UNIVERSIDAD AUTÓNOMA METROPOLITANA
UNIDAD AZCAPOTZALCO

DIVISIÓN DE CIENCIAS BÁSICAS E INGENIERÍA

DMR: DETECTOR DE MOVIMIENTO REMOTO PARA UN SISTEMA DE VIGILANCIA MULTISITIO.

PROYECTO TERMINAL:

VÁZQUEZ GARCÍA MIGUEL ÁNGEL
MATRÍCULA: 206300159

INGENIERÍA EN COMPUTACIÓN

ASESOR DEL PROYECTO:

DR. ARMANDO JIMÉNEZ FLORES

MÉXICO, D.F. JULIO, 2012

RESUMEN

El presente proyecto denominado DMR, tiene como objetivo primordial la detección de movimiento en un ambiente *remoto* y multisitio. Este sistema está constituido por dos tipos de aplicaciones que se ejecutan en diferentes lugares, se complementan y se comunican mediante la Internet. Estas aplicaciones trabajan en conjunto y brindan al usuario un sistema sencillo de manejar.

Un tipo de aplicación, llamada Servidor, se encarga de gestionar las conexiones de los clientes ubicados en diferentes sitios y actúa como interfaz gráfica de usuario para recibir información sobre el estado del sistema.

El segundo tipo de aplicación del sistema DMR, corresponde al Cliente. Esta aplicación se sitúa y ejecuta en diferentes sitios que dispongan de una *cámara web*. Al igual que el Servidor, el Cliente maneja una interfaz gráfica para el usuario, en donde se pueden configurar algunos modos de funcionamiento.

El Cliente tiene la función de detectar si ocurre un movimiento en el lugar vigilado. Cabe mencionar que esta aplicación puede trabajar de manera autónoma, es decir, se puede ejecutar sin necesidad de estar conectado al Servidor. En este caso el Cliente realizará la misma función de detección independientemente del Servidor.

En caso de detectar algún tipo de movimiento, el Cliente notifica al Servidor sobre este evento y le envía las imágenes de su escena. Asimismo, las imágenes del Cliente son capturadas localmente durante el intervalo de tiempo en que persista algún movimiento.

Del lado del Servidor se puede visualizar la escena de los clientes en cualquier momento. El Servidor puede desplegar las imágenes recibidas por parte del Cliente en el momento en que el usuario lo solicite. De igual manera, desde el Servidor se pueden deshabilitar cada una de las conexiones con los clientes y desplegar una lista de clientes activos.

A mis padres, que siempre me han dado su apoyo incondicional, por todo su trabajo y dedicación para darme una formación académica y sobre todo humanista y espiritual.

ÍNDICE GENERAL

1 INTRODUCCIÓN **1**

1.1 ANTECEDENTES **2**

1.2 JUSTIFICACIÓN **2**

1.3 OBJETIVOS **3**
1.3.1 OBJETIVO GENERAL 3
1.3.2 OBJETIVOS PARTICULARES 3

1.4 ORGANIZACIÓN DEL DOCUMENTO **4**

2 FUNDAMENTOSTEORICOS Y HERRAMIENTAS UTILIZADAS **5**

2.1 FUNDAMENTOS TEORICOS **5**
2.1.1 ANTECEDENTES 5
2.1.2 TIPOS DE SENSORES 5

2.2 HERRAMIENTA OpenCV **6**
2.2.1 ANTECEDENTES 6
2.2.2 ESTRUCTURAS Y CARACTERISTICAS DE OpenCV 7

2.3 HERRAMIENTA QT **9**
2.3.1 ANTECEDENTES 9
2.3.2 CARACTERÍSTICAS Y APLICACIONES 9

3 DESARROLLO DEL SISTEMA DMR **11**

3.1 DESCRIPCIÓN DEL SISTEMA DMR **11**
3.1.1FUNCIONAMIENTO 11
3.1.2ESTRUCTURA INTERNA 11
3.1.3ESPECIFICACIONES TÉCNICAS 13

3.2 DISEÑO DEL SISTEMA DMR **15**
3.2.1SERVIDOR DMR 15
3.2.1.1 INTERFAZ PRINCIPAL 15
3.2.1.2 RECEPCIÓN 16
3.2.1.3 MUESTRA 17
3.2.2 CLIENTE DMR 19
3.2.2.1 INTERFAZ DE CLIENTE 19
3.2.2.2 CAPTURA 19
3.2.2.3 CONVERSIÓN 21
3.2.2.4 RECONOCIMIENTO 22

3.2.2.5 GRABACIÓN 24
3.2.2.6 ENVÍO 26

4 PRUEBAS Y RESULTADOS 29

5 CONCLUSIONES 41

BIBLIOGRAFÍA 42

ANEXOS 43

ANEXO A. CÓDIGO FUENTE DEL SISTEMA DMR 43
Servidor DMR 43
Cliente DMR 63

ANEXO B. MANUAL DE INSTALACIÓN Y EJECUCIÓN 89
Instalar el ***Compilador* para C++** y las dependencias útiles. 89
Instalar **QTCreator.** 90
Instalar biblioteca **OpenCV**. 90
Ejecución 92

ANEXO C. MANUAL DE USUARIO 98

GLOSARIO 106

CAPÍTULO 1

INTRODUCCIÓN

Actualmente, el problema de la inseguridad se ha estado incrementando de manera alarmante. Por tal motivo, es importante utilizar las nuevas tecnologías de la información y las comunicaciones para desarrollar sistemas de vigilancia automáticos y seguros capaces de ofrecer un razonable nivel de protección a las personas y a los bienes materiales contra intrusos o delincuentes en las empresas, comercios y casas habitación. El aumento del potencial de computo de los sistemas, y al mismo tiempo su reducción de precios, facilita cada vez más la implementación de herramientas que pueden actuar como atentos y permanentes vigilantes durante las 24 horas de cada día.

Los *sensores* juegan un papel importante en nuestras vidas, y en ocasiones nuestra integridad física puede depender de ellos. Un ejemplo de esto es el *sensor* de detección de fuego, que podría estar instalado en nuestro ambiente laboral y evitarnos accidentes graves. Los *sensores* han existido desde siempre, y los seres humanos poseemos varios de ellos desde que nacemos. Gracias a nuestros *sensores* nos es posible distinguir diferentes tipos de sensaciones originadas por variaciones de temperatura, luz, sonido, entre otras.

Un detector de movimiento es un dispositivo electrónico compuesto por *sensores* que revelan un movimiento físico. Este tipo de detectores, generalmente se utilizan en sistemas de seguridad o en circuitos cerrados. La mayoría de los circuitos que detectan movimiento, utilizan *sensores* que habitualmente requieren de operadores especializados para hacer los ajustes o calibraciones correspondientes.

Existen varios tipos de *sensores* que pueden ser utilizados en la detección de movimiento de objetos o personas. Como ejemplos se encuentran los basados en señales de ultrasonido que aprovechan el efecto Doppler, los basados en radio frecuencia que utilizan efectos capacitivos, *sensores* de presión que censan la fuerza de gravedad ejercida sobre un piso, emisores-detectores laser que se apoyan en la interrupción de haz de luz y, actualmente, los basados en imágenes que detectan cambios en las imágenes.

El DMR, tiene las cualidades de ser un sistema de bajo costo y baja complejidad de operación, además de que no sólo nos permite detectar movimiento sino que también es capaz de capturar imágenes y video en tiempo real, para posteriormente ser analizadas.

1.1 ANTECEDENTES

Algunos antecedentes relacionados con la temática del presente proyecto se citan a continuación y han sido realizados dentro de la Universidad Autónoma Metropolitana Azcapotzalco:

Sistema de vigilancia remota programado con C++ y OpenCV [1]. En este proyecto se implementa un sistema de vigilancia remota, que es capaz de detectar movimiento en una zona determinada. El sistema captura, almacena y envía imágenes a una dirección de *e-mail* específica.

Sistema de aprendizaje del alfabeto dactilógico mediante procesamiento de imágenes utilizando software libre [2]. Este sistema reconoce y procesa ciertas posiciones de la mano para poder generar un alfabeto útil para personas sordomudas.

Clasificador de objetos en banda infinita por medio de procesamiento digital de imágenes [3]. Este proyecto trata una Identificación de una serie de objetos los cuales pasan frente a una cámara por medio de una banda infinita, simulando un proceso de producción.

En cuanto a proyectos externos relacionados con nuestro sistema DMR podemos mencionar los siguientes proyectos:

Webcam Surveillance Standard [4]. Un programa de vigilancia por video en donde el usuario puede monitorear una zona específica. Posee un registro de la hora en las imágenes que permite informar a los usuarios detalles de los eventos en el momento que sucedieron.

Move Action [5]. Aplicación para vigilancia utilizando una cámara *web*. Cuando esta aplicación detecta algún movimiento puede realizar diversas acciones como bloquear el equipo, reproducir sonidos, ejecutar programas, tomar fotografías, o enviarlas a una dirección de correo electrónico.

1.2 JUSTIFICACIÓN

Las computadoras se han convertido en una herramienta indispensable en nuestra vida cotidiana, por lo que existen diversos complementos para una computadora capaces de detectar movimiento.

Actualmente, existen sistemas de vigilancia en el mercado que nos permiten monitorear los eventos que se efectúan en algún lugar en específico. Desafortunadamente, estos sistemas exigen recursos adicionales, como una capacidad de almacenamiento suficientemente grande, el uso de tecnología cara o simplemente un alto consumo de energía. Tenemos el caso típico de un sistema que graba video en todo momento sin importar si hay o no movimiento para hacer una revisión manual posterior de los eventos captados. En este tipo de sistemas los recursos son desperdiciados o se requiere de conocimientos adicionales que el usuario tal vez no tenga para configurarlos y manipularlos.

Lo que se pretende con este proyecto, es obtener un sistema que permita a los usuarios vigilar lugares en los cuales se necesite algún tipo de supervisión, mientras que el usuario está ausente. La importancia del proyecto radica en que la aplicación del DMR ayudará a los usuarios a monitorear zonas que requieran ser vigiladas de manera sencilla, con la ventaja de ser un detector de movimiento accesible y capaz de alertar al usuario remotamente, sólo con la ayuda de *cámaras web* ordinarias y computadoras.

La licenciatura en Ingeniería en Computación ofrece los conocimientos suficientes para realizar este proyecto, pues además de requerirse conocimientos de programación de interfaces gráficas de usuario y tratamiento de imágenes, también se requiere un alto grado de conocimientos en programación en lenguaje C/C++, Sistemas Operativos y Sistemas Distribuidos, por lo que este proyecto aborda un problema que requiere de un ingeniero en computación para ser resuelto.

1.3 OBJETIVOS

1.3.1 OBJETIVO GENERAL

Desarrollar un sistema computacional (DMR), para detectar movimientos en sitios *remotos* y generar señales de aviso, además de activar y monitorear en tiempo real la grabación de imágenes durante el tiempo que dure el movimiento.

1.3.2 OBJETIVOS PARTICULARES

- Implementar un módulo para capturar imágenes de una cámara *web*.
- Implementar un módulo para convertirlas imágenes capturadas en una escala de grises.
- Desarrollar un módulo para detectar movimiento.
- Implementar un módulo para grabar video desde una cámar*a web*.
- Implementar un módulo capaz de enviar una imagen y un mensaje textual de aviso hacia una computadora remota, bajo un modelo de comunicación cliente-servidor.
- Desarrollar una interfaz gráfica de usuario para la máquina servidor del sistema DMR, objeto de este proyecto terminal.

1.4 ORGANIZACIÓN DEL DOCUMENTO

Este documento se compone de cinco capítulos; a continuación se describe brevemente el contenido de cada uno de ellos:

Capítulo 2. FUNDAMENTOS TEÓRICOS Y HERRAMIENTAS UTILIZADAS: Contiene los principios teóricos en los que se apoya este proyecto, incluyendo el principio de funcionamiento de los detectores de movimiento. Se presenta la herramientade visión por computadora OpenCV, su estructura y sus características. En este capítulo también se da a conocer el uso, la importancia y las características de QT para el desarrollo de las interfaces gráficas del sistema.

Capítulo 3. DESARROLLO DEL SISTEMA DMR: Dentro de este capítulo se da a conocer el funcionamiento y la estructura del sistema, así como la descripción y especificación técnica de cada uno de sus bloques. Se aborda el diseño de los módulos y se explica su código correspondiente.

Capítulo 4. PRUEBAS Y RESULTADOS: En este capítulo se muestran los resultados obtenidos después de haber realizado las pruebas de detección de movimiento en sitios *remotos*.

Capítulo 5. CONCLUSIONES: En este último capítulo se habla de los objetivos planteados y los objetivos alcanzados tras el desarrollo del presente proyecto, así como los trabajos futuros que pueden darle continuidad al proyecto.

Anexo A: Código fuente del sistema DMR. En este apartado se incluye el código fuente de las aplicaciones Cliente y Servidor.

Anexo B: Manual de instalación y ejecución. En este apartado se guía al usuario para poder instalar y ejecutar el sistema DMR.

Anexo C: Manual de usuario. En este apartado se guía al usuario para que pueda utilizar de manera correcta el sistema DMR.

CAPÍTULO 2

FUNDAMENTOSTEORICOS Y HERRAMIENTAS UTILIZADAS

2.1 FUNDAMENTOS TEORICOS

2.1.1 ANTECEDENTES

Los seres humanos contamos con varios tipos de *sensores* naturales, que nos ayudan de alguna manera a detectar las condiciones en las que vivimos. Estos detectores nos son de gran utilidad en la vida diaria, debido a que gracias a ellos nosotros podemos tomar decisiones para nuestro bienestar.

Los *sensores* son elementos físicos que pertenecen a un tipo de dispositivo llamado *transductor*. Los *transductores* son dispositivos capaces de transformar una magnitud física a una señal eléctrica. Los *sensores* captan las señales necesarias para conocer el estado del proceso y decidir su desarrollo futuro. Detectan posición, presión, temperatura, velocidad y aceleración entre otras variables. En la **Figura1** se puede apreciar la relación del ser humano y los *sensores*.

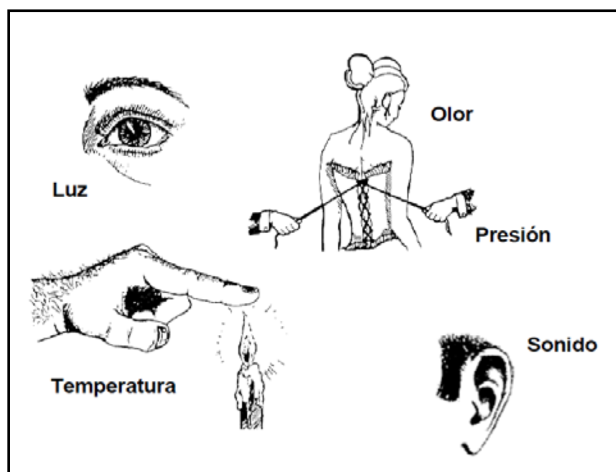


Figura1. Relación del hombre con los *sensores*.

2.1.2 TIPOS DE SENSORES

Existen diferentes tipos de *sensores* que en la vida diaria nos son de gran utilidad para realizar nuestras actividades, estos dispositivos poco a poco han ido evolucionando para alcanzar un mejor desempeño. A continuación se describen algunos tipos de *sensores* más comunes.

Detectores de ultrasonidos: los detectores de ultrasonidos resuelven los problemas de detección de objetos de prácticamente cualquier material. Trabajan en ambientes secos y polvorientos. Normalmente se usan para control de presencia/ausencia, distancia o rastreo.

Productos infrarrojos: son componentes optoelectrónicos fiables y económicos que integran los principios ópticos y la electrónica de semiconductores.

Sensores de corriente: los *sensores* de corriente son aquellos dispositivos que monitorizan corriente continua o alterna. Entre ellos se encuentran los *sensores* de corriente lineales ajustables, de balance nulo, digitales y lineales.

Sensores de humedad: los *sensores* de humedad relativa están compuestos por circuitos integrados que proporcionan una señal acondicionada. En su interior se encuentra un elemento sensible capacitivo en base de polímeros que interactúa con *electrodos* de platino.

Sensores de presión y fuerza: este tipo de *sensores* generalmente son pequeños, fiables y de bajo costo. Tienen una excelente repetitividad y una alta precisión y fiabilidad en condiciones ambientales variables. Algunos de estos dispositivos son combinados con *microcontroladores* que proporcionan una alta precisión y capacidad de comunicación digital con una PC.

Sensores de temperatura: estos dispositivos consisten en una fina película de resistencia variable con la temperatura y están calibrados por láser para una mayor precisión e intercambiabilidad.

En el contexto de este proyecto, podríamos hablar de otro tipo de *sensor* basado en software cuya función básica es generar un mensaje como consecuencia de la detección de movimiento. De esta forma podemos ver al subsistema cliente, desde un alto nivel, como un *sensor* sofisticado que se encarga de enviar un mensaje de aviso al subsistema servidor cada vez que se detecta un movimiento en el subsistema cliente.

2.2 HERRAMIENTA OpenCV

2.2.1 ANTECEDENTES

OpenCV [6] es una *biblioteca* libre para visión artificial desarrollada por Intel. Se ha utilizado en infinidad de aplicaciones desde su aparición en el mes de enero de 1999. Su publicación se da bajo licencia *BSD*, que permite que sea usada libremente para propósitos comerciales y de investigación con las condiciones en ella expresadas.

OpenCV es una herramienta *multiplataforma* que dispone de versiones para GNU/Linux, Mac OS X y Windows. Esta *biblioteca* contiene más de 500 funciones que abarcan una gran gama de áreas en el proceso de visión, como son:

- Captura en tiempo real.
- Importación de archivos de video.
- El tratamiento básico de imágenes.
- Detección de objetos.

2.2.2 ESTRUCTURAS Y CARACTERISTICAS DE OpenCV

La *biblioteca* OpenCV está enfocada a la visión por computadora en tiempo real. Entre sus diversas áreas de aplicación destaca la interacción hombre-máquina, la segmentación y reconocimiento de objetos o el reconocimiento de gestos entre otras. Como se puede observar en la **Figura 2**, OpenCV proporciona numerosos elementos de alto y bajo nivel.

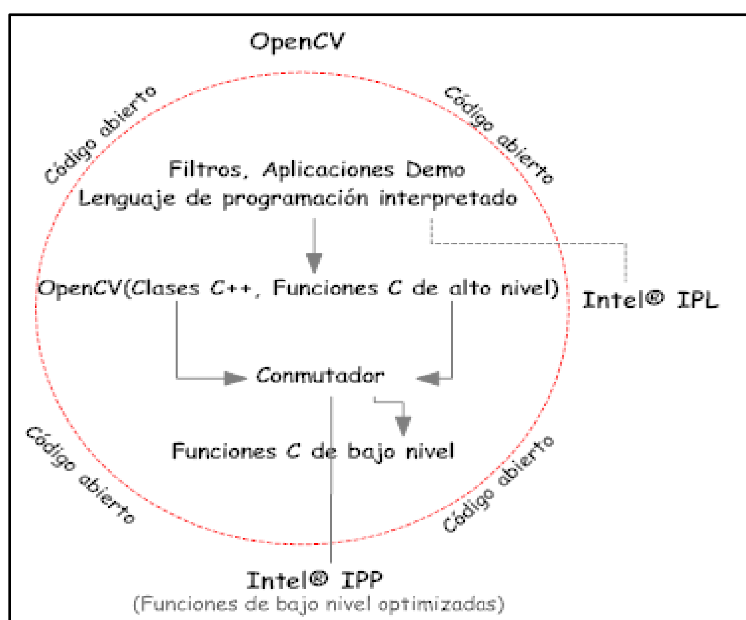


Figura 2. Posicionamiento de la biblioteca en el entorno de programación

Para explicar de forma más detallada el funcionamiento de la biblioteca podemos observar la **Figura 3**, en la cual se observan las partes en las que está dividida.

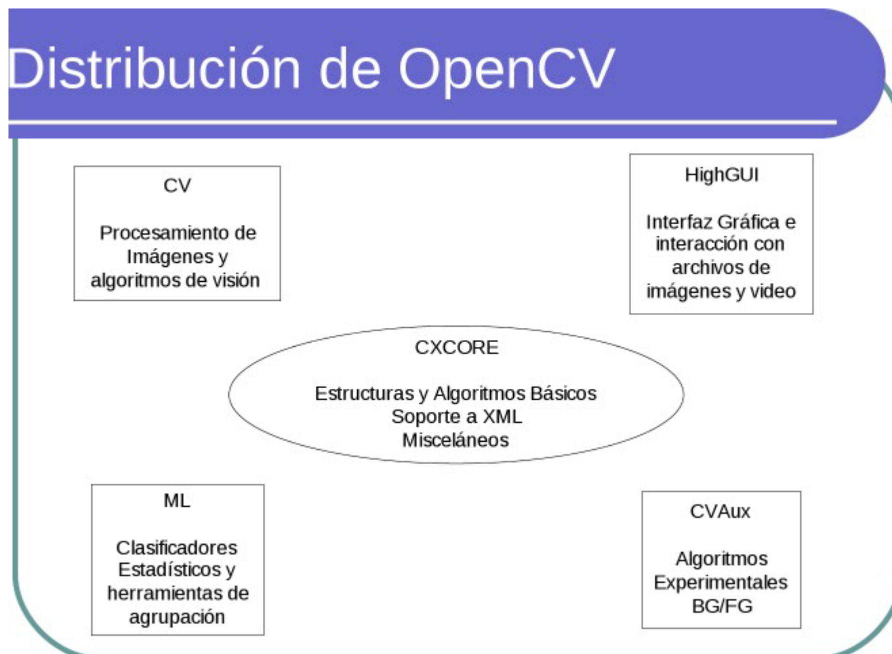


Figura 3. Estructura de OpenCV.

Como se puede apreciar en la figura anterior, OpenCV cuenta con cinco módulos, teniendo cada uno de ellos su funcionamiento individual, como a continuación se describen:

- El módulo **CV** es el encargado de proveer las funciones de operaciones matriciales, histogramas, funciones especiales o calibración, entre otras.
- Dentro del módulo **CVAUX** están las funciones en fase de pruebas o experimentales que aún no han sido declaradas estables.
- **HIGHGUI** este módulo permite la lectura y escritura de imágenes en diversos formatos y lacaptura de video en *stream* de cámaras. Proporciona la creación de ventanas para mostrar imágenes en ellas, y métodos para interactuar condichas ventanas como son los *trackbars* y capturando de eventos del teclado y ratón.
- **CXCORE** donde se ubican lasestructuras y algoritmos básicos que usan las demás funciones como suma, media, operaciones binarias entre otras.
- En el módulo **ML** se cuenta con algoritmos deaprendizaje y clasificadores.

2.3 HERRAMIENTA QT

2.3.1 ANTECEDENTES

Qt [7] es una biblioteca *multiplataforma* usada para crear aplicaciones con interfaz gráfica de usuario. La herramienta Qt fue desarrollada inicialmente por Haavard Nord y Eirik Chambe-Eng. Inicialmente apareció como biblioteca desarrollada por Trolltech, una compañía de software de computadoras de Oslo, Noruega, en 1992 siguiendo un desarrollo basado en el código abierto, pero no completamente libre.

Una de sus aplicaciones destacadas fue su participación en el desarrollo del escritorio *KDE*, que tuvo un gran éxito y rápida expansión, puesto que se convertiría en uno de los escritorios más populares de GNU/Linux.

En 1998 los desarrolladores de KDE junto con Trolltech establecieron la KDE Free Qt Foundation, que establecía que si Trolltech dejaba de desarrollar la versión gratuita y semi-libre de Qt la propia Fundación podría liberar la última versión publicada de la biblioteca Qt bajo una licencia tipo *BSD*.

Al llegar a su versión 2.0 se cambió a la licencia *QPL*, considerada de código abierto. Este cambio intentaba calmar las críticas a Qt y KDE que fundamentaban que no era software libre. Desafortunadamente, QPL no era compatible con la licencia *GPL* que usaba KDE, por lo que en algún momento se afirmaban que se estaba violando la licencia GPL.

Actualmente esta biblioteca cuenta con un sistema de triple licencia: GPL v2/v3 para el desarrollo de software de código abierto y software libre; la licencia de pago QPL que es usada para el desarrollo de aplicaciones comerciales; y a partir de la versión 4.5, una licencia gratuita pensada para aplicaciones comerciales.

2.3.2 CARACTERÍSTICAS Y APLICACIONES

El *SDK* de Qt se mezcla con herramientas diseñadas para simplificar la creación de aplicaciones para teléfonos *Symbian* y plataformas de escritorio, como Microsoft de Windows, Mac OS X y Linux. Las aplicaciones que ofrece esta biblioteca son las siguientes:

- Qt Creator IDE: entorno de desarrollo integrado de gran alcance *multiplataforma*, incluye herramientas de diseño de interfaz de usuario y depuración.
- Herramientas y cadenas de herramientas: simulador, compiladores locales y remotos, soporte de internacionalización entre otros.

Una de las ventajas importantes de esta biblioteca es que se puede reutilizar el código de manera eficiente para atacar múltiples plataformas con una base de código. Actualmente cuenta con una *UI* para todo tipo de aplicación, manejando los siguientes lenguajes:

- C++
- QML/Java Script
- HTML/Java Script/CSS

Otra ventaja importante es que sus modelos corren de acuerdo con el sistema operativo, se comporta como lenguaje descriptivo y se pueden utilizar *elementos Web*.

CAPÍTULO 3

DESARROLLO DEL SISTEMA DMR

3.1 DESCRIPCIÓN DEL SISTEMA DMR

3.1.1FUNCIONAMIENTO

El sistema de Detección de movimiento que se propone, detecta cambios a través del reconocimiento de escenarios e imágenes adquiridas directamente de una *cámara web*, e informa de un posible movimiento a una interfaz gráfica de usuario ubicada en otra computadora en un sitio *remoto*.

