QTextStream*, pero se puede leer con el *readBlock()* y *readLine()* y escribir con el *writeBlock()*. QFile también soporta las funciones *getch()*, *ungetch()* y *putch()*.

El tamaño del archivo es retornado por *size()*. Se puede conseguir la posición actual del archivo o moverse a una nueva posición con la función *at()*. Si se ha alcanzado el fin del archivo, la función *atEnd()* retorna verdadero. El manejador del archivo es retornado por *handle()*.

QFile::QFile (const QString & name)
Construye un objeto de tipo QFile con el nombre especificado en el argumento **name**.

bool QFile::open (int m) [virtual]
Abre un fichero con el nombre anteriormente establecido (ya sea en la construcción del objeto QFile o utilizando la función setName()), en modo **m**. Retorna verdadero en caso de éxito o falso en caso contrario.
El parámetro del modo m, debe ser una combinación de las banderas siguientes:

Tabla 3 Modos de abrir un fichero.

Bandera	Significado
IO_Raw	Acceso en crudo al fichero. Sin buffer intermedio.
IO_ReadOnly	Abre el archivo en modo solo lectura.
IO_WriteOnly	Abre el archivo en modo solo escritura. Si esta bandera se usa con otra bandera, por ejemplo. IO_ReadOnly o IO_Raw o IO_Append, el archivo no se trunca; pero si se utiliza sola (o con IO_Truncate), el archivo es truncado.
IO_ReadWrite	Abre el archivo en modo de lectura/escritura, equivalente a (IO_ReadOnly | IO_WriteOnly).
IO_Append	Abre el archivo .para hacerlo editable e ir añadiendo al final de éste (Realmente se debe usar (IO_WriteOnly | IO_Append). Este modo es muy útil cuando se quiere añadir algo a un archivo. El descriptor del archivo se pone al final de éste.
IO_Truncate	Trunca el archivo.

void QFile::close () [virtual]
Cierra un archivo abierto.

*Q_LONG QIODevice::readBlock (char * data, Q_ULONG maxlen) [pure virtual]* es necesario QIODevice
Lee a lo sumo **maxlen** bytes del dispositivo de I/O, los almacena en **data** y retorna el número de bytes que leyó realmente. Esta función debe devolver -1 si un error fatal ocurre y debe devolver 0 si no hay ningún byte para leer. El dispositivo debe abrirse para lectura. Esta función virtual debe ser reimplementada por todas las subclases.

*Q_LONG QIODevice::writeBlock (const char * data, Q_ULONG len) [pure virtual]*

Escribe **len** bytes contenidos en **data** al dispositivo de I/O y retorna el número de bytes realmente escritos. Esta función debe devolver -1 si un error fatal ocurre. Esta función virtual debe ser los reimplementada por todas las subclases.

QFileDialog.

La clase de QFileDialog proporciona los diálogos que permiten que los usuarios seleccionen archivos o directorios.

*QString QFileDialog::getOpenFileName (const QString & startWith = QString::null, const QString & filter = QString::null, QWidget * parent = 0, const char * name = 0, const QString & caption = QString::null, QString * selectedFilter = 0, bool resolveSymlinks = TRUE) [static]*

Esta es una función estática que retorna el nombre de un archivo existente seleccionando por el usuario, si el usuario presiona Cancelar devuelve una cadena nula. (*QString::null*)

```
QString s =     QFileDialog::getOpenFileName(
                "/home",
                "Images (*.png *.xpm *.jpg)",
                this,
                "open file dialog",
                "Choose a file to open" );
```

La función crea un diálogo modal, visualizado encima de la ventana padre, el directorio activo del dialogo se pondrá al valor de **startWith** (home), si el **startWith** incluye un nombre de archivo el archivo se seleccionará, los filtros se ponen para filtrar el tipo de archivos que se desean mostrar en el diálogo, el filtro seleccionado se pone en **selectedFilter**. El **selectedFilter** y el **startWith** pueden ser cadenas nulas Qstring::null. Si el **caption** no se especifica se utiliza el título predeterminado.

QListBox.

El widget QListBox proporciona una lista de items (elementos) seleccionables, inalterables. Esta es típicamente una lista de una sola columna.