Como primer paso, se obtendrá la imagen captada por una cámara web, localizada en una computadora cliente. Esta imagen se convertirá a escala de grises para su tratamiento y para su reconocimiento. En caso de haber detectado un cambio, se comenzarán a captar las imágenes a partir de ese instante; la aplicación dejará de grabar en cuanto no haya cambio. Finalmente, se enviará un aviso *remoto* hacia la interfaz gráfica de usuario, ubicada en el servidor, informando la detección de movimiento, en caso de haberlo, y se mostrará la imagen capturada en el momento del cambio.

3.1.2ESTRUCTURA INTERNA

El DMR, se divide en dos aplicaciones que corresponden a un modelo cliente-servidor. Dichas aplicaciones se deberán ejecutar en computadoras distintas, ambas con acceso a internet. El programa cliente se puede ejecutar en varias computadoras ubicadas en los sitios a vigilar.

En la **Figura 4** se muestra un diagrama conceptual del sistema DMR, en el cual se observa una entrada por donde se reciben las imágenes captadas por la *cámara web*. Así mismo, se observan dos salidas, una con el video que se almacenará en el sitio local donde se encontrará la cámara, y otra con las imágenes que se envían a una computadora servidor. También se muestran un esquema general de los módulos que forman parte de la aplicación DMR.

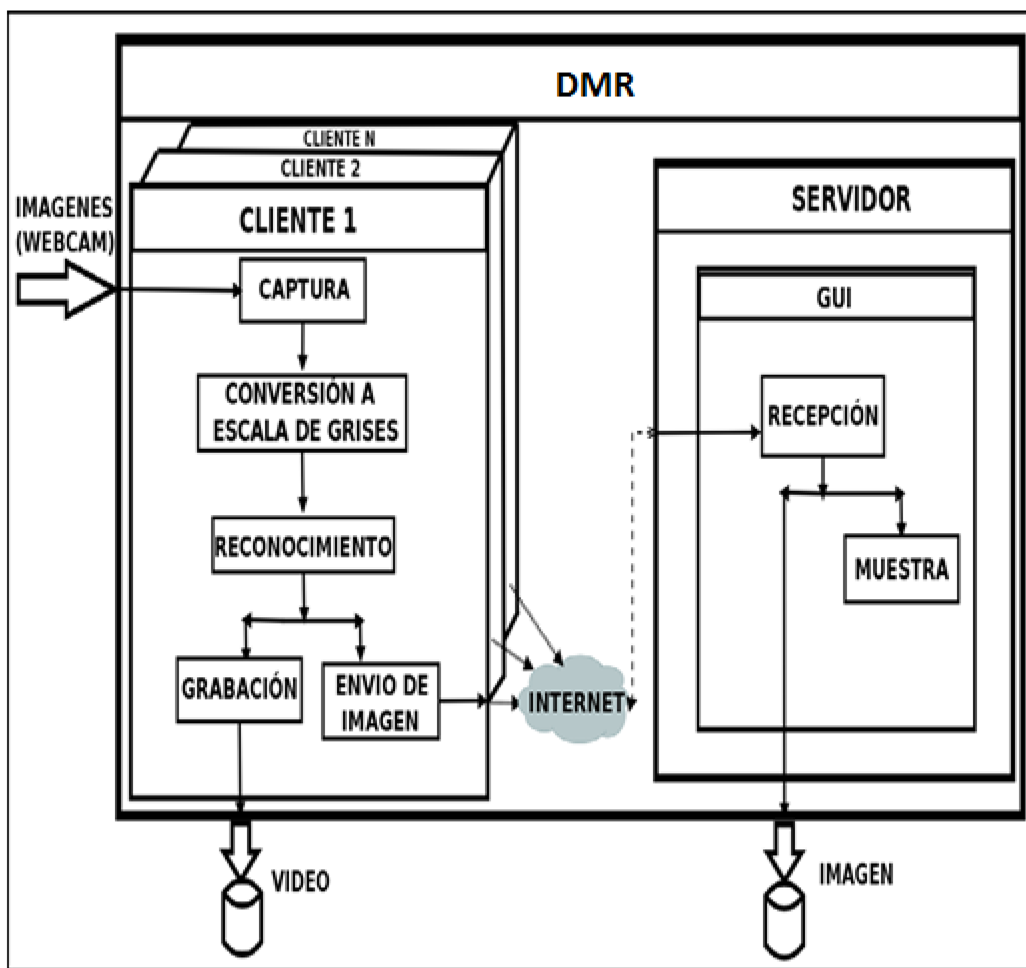


Figura 4. Componentes del sistema DMR.

Al iniciar la ejecución de la aplicación del servidor, el usuario tendrá la posibilidad de configurar los parámetros necesarios para que el DMR se comporte como mejor convenga. Sin embargo, se establecerán los parámetros por defecto, como son: el tiempo de inicio de activación del sistema y la sensibilidad al movimiento.

El módulo CAPTURA, será el encargado de obtener la imagen directamente de la *cámara web*. Posteriormente, se enviará esta imagen al módulo CONVERSIÓN A ESCALA DE GRISES. El módulo RECONOCIMIENTO, será el encargado de detectar el movimiento. En caso de haber detectado un cambio, dará una orden al módulo GRABACIÓN, y sólo si se ha establecido una comunicación con el servidor, enviará una orden al módulo ENVÍO DE IMAGEN. Además, determinará durante qué tiempo se grabará el video. En este caso, hasta que no se perciba movimiento. En la **Figura 5** se puede apreciar el comportamiento de éste módulo.

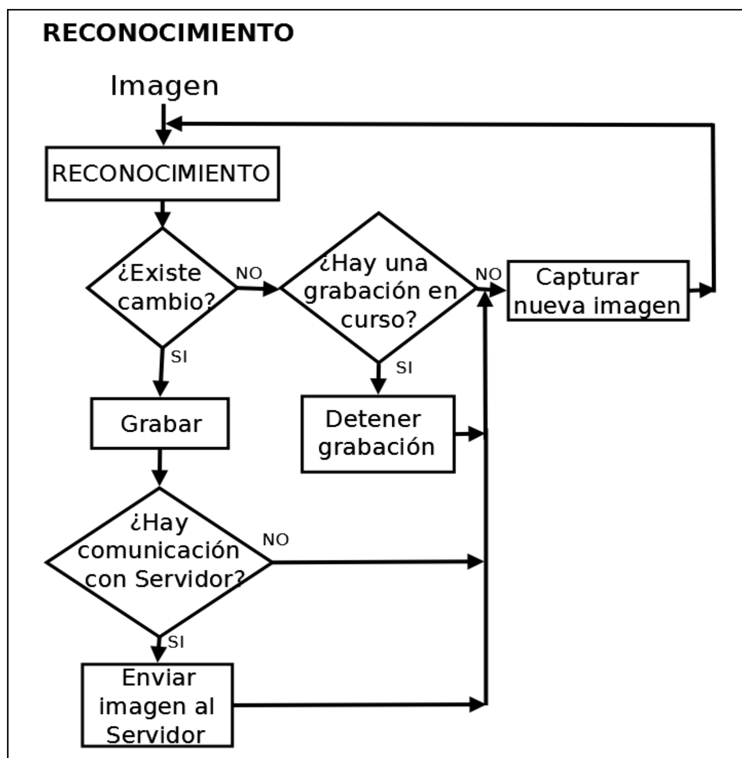


Figura 5. Comportamiento del módulo RECONOCIMIENTO.

En la figura anterior se puede apreciar el comportamiento del módulo RECONOCIMIENTO, perteneciente a la aplicación DMR. El módulo GRABACIÓN, recibirá una orden para inicializar y finalizar la captura de video. La captura se realizará desde la *cámara web* y se almacenará en la máquina cliente. Los módulos ENVÍO DE IMAGEN y RECECEPCIÓN, serán los encargados de enviar y recibir la imagen a través de Internet, respectivamente. Además, el módulo RECEPCIÓN, también se encargará de almacenar la imagen en la máquina servidor. Finalmente, el módulo MUESTRA informará textualmente al usuario, mediante la interfaz gráfica del servidor, que se ha generado un movimiento en el lugar del cliente y mostrará la imagen de la escena recibida.

3.1.3ESPECIFICACIONES TÉCNICAS

El presente proyecto con todos sus módulos, se desarrolla bajo una plataforma Linux, con sistema operativo Ubuntu, versión 10.10, bajo el entorno de desarrollo integrado **QT Creator [8]**. Se utiliza el lenguaje de programación C++ y la biblioteca OpenCV, versión 2.2, para el tratamiento de imágenes. La resolución de las imágenes a procesar, será de 640x480 pixeles, en formato JPEG, mientras que el video de salida será de la misma resolución en formato AVI. No se contempla el manejo de compresión de imágenes.

Para el módulo CAPTURA, se usarán dos variables, una de tipo *booleana*, que determinará el inicio de la captura de imágenes y con ello toda la ejecución de la aplicación, la otra variable será la imagen obtenida de la *cámara web*. Para este módulo se usarán bibliotecas de OpenCV y el lenguaje C++.

En en módulo CONVERSIÓN A ESCALA DE GRISES, se utilizará una variable de entrada correspondiente a la imagen capturada en color, y una variable de salida, perteneciente a la imagen convertida a escala de grises. Para este módulo se usará OpenCV y el lenguaje de programación C++.

Dentro del módulo RECONOCIMIENTO, se usarán tres variables *booleanas*; la primera será utilizada para determinar si hay una comunicación entre servidor y cliente, la segunda determinará si hay cambio con base en el reconocimiento de la escena, y la tercera variable indicará si se está grabando un video actualmente. Durante el desarrollo de este módulo se utilizará el lenguaje de programación C++ y bibliotecas de OpenCV.

El módulo GRABACIÓN, será controlado por una variable *booleana*, que determinará la inicialización y finalización de captura de video. Contendrá una entrada correspondiente al flujo de imágenes capturadas directamente de la *cámara web* y una salida que será un video en formato AVI. En el desarrollo de este módulo se usará el lenguaje C++ y bibliotecas de OpenCV.

Para el módulo ENVÍO DE IMAGEN se usará una variable *booleana*, que servirá para ordenar el envío de la imagen. Además, este módulo manejará una salida correspondiente a la imagen en donde se captó el movimiento. La comunicación entre las aplicaciones cliente y servidor, se hará mediante *sockets*. Cabe mencionar que se implementará un servidor *multihilo* capaz de atender a más de un cliente a la vez. En este módulo se utilizará el lenguaje de programación C++ y la biblioteca QT.

En el módulo RECEPCIÓN, se hará uso de una entrada con la imagen recibida del cliente y una salida, utilizada para enviar la imagen al módulo MUESTRA y almacenarla. Para realizar este módulo se utilizará el lenguaje de programación C++ y la biblioteca QT.

El módulo MUESTRA contiene una entrada con la imagen del cliente, previamente recibida, y se encargará de desplegar dicha imagen mediante la interfaz gráfica de usuario, en el servidor. En la realización de este módulo se utilizarán bibliotecas de QT y el lenguaje C++. La interfaz gráfica de usuario se desarrollará utilizando el *IDE QT Creator* y el lenguaje de programación C++.

3.2 DISEÑO DEL SISTEMA DMR

El diseño del DMR se basó en la construcción de los módulos mostrados en la **Figura 4** del cliente y servidor, para ser incorporados finalmente junto con las interfaces gráficas de usuario en un solo sistema. A continuación se describe brevemente la implementación de los mismos con sus diferentes interfaces creadas.

3.2.1SERVIDOR DMR

3.2.1.1 INTERFAZ PRINCIPAL

La interfaz principal del servidor perteneciente al sistema DMR actúa como intermediaria entre las funcionalidades reales del sistema y el usuario. En la **Figura 6** se puede apreciar dicha ventana.

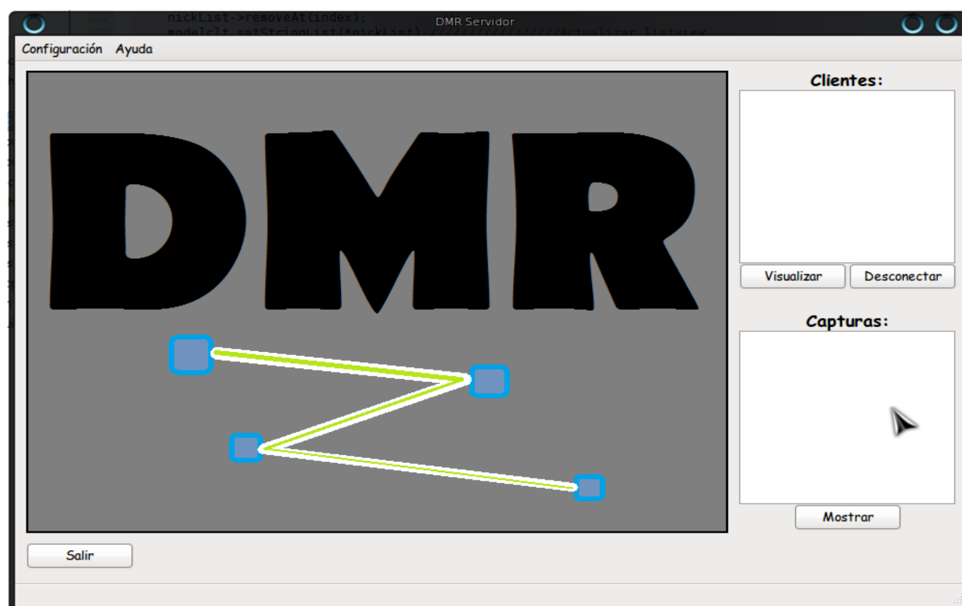


Figura 6. Ventana principal del servidor DMR.

Como se puede observar en la figura anterior, la ventana principal está dividida en secciones permitiéndole al usuario ver las imágenes capturadas y los clientes conectados. El códigoque implementa esta interfaz se puede consultar en los anexos *código fuente Servidor* dentro de mainwindow.h*,* *mainwindow.cpp* y *mainwindow.ui*.

3.2.1.2 RECEPCIÓN

Como su nombre lo indica, este módulo del servidor es el encargado de recibir imágenes o *comandos* provenientes del cliente. La entrada para este bloque son los datos provenientes del *socket*, en específico, la imagen o un comando. La salida es la imagen recibida o el comando para procesar. En el **Figura 7** se puede observar el diagrama de flujo de este módulo.

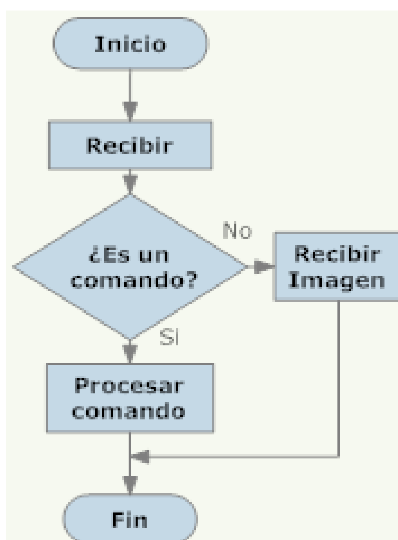


Figura 7. Diagrama de flujo de *Recepción*.

Como se puede ver en la imagen anterior, se toma una decisión para saber qué es lo que se va a recibir, un *comando* ó una imagen. En el caso de ser una imagen, se almacena en un archivo y se emite una orden al bloque *Muestra* para ser mostrada al usuario junto con un aviso textual. En caso de ser un mensaje, se procesa para obtener el *comando*. A continuación se muestra parte del código de este módulo. El código completo de *Recepción* se encuentra dentro de anexos *código fuente Servidor* en el archivo *conexión.cpp.*

Código *Recepción.*

```
20 void conexion::recibir()
21 {
        //Si se trata de una imagen
22      if(this->sup)
23      {
                //Si aún no se sabe el tamaño de la imagen
24              if(f)
25              {
                        //Se crea un nuevo archivo para la imagen
26                      Rimg = new QFile(Cruta + "Cap"+
                        QDateTime::currentDateTime().toString("dd-MM-yy_hh.mm.ss") +

                        nick + ".jpeg");
                        //Se recibe el tamaño de la imagen
```

```
27                      QByteArray datos2 = socket->readAll();
28                      bytes = datos2.toInt();
29                      qDebug()<<"Bytes: "<<bytes;
                        //Se abre el archivo de la imagen
30                      Rimg->open(QIODevice::Append);
31                      t=0;
32                      f=false;
33              }
                //Si ya se sabe el tamaño de la imagen
34              else
35              {
                        //Se reciben los datos de la imagen
36                      QByteArray datos = socket->readAll();
37                      t= t + datos.size();
38                      qDebug()<<"Recibiendo "<<t<<" de "<<bytes <<" bytes";
                        //Se escriben los datos de la imagen al archivo
39                      Rimg->write(datos);

                        //Si ya se recibió el total de datos de la imagen
40                      if(t>=bytes)
41                      {
                                //Se cierra el archivo de la imagen
42                              Rimg->close();
43                              f=true;
                                //Se emite una orden para mostrar la imagen
44                              emit imgcap(Rimg->fileName(),nick);
45                      }
46              }
47      }
        //Si se trata de un comando
48      else
49      {
                //Se leen los datos
50              mensaje.append(socket->readAll());
51              int pos;
                //Se obtiene el comando de los datos recibidos
52              while((pos = mensaje.indexOf("\n\r")) > -1)
53              {
                        //Se procesa el comando
54                      procesar(QString(mensaje.left(pos+2)));
55                      mensaje = mensaje.mid(pos+2);
56              }
57      }
58 }
```

3.2.1.3 MUESTRA

La función del bloque *Muestra* es muy básica, ya que solo se ocupa de mostrar la imagen recibida en la ventana principal y de informar mediante un aviso textual al usuario que se ha generado un movimiento. La entrada para este bloque es la imagen anteriormente recibida. En la **Figura 8** se observa el diagrama de flujo correspondiente al bloque *Muestra.*

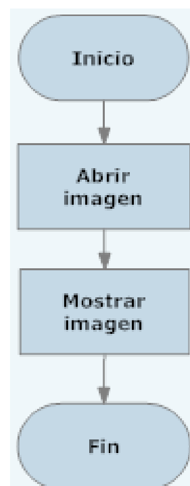


Figura 8. Diagrama de flujo de módulo *Muestra.*

El código fuente completo de este bloque se puede localizar en anexos *código fuente Servidor* en el archivo *mainwindow.cpp*. Las instrucciones específicas para este módulo son las siguientes:

Código *Muestra*.

```
142 void MainWindow::most(QString idirec, QString n)
143 {
144     //Se actualiza el contenido de la interfaz principal
145     modelcap.setFilter(QDir::Files);
146     ui->listCap->setModel(&modelcap);
147     ui->listCap->setViewMode(QListView::IconMode);
148     ui->listCap->setRootIndex(modelcap.index(ruta));

        //Si se dio la orden de visualizar al cliente
150     if (vis)
151     {
                //Se muestra un aviso en la interfaz principal
152             vis = false;
153             statusBar()->showMessage("Mostrando imagen capturada en cliente " + n ,5000);
154     }
        //Si se detectó movimiento
155     else
156     {
                //Se muestra un aviso en la interfaz principal
157             statusBar()->showMessage("Se ha detectado movimiento en cliente " + n ,5000);
158     }
        //Se muestra la imagen recibida en la interfaz principal
159     QImage imgM(idirec);
160     ui->labImg->setPixmap(QPixmap::fromImage(imgM));
161 }
```

3.2.2 CLIENTE DMR

3.2.2.1 INTERFAZ DE CLIENTE

Para un cómodo manejo de la aplicación cliente, se implementó su propia interfaz gráfica, en donde el usuario puede configurar la conexión hacia el servidor o establecer ciertas funciones como la sensibilidad o el retardo de inicio. En la **Figura 9** se puede apreciar dicha ventana.

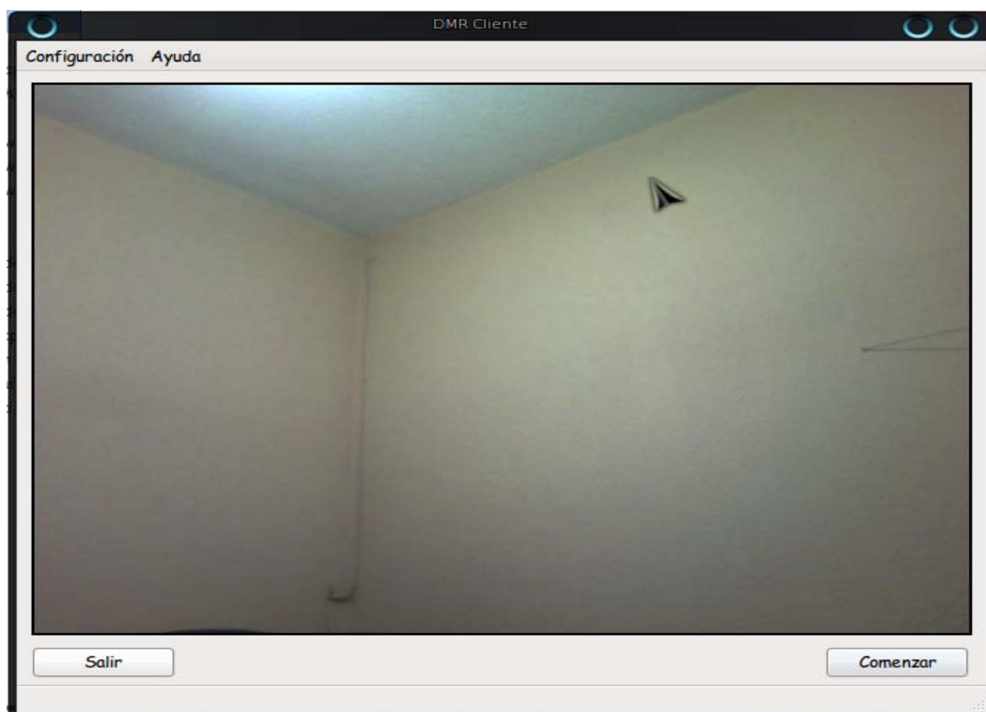


Figura 9. Ventana del cliente DMR.

El código fuente completo de esta interfaz se encuentra en anexos *Código fuente Cliente* dentro de los archivos *mainwindow.h*, *mainwindow.cpp* y *mainwindow.ui*.

3.2.2.2 CAPTURA

Para la aplicación cliente, este bloque es sencillo pero fundamental, ya que con él se obtiene la entrada principal que es la imagen. Este código se puede encontrar en el apartado de anexos *código fuente Cliente* dentro del archivo *mainwindow.cpp.* En la **Figura 10** se observa el diagrama de flujo para este bloque.

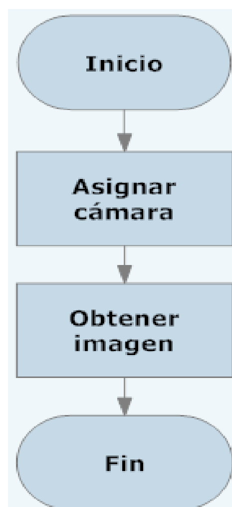


Figura 10. Diagrama de flujo del bloque *Captura*.

Las instrucciones específicas de este módulo son:

Código *Captura.*

```
71 {
        //Se crea una estructura de captura de imagen
72      CvCapture* captura;
        //Se crean las matrices para la imagen y una auxiliar
73      IplImage* img;
74      IplImage* otr;
        //Se crea una referencia a un archivo imagen
75      QImage image;
76
        //Se le asigna la cámara por default a la estructura de captura
77      captura = cvCaptureFromCAM(0);

        //Se obtiene la imagen de la cámara
78      img = cvQueryFrame( captura );

        //Se construye una nueva imagen con el tamaño de la captura
        //en formato RGB de 24 bits.
79      image=QImage (QSize(img->width,img->height),QImage::Format_RGB888);

        //Se crea la estructura de una imagen del tamaño de la captura
        //de profundidad 8 bits y con 3 canales = 24 bits.
80      otr=cvCreateImageHeader(cvSize(image.width(),image.height()),8,3);

        //Se almacenan los datos de la imagen en la auxiliar
81      otr->imageData=(char*)image.bits();

        //Si los datos de la imagen están en el orden adecuado
82      if(img->origin==IPL_ORIGIN_TL)
                //se copian los datos de la imagen a la auxiliar
83              cvCopy(img,otr,0);
        //Si el orden no es el adecuado
84      else
                //Se cambia el orden
85              cvFlip(img,otr,0);
```

```
    //Se convierte la imagen auxiliar de BGR a RGB
86  cvCvtColor(otr,otr,CV_BGR2RGB);

    //Se muestra la imagen capturada en la interfaz principal cliente
87  ui->caplabel->setPixmap(QPixmap::fromImage(image));
88  ui->caplabel->show();
89 }
```

3.2.2.3 CONVERSIÓN

El módulo *conversión* a escala de grises se combinó con el bloque *reconocimiento*. Así, en un mismo bloque se transforma la imagen a grises y se realiza el reconocimiento, ahorrando el tiempo de traslado hacia otro módulo diferente. La entrada para este bloque es una imagen a color y su salida es una imagen en escala de grises. En la **Figura 11** se muestra el diagrama de flujo correspondiente.

Figura 11. Diagrama de flujo de módulo *Conversión.*

El código específico que nos ayuda a la conversión es:

Código *Conversión.*

```
49  //Se crea un variable llamada grisSize para el tamaño de la imagen
50  CvSize grisSize;

    //Se le pasan el ancho y el alto de la imagen original
51  grisSize.width = img->width;
52  grisSize.height = img->height;
53
    //Se crea una estructura tipo imagen con el ancho y alto de la imagen original,
    //se le asigna la misma profundidad pero con un solo canal
54  gris = cvCreateImage( grisSize, img->depth, 1 );

    //Se transforma la imagen a escala de grises
55  cvCvtColor(img,gris,CV_RGB2GRAY);
```

Estas instrucciones están ubicadas en los anexos *código fuente Cliente* dentro del archivo *detectar.cpp.*

3.2.2.4 RECONOCIMIENTO

En este bloque se implementa la detección de movimiento, y es el más extenso de todos. Además de detectar el movimiento, también tiene la tarea de emitir órdenes de envió y grabación a sus módulos correspondientes. La entrada para este bloque es la imagen en escala de grises y las salidas son las señales que genere para la captura y la grabación de imágenes con movimiento. El comportamiento de este bloque se puede observar en el diagrama de flujo de la **Figura 12**.

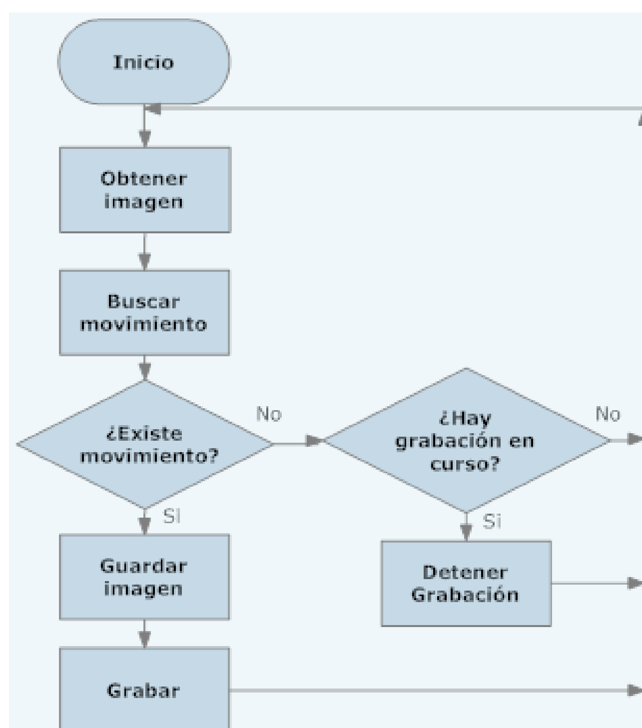


Figura 12. Diagrama de flujo de módulo *Reconocimiento.*

La implementación de este bloque se muestra a continuación, misma que se puede consultar en los anexos *código fuente Cliente* dentro del archivo *detectar.cpp*.

Código *Reconocimiento.*

```
29 void Detectar::run()
30 {
       //Se inicializan las variables booleanas necesarias
31     mov = false;
32     pmax = 0;
33     inicia = true;
34
```

```
        //Se espera un tiempo para iniciar
35      for(int i=retardo; i>0; i--)
36      {
                //Se informa en interfaz principal el tiempo restante
37              emit statusBar("Iniciando en: "+ QString::number(i));
38              sleep(1);
39      }
        //Se informa que el detector se ha inicializado
40      emit statusBar("Detector iniciado");
41
        //Se crean matrices para almacenar la imagen
42      Mat foreground;
43      Mat structure = getStructuringElement(MORPH_RECT, Size(S[sens-1],S[sens-1]));
44
        //Se crean vectores punto para recorrer la estructura de la imagen
45      vector<vector<Point>> contours;
46      vector<Vec4i> hierarchy;
47
48      //////////INICIO CONVERSIÓN//////////
49
50      CvSize grisSize;
51      grisSize.width = img->width;
52      grisSize.height = img->height;
53
54      gris = cvCreateImage( grisSize, img->depth, 1 );
55      cvCvtColor(img,gris,CV_RGB2GRAY);
        //////////FIN CONVERSIÓN/////////
56
        //Se inicia el reconocimiento, detector de movimiento
        //se guarda una referencia a la imagen actual y
        //otra que representa las partes móviles en cada momento
57      CvBGStatModel * bgModel = cvCreateGaussianBGModel(img,NULL);
58
        //Se almacena las propiedades de la imagen como son el ancho y el alto
59      size = cvSize( (int)cvGetCaptureProperty(captura,CV_CAP_PROP_FRAME_WIDTH),
        (int)cvGetCaptureProperty(captura, CV_CAP_PROP_FRAME_HEIGHT));
60
61      while(1)
62      {
                //Se obtiene una nueva imagen de la cámara
63              img = cvQueryFrame(captura);

                //Se actualiza la imagen actual y
                //se almacenan las partes móviles de la imagen
64              cvUpdateBGStatModel(img,bgModel);
65              foreground = bgModel->foreground;
66

                //Se quita el ruido o basura de la imagen
67              erode(foreground,foreground,structure);
68              dilate(foreground,foreground,structure);
69
                //Se busca un cambio en la estructura de movimiento
70              findContours(foreground,contours,hierarchy,
                CV_RETR_CCOMP,CV_CHAIN_APPROX_SIMPLE);
71
                //Si existe un cambio, movimiento
72              if(contours.size()>0)
73              {
                        //////////INICIO GRABACIÓN//////////
74                      pix=contours.size();
75
76                      if(inicia)
77                      {
78                              nom= ruta + "Mov" + QString::number(cap);
79                              writer = cvCreateVideoWriter(nom.toAscii()+".avi",
                                CV_FOURCC('M','J','P','G'), fps ,size, 1);
80                              inicia = false;
81                              mov = true;
```

```
82                          cap++;
83                   }
84                   if(pix > pmax)
85                   {
86                          cvSaveImage(nom.toAscii()+".jpeg",img);
87                          pmax=pix;
88                          //qDebug () << " pmax :"<< pmax; ////////////////////////////
89                   }
90                   cvWriteFrame( writer, img );
                     ///////////FIN GRABACIÓN///////////
91            }

              //Si no existe cambio, movimiento
92            else
93            {
                     //Si anteriormente se detecto movimiento
94                   if(mov)
95                   {

96                   //Se envía la orden para enviar la imagen
97                          emit imagNom(nom);

                     //Se restablecen las variables a su normalidad
98                          inicia = true;
99                          mov=false;
100                         pmax=0;
101                  }
102           }
103    }
104 }
```

3.2.2.5 GRABACIÓN

Dentro de este bloque se implementa la función de grabación de un video, que es simplemente la captura de imágenes continuas. La entrada para este bloque es una secuencia de imágenes de movimiento y la salida es un video en formato AVI. En la **Figura 13** se observa el diagrama de flujo correspondiente.

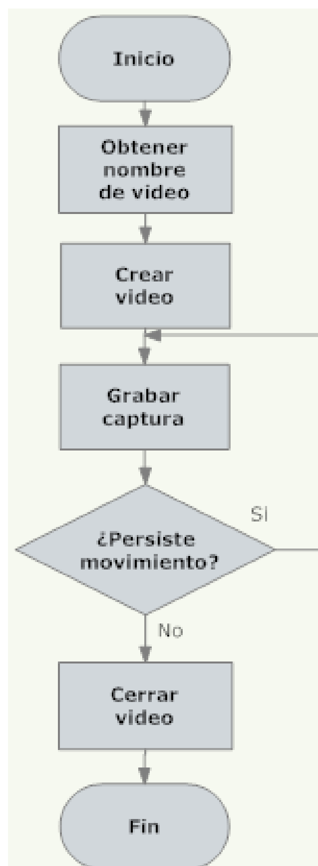


Figura 13. Diagrama de flujo de módulo *Grabación.*

La implementación de esta función se puede observar en el siguiente código, también ubicado dentro de los anexos *código fuente Cliente* en el archivo *detectar.cpp.*

Código *Grabación.*

```
        //Se obtiene el tamaño de los vectores en movimiento
74      pix=contours.size();
75
        //Si no hay un movimiento actualmente
76      if(inicia)
77      {
                //Se crea el nombre del video con fecha actual
78              nom= ruta + "Mov" + QString::number(cap);

                //Se crea el video
79              writer = cvCreateVideoWriter(nom.toAscii()+
".avi",CV_FOURCC('M','J','P','G'), fps ,size, 1);

//Se establecen las variables para comenzar a capturar
80              inicia = false;
81              mov = true;
82              cap++;
83      }

        //Si la imagen actual tiene un mayor cambio
84      if(pix > pmax)
85      {
                //Se guarda la imagen
```

```
86            cvSaveImage(nom.toAscii()+".jpeg",img);
87            pmax=pix;
88        }
          //Se guarda anexa la imagen al video
89        cvWriteFrame( writer, img );
```

3.2.2.6 ENVÍO

Como complemento del bloque *Recepción*, este bloque envía las imágenes capturadas o los posibles *comandos* del sistema a la aplicación Servidor. Estos bloques son de gran importancia para el sistema, ya que facilita la comunicación remota de las dos aplicaciones. La entrada para este bloque es la imagen o el comando a ser enviado mediante el *socket* de comunicación. En la **Figura 14** se observa el diagrama de flujo del módulo *Envío.*

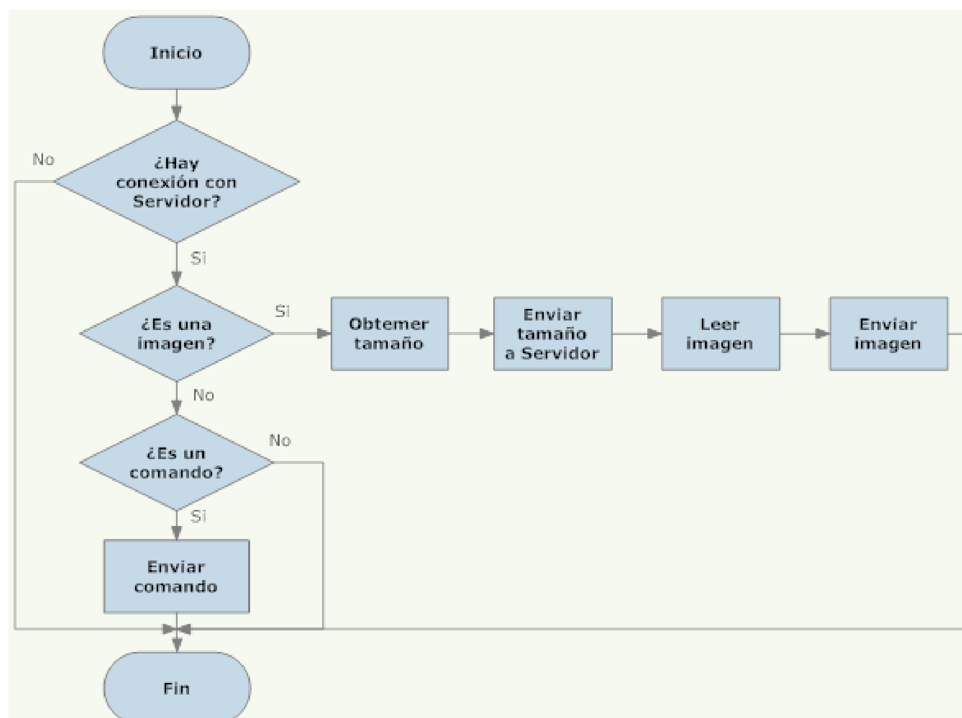


Figura 14. Diagrama de flujo de módulo *Envío.*

La implementación de esta función se encuentra en el archivo *sock.cpp* ubicado en anexos *código fuente Cliente*. El código que implementa este módulo en específico se observa a continuación:

Código *Envío.*

```
73 void sock::sendMessage(QString msg)
74 {
       //Si el socket esta activo y existe una conexión
75     if (socket->isValid())
76     {
```

```
                //Si se tiene la ruta de la imagen
77              if(msg.contains(sruta))
78              {
                        //Se crea una referencia hacia la imagen
79                      img = new QFile(msg);

                        //Se obtiene el tamaño de la imagen
80                      int tam = img->size();
81                      QString tama = QString::number(tam);

                        //Se envía el tamaño al Servidor
82                      socket->write(tama.toAscii());
83                      socket->flush();
84
                        //Se abre la imagen para ser leída
85                      img->open(QIODevice::ReadOnly);

                        //Se envían los datos de la imagen
86                      socket->write(img->readAll());
87                      socket->flush();

                        //Se elimina la referencia de la imagen
88                      delete img;
89              }
                //Si se envía un comando
90              else
91              {
                        //Se envía el comando al Servidor
92                      socket->write(msg.toAscii());
93                      socket->flush();
94              }
95      }
96 }
```

En la **Figura 15** se puede apreciar el comportamiento a grandes rasgos de los módulos de la aplicación Cliente trabajando en conjunto.

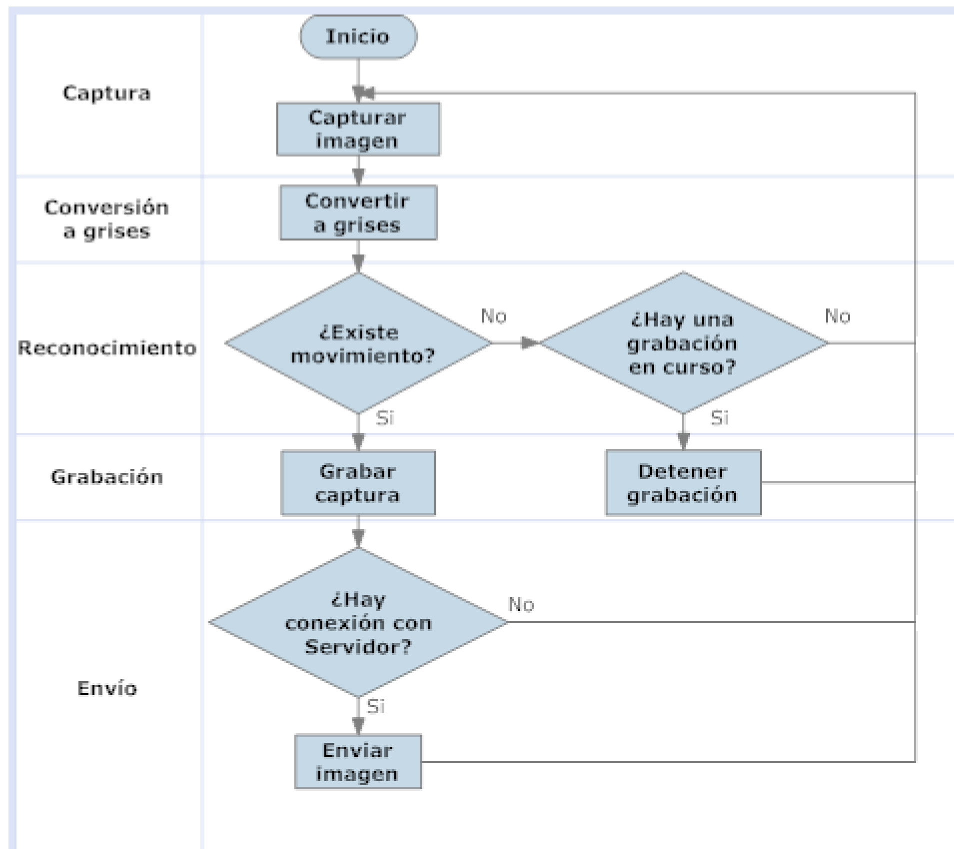


Figura 15. Diagrama de los módulos de Cliente.

CAPÍTULO 4

PRUEBAS Y RESULTADOS

Al terminar el desarrollo del sistema DMR, se realizaron pruebas del funcionamiento. Estas pruebas se llevaron a cabo en dos computadoras portátiles con sistema operativo Linux Ubuntu versión 10.10, en donde se ejecutaron aplicaciones cliente. Mientras que, la aplicación servidor del sistema DMR se ejecutó en una computadora PC con la misma versión de sistema operativo. La especificación de los equipos es:

Cliente 1: Computadora *Laptop* HP DV4, sistema operativo Linux Ubuntu versión 10.10, procesador Intel Core 2 Duo @ 3.3 GHz, memoria RAM 4GB, *cámara web* integrada, tarjeta de red inalámbrica integrada. La conexión a Internet de este cliente se logró mediante un modem Thomson de la compañía Telmex de forma inalámbrica.

Cliente 2: Computadora *Laptop* DELL M1330, sistema operativo Linux Ubuntu versión 10.10, procesador Intel Core 2 Duo @ 3.3 GHz, memoria RAM 2GB, *cámara web* integrada, tarjeta de red inalámbrica integrada. La conexión de este equipo a Internet se logró gracias a un modem USB Banda Ancha de la compañía Telcel.

Servidor: Computadora PC, sistema operativo Linux Ubuntu versión 10.10, procesador AMD Phenom II X6 @ 3.2 GHz, memoria RAM 4GB. La conexión a internet del servidor se llevo a cabo gracias a un modem 2Wire de la compañía Telmex de forma alambrica.

A continuación se muestra el proceso ejecución y conexión de los equipos:

Como primer paso se ejecutó la aplicación servidor en nuestra máquina PC. En la **Imagen 1** se puede observar la ejecución y despliegue de nuestra interfaz principal servidor.

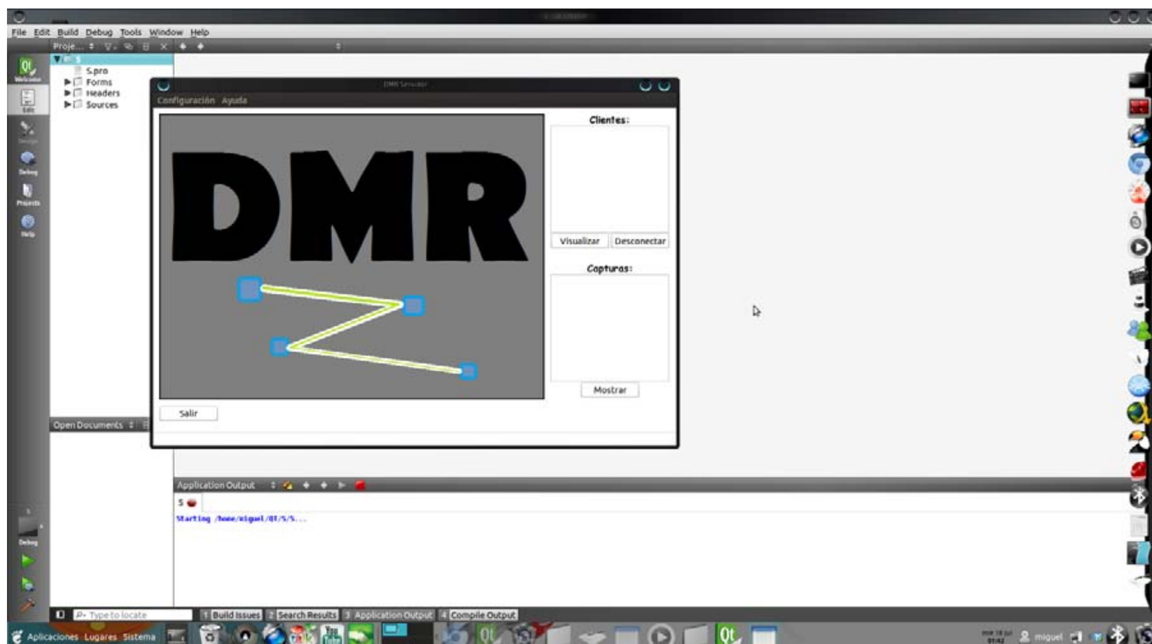


Imagen 1. Despliegue del Servidor en PC.

Para que el servidor pudiera ser visible por los clientes en la Internet, fue necesario establecer una dirección IP pública en el equipo, en la **Imagen 2** se puede observar la dirección IP 201.102.204.163, con la cual se trabajó.

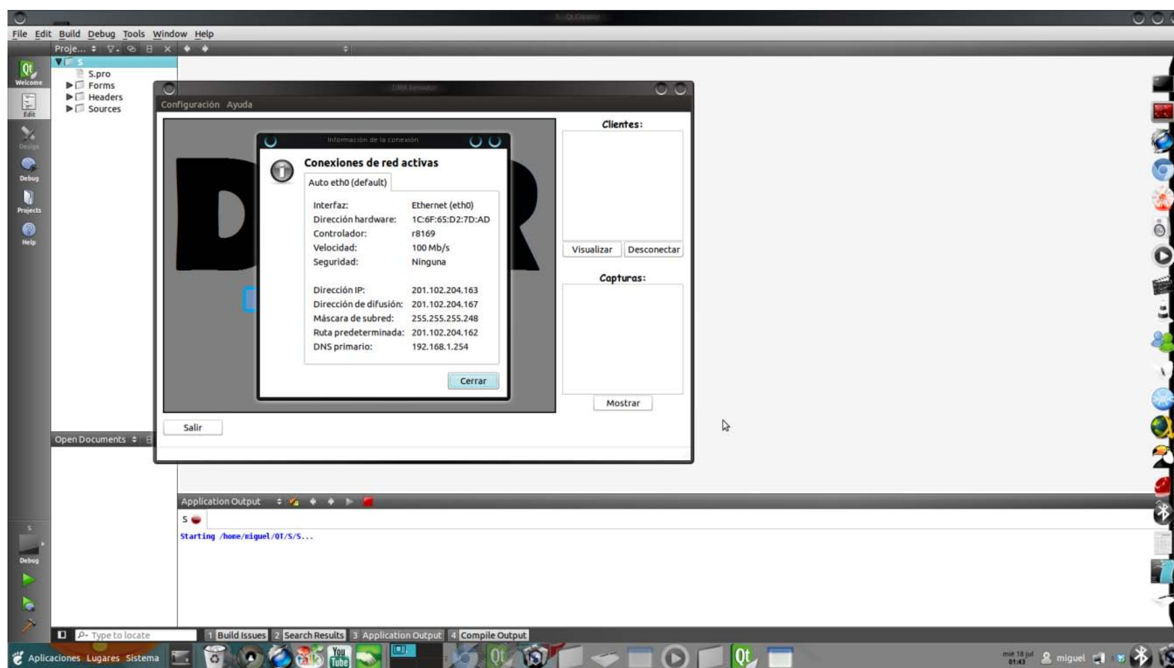


Imagen 2. Dirección IP de PC.

El siguiente paso fue iniciar nuestro servidor, para que estuviera a la espera de los clientes. En la **Imagen 3** se puede observar la configuración de inicio de nuestro servidor, para esto fue necesario especificarle el puerto en el cual escucharía peticiones de los clientes, en este caso le asignamos el puerto 55555 que tiene por defecto.

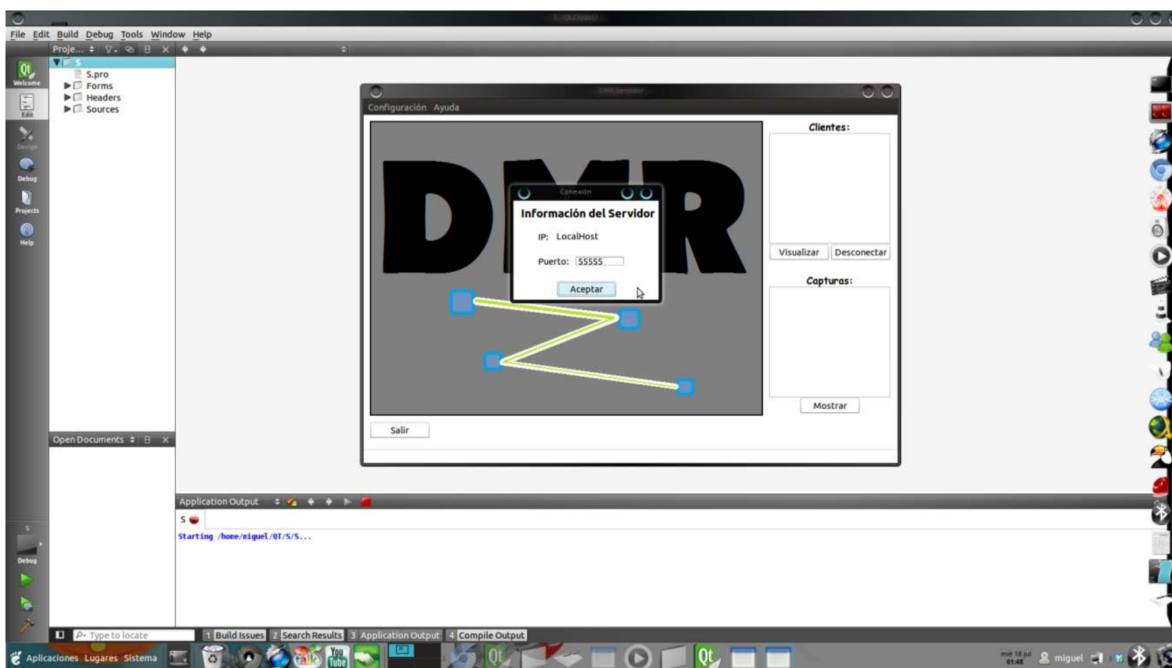


Imagen 3. Configuración de Servidor.

En este paso, específicamente pusimos el *socket* de comunicación del sistema DMR a escuchar conexiones entrantes de los clientes por Internet. En la **Imagen 4** se observa un pequeño aviso en la interfaz al momento de iniciar el servidor de manera exitosa.