QListBox agregará barras de desplazamiento, cuando sean necesarias, pero no se piensa para listas realmente grandes. Si se desea unos mil items, es mejor utilizar otro tipo de widget, principalmente porque las barras de desplazamiento no proporcionarán una buena navegación,

pero también porque QListBox puede llegar a ser lento con las listas enormes. Algunas alternativas posibles pueden ser: *QListView* y *QTable*.

void QListBox::clear () [slot]

Anula todos los items en la lista.

uint QListBox::count () const

Retorna el número de items que posee la lista.

int QListBox::currentItem () const

Retorna el índice del item seleccionado actualmente.

QString QListBox::currentText () const

Retorna el texto del item seleccionado actualmente.

void QListBox::insertItem (const QString & text, int index = -1)

Función miembro sobrecargada. Inserta un Nuevo item **txt** en la Lista, en el índice especificado por **index**.

void QListBox::removeItem (int index)

Remueve y quita el item especificado en **index**.

QMessegeBox.

La clase QMessageBox proporciona un diálogo modal con un mensaje corto, un icono, y algunos botones. Las cajas de mensajes se utilizan para proporcionar mensajes informativos y para hacer preguntas simples. QMessageBox proporciona una gama de diversos mensajes:

Tabla 4 Tipos de QMessageBox.

	Question	Para las cajas de mensajes que hacen una pregunta, como parte de la operación normal.
	Information	Para las cajas de mensaje que son parte de la operación normal.
	Warning	Para las cajas de mensajes que informan a los usuarios sobre errores inusuales.
	Critical	Para las cajas de mensajes que informan al usuario sobre errores críticos.

*void QMessageBox::about (QWidget * parent, const QString & caption, const QString & text) [static]*

Muestra una caja de mensaje simple con título **caption** y el texto o mensaje **text**. El padre de la caja es **parent**.

La función *about()* busca un icono conveniente en cuatro localizaciones:

- Busca el *parent->icon()* si éste existe.
- Si no, intenta con el widget de nivel superior que contiene al padre.
- Si eso falla, intenta con el widget principal.
- Como último recurso utiliza el icono de Information.

La caja contiene un solo botón etiquetado "OK".

*int QMessageBox::question (QWidget * parent, const QString & caption, const QString & text, int button0, int button1 = 0, int button2 = 0) [static]*

Abre un diálogo de pregunta con título **caption** y texto **text**. El diálogo puede tener hasta tres botones. Retorna el valor numérico del botón que fue pulsado por el usuario Cada uno de los botones, button0, button1 y button2 se pueden fijar a uno de los valores siguientes:

- QMessageBox::NoButton
- QMessageBox::Ok
- QMessageBox::Cancel
- QMessageBox::Yes
- QMessageBox::No
- QMessageBox::Abort
- QMessageBox::Retry
- QMessageBox::Ignore
- QMessageBox::YesAll
- QmessageBox::NoAll

QString.

*const char * QString::ascii () const*

Devuelve una representación en 8-bit del ASCII de la cadena.

bool QString::isEmpty () const

Devuelve TRUE si la cadena es vacía, es decir si el length() == 0; si no devuelve FALSE. Las cadenas nulas son también vacías.

QTimer.

La clase QTimer proporciona señales del contador de tiempo. Utiliza acontecimientos del contador de tiempo internamente para proporcionar un contador de tiempo más versátil. QTimer es muy fácil de utilizar: se crea un QTimer, se llama a la función *start()* para comenzarlo y para conectarlo con el slot apropiado se llama a la función *timeout().* Cuando el tiempo está encima de él emitirá la señal del *timeout().* Un objeto QTimer se destruye automáticamente cuando se destruye su objeto padre.

void QTimer::timeout () [signal]

Se emite esta señal cuando se activa el contador de tiempo.

int QTimer::start (int msec, bool sshot = FALSE)

Comienza el contador de tiempo con un descanso de **msec** milisegundos, retorna el identificador del contador de tiempo, o cero cuando al comenzar el contador de tiempo ocurrió un falló. Si el **sshot** es TRUE, el contador de tiempo será activado solamente una vez; si no, continuará hasta que se para.

void QTimer::stop ()

Para el contador de tiempo.

QWidget.

La clase QWidget es la clase base de todos los objetos de la interfaz de usuario. Recibe eventos del ratón, del teclado y otros acontecimientos del sistema de ventanas, y pinta una representación de sí mismo en la pantalla.

bool QWidget::close () [slot]

Cierra el widget. Retorna *TRUE* si el widget fue cerrado; de lo contrario retorna *FALSE*. Primero el widget envía un *QCloseEvent*. Se oculta el widget si acepta el evento de cierre.