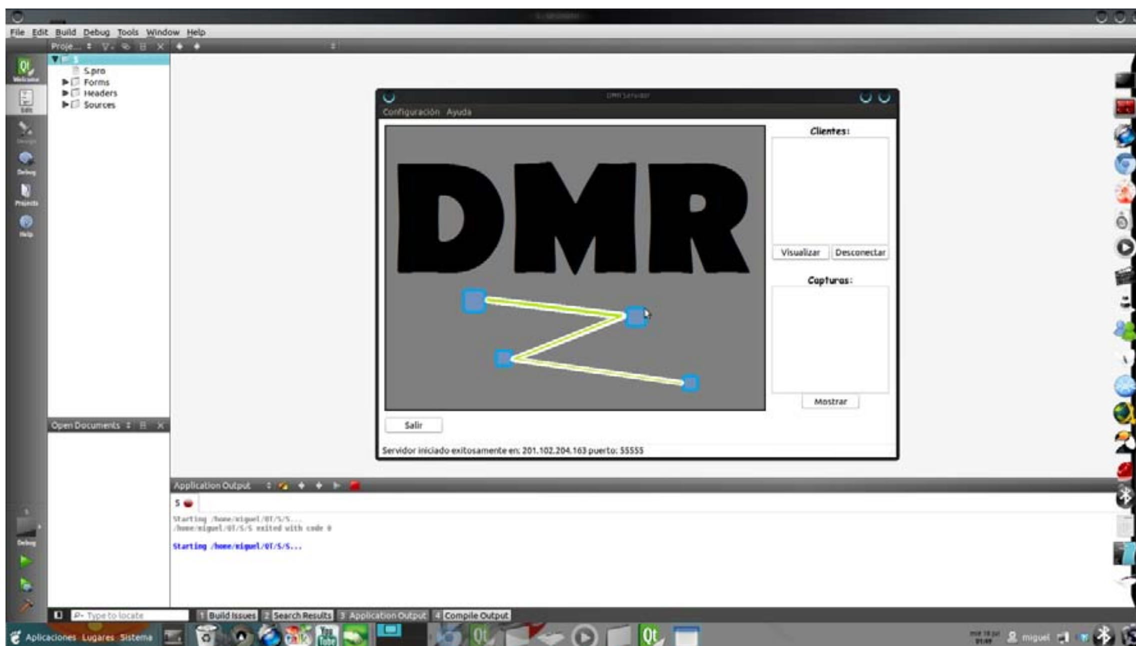


Imagen 4. Inicio de Servidor.

Una vez conectado nuestro servidor, el siguiente paso fue conectar el Cliente 1 a Internet y ejecutar la aplicación Cliente. En la **Imagen 5** se puede ver la conexión establecida a Internet y en la **Imagen 6** la interfaz desplegada del Cliente tras su ejecución.

Imagen 5. Conexión a Internet de Cliente 1.

Imagen 6. Interfaz de aplicación Cliente 1.

Despues de ejecutar el Cliente 1, se estableció la conexión al servidor. Para esto fue necesario configurar la aplicación, especificando la dirección y puerto del servidor, asi como un nombre para identificar desde el servidor a nuestro cliente. En la **Imagen 7** se observa la configuración del Cliente 1.

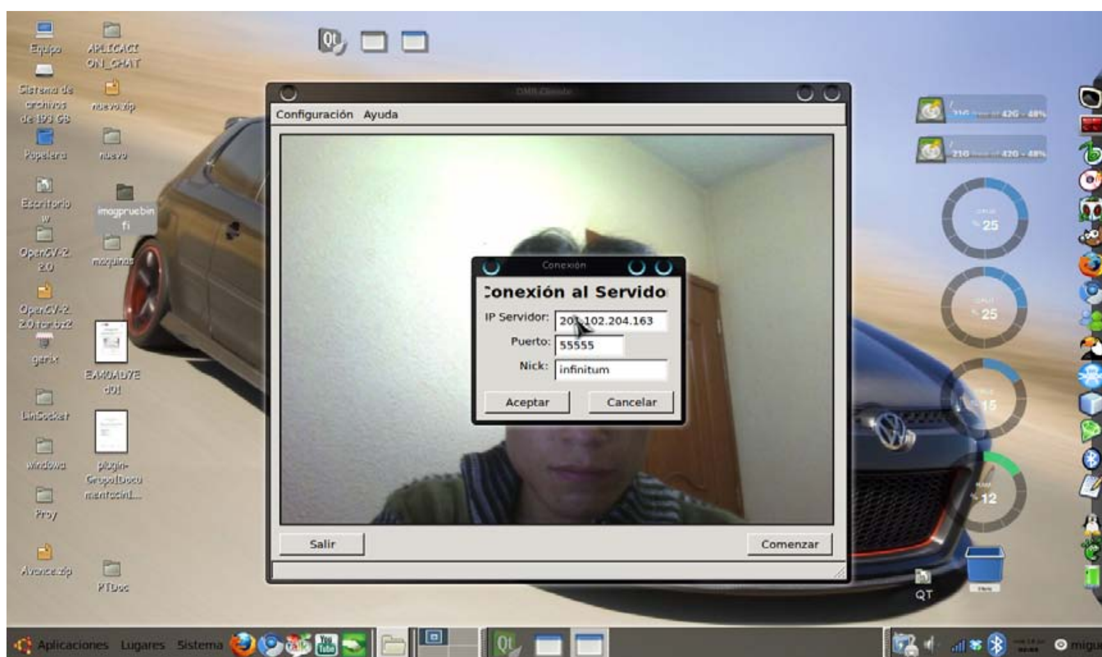


Imagen 7. Configuración de Cliente 1.

Como podemos observar en la **Imagen 8**, la conexión del Cliente 1 se estableció de manera exitosa. En la **Imagen 9** se observa dentro de la interfaz servidor, en el apartado de clientes, el nombre del Cliente 1 recientemente conectado.

Imagen 8. Conexión exitosa de Cliente 1.

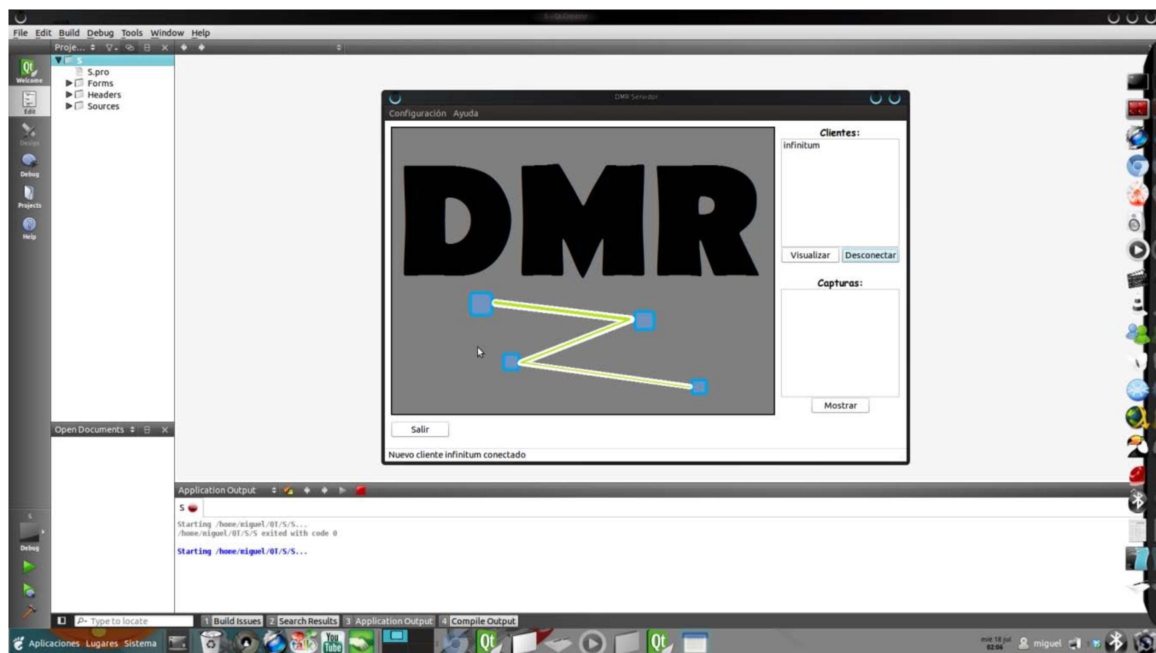


Imagen 9. Cliente 1 conectado al Servidor.

Para la conexión del Cliente 2, el equipo se conecto a Internet por parte de la compañía Telcel. En la **Imagen 10** podemos ver la conexión establecida.

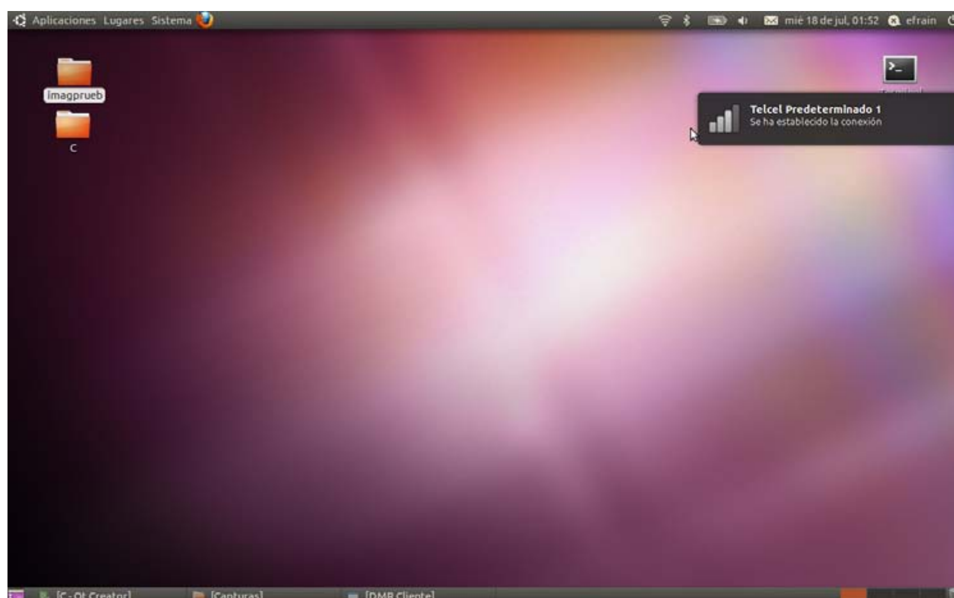


Imagen 10. Conexión a Internet de Cliente 2.

Una vez establecida la conexión a Internet de nuestro Cliente 2, se ejecutó la aplicación correspondiente al cliente nuevamente, y de igual manera se configuró para conectarse al servidor. En la **Imagen 11** se puede observar la configuración usada.

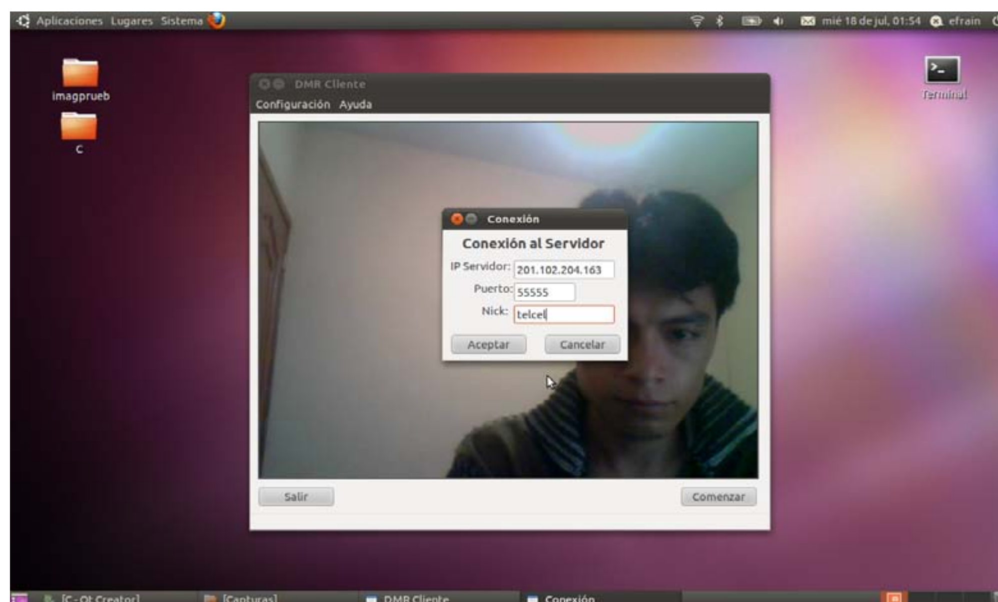


Imagen 11. Configuración de Cliente 2.

De igual manera que con el Cliente 1, el Cliente 2 se logró conectar al servidor. En la **Imagen 12** podemos observar el nuevo cliente conectado dentro del apartado clientes del la interfaz servidor.

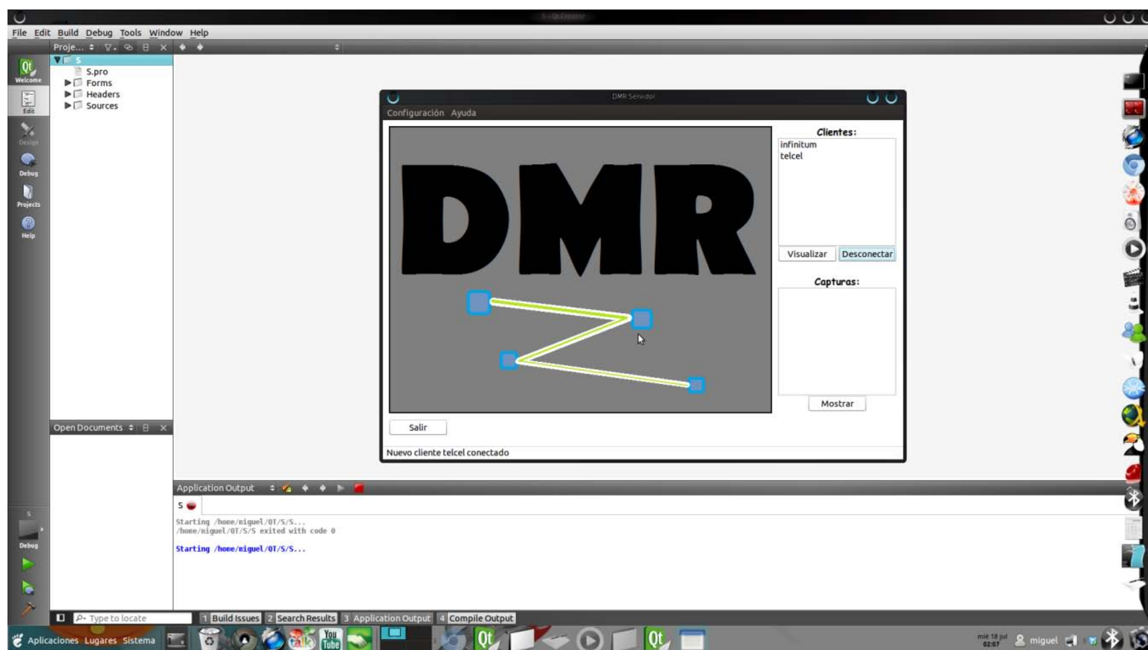


Imagen 12. Cliente 2 conectado al Servidor.

Para entender mejor las conexiones de nuestras aplicaciones, se puede observar en la **Figura 16** la topología usada para esta prueba.

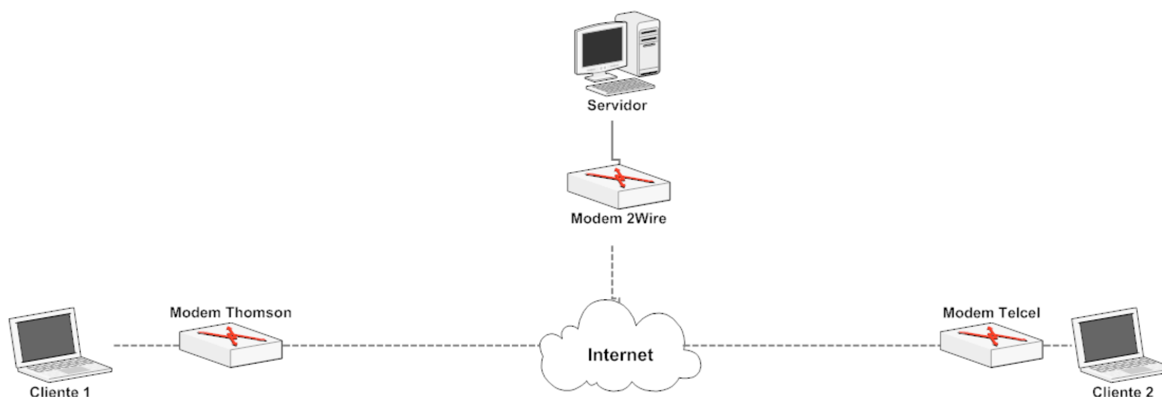


Figura 16. Topología usada en la prueba.

Una vez conectados nuestros dos clientes al servidor, se inicio la detección de movimiento. Para esto se dio click en el botón "Comenzar" dentro de la interfaz de ambos clientes. En la imagen **Imagen 13** se aprecia el inicio de detección para el Cliente 2.

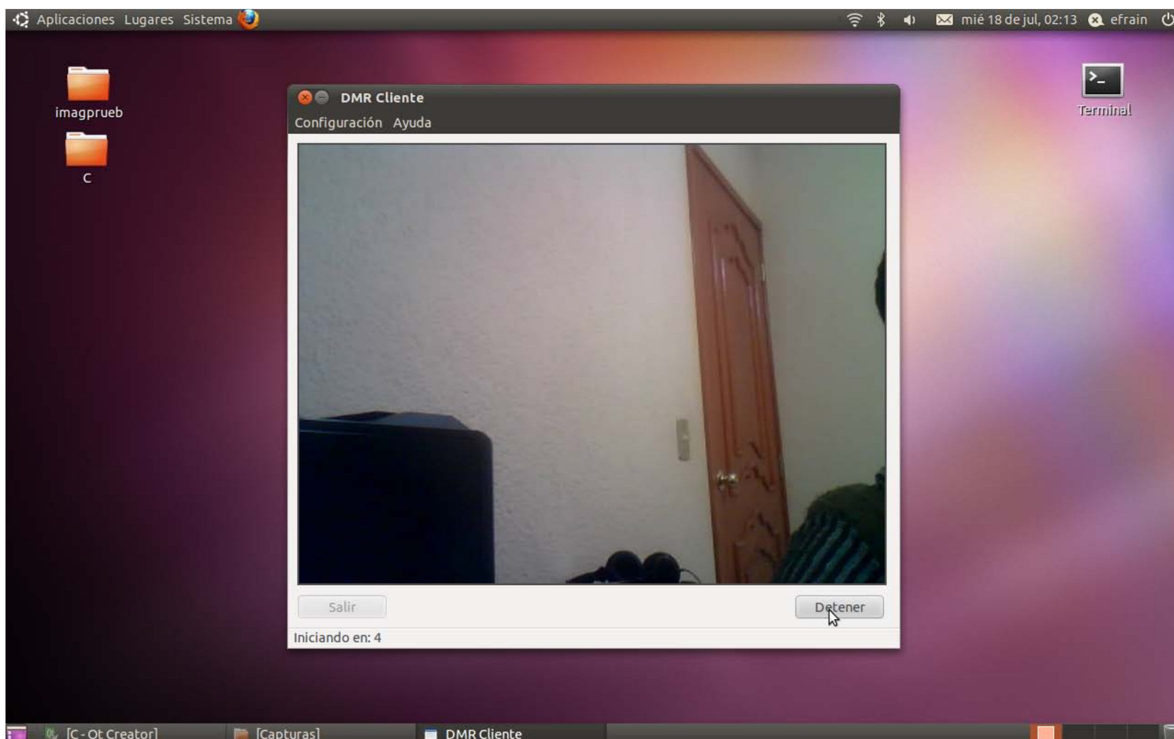


Imagen 13. Inicio de detección de movimiento en Cliente 2.

Para la detección de movimiento en esta prueba se pasaron ciertos objetos dentro de la escena que capturaba cada cliente. En primer lugar se paso una funda de teléfono en la escena del Cliente 1. En la **Imagen 14** se muestra una imagen capturada por el Cliente 1 al momento de detectar el movimiento.

Imagen 14. Imagen capturada por Cliente 1 al detectar movimiento.

Al detectar movimiento, el Cliente 1 informó al servidor del evento, mostrando un aviso textual y la imagen correspondiente. En la **Imagen 15** se puede apreciar la imagen recibida en la aplicación servidor.

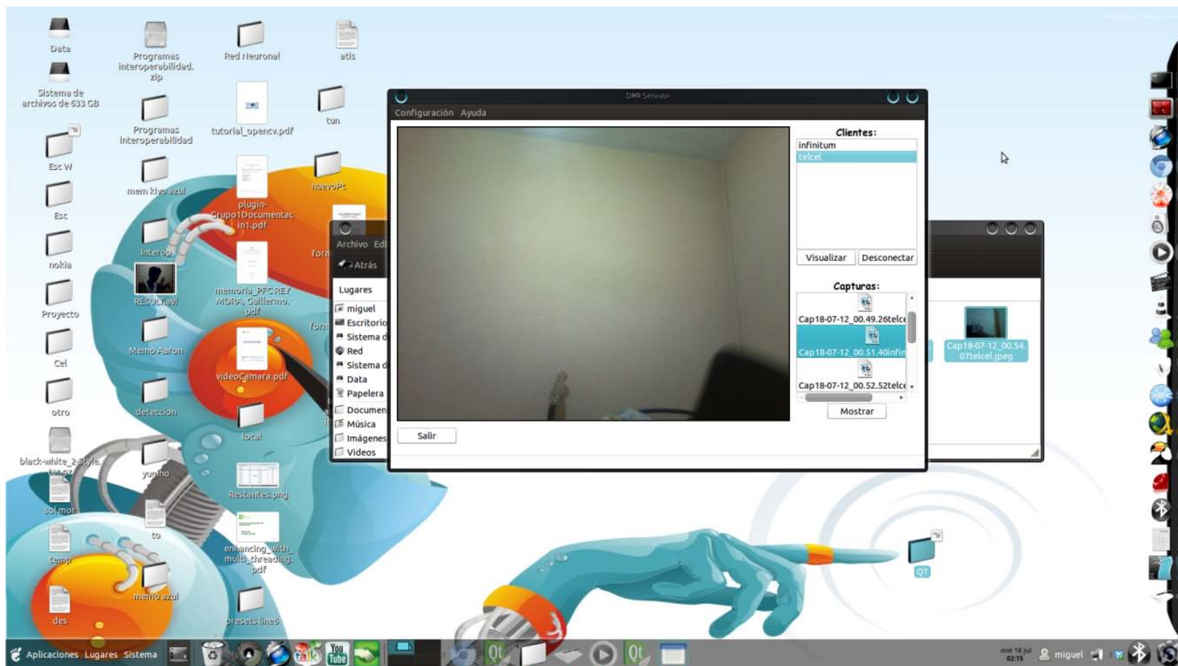


Imagen 15. Imagen recibida de Cliente 1 en Servidor.

Un segundo movimiento se generó en el Cliente 2, esta vez se atravesó una memoria flash (con puertoUSB) en la escena vigilada. En la **Imagen 16** podemos observar la captura por parte del Cliente 2 el cual detecta movimiento y en la **Imagen 17** dicha captura en el servidor.

Imagen 16. Imagen capturada por Cliente 2 al detectar movimiento.

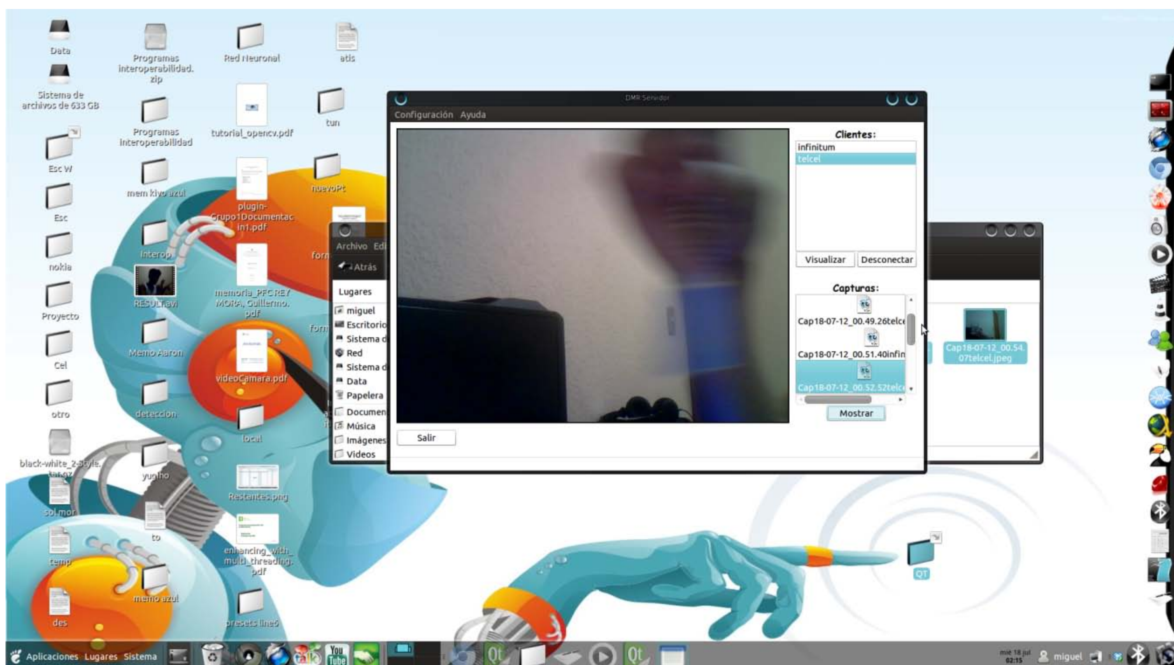


Imagen 17. Imagen recibida de Cliente 2 en Servidor.

Un tercer movimiento se generó en Cliente 2. Atravesando un vaso de vidrio en la escena vigilada se pudo detectar dicho movimiento. En la **Imagen 18** e **Imagen 19** se muestra la captura del Cliente y la imagen recibida en el servidor respectivamente.

Imagen 18. Segunda imagen capturada por Cliente 2 al detectar movimiento.

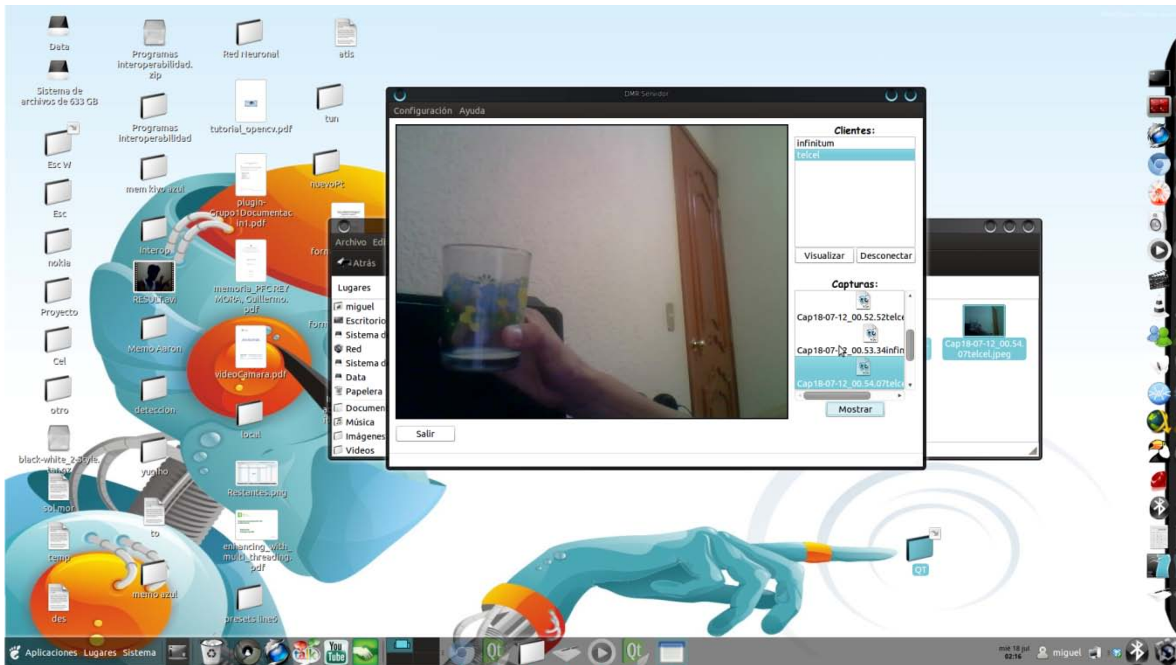


Imagen 19. Imagen recibida de Cliente 2 en Servidor.

Al terminar esta prueba, los resultados fueron satisfactorios, debido a que se detectó movimiento en ambos clientes simultáneamente y se notificó en el servidor del evento ocurrido, mostrando la imagen relacionada al movimiento generado en el cliente correspondiente.

CAPÍTULO 5

CONCLUSIONES

El objetivo de este proyecto fue desarrollar un sistema detector de movimientos en múltiples sitios remotos, con la finalidad de vigilarlos en nuestra ausencia. Para ello se diseñaron e implementaron dos aplicaciones bajo un modelo cliente-servidor. Una aplicación, llamada cliente, se encarga de detectar movimientos y reportarlos a otra aplicación llamada servidor, de manera remota. Mediante la realización de las pruebas hechas con el detector de movimiento se obtuvieron resultados satisfactorios, ya que los resultados arrojados fueron los esperados.

Se puede decir que los objetivos planteados en este proyecto se han cumplido, puesto que, se ha logrado implementar un sistema de detección de movimiento remoto con una funcionalidad razonablemente satisfactoria, de acuerdo con los objetivos propuestos. Aunque el DMR no se probó para más de dos clientes, se asume que su comportamiento con más clientes debe ser similar al observado con sólo dos clientes.

Como posibles trabajos futuros que den continuidad al presente proyecto está la integración y uso de *cámaras IP* dedicadas, de esta manera no habría la necesidad de requerir una computadora por cada cliente a vigilar. También, ampliar la funcionalidad de la aplicación Servidor para manipular mejor a los clientes conectados. Por ejemplo, sería deseable determinar el momento de inicio y terminación de la detección de movimiento de los clientes, desde el propio servidor, mostrando una lista de clientes activos y no activos. Además, se podrían mejorar los avisos de alarma, integrando al sistema la capacidad de reproducir sonidos personalizados cuando un movimiento se genere en algún cliente activo específico.

BIBLIOGRAFÍA

[1] I. Elizarrarás, **“Sistema de vigilancia remota programado con C++ y OpenCV”**, Proyecto Terminal Ingeniería en Computación, División de CBI, Universidad Autónoma Metropolitana Azcapotzalco, D.F, México, 2010.

[2] L. Marín**. “Sistema de aprendizaje del alfabeto dactilógico mediante procesamiento de imágenes utilizando software libre”**, Proyecto Terminal Ingeniería en Computación, División de CBI, Universidad Autónoma Metropolitana Azcapotzalco, D.F., México, 2009.

[3] F. M. Díaz**, “Clasificador de objetos de banda infinita por medio de procesamiento digital de imágenes”**, Proyecto Terminal Ingeniería en Computación, División de CBI, Universidad Autónoma Metropolitana Azcapotzalco, D.F., México, 2009.

[4]**Webcam Surveillance Standard**, Software de vigilancia. Disponible:
http://www.athtek.com/webcam-surveillance-standard.html.

[5] **Move Action**, Software de vigilancia. Disponible: http://move-action.softonic.com/

[6]**OpenCV,** Biblioteca de visión por computadora. Disponible:
http://opencv.willowgarage.com/wiki/

[7] **QT,** Biblioteca para desarrollo de aplicaciones con interfaces gráficas. Disponible:
http://qt.nokia.com/products/

[8] **QT Creator**, Entorno de desarrollo integrado. Disponible:
http://qt.nokia.com/products/developer-tools/

ANEXOS

ANEXO A. CÓDIGO FUENTE DEL SISTEMA DMR

Servidor DMR

```
//////////////////////////////////////////////////////////////
/////////////////////////CONEXION.H////////////////////////
#ifndef CONEXION_H
#define CONEXION_H
//Clases necesarias//
#include <QObject>
#include <QByteArray>
#include <QString>
#include <QStringList>
#include <QTcpSocket>
#include <QFile>
//Inicio de clase conexion//
class conexion : public QObject
{
	Q_OBJECT
	public:
	//Constructor que recibe como parámetro el socket
	//para la comunicación
	conexion(QTcpSocket *_socket);
	//Socket que mantiene la comunicación
	QTcpSocket *socket;
	//Array de bytes para leer y escribir
	QByteArray mensaje;
	//Nick asociado a la conexión
	QString nick;
	//Índice dentro la lista de conexiones y otras variables
	//para el manejo de la comunicación
	int indice_lista_conexion, k, t, bytes;
	bool sup;
	bool f;
	private:
	//Referencia a archivo imagen
	QFile *Rimg;
	//Ruta de la carpeta de almacenamiento
	QString Cruta;
	//Función para procesar un comando
	void procesar(QString msg);
	private slots:
	//Función que implementa módulo Recepción
	void recibir();
	public slots:
	//Función para enviar mediante el socket
	void enviar(QString);
	//Función para desconectar un cliente
	void des();
	signals:
	//Señal para crear una nueva conexión de un cliente
	void conec(conexion*, QString);
	//Señal para desconectar un cliente
```

```
        void desconec(conexion*, QString);
        //Señal para informar una nueva imagen recibida
        void imgcap(QString,QString);
};
//Fin de clase conexion//
#endif // CONEXION_H
//////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
//////////////////////CONEXION.CPP/////////////////////////
//Clases necesarias
//y definición de clase conexion//
#include "conexion.h"
#include <QDateTime>
#include <QDebug>
//Constructor de clase conexion//
conexion::conexion(QTcpSocket *_socket)
{
        //Se asigna el socket recibido
        socket = _socket;
        //Se conectan las señales recibidas con las funciones
        //correspondientes
        connect(socket, SIGNAL(readyRead()), this, SLOT(recibir()));
        connect(socket, SIGNAL(disconnected()), this, SLOT(des()));
        //Se inicializa el índice de la lista de conexiones
        indice_lista_conexion = -1;
        //Se inicializa la ruta de almacenamiento
        Cruta="../S/Capturas/";
        //Se inicializan las variables de comunicación
        sup=false;
        f=true;
        k=0;
}
//Fin de constructor de clase conexion//

//////////Módulo Recepción//////////
void conexion::recibir()
{
        //Si se trata de una imagen
        if(this->sup)
        {
                //Si aún no se sabe el tamaño de la imagen
                if(f)
                {
                        //Se crea un nuevo archivo para la imagen
                        Rimg = new QFile(Cruta + "Cap"
                        +QDateTime::currentDateTime().toString("dd-MM-
                        yy_hh.mm.ss") + nick +".jpeg");
                        //Se recibe el tamaño de la imagen
                        QByteArray datos2 = socket->readAll();
                        bytes = datos2.toInt();
                        qDebug()<<"Bytes: "<<bytes;
                        //Se abre el archivo de la imagen
                        Rimg->open(QIODevice::Append);
                        t=0;
```

```
                f=false;
            }
            //Si ya se sabe el tamaño de la imagen
            else
            {
                //Se reciben los datos de la imagen
                QByteArray datos = socket->readAll();
                t= t + datos.size();
                qDebug()<<"Recibiendo "<<t<<" de "<<bytes <<" bytes";
                //Se escriben los datos al archivo
                Rimg->write(datos);
                //Si ya se recibió el total de los datos
                if(t>=bytes)
                {
                    //se cierra el archivo de la imagen
                    Rimg->close();
                    f=true;
                    //Se emite una orden para mostrar la imagen
                    emit imgcap(Rimg->fileName(),nick);
                }
            }
        }
        //Si se trata de un comando
        else
        {
            //Se leen los datos
            mensaje.append(socket->readAll());
            int pos;
            //Se obtiene el comando de los datos recibidos
            while((pos = mensaje.indexOf("\n\r")) > -1)
            {
                //Se procesa el comando
                procesar(QString(mensaje.left(pos+2)));
                mensaje = mensaje.mid(pos+2);
            }
        }
}
//////////Fin de módulo Recepción/////////

//Función procesar de la clase conexion//
void conexion::procesar(QString msg)
{
        //Si el comando indica una nueva conexión
        if(msg.startsWith("CON:"))
        {
            //Se obtiene el nick de la conexión
            QString resto = msg.mid(4 );
            nick = resto.mid(0,resto.length()-2) ;
            //Se emite una señal a clase servidor con la
            //información de la nueva conexión
            emit conec(this, nick);
        }
        //Si el comando indica desconexión
        if(msg.startsWith("DES:"))
        {
            //Se emite una señal a la clase servidor
```

```
            //para cerrar la conexión del cliente
            emit desconec(this, nick);
      }
}
//Fin de función procesar de la clase conexion//

//Función enviar de la clase conexion//
void conexion::enviar(QString msg)
{
      //Si el socket de la conexión es valido
      if (socket->isValid())
      {
            //Se envía el mensaje en Ascii
            socket->write(msg.toAscii());
            socket->flush();
      }
}
//Fin de función enviar de la clase conexion//

//Función desconectar de la clase conexion//
void conexion::des()
{
      //Emitir una señal a clase servidor
      //para desconectar al cliente
      emit desconec(this, nick);
}
//Fin de función desconectar de la clase conexion//
//////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////


//////////////////////////////////////////////////////////////
//////////////////////MAIN.CPP///////////////////////////////
//Clase necesaria y definición
//de clase mainwindow//
#include <QtGui/QApplication>
#include "mainwindow.h"

//Función principal//
int main(int argc, char *argv[])
{
    //Se crea una instancia de mainwindow
    QApplication a(argc, argv);
    MainWindow w;

    //Se muestra la interfaz
    w.show();
    return a.exec();
}
//Fin de función principal//
//////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////
```

```
//////////////////////////////////////////////////////////////
//////////////////////MAINWINDOW.H//////////////////////////
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

//Clases necesarias//
#include <QMainWindow>
#include <QDirModel>
#include <QStringListModel>
#include <servidor.h>
#include <winconexs.h>
#include <wininfos.h>

//Uso de la interfaz mainwindow.ui//
namespace Ui
{
    class MainWindow;
}

//Inicio de la clase mainwindow//
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:

    //Constructor y destructor de la clase
    MainWindow(QWidget *parent = 0);
    ~MainWindow();

    //Variable para el estado del servidor
    bool estadoServidor;
    bool vis;

private:
    //Referencia a la interfaz gráfica
    Ui::MainWindow *ui;
    //Ruta de almacenamiento
    QString ruta;
    //Nombre de imagen
    QString imgnom;
    //Puerto e ip para la conexión
    QString Puerto, Ip;
    //Modelo directorio para acceder al
    //contenido del directorio de almacenamiento
    QDirModel modelcap;
    //Modelo de lista para clientes
    QStringListModel modelclt;
    //Referencias a clases usadas
```

```
    servidor *serv;
    winconexs *wincon;
    wininfos * wininf;
    //Lista de nicks de clientes
    QStringList *nickList;

    //Función para inicializar servidor a estado inicial
    void Default();

private slots:
    //Función para cerrar interfaz
    void on_BotSalir_clicked();
    //Función para desplegar información de ayuda
    void on_actionAcerca_de_triggered();
    //Función para desplegar información de conexión
    void on_actionConexi_n_triggered();
    //Función para desconectar un cliente
    void on_BotDesconec_clicked();
    //Función para mostrar imagen recibida
    void on_BotMostrar_clicked();
    //Función para solicitar escena de cliente
    void on_BotVisual_clicked();
    //Función para establecer una conexión
    void configconex(QString puerto);
    //Función para mostrar información en barra de estado
    void mensajeStatus(QString msj);
    //Función para agregar un cliente
    void agregar(QString nick);
    //Función para eliminar un cliente
    void eliminar(QString nick);
    //Función que implementa módulo Mostrar
    void most(QString idirec, QString n);
};
//Fin de la clase mainwindow//

#endif
//////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
///////////////////MAINWINDOW.CPP/////////////////////////
//Definiciones de clases//
#include "mainwindow.h"
#include "ui_mainwindow.h"

//Constructor de la clase mainwindow//
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
```

```
{
    //Se inicializa la interfaz
    ui->setupUi(this);
    //Se asigna el tamaño de la ventana
    this->setFixedSize(869,592);

    //Se asigna el servidor como inactivo
    estadoServidor=false;

    //Se crea una instancia de la clase servidor
    serv = new servidor();
    //Se crea una lista de nicks
    nickList = new QStringList();

    //Se inicializa el servidor a su estado inicial
    Default();

    //Se enlaza el directorio de almacenamiento
    //con la lista de contenido de la interfaz
    modelcap.setFilter(QDir::Files);
    ui->listCap->setModel(&modelcap);
    ui->listCap->setViewMode(QListView::IconMode);
    ui->listCap->setRootIndex(modelcap.index(ruta));

    //Se enlaza la lista de clientes con
    //la lista de clientes de la interfaz
    modelclt.setStringList(*nickList);
    ui->listClient->setModel(&modelclt);

    //Se conectan las señales recibidas con sus correspondientes
    //funciones
    connect(serv,SIGNAL(statusMensaje(QString)),this,
    SLOT(mensajeStatus(QString)));
    connect(serv,SIGNAL(agregar(QString)),this,SLOT(agregar(QString)));
    connect(serv,SIGNAL(eliminar(QString)),this,SLOT(eliminar(QString)));
    connect(serv,SIGNAL(most(QString,QString)),this,SLOT(most
(QString,QString)));
}
//Fin de constructor de clase mainwindow//

//Destructor de la clase mainwindow//
MainWindow::~MainWindow()
{
    delete ui;
    delete this->serv;
    delete this->nickList;
}
//Fin de destructor de clase mainwindow//
```

```
//Función para solicitar escena de cliente//
void MainWindow::on_BotVisual_clicked()
{
      //Se envía el comando de visualizar escena al cliente
      serv->enviar("SUP:\n\r",ui->listClient->
      currentIndex().data().toString());
      //Se envía una notificación a la barra de estado en la interfaz
      statusBar()->showMessage("Orden Visualizar",5000);
      vis=true;
}
//Fin de función para solicitar escena de cliente//

//Función para mostrar una imagen recibida//
void MainWindow::on_BotMostrar_clicked()
{
    //Se obtiene el nombre de la imagen seleccionada
    imgnom=ruta + ui->listCap->currentIndex().data().toString();
    //Se crea una referencia a imagen seleccionada
    QImage img(imgnom);
    //Se muestra la imagen seleccionada
    ui->labImg->setPixmap(QPixmap::fromImage(img));
    //Se envía una notificación a la interfaz
    statusBar()->showMessage("Mostrando imagen",5000);
}
//Fin de función para mostrar una imagen recibida//

//Función para desconectar un cliente//
void MainWindow::on_BotDesconec_clicked()
{
    //Se obtiene el nick del cliente a desconectar
    QString nom = ui->listClient->currentIndex().data().toString();
    //Se elimina la conexión al cliente
    eliminar(nom);
    serv->buscar(nom);
    //Se envía una notificación a la interfaz
    statusBar()->showMessage("Orden Desconectar",5000);
}
//Fin de función para desconectar un cliente//

//Función para desplegar información de conexión//
void MainWindow::on_actionConexi_n_triggered()
{
    //Se crea una nueva ventana de la clase winconexs
    wincon = new winconexs(this);
    //Se copia la ip y el puerto de la conexión a la ventana
    wincon->lconxsIP->setText(Ip);
    wincon->lepuerto->setText(Puerto);
    //Si el servidor esta activo
    if (estadoServidor)
```

```
    {
        //Se muestra ip en ventana winconexs
        wincon->lconxsIP->setText(serv->dirIP);
        wincon->lepuerto->setEnabled(false);
    }
    //Se muestra la ventana de información de conexión
    wincon->show();
      //Se conecta una señal para asignar el puerto
      connect(wincon,SIGNAL(enviarconfig(QString)),this,SLOT(configconex(
      QString)));
}
//Fin de función información de conexión//

//Función para asignar el puerto//
void MainWindow::configconex(QString puerto)
{
    //Si el servidor está inactivo
    if (!estadoServidor)
    {
        //Se asigna el nuevo puerto
        Puerto = puerto;
        //Se inicia el servidor
        serv->iniciar(Puerto.toInt());
    }
}
//Fin de función para asignar puerto//

//Función para desplegar información ayuda//
void MainWindow::on_actionAcerca_de_triggered()
{
    //Se crea y se muestra una nueva ventana
    //de la clase wininfos
    wininf = new wininfos(this);
    wininf->show();
}
//Fin de función para desplegar ayuda//

//Función para cerrar la interfaz//
void MainWindow::on_BotSalir_clicked()
{
    //Se detiene el servidor
    serv->detener();
    //Se cierra la ventana interfaz
    this->close();
}
//Fin de función para cerrar la interfaz//

//Función para mostrar texto en barra de estado de interfaz//
void MainWindow::mensajeStatus(QString msj)
```

```
{
    //Si se inicia el servidor con éxito
    if(msj.startsWith("Servidor iniciado exitosamente"))
    {
        //Se pone la variable de estado como activa
        estadoServidor=true;
    }
    //Se muestra el mensaje en la barra de estado
    ui->statusBar->showMessage(msj,5000);
}
//Fin de función mostrar información de estado//

//Función para agregar un cliente a la lista//
void MainWindow::agregar(QString nick)
{
    //Se integra el nick del cliente a la lista
    nickList->push_back(nick);
    //Se actualiza el contenido de la lista en la interfaz
    modelclt.setStringList(*nickList);
    //Se envía una notificación del nuevo cliente conectado
    ui->statusBar->showMessage("Nuevo cliente "+nick+" conectado",5000);
}
//Fin de función para agregar cliente//

//Función para eliminar un cliente de la lista//
void MainWindow::eliminar(QString nick)
{
    //Se obtiene el índice del cliente en la lista
    int index = nickList->indexOf(nick);
    //Se elimina el cliente con su índice
    nickList->removeAt(index);
    //Se actualiza el contenido de la lista en la interfaz
    modelclt.setStringList(*nickList);
    //Se envía una notificación del cliente desconectado
    ui->statusBar->showMessage("Cliente "+nick+" desconectado",5000);
}
//Fin de función para eliminar cliente//

//Función para poner el servidor en estado inicial//
void MainWindow::Default()
{
    //Se abre imagen de logo del sistema
    QImage logo("DMRlogo.png");
    //Se muestra el logo del sistema en la interfaz
    ui->labImg->setPixmap(QPixmap::fromImage(logo));

    //Se inicializan las variables por default
    Ip = "LocalHost";
    Puerto = "55555";
```

```
    ruta = "../S/Capturas/";
    vis = false;

    //Si el directorio de almacenamiento no existe
    if(! (QDir("../S/Capturas/").exists()))
    {
        //Se crea el directorio de almacenamiento
        QDir().mkdir("../S/Capturas/");
    }
}
//Fin de función para poner servidor en estado inicial//

//////////Módulo Mostrar/////////
void MainWindow::most(QString idirec, QString n)
{
       //Se actualiza el contenido de la lista de imágenes
       modelcap.setFilter(QDir::Files);
       ui->listCap->setModel(&modelcap);
       ui->listCap->setViewMode(QListView::IconMode);
       ui->listCap->setRootIndex(modelcap.index(ruta));

    //Si se solicitó una imagen de escena de cliente
    if (vis)
    {
            //Se informa de la imagen recibida en la interfaz
            vis = false;
            statusBar()->showMessage("Mostrando imagen capturada en
      cliente " + n ,5000);
    }
    //Si se recibió la imagen como movimiento
    else
    {
            //Se informa de la imagen recibida con movimiento
              statusBar()->showMessage("Se ha detectado movimiento en
      cliente " + n ,5000);
    }
    //Se muestra la imagen en la interfaz
    QImage imgM(idirec);
    ui->labImg->setPixmap(QPixmap::fromImage(imgM));
}
//////////Fin de módulo Mostrar//////////
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////
/////////////////////////SERVIDOR.H////////////////////////
#ifndef SERVIDOR_H
#define SERVIDOR_H
```

```
//Clases necesarias
//y definición de clase conexion//
#include <QTcpServer>
#include <QStringList>
#include <QList>
#include "conexion.h"

//Inicio de la clase servidor//
class servidor : public QTcpServer
{
    Q_OBJECT

public:
    //Constructor y destructor de la clase
    servidor();
    ~servidor();

    //Variables para almacenar ip y puerto
    QString dirIP;
    int serPuerto;

    //Función para iniciar servidor
    void iniciar(int puerto);
    //Función para desconectar a un cliente
    void buscar(QString nick);
    //Función para detener el servidor
    void detener();
    //Función para enviar mensaje a un cliente
    void enviar(QString mensaje, QString nick);
    //Lista de conexiones de los clientes
    QList<conexion *> Lista_conexiones;
    //Variable de estado de servidor
    bool estado();

public slots:
        //Función para enlazar una nueva conexión
        void nuevaConexion();
        //Función para agregar un nuevo cliente
        void conec(conexion *con, QString nick);
        //Función para desconectar un cliente
        void desconec(conexion *con, QString nick);
        //Función para mostrar imagen recibida
        void imgcap(QString direc, QString n);

private:
        //Variable que indica la conexión del servidor
        bool servConectado;
signals:
        //Señal para mostrar mensaje de estado
```

```
        void statusMensaje(QString);
        //Señal para agregar cliente
        void agregar(QString);
        //Señal para eliminar cliente
        void eliminar(QString);
        //Señal para mostrar imagen recibida
        void most(QString,QString);
};
//Fin de clase servidor//

#endif
//////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
////////////////////////SERVIDOR.CPP////////////////////////
//Clases necesarias y definición
//de clase servidor//
#include "servidor.h"
#include "QHostInfo"

//Constructor de clase servidor//
servidor::servidor()
{
    //Se inicializa el servidor como inactivo
    servConectado = false;
    //Se espera una señal de nueva conexión
    connect(this, SIGNAL(newConnection()), this, SLOT(nuevaConexion()));
    //Se limpia la lista de conexiones
    Lista_conexiones.clear();
}
//Fin de constructor de clase servidor//

//Destructor de clase servidor//
servidor::~servidor()
{
    //Se detiene el servidor
    this->detener();
}
//Fin de destructor de clase servidor//

//Función para iniciar servidor//
void servidor::iniciar(int puerto)
{
    //Si el servidor esta desconectado
    if (servConectado==false)
    {
        //Si se inicia el servidor con éxito
        //con ip y puerto asignado
```

```
        if (listen(QHostAddress::Any, puerto))
        {
           //Se establece el servidor como activo
           servConectado = true;
           //Se limpia la lista de conexiones
           Lista_conexiones.clear();

           //Se obtiene la ip de conexión
           QHostInfo info;
           QList<QHostAddress> ip;

           info = QHostInfo::fromName( QHostInfo::localHostName()  );
           ip = info.addresses();
           dirIP = ip[2].toString();
           //Se obtiene el puerto
           serPuerto = puerto;

           //Se informa a la interfaz que se inició el servidor
                 emit statusMensaje("Servidor iniciado exitosamente en:
      "+ dirIP + " puerto: "+ QString::number(serPuerto));
        }
        //Si no se inicia el servidor con éxito
        else
        {
            //Se establece el servidor como inactivo
            servConectado = false;
            //Se informa en la interfaz
            emit statusMensaje("Servidor no se pudo iniciar");
        }
    }
}
//Fin de función para iniciar servidor//

//Función para detener servidor//
void servidor::detener()
{
    //Si el servidor está activo
    if (servConectado)
    {
      //Se estable servidor como inactivo
      servConectado = false;
      //Se limpia la lista de conexiones
       while (!Lista_conexiones.isEmpty())
        {
             //Se envía comando para desconectar a cada cliente
             conexion *cTemp= Lista_conexiones.takeFirst();
             cTemp->enviar("DET:\n\r");
             delete cTemp;
         }
```

```
        //Se cierra el socket del servidor
        this->close();
        //Se informa a la interfaz que el servidor se ha detenido
        emit statusMensaje("Servidor detenido");
    }
}
//Fin de función para detener servidor//

//Función para obtener el estado del servidor//
bool servidor::estado()
{
    //Si el servidor está conectado
    if(servConectado)
    //Función regresa verdadero
    return true;
            //Si no está conectado
            else
                //Función regresa falso
                return false;
}
//Fin de función para obtener estado de servidor//

//Función para enlazar una nueva conexión//
void servidor::nuevaConexion()
{
        //Se crea una nueva instancia de la clase conexion
        conexion *con = new conexion(nextPendingConnection());
        //Se agrega la nueva conexión a la lista
        Lista_conexiones.push_back(con);
        con->indice_lista_conexion = Lista_conexiones.count()-1;

        //Se conectan las señales recibidas con sus correspondientes
      //funciones
        connect(con, SIGNAL(imgcap(QString,QString)), this,
      SLOT(imgcap(QString,QString)));
        connect(con, SIGNAL(conec(conexion*, QString)), this,
      SLOT(conec(conexion*, QString)));
        connect(con, SIGNAL(desconec(conexion*, QString)), this ,
      SLOT(desconec(conexion*, QString)));
}
//Fin de función para enlazar nueva conexión//

//Función para agregar un nuevo cliente//
void servidor::conec(conexion* con, QString nick)
{
        //Se asigna el nuevo nick recibido
        con->nick=nick;
        //Se envía una señal para agregar a la lista de nicks
        emit agregar(nick);
```

```
}
//Fin de función para agregar nuevo cliente//

//Función para desconectar a un cliente//
void servidor::desconec(conexion* con, QString nick)
{
        //Se cierra el socket de comunicación del cliente
        Lista_conexiones.at(con->indice_lista_conexion)->socket->close();
        //Se envía una señal para eliminar de la lista
        emit eliminar(nick);
}
//Fin de función para desconectar a un cliente//

//Función para enviar un mensaje a un cliente//
void servidor::enviar(QString mensaje, QString nick)
{
   int i=0;
   //Se recorre la lista de conexiones
    while (i<Lista_conexiones.count())
   {
        //Si la conexión actual esta activa
        if(Lista_conexiones.at(i)->socket->isValid())
        {
            //Si el nick de la conexión es igual al de destino
            if(QString::compare(Lista_conexiones.at(i)->nick,nick) ==0)
            {
                //Se envía el mensaje al cliente
                Lista_conexiones.at(i)->sup=true;
                Lista_conexiones.at(i)->enviar(mensaje);
                emit statusMensaje("Mensaje Sup enviado");
                Lista_conexiones.at(i)->k++;
                break;
            }
        }
        i++;
   }
}
//Fin de función para enviar mensaje a cliente//

//Función para desconectar a un cliente//
void servidor::buscar(QString nick)
{
    int i=0;
    //Se recorre la lista de conexiones
    while (i<Lista_conexiones.count())
   {
        //Si la conexión actual esta activa
        if(Lista_conexiones.at(i)->socket->isValid())
        {
```

```
            //Si el nick de la conexión es igual al cliente a desconectar
            if(QString::compare(Lista_conexiones.at(i)->nick,nick) ==0)
            {
                //Se envía comando de desconexión
                Lista_conexiones.at(i)->enviar("DES:\n\r");
                //Se cierra el socket correspondiente al cliente
                Lista_conexiones.at(i)->socket->close();
                break;
            }
        }
        i++;
    }
}
//Fin de función para desconectar cliente//

//Función para mostrar imagen recibida//
void servidor::imgcap(QString direc, QString n)
{
    //Se envía una señal para mostrar la imagen
    emit most(direc,n);
}
//Fin de función para mostrar imagen//
//////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
/////////////////////////WINCONEXS.H/////////////////////////
#ifndef WINCONEXS_H
#define WINCONEXS_H

//Clases necesarias
#include <QMainWindow>
#include <QLabel>
#include <QLineEdit>

//Uso de la interfaz winconexs.ui
namespace Ui {
    class winconexs;
}

//Inicio de la clase winconexs//
class winconexs : public QMainWindow
{
    Q_OBJECT

public:
    //Constructor y destructor de la clase
    explicit winconexs(QWidget *parent = 0);
    ~winconexs();
```

```
    //Etiqueta y campo de texto para ip y puerto
    QLabel *lconxsIP;
    QLineEdit *lepuerto;

signals:
    //Señal para asignar configuración de conexión
    void enviarconfig(QString _ip);

private:
    //Referencia a la interfaz winconexs.ui
    Ui::winconexs *ui;

private slots:
    //Función para enviar la configuración de conexión
    void on_bconxsacep_clicked();
};
//Fin de clase winconexs//

#endif
//////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
//////////////////////WINCONEXS.CPP//////////////////////////
//Clases necesarias y
//definición de clase e interfaz
#include "winconexs.h"
#include "ui_winconexs.h"
#include <QMessageBox>

//Constructor de la clase winconexs//
winconexs::winconexs(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::winconexs)
{
    //Se inicializa la interfaz
    ui->setupUi(this);
    //Se asigna el tamaño de la ventana
    this->setFixedSize(236,161);

    //Se inicializa el tamaño y contenido
    //de los elementos de la interfaz
    lconxsIP = new QLabel ("",this);
    lconxsIP->setGeometry(70,50,121,16);
    lepuerto = new QLineEdit("",this) ;
    lepuerto->setGeometry(100,90,81,16);
}
//Fin de constructor de la clase//
```

```
//Destructor de la clase winconexs//
winconexs::~winconexs()
{
    //Eliminar interfaz
    delete ui;
}
//Fin de destructor de la clase//

//Función para enviar la configuración de conexión//
void winconexs::on_bconxsacep_clicked()
{
        //Si el contenido del puerto está vacio
        if(lepuerto->text().isEmpty())
        {
            //Informar el error en cuadro de dialogo
                   QMessageBox::information(this,"Campo de Puerto
      vacio","Especifique un Puerto");
        }
        //Si el contenido del campo no está vacio
        else
        {
            //Enviar señal para asignar parámetros de conexión
            emit enviarconfig(lepuerto->text());
            //Cerrar ventana winconexs
            close();
        }
}
//Fin de función enviar configuración//
//////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
//////////////////////WININFOS.H/////////////////////////
#ifndef WININFOS_H
#define WININFOS_H

//Clase necesaria para ventana principal
#include <QMainWindow>

//Uso de interfaz wininfos
namespace Ui {
    class wininfos;
}

//Inicio de clase wininfos//
class wininfos : public QMainWindow
{
    Q_OBJECT
```

```
public:
    //Constructor y destructor de la clase
    explicit wininfos(QWidget *parent = 0);
    ~wininfos();

private:
    //Referencia a la interfaz wininfos.ui
    Ui::wininfos *ui;
};
//Fin de clase wininfos//

#endif
//////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////

//////////////////////////////////////////////////////////////
///////////////////////WININFOS.CPP/////////////////////////
//Definición de clase e interfaz
#include "wininfos.h"
#include "ui_wininfos.h"

//Constructor de clase wininfos//
wininfos::wininfos(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::wininfos)
{
    //Se inicializa la interfaz
    ui->setupUi(this);
    //Se asigna el tamaño de la ventana
    this->setFixedSize(396,179);

    //Se conecta una señal para cerrar ventana
    connect(ui->binfosacep, SIGNAL(clicked()),this, SLOT(close()));
}
//Fin de constructor de la clase wininfos//

//Destructor de la clase wininfos
wininfos::~wininfos()
{
    //Elimina interfaz wininfos
    delete ui;
}
//Fin de destructor de la clase wininfos//
//////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////
```

Cliente DMR

```
///////////////////////////////////////////////
////////////////////DETECTAR.H////////////////
#ifndef DETECTAR_H
#define DETECTAR_H

//Clases necesarias//
#include <QtCore/QThread>
#include <opencv/cv.h>
#include <opencv/highgui.h>
#include <opencv/cvaux.h>

//Inicio clase detectar//
class Detectar : public QThread
{
    Q_OBJECT
public:
    //Constructor y destructor de la clase
    Detectar(QObject *parent = 0);
    ~Detectar();

    //Función para asignar cámara
    void asignar(CvCapture * c);

    //variable de retardo de inicio
    int retardo;
    //Variable de sensibilidad
    int sens;
    //Variable para video, cuadros por segundo
    double fps;

protected:
    //Ruta de almacenamiento
    QString ruta;
    //Nombre de imagen
    QString nom;
    //Instancia de estructura de captura
    CvCapture *captura;
    //Instancia de estructuras de imágenes
    IplImage * img;
    IplImage * gris;
    //Variable para nombre de imagen
    int cap;
    //Variables de acumulación
    int pix,pmax;
    //Array de sensibilidad
    int S[9];
```

```
    //Variables de estado de movimiento
    bool mov;
    bool inicia;
    //Estructura de tamaño de imagen
    CvSize size;
    //Estructura de captura de vídeo
    CvVideoWriter *writer;
    CvBGStatModel * bgModel;

    //Función que implementa módulo Reconocimiento
    void run();

signals:
    //Señal para enviar imagen a servidor
    void imagNom(QString nom);
    //Señal para notificar en interfaz
    void statusBar(QString msj);
};
//Fin de clase detectar//
#endif
///////////////////////////////////////////////
///////////////////////////////////////////////

///////////////////////////////////////////////
///////////////////DETECTAR.CPP//////////////
//Definición de clase detectar
#include "detectar.h"

using namespace cv;

//Constructor de clase detectar//
Detectar::Detectar(QObject *parent)
        : QThread(parent)
{
    //Inicialización de variables por default
    pix = 0;
    fps = 30;
    sens = 8;
    retardo = 5;
    cap = 0;
    ruta = "../C/Capturas/";
    S[0]=50; S[1]=45; S[2]=40; S[3]=35; S[4]=30;
    S[5]=25; S[6]=20; S[7]=15; S[8]=10; S[9]=5;
}
//Fin de constructor de la clase detectar//

//Destructor de clase detectar//
Detectar::~Detectar()
{
```

```
}
//Fin de destructor de la case detectar//

//////////Módulo reconocimiento//////////
void Detectar::run()
{
        //Se inicializan variables de estado inicial
        mov = false;
        pmax = 0;
        inicia = true;

        //Se espera un tiempo para iniciar la detección
        for(int i=retardo; i>0; i--)
        {
            //Se informa en interfaz el tiempo restante
            emit statusBar("Iniciando en: "+ QString::number(i));
            sleep(1);
        }
        //Se informa el inicio de la detección
        emit statusBar("Detector iniciado");

        //Se crean matrices para almacenar la imagen
              Mat foreground;
              Mat structure = getStructuringElement(MORPH_RECT, Size
      (S[sens-1],S[sens-1]));

        //Se crean vectores punto para recorrer la estructura de la
      //imagen
        vector<vector<Point> > contours;
        vector<Vec4i> hierarchy;

        //Se captura una nueva imagen de la cámara
        img=cvQueryFrame(captura);

        //////////Inicio de módulo Conversión//////////
        //Variable para el tamaño de la imagen en escala de grises
        CvSize grisSize;
        //Se le pasa el ancho y alto de imagen original
        grisSize.width = img->width;
        grisSize.height = img->height;
       //Se crea una estructura tipo imagen con el ancho y alto de
      //la imagen original,
        //se le asigna la misma profundidad pero con un solo canal
        gris = cvCreateImage( grisSize, img->depth, 1 );
        //Se transforma la imagen a escala de grises
        cvCvtColor(img,gris,CV_RGB2GRAY);
        //////////Fin de módulo Conversión//////////

        //Se inicia el reconocimiento, detector de movimiento
```

```
  //se guarda una referencia a la imagen actual y
  //otra que representa las partes móviles en cada momento
  CvBGStatModel * bgModel = cvCreateGaussianBGModel(img,NULL);

  //Se almacena las propiedades de la imagen como son el ancho y el
  //alto
  size = cvSize( (int)cvGetCaptureProperty(captura,
CV_CAP_PROP_FRAME_WIDTH),(int)cvGetCaptureProperty(captura,
CV_CAP_PROP_FRAME_HEIGHT));

  while(1)
  {
      //Se obtiene una nueva imagen de la cámara
      img = cvQueryFrame(captura);
      //Se actualiza la imagen actual y
      //se almacenan las partes móviles de la imagen
      cvUpdateBGStatModel(img,bgModel);
      foreground = bgModel->foreground;
      //Se quita el ruido o basura de la imagen
      erode(foreground,foreground,structure);
      dilate(foreground,foreground,structure);
      //Se busca un cambio en la estructura de movimiento

      findContours(foreground,contours,hierarchy,CV_RETR_CCOMP,CV_C
      HAIN_APPROX_SIMPLE);
      //Si existe un cambio, movimiento
      if(contours.size()>0)
      {
          //////////Inicio de módulo Grabación//////////
          //Se obtiene el tamaño de los vectores en movimiento
          pix=contours.size();
          //Si no hay un movimiento actualmente
          if(inicia)
          {
              //Se crea el nombre del video con fecha actual
              nom= ruta + "Mov" + QString::number(cap);
              //Se crea el video
             writer = cvCreateVideoWriter(nom.toAscii()+".avi",
            CV_FOURCC('M','J','P','G'), fps ,size, 1);
              inicia = false;
              mov = true;
              cap++;
          }
          //Si la imagen actual tiene un mayor cambio
          if(pix > pmax)
          {
              //Se guarda la imagen
              cvSaveImage(nom.toAscii()+".jpeg",img);
              pmax=pix;
```

```
                }
                //Se almacena la imagen al video
                cvWriteFrame( writer, img );
                //////////Fin de módulo Grabación/////////
            }
            //Si no existe movimiento
            else
            {
                //Si anteriormente se detectó movimiento
                if(mov)
                {
                    //Se envía una señal para enviar la imagen
                    emit imagNom(nom);
                    //Se establecen las variables a su normalidad
                    inicia = true;
                    mov=false;
                    pmax=0;
                }
            }
        }
        this->exec();
}
//////////Fin de módulo Reconocimiento//////////

//Función para asignar cámara//
void Detectar::asignar(CvCapture * c)
{
    //Asigna la cámara por default c
    captura = c;
}
//Fin de función asignar cámara//
///////////////////////////////////////////////
///////////////////////////////////////////////

///////////////////////////////////////////////
/////////////////////MAIN.CPP///////////////
//Clase necesaria y definición
//de clase mainwindow//
#include <QtGui/QApplication>
#include "mainwindow.h"

//Función principal//
int main(int argc, char *argv[])
{
    //Se crea una instancia de mainwindow
    QApplication a(argc, argv);
    MainWindow w;

    //Se muestra la interfaz
```

```
    w.show();
    return a.exec();
}
//Fin de función principal//
//////////////////////////////////////////////
//////////////////////////////////////////////

//////////////////////////////////////////////
//////////////MAINWINDOW.H//////////////////
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

//Clases y definiciones necesarias
#include <QMainWindow>
#include <opencv/cv.h>
#include <opencv/highgui.h>
#include <QFile>
#include <winavanzado.h>
#include <winconex.h>
#include <wininfo.h>
#include <sock.h>
#include <detectar.h>

//Uso de interfaz mainwindow.ui
namespace Ui {
    class MainWindow;
}

//Inicio de clase mainwindow//
class MainWindow : public QMainWindow
{

    Q_OBJECT

public:
    //Constructor y destructor de la clase
    MainWindow(QWidget *parent = 0);
    ~MainWindow();

    //Variable contador de capturas
    int cap;

private:
    //Referencia a mainwindow.ui
    Ui::MainWindow *ui;

    //Estructura de captura de imagen
    CvCapture* captura;
    //Estructuras de imágenes
```

```
    IplImage* img;
    IplImage* otr;
    //Elementos de la interfaz
    QImage image;
    QPixmap* pixmap;
    QTimer* timer;
    QString Puerto, Ip, Nombre;
    QString ruta;
    QByteArray datos;
    QFile *archivo;
    //Referencia a clase wincon
    winconex *wincon;
    //Referencia a clase wininf
    wininfo *wininf;
    //Referencia a clase avanz
    winavanzado *avanz;
    //Referencia a clase detec
    Detectar * detec;
    //Referencia a clase sock
    sock* socket;

    //Variable de estado de conexión
    bool estadoclt;
    //Variable de estado de detector
    bool activo;
    //Función para enviar imagen
    void enviarImg();
    //Función para establecer el estado inicial
    void Default();

public slots:
    //Función para recibir del socket
     void recibir(QString comando);
     //Función para informar un error
     void error(sock *, QString titulo,QString mensaje);

private slots:
    //Función para desplegar ventana de configuración avanzada
    void on_actionAvanzado_triggered();
    //Función para cerrar la interfaz
    void on_botonSalir_clicked();
    //Función para configurar de la conexión
    void configconex(QString ip, QString puerto, QString nick);
    //Función para asignar la configuración avanzada
    void configavanz(int in, double fr, int sns);
    //Función para desplegar ventana de ayuda
    void on_actionAcerca_de_triggered();
    //Función para desplegar ventana de conexión
    void on_actionConexi_n_2_triggered();
```

```
    //Función que implementa módulo Captura
    void mostrar();
    //Función para iniciar el detector
    void capturar();
    //Función para notificar mensaje en interfaz
    void statusMensaje(QString msj);
    //Función para enviar imagen de movimiento
    void recibirNom(QString nom);
};
//Fin de clase mainwindow//

#endif
///////////////////////////////////////////////
///////////////////////////////////////////////

///////////////////////////////////////////////
////////////////MAINWINDOW.CPP//////////////
//Clases y definiciones necesarias
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include "opencv/cv.h"
#include "opencv/highgui.h"
#include <QTimer>
#include <QDir>

using namespace cv;

//Constructor de la clase mainwindow//
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent), ui(new Ui::MainWindow)
{
    //Se inicializa la interfaz
    ui->setupUi(this);
    //Se asigna el tamaño de la ventana
    this->setFixedSize(664,583);

    //Se establecen variables de estado inicial
    estadoclt = false;
    activo = false;

    //Se asigna a estructura de captura la cámara por default
    captura = cvCaptureFromCAM(0);

    //Se crea un temporizador
    timer = new QTimer(this);
    //Se espera la señal del temporizador para mostrar captura
    connect(timer, SIGNAL(timeout()), this, SLOT(mostrar()));
    timer->start(1);
    //Se crea un nuevo socket para la comunicación
```

```
    socket = new sock();
    //Función para establecer estado inicial
    Default();

    //Se crea un instancia de la clase detectar
    detec = new Detectar(this);
    //Se le asigna la cámara por default
    detec->asignar(captura);

    //Se conectan las señales recibidas con sus respectivas funciones

connect(detec,SIGNAL(imagNom(QString)),this,SLOT(recibirNom(QString)),
Qt::QueuedConnection);
    connect(ui->botonComenzar,SIGNAL(clicked()),this,SLOT(capturar()));
    connect(socket, SIGNAL(newMessage(QString)),this,
SLOT(recibir(QString)));
    connect(socket, SIGNAL(errorconec(sock*, QString,QString)),this ,
SLOT(error(sock*, QString,QString)));
    connect(socket, SIGNAL(statusBar(QString)),this,
SLOT(statusMensaje(QString)));
    connect(detec, SIGNAL(statusBar(QString)),this,
SLOT(statusMensaje(QString)));
}
//Fin de constructor de la clase mainwindow//

//Destructor de la clase mainwindow//
MainWindow::~MainWindow()
{
    //Se libera la memoria para los objetos creados
    cvReleaseImage(&otr);
    cvReleaseImage(&img);
    cvReleaseCapture(&captura);
    delete this->detec;
    delete this->socket;
    delete ui;
}
//Fin de destructor de la clase mainwindow//

//Función para iniciar la detección//
void MainWindow::capturar()
{
    //Si el detector está ejecutándose
    if(detec->isRunning())
    {
        //Se detiene el detector
        detec->terminate();
        detec->wait();
        //Se cambia el texto del botón de la interfaz
        ui->botonComenzar->setText("Comenzar");
```

```
        //Se habilita el botón de salir de la interfaz
        ui->botonSalir->setEnabled(true);
        //Se establece el detector como inactivo
        activo = false;
    }
    //Si el detector no está ejecutándose
    else
    {
        //Se cambia el texto del botón de la interfaz
        ui->botonComenzar->setText("Detener");
        //Se deshabilita el botón salir de la interfaz
        ui->botonSalir->setEnabled(false);
        //Se inicia el detector
        detec->start();
        //Se establece el detector como activo
        activo = true;
    }
}
//Fin de función para iniciar detección//

//////////Inicio de módulo Captura/////////
void MainWindow::mostrar()
{
    //Se obtiene una imagen de la cámara
    img = cvQueryFrame( captura );

    //Se construye una nueva imagen con el tamaño de la captura
    //en formato RGB de 24 bits.
    image=QImage (QSize(img->width,img->height),QImage::Format_RGB888);
    //Se crea la estructura de una imagen del tamaño de la captura
    //de profundidad 8 bits y con 3 canales = 24 bits.
    otr=cvCreateImageHeader(cvSize(image.width(),image.height()),8,3);
    //Se almacenan los datos de la imagen en la auxiliar
    otr->imageData=(char*)image.bits();

    //Si los datos de la imagen están en el orden adecuado
    if(img->origin==IPL_ORIGIN_TL)
        //se copian los datos de la imagen a la auxiliar
        cvCopy(img,otr,0);
    //Si el orden no es el adecuado
    else
        //Se cambia el orden
        cvFlip(img,otr,0);
    //Se convierte la imagen auxiliar de BGR a RGB
    cvCvtColor(otr,otr,CV_BGR2RGB);
    //Se muestra la imagen capturada en la interfaz principal cliente
    ui->caplabel->setPixmap(QPixmap::fromImage(image));
    ui->caplabel->show();
}
```

```
//////////Fin de módulo Captura//////////

//Función para desplegar ventana de conexión//
void MainWindow::on_actionConexi_n_2_triggered()
{
    //Se crea una nueva instancia de la clase winconex
    wincon = new winconex(this);
    //Si la conexión está activa
    if(estadoclt)
    {
        //Se muestra en la ventana creada el puerto, la ip y el nick
        wincon->leipservidor->setText(Ip);
        wincon->lepuerto->setText(Puerto);
        wincon->lenick->setText(Nombre);
        //Se deshabilitan los campos para ser editados
        wincon->leipservidor->setEnabled(false);
        wincon->lepuerto->setEnabled(false);
        wincon->lenick->setEnabled(false);
    }
          //Se espera una señal para almacenar una nueva configuración de
      //conexión

connect(wincon,SIGNAL(enviarconfig(QString,QString,QString)),this,SLOT(co
nfigconex(QString,QString,QString)));
    //Se muestra la ventana creada
    wincon->show();
}
//Fin de función para desplegar ventana de conexión//

//Función para configurar la conexión//
void MainWindow::configconex(QString ip, QString puerto, QString nick)
{
    //Si la conexión está inactiva
    if(!estadoclt)
    {
        //Se asigna la ip, puerto y nick recibido
        Ip=ip;
        Puerto=puerto;
        Nombre=nick;

        //Se intenta conectar al servidor con la configuración recibida
        socket->conectarse(ip,puerto.toInt(),nick);
    }
}
//Fin de función para configurar conexión//

//Función para desplegar ventana de ayuda//
void MainWindow::on_actionAcerca_de_triggered()
{
```

```
    //Se crea una instancia de clase wininfo
    wininf = new wininfo(this);
    //Se muestra la ventana
    wininf->show();
}
//Fin de función para desplegar ventana ayuda//

//Función para cerrar interfaz//
void MainWindow::on_botonSalir_clicked()
{
    //Se cierra la interfaz
    this->close();
}
//Fin de función para cerrar interfaz//

//Función para recibir del socket//
void MainWindow::recibir(QString comando)
{
    //Si se recibe comando para obtener escena de cliente
    if(comando=="SUP")
    {
        //Se notifica a la interfaz del comando recibido
        ui->statusBar->showMessage("Orden Visualizar recibida",5000);
        //Se envía una imagen de la cámara
        enviarImg();
    }
    //Si se recibe comando para desconectar al cliente
    if(comando=="DES")
    {
        //Se notifica a la interfaz del comando recibido
        ui->statusBar->showMessage("Orden Desconectar recibida", 5000);
        //Se cierra la conexión del cliente
        socket->disconnect();
        //Se notifica a la interfaz
        ui->statusBar->showMessage("Desconectado",5000);
    }
    //Si se recibe el comando de detener
    if(comando=="DET")
    {
        //Se notifica a interfaz del comando recibido
        ui->statusBar->showMessage("Orden Detener recibida", 5000);
        //Se cierra la conexión del cliente
        socket->disconnect();
        //Se notifica a la interfaz
        ui->statusBar->showMessage("Desconectado",5000);
    }
}
//Fin de función para recibir del socket//
```

```
//Función para enviar una imagen//
void MainWindow::enviarImg()
{
    //Si existe una estructura de captura activa
    if(img)
    {
        //Se obtiene el nombre de la imagen
        QString nom = ruta +"Cap" + QString::number(cap) + "_" + Nombre
+".jpeg";
        //Se guarda la imagen
        cvSaveImage(nom.toAscii(),img);
        //Se envía la imagen
        socket->sendMessage(nom);
        cap++;
    }
}
//Fin de función para enviar una imagen//

//Función para establecer estado inicial//
void MainWindow::Default()
{
    //Se limpia el contenido de las variables
    Ip="";
    Puerto="";
    Nombre="";
    cap=0;

    //Si no existe el directorio de almacenamiento
    if(! (QDir("../C/Capturas/").exists()))
    {
        //Se crea el directorio de almacenamiento
        QDir().mkdir("../C/Capturas/");
    }

    //Se establece la ruta del directorio de almacenamiento
    ruta = "../C/Capturas/";
    socket->sruta = ruta;
}
//Fin de función de estado inicial//

//Función para notificar un error//
void MainWindow::error(sock*, QString titulo,QString mensaje)
{
    //Se notifica el error en interfaz
    ui->statusBar->showMessage("ERROR:"+titulo + ">> "+ mensaje,5000);
}
//Fin de función para notificar error//

//Función para notificar mensaje en interfaz//
```

```
void MainWindow::statusMensaje(QString msj)
{
    //Si el servidor se conecto con éxito
    if(msj.startsWith("Conectado exitosamente"))
    {
        //Se establece la conexión como activa
        estadoclt=true;
    }
    //Si se desconecto el cliente
    if(msj.startsWith("Desconectado"))
    {
        //Se establece la conexión como inactiva
        estadoclt=false;
    }
    //Se muestra el mensaje en la interfaz
    ui->statusBar->showMessage(msj,5000);
}
//Fin de función para notificar mensaje en interfaz//

//Función para enviar imagen de movimiento//
void MainWindow::recibirNom(QString nom)
{
    //Si hay conexión con servidor
    if(estadoclt)
    {
        //Se envía mensaje con nombre de imagen
        socket->sendMessage(nom + ".jpeg");
        //Se notifica del envío de imagen en interfaz
        ui->statusBar->showMessage("Enviando imagen "+nom+".jpeg" + " al
Servidor",5000);
    }
    //Si no hay una conexión activa
    else
    {
        //Se notifica que no hay una conexión en interfaz
        ui->statusBar->showMessage("No hay conexion con el Servidor",
5000);
    }
}
//Fin de función para enviar imagen de movimiento//

//Función para desplegar ventana de configuración avanzada//
void MainWindow::on_actionAvanzado_triggered()
{
    //Se crea una instancia de clase winavanzado
    avanz = new winavanzado(this);

    //Se muestran en la ventana las variables por default
    //de tiempo de retardo, cuadros por segundo y sensibilidad
```

```
    avanz->leiniseg->setText(QString::number(detec->retardo));
    avanz->lefps->setText(QString::number(detec->fps));
    avanz->slidesensi->setValue(detec->sens);

    //Si el detector está en ejecución
    if(activo)
    {
        //deshabilita la los campos para editarlos
        avanz->leiniseg->setEnabled(false);
        avanz->lefps->setEnabled(false);
        avanz->slidesensi->setEnabled(false);
    }
    //Si el detector está inactivo
    else
    {
        ///Se puede editar y guardar la nueva configuración
        avanz->leiniseg->setEnabled(true);
        avanz->lefps->setEnabled(true);
        avanz->slidesensi->setEnabled(true);
    }
    //Se espera una señal para recibir la nueva configuración

connect(avanz,SIGNAL(enviaravanz(int,double,int)),this,SLOT(configavanz(i
nt,double,int)));
    //Se muestra la ventana creada
    avanz->show();
}
//Fin de función para desplegar ventana de configuración avanzada//

//Función para asignar la configuración avanzada//
void MainWindow::configavanz(int in, double fr, int sns)
{
    //Asignar los valores recibidos de
    //retardo, cuadros por segundo y sensibilidad
    detec->retardo = in;
    detec->fps = fr;
    detec->sens = sns;
}
//Fin de función para asignar configuración avanzada//
///////////////////////////////////////////////
//////////////////////////////////////////////

///////////////////////////////////////////////
//////////////////SOCK.H/////////////////////
#ifndef SOCK_H
#define SOCK_H

//Clases necesarias
#include <QObject>
```

```
#include <QTcpSocket>
#include <QFile>

//Inicio de clase sock//
class sock : public QObject
{
   Q_OBJECT
public:
        //Constructor de clase sock
        sock();
        //Socket tipo tcp para la comunicación
        QTcpSocket *socket;
        //Referencia a archivo imagen
        QFile *img;
        //Buffer para leer el socket
        QByteArray message;
        //Nick asociado a la conexión
        QString nick;
        //Ruta de almacenamiento
        QString sruta;
        //Estado de la conexión
        bool estaConectado;
        //Función que implementa módulo Envío
        void sendMessage(QString);
        //Función para conectarse al servidor
        void conectarse(QString host,int port,QString nick);

private:
        //Función para procesar un comando recibido
        void parseMessage(QString msg);

private slots:
        //Función para recibir datos del socket
        void recv();
        //Función para confirmar conexión
        void conectado();
        //Función para mostrar un error de conexión
        void displayError(QAbstractSocket::SocketError socketError);

public slots:
        //Función para cerrar la conexión
        void disconnect();

signals:
        //Señal para enviar un nuevo mensaje
        void newMessage(QString);
        //Señal para establecer la conexión
        void connected(sock*, QString);
        //Señal para cerrar la conexión
```

```
        void disconnected(sock*, QString);
        //Señal para informar un error de conexión
        void errorconec(sock*, QString,QString);
        //Señal para notificar un mensaje en interfaz
        void statusBar(QString);
    };
//Fin de clase sock//

#endif
///////////////////////////////////////////////
///////////////////////////////////////////////

///////////////////////////////////////////////
///////////////SOCK.CPP//////////////////////
//Definición de clase sock
#include "sock.h"

//Constructor de clase sock//
sock::sock()
{
    //Se crea una instancia de socket tcp
    socket = new QTcpSocket(this);
    //Se conectan las señales recibidas con sus respectivas funciones
    connect(socket, SIGNAL(connected()), this, SLOT(conectado()));
    connect(socket, SIGNAL(readyRead()), this, SLOT(recv()));
    connect(socket, SIGNAL(disconnected()), this, SLOT(disconnect()));
    connect(socket, SIGNAL(error(QAbstractSocket::SocketError)),
            this, SLOT(displayError(QAbstractSocket::SocketError)));

    //Se inicializan las variables a su valor por default
    nick="";
    sruta="";
    estaConectado =false;
}
//Fin de constructor de clase sock//

//Función para recibir datos del socket
void sock::recv()
{
        //Se leen los datos del socket
        message.append(socket->readAll());

        int pos;
        //Se obtiene el comando del mensaje
        while((pos = message.indexOf("\n\r")) > -1)
        {
                //Se procesa el comando
                parseMessage(QString(message.left(pos+2)));
                message = message.mid(pos+2);
```

```
        }
}
//Fin de función recibir datos del socket//

//Función para conectarse al servidor//
void sock::conectarse(QString host, int port,QString nick)
{
  //Se inicializa el socket
  socket->abort();
  //Se asigna el nick recibido
  this->nick = nick;
  //Se intenta conectar con ip y puerto recibido
  socket->connectToHost(host, port);
}
//Fin de función para conectarse al servidor//

//Función para confirmar conexión//
void sock::conectado()
{
    //Se envía comando de conexión al servidor
    sendMessage("CON:" + nick + "\n\r");
    //Se establece la conexión como activa
    estaConectado = true;
    //Se notifica a la interfaz del estado de la conexión
    emit statusBar("Conectado exitosamente");
}
//Fin de función para confirmar conexión

//Función para procesar un comando recibido//
void sock::parseMessage(QString msg)
{
       //Si el comando indica visualizar la escena del cliente
       if(msg.startsWith("SUP:"))
       {
           //Se envía una señal para obtener imagen de escena
           msg = msg.mid(4);
           msg = msg.mid(0,msg.length()-2);
           emit newMessage("SUP");
       }
       //Si el comando indica desconectar al cliente
       if(msg.startsWith("DES:"))
       {
           //Se envía una señal para cerrar su conexión
           msg = msg.mid(4);
           msg = msg.mid(0,msg.length()-2);
           emit newMessage("DES");
       }
       //Si el comando indica detener el cliente
       if(msg.startsWith("DET:"))
```

```
        {
            //Se envía una señal para detener el cliente
            msg = msg.mid(4);
            msg = msg.mid(0,msg.length()-2);
            emit newMessage("DET");
        }
}
//Fin función para procesar un comando recibido//

//////////Módulo Envío/////////
void sock::sendMessage(QString msg)
{
    //Si el socket esta activo y existe una conexión
    if (socket->isValid())
    {
        //Si se tiene la ruta de la imagen
        if(msg.contains(sruta))
        {
            //Se crea una referencia hacia la imagen
            img = new QFile(msg);
            //Se obtiene el tamaño de la imagen
            int tam = img->size();
            QString tama = QString::number(tam);
            //Se envía el tamaño al Servidor
            socket->write(tama.toAscii());
            socket->flush();
            //Se abre la imagen para ser leída
            img->open(QIODevice::ReadOnly);
            //Se envían los datos de la imagen
            socket->write(img->readAll());
            socket->flush();
            //Se elimina la referencia de la imagen
            delete img;
        }
        //Si se envía un comando
        else
        {
            //Se envía el comando al Servidor
            socket->write(msg.toAscii());
            socket->flush();
        }
    }
}
//////////Fin de módulo Envío//////////

//Función para cerrar la conexión//
void sock::disconnect()
{
     //Se establece la conexión como inactiva
```

```
     estaConectado = false;
     //Se limpia el nick
     nick = "";
     //Se cierra el socket de comunicación
     socket->close();
     //Se notifica a la interfaz del estado de conexión
     emit statusBar("Desconectado");
}
//Fin de función para cerrar la conexión//

//Función para mostrar un error de conexión//
void sock::displayError(QAbstractSocket::SocketError socketError)
{
    //Opción de error de socket
    switch (socketError)
    {
        //En caso de error del servidor
        case QAbstractSocket::RemoteHostClosedError:
            break;
        //En caso de que no haya encontrado el servidor
        case QAbstractSocket::HostNotFoundError:
          //Notificar el error
          emit errorconec(this, tr("DMR"),
                                 tr(" Servidor no encontrado verifique "
                                    " Direccion IP y Puerto."));
            break;
        //En caso de que la conexión no se establezca adecuadamente
        case QAbstractSocket::ConnectionRefusedError:
        //Notificar el error
        emit errorconec(this, tr("DMR"),
                                 tr(" Error de conexion. verifique "
                                    " Direccion IP y Puerto."));
            break;
        //Caso por default
        default:
        //Notificar el error específico
        emit errorconec(this, tr("DMR"),
                                 tr("Error: %1.")
                                 .arg(socket->errorString()));
    }
}
//Fin de función para mostrar un error de conexión//
/////////////////////////////////////////////////
/////////////////////////////////////////////////

/////////////////////////////////////////////////
////////////////WINAVANZADO.H////////////////
#ifndef WINAVANZADO_H
#define WINAVANZADO_H
```

```
//Clases necesarias
#include <QMainWindow>
#include <QLineEdit>
#include <QSlider>

//Usar interfaz winavanzado.ui
namespace Ui {
    class winavanzado;
}

//Inicio clase winavanzado//
class winavanzado : public QMainWindow
{
    Q_OBJECT

public:
    //Constructor y destructor de la clase
    explicit winavanzado(QWidget *parent = 0);
    ~winavanzado();

    //Referencias a elementos de la interfaz
    QLineEdit *leiniseg;
    QLineEdit *lefps;
    QSlider *slidesensi;

private:
    //Referencia a la interfaz winanvanzado.ui
    Ui::winavanzado *ui;

signals:
    //Señal para enviar configuración avanzada
    void enviaravanz(int in, double fr, int sns);

private slots:
    //Función para configuración avanzada
    void on_Bavanzaceptar_clicked();
};
//Fin de clase winavanzado//
#endif
///////////////////////////////////////////////
///////////////////////////////////////////////

///////////////////////////////////////////////
/////////////////WINAVANZADO.CPP/////////////
//Clases necesarias y
//definición de clase winavanzado
#include "winavanzado.h"
#include "ui_winavanzado.h"
#include <QMessageBox>
```

```
//Constructor de la clase winavanzado//
winavanzado::winavanzado(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::winavanzado)
{
    //Se inicializa la interfaz
    ui->setupUi(this);
    //Se asigna el tamaño de la ventana
    this->setFixedSize(282,275);

    //Se inicializan los elementos de la interfaz
    leiniseg = new QLineEdit("",this);
    leiniseg->setGeometry(20,80,41,21);
    this->lefps = new QLineEdit("",this);
    lefps->setGeometry(20,180,41,21);
    this->slidesensi = new QSlider(Qt::Vertical,this);
    slidesensi->setGeometry(210,70,16,151);
    slidesensi->setTickPosition(QSlider::TicksBothSides);
    slidesensi->setTickInterval(1);
    slidesensi->setMinimum(1);
    slidesensi->setMaximum(10);

    //Se espera una señal para cerrar la ventana
    connect(ui->Bavanzcancelar, SIGNAL(clicked()),this, SLOT(close()));
}
//Fin de constructor de clase winavanzado//

//Destructor de clase winavanzado//
winavanzado::~winavanzado()
{
    //Se elimina la referencia de la interfaz
    delete ui;
}
//Fin de destructor de la clase winavanzado//

//Función para configuración avanzada//
void winavanzado::on_Bavanzaceptar_clicked()
{
    //Si los campos de la ventana están vacios
    if(    leiniseg->text().trimmed()==""
       || lefps->text().trimmed()=="")
    {
        //Se lanza un cuadro de dialogo error
         QMessageBox::information(this,"Campos vacios "," Segundos o Fps
vacios");
    }
    //Si los campos se han llenado
    else
```

```
    {
        //Se envía la señal para establecer la nueva configuración
        emit enviaravanz(leiniseg->text().toInt(),lefps-
>text().toDouble(),slidesensi->value());
        //Se cierra la ventana
        close();
    }
}
//Fin de función para configuración avanzada//
///////////////////////////////////////////////
///////////////////////////////////////////////

///////////////////////////////////////////////
////////////////WINCONEX.H////////////////////
#ifndef WINCONEX_H
#define WINCONEX_H
//Clases necesarias
#include <QMainWindow>
#include <QMessageBox>
#include <QLineEdit>

//Uso de la interfaz winconex.ui
namespace Ui {
    class winconex;
}

//Inicio de clase winconex//
class winconex : public QMainWindow
{
    Q_OBJECT

signals:
    //Señal para enviar los datos de la configuración
    void enviarconfig(QString _ip, QString _puerto, QString _nick);

public:
    //Constructor y destructor de la clase
    explicit winconex(QWidget *parent = 0);
    ~winconex();

    //Elementos de la interfaz
    QLineEdit *leipservidor;
    QLineEdit *lepuerto;
    QLineEdit *lenick;

private:
    //Referencia a interfaz de la clase winconex
    Ui::winconex *ui;
```

```
private slots:
    //Función para establecer la configuración
    void on_bconxaceptar_clicked();
};
//Fin de clase winconex//

#endif
///////////////////////////////////////////////
///////////////////////////////////////////////

///////////////////////////////////////////////
//////////////WINCONEX.CPP//////////////////
//Definición de clase winconex//
#include "winconex.h"
#include "ui_winconex.h"

//Constructor de la clase winconex//
winconex::winconex(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::winconex)
{
    //Se inicializa la interfaz
    ui->setupUi(this);
    //Se asigna el tamaño de la ventana
    this->setFixedSize(236,175);

    //Se inicializan los elementos de la interfaz
    leipservidor = new QLineEdit("",this);
    leipservidor->setGeometry(90,40,131,27);
    lepuerto = new QLineEdit("",this);
    lepuerto->setGeometry(90,70,81,27);
    lenick = new QLineEdit("",this);
    lenick->setGeometry(90,100,131,27);

    //Se espera una señal para cerrar la ventana
    connect(ui->bconxcancelar, SIGNAL(clicked()),this, SLOT(close()));
}
//Fin de constructor de la clase winconex//

//Destructor de la clase winconex//
winconex::~winconex()
{
    //Se elimina la referencia a la interfaz
    delete ui;
}
//Fin de destructor winconex//

//Función para establecer la configuración//
void winconex::on_bconxaceptar_clicked()
```

```
{
    //Si los campos de la ventana están vacios
    if(   leipservidor->text().trimmed()==""
       || lepuerto->text().trimmed()==""
       || lenick->text().trimmed()=="")
    {
        //Se lanza un cuadro de dialogo error
         QMessageBox::information(this,"Campos vacios ","Nick, Server o
puerto vacios");
    }
    //Si los campos se han llenado
    else
    {
        //Se envía una señal para establecer la nueva configuración
        emit enviarconfig(leipservidor->text(), lepuerto->text(), lenick-
>text());
        //Se cierra la ventana
        close();
    }
}
//Fin de función para establecer la configuración//
///////////////////////////////////////////////
///////////////////////////////////////////////

///////////////////////////////////////////////
///////////////WININFO.H////////////////////
#ifndef WININFO_H
#define WININFO_H
//Clase necesaria
#include <QMainWindow>

//Uso de interfaz wininfo.ui
namespace Ui {
    class wininfo;
}

//Inicio de clase wininfo//
class wininfo : public QMainWindow
{
    Q_OBJECT

public:
    //Constructor y destructor de la clase
    explicit wininfo(QWidget *parent = 0);
    ~wininfo();

private:
    //Referencia a interfaz wininfo.ui
    Ui::wininfo *ui;
```

```
};
//Fin de la clase wininfo//
#endif
//////////////////////////////////////////////
//////////////////////////////////////////////

//////////////////////////////////////////////
////////////////WININFO.CPP//////////////////////
//Definición de clase wininfo//
#include "wininfo.h"
#include "ui_wininfo.h"

//Constructor de la clase wininfo//
wininfo::wininfo(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::wininfo)
{
    //Se inicializa la interfaz
    ui->setupUi(this);
    //Se asigna el tamaño de la ventana
    this->setFixedSize(396,179);

    //Se espera una señal para cerrar la ventana
    connect(ui->binfoaceptar, SIGNAL(clicked()),this, SLOT(close()));
}
//Fin de constructor de clase wininfo//

//Destructor de clase wininfo//
wininfo::~wininfo()
{
    //Elimina referencia a interfaz
    delete ui;
}
//Fin de destructor de la clase wininfo//
//////////////////////////////////////////////
//////////////////////////////////////////////
```

ANEXO B. MANUAL DE INSTALACIÓN Y EJECUCIÓN

Este manual explica cómo instalar y ejecutar el sistema DMR, el contenido del mismo se divide en dos partes que son Instalación y Ejecución.

Instalación

Para poder poner en funcionamiento el sistema DMR es necesario instalar previamente tres principales herramientas:

1. ***Compilador* para C++**.
2. Entorno de Desarrollo Integrado **QTCreator**.
3. Biblioteca **OpenCV**.

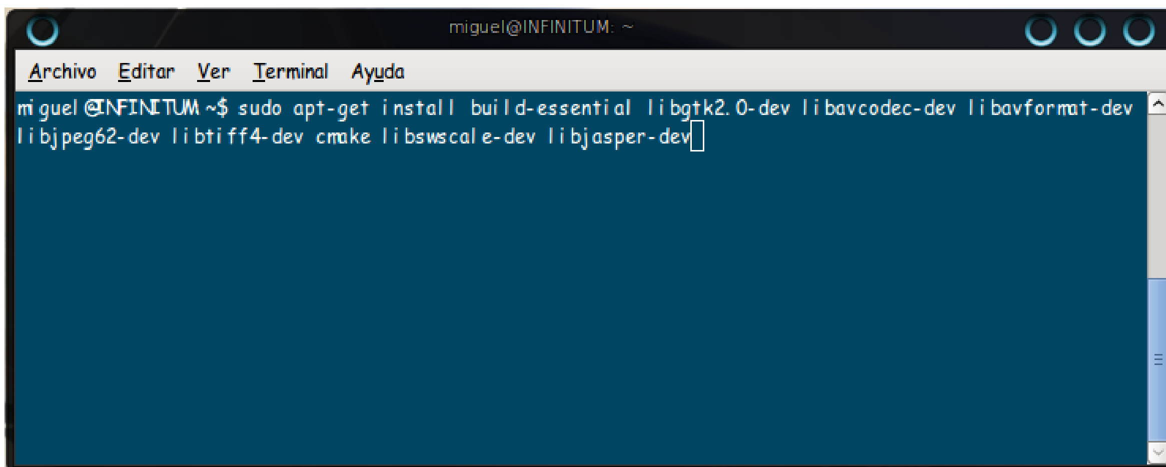
A continuación se muestran los pasos a seguir para instalar las herramientas anteriores.

1. Instalar el ***Compilador* para C++** y las dependencias útiles.

Para poder compilar y ejecutar nuestros códigos en lenguaje C++ se debe instalar el paquete denominado "*build-essential*". Lo que este paquete contiene es una lista informativa de los paquetes considerados esenciales para la creación de paquetes Debian. Este paquete incluye entre otras cosas los *compiladores gcc/g++* que es lo que necesitamos.

Para su instalación simplemente abrimos una terminal y tecleamos la siguiente instrucción:

```
$ sudo apt-get install build-essential libgtk2.0-dev libavcodec-dev libavformat-dev libjpeg62-dev libtiff4-dev cmake libswscale-dev libjasper-dev
```

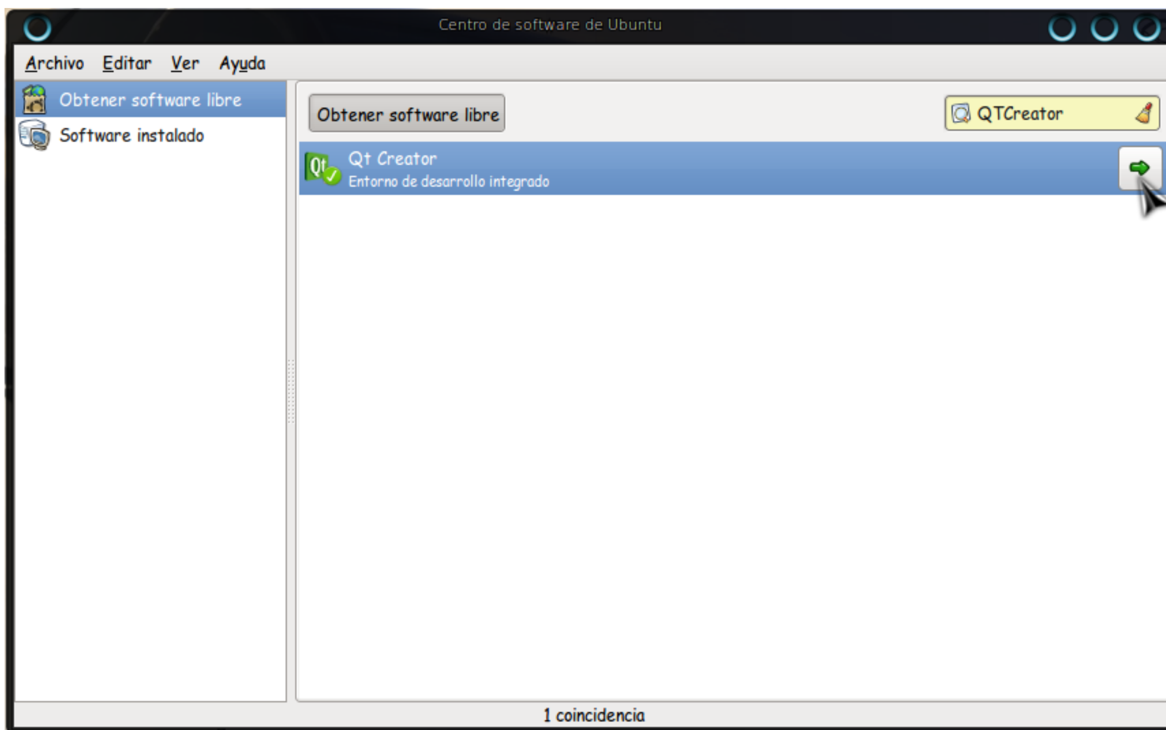


Nota: Se debe ingresar la contraseña de súper usuario o administrador en caso de que se requiera.

Si la instrucción anterior se ejecuta correctamente, el paquete comenzará a instalarse junto con sus dependencias. Al terminar la instalación ya podemos compilar códigos en lenguaje C++.

2. Instalar **QTCreator.**

Para instalar esta herramienta basta con ir a el *Centro de Software de Ubuntu*, dentro del menú *Aplicaciones* e introducir en el recuadro de búsqueda de la parte superior derecha la palabra "*QTCreator*". Una vez localizado el entorno, seleccionar y dar click en el botón para instalar.

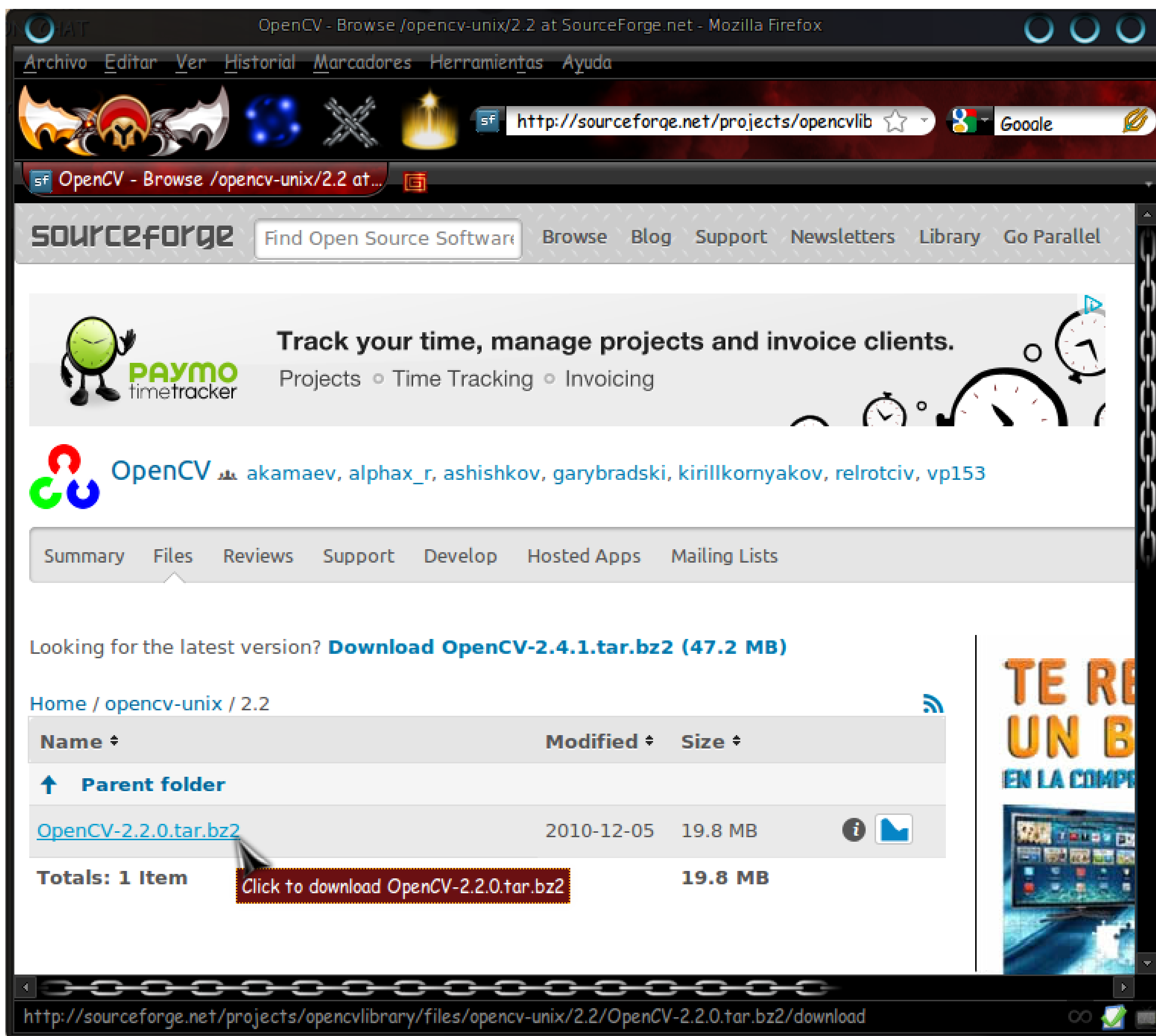


Con estos simples pasos instalamos nuestro IDE, que se encargará de ejecutar nuestro sistema de cliente o servidor de manera rápida y fácil.

3. Instalar biblioteca **OpenCV**.

Para la instalación de la biblioteca OpenCV se requiere tener previamente descargado el paquete, en este caso la versión 2.2 de la página:

http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/2.2/



El paquete consiste de un archivo comprimido *.tar.bz2*.Una vez descargada la biblioteca, debemos moverla a la carpeta personal del usuario, abrir una terminal y extraer el contenido del paquete con el siguiente código:

```
$ tar xfv OpenCV-2.2.0.tar.bz2
```

Entrar al directorio extraído en el paso anterior:

```
$ cd OpenCV-2.2.0
```

Creamos un nuevo directorio en donde construiremos nuestro paquete:

```
$ mkdir opencv.build
```

Entramos al directorio creado con:

```
$ cd opencv.build
```

Configuramos el paquete a construir:

```
$ cmake ..
```

Construimos el paquete:

```
$ make
```

Instalamos el paquete:

```
$ sudo make install
```

Una vez instalada nuestra biblioteca es necesario configurarla para que se pueda ejecutar de manera correcta para ello editamos el archivo *opencv.conf*:

```
$ sudo gedit /etc/ld.so.conf.d/opencv.conf
```

Y agregamos la siguiente línea de texto:

```
/usr/local/lib
```

Guardamos los cambios y cargamos la configuración para que tenga efecto con:

```
$ sudo ldconfig
```

Editamos el archivo *bash.bashrc* con la siguiente instrucción:

```
$ sudo gedit /etc/bash.bashrc
```

Y agregamos al final del contenido las líneas de texto siguientes:

```
PKG_CONFIG_PATH=$PKG_CONFIG_PATH:/usr/local/lib/pkgconfig
    export PKG_CONFIG_PATH
```

Guardamos los cambios y cerramos. Con estos pasos ejecutados adecuadamente ya tenemos lo necesario para ejecutar nuestro sistema en nuestra computadora.

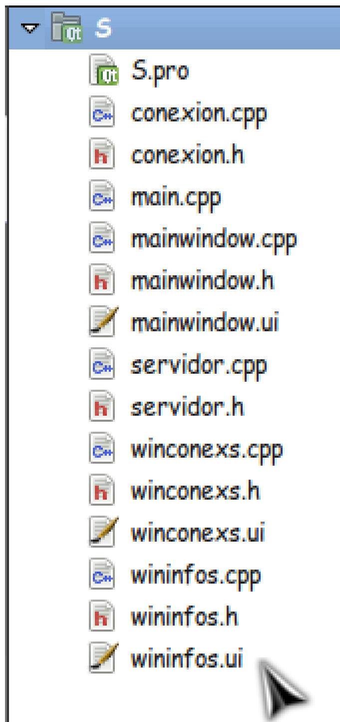
Ejecución

Para ejecutar correctamente nuestro sistema, ya sea la parte del servidor o cliente es necesario tener los archivos para cada aplicación, abrir nuestro IDE *QTCreator* anteriormente instalado. *QTCreator* se encuentra ubicado en el menú *Aplicaciones*, dentro del apartado *Programación*. A continuación se muestran los archivos necesarios en cada aplicación para una ejecución adecuada.

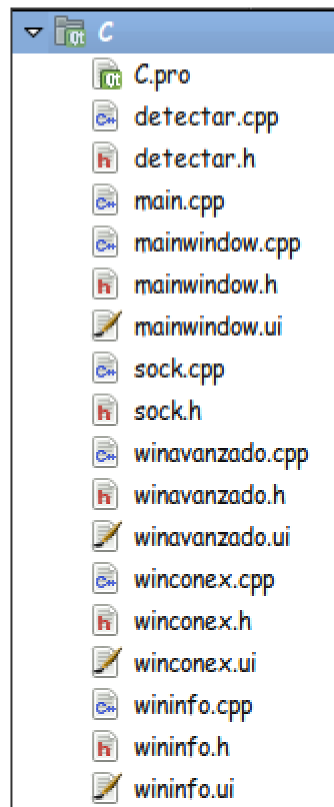
Para abrir el proyecto damos click en *Archivo*, *Abrir* y seleccionamos nuestro proyecto con la extensión *.pro*.

Al abrir el proyecto se abrirán también los archivos *.h*, *.cpp* y los archivos *.ui* relacionados al proyecto. Estos son los elementos que conforman nuestro proyecto y de alguna manera son necesarios para una correcta ejecución del mismo. En la siguiente imagen se muestran el total de archivos que debe tener el servidor y el cliente.

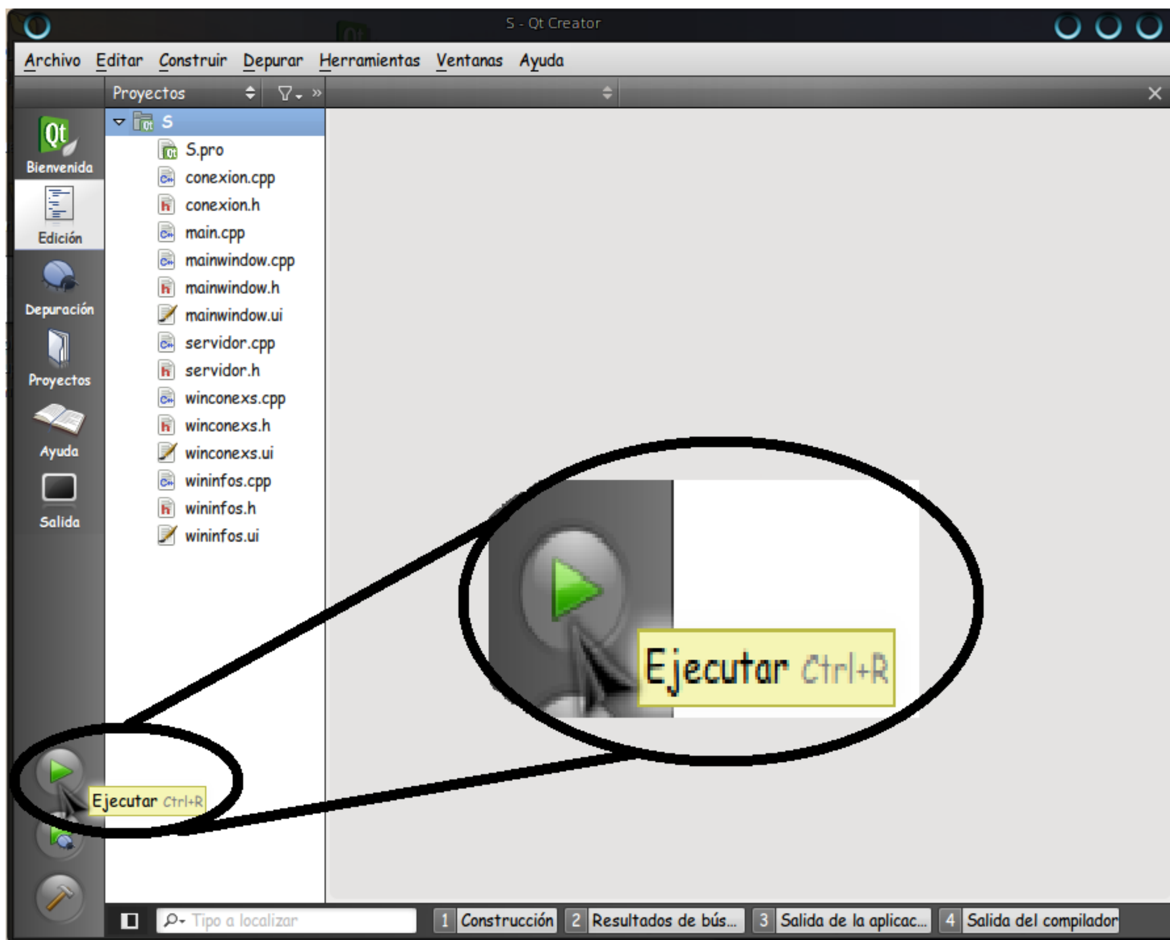
Servidor:



Cliente:



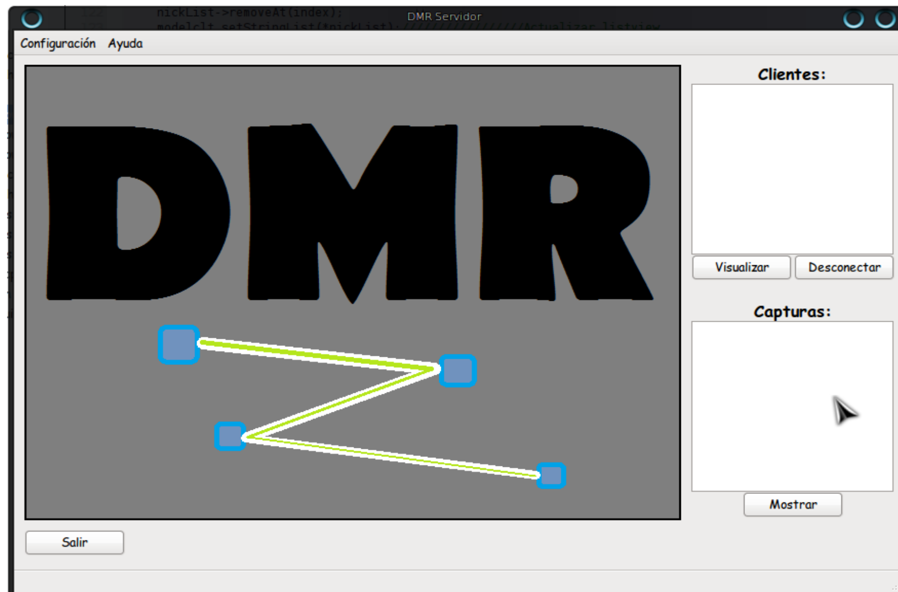
Finalmente para ejecutar un proyecto, en este caso el Cliente o el Servidor, basta con tener abierto el archivo *.pro* en nuestro IDE *QTCreator* y dar click en el menú *Construir* y dar un click en la opción *Construir todo,* seguido del botón ejecutar, representado por una flecha verde en la parte inferior izquierda de la pantalla principal.



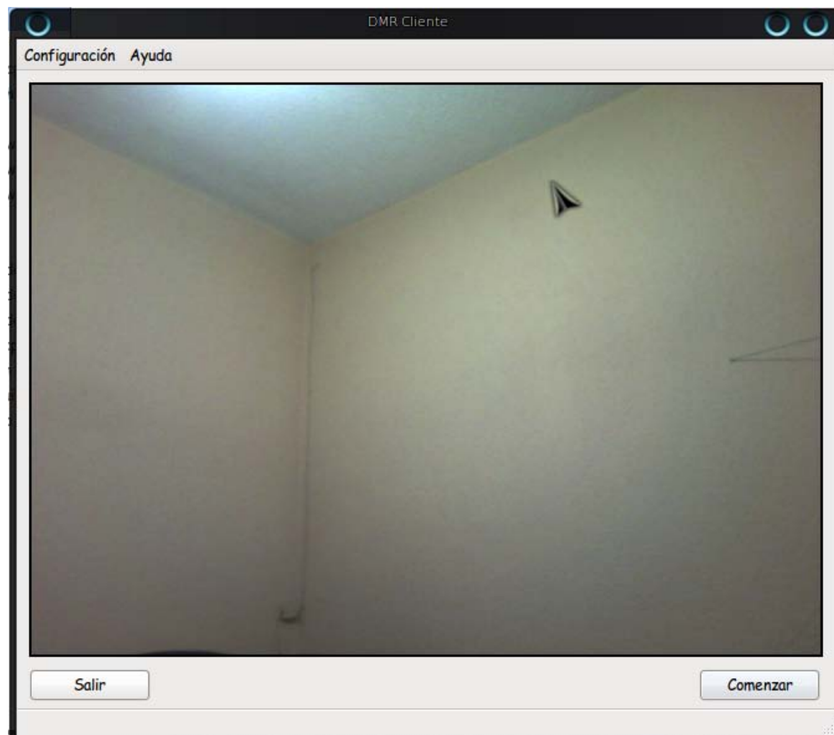
Una vez hecho esto, el proyecto comenzará a construirse y en caso de construirse de manera correcta, el IDE mostrará la pantalla principal de nuestro sistema DMR.

A continuación se muestran las pantallas principales del Cliente y Servidor, lanzadas por el IDE después de haber sido construido y ejecutado el proyecto.

Pantalla principal del DMR Servidor:

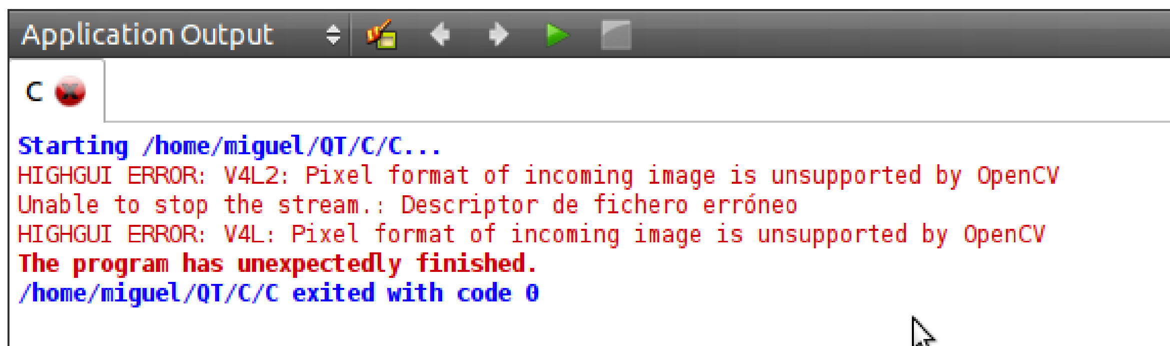


Pantalla principal del DMR Cliente:



Nota: Algunas *cámaras web* no trabajan bien con la API de captura de video *V4L2* (Video For Linux 2), por lo cual se lanza un error dentro del IDE al tratar de ejecutar el Cliente, que es el encargado de la captura de video. En seguida se muestra el código del error.

Error:



La solución más sencilla es utilizar la primer versión V4L1, exportándola y ejecutando nuestro Cliente fuera del IDE con los siguientes pasos:

1. Abrimos una terminal y nos dirigimos a donde tenemos nuestro Cliente.
2. Exportamos la API V2L, con la siguiente instrucción en la terminal:
   ```
   $ export LD_PRELOAD=/usr/lib/libv4l/v4l1compat.so
   ```
3. Ejecutamos nuestro Cliente con:
   ```
   $./C
   ```
   Donde *C* es el nombre de nuestro ejecutable.

ANEXO C. MANUAL DE USUARIO

Este manual explica cómo usar el sistema DMR. Como ya sabemos, nuestro sistema se divide en dos aplicaciones que corresponden al Cliente y al Servidor.

Una vez ejecutada nuestra aplicación Servidor, se mostrará la interfaz principal, en donde se tienen tres secciones. La primera sección se trata de un recuadro en donde el usuario podrá ver las imágenes obtenidas de los clientes. Una segunda sección es la lista de clientes conectados al servidor. Y finalmente una sección en donde se tendrán las capturas anteriormente hechas para ser mostradas en el momento que el usuario lo desee. En la **Imagen A** se observa dicha interfaz.

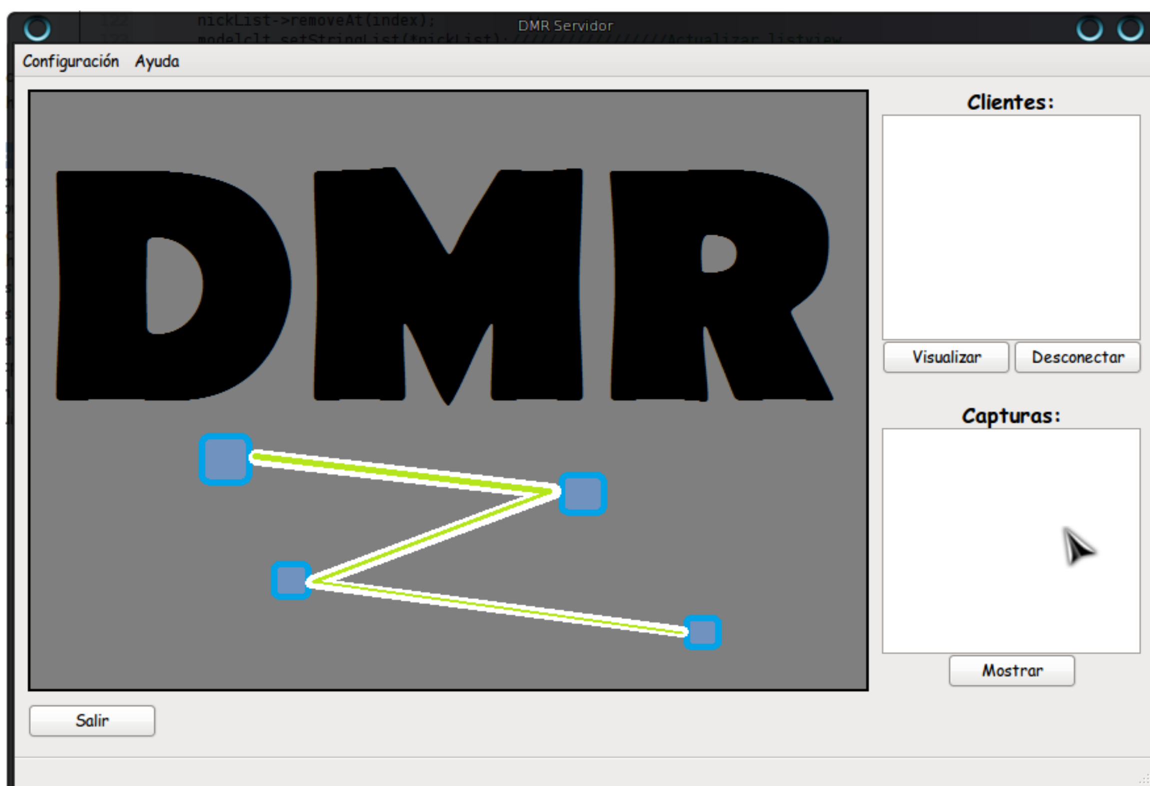
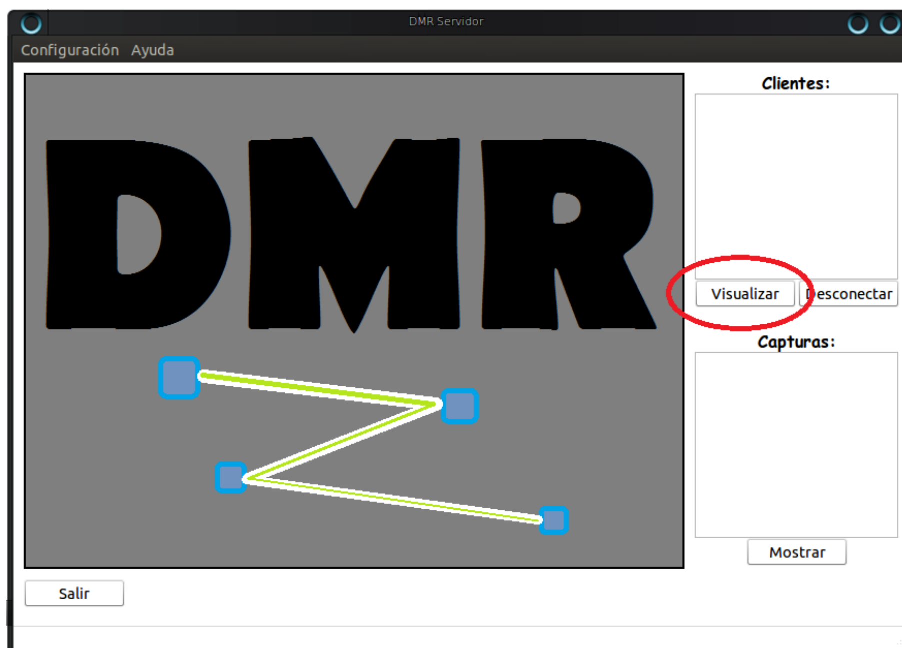


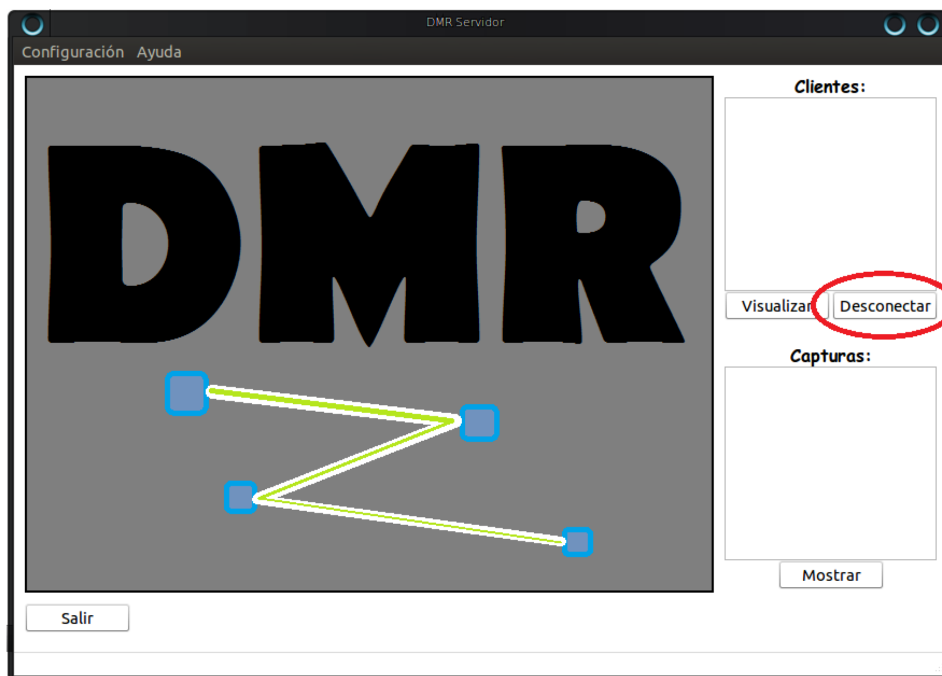
Imagen A. Interfaz del Servidor DMR.

Como se puede observar en la imagen anterior, también se cuenta con dos menús en la parte superior de la ventana y cinco botones de acción dentro de nuestra interfaz. A continuación se describe el funcionamiento de cada botón y el flujo de ventanas que llevan a la conexión de un cliente al servidor.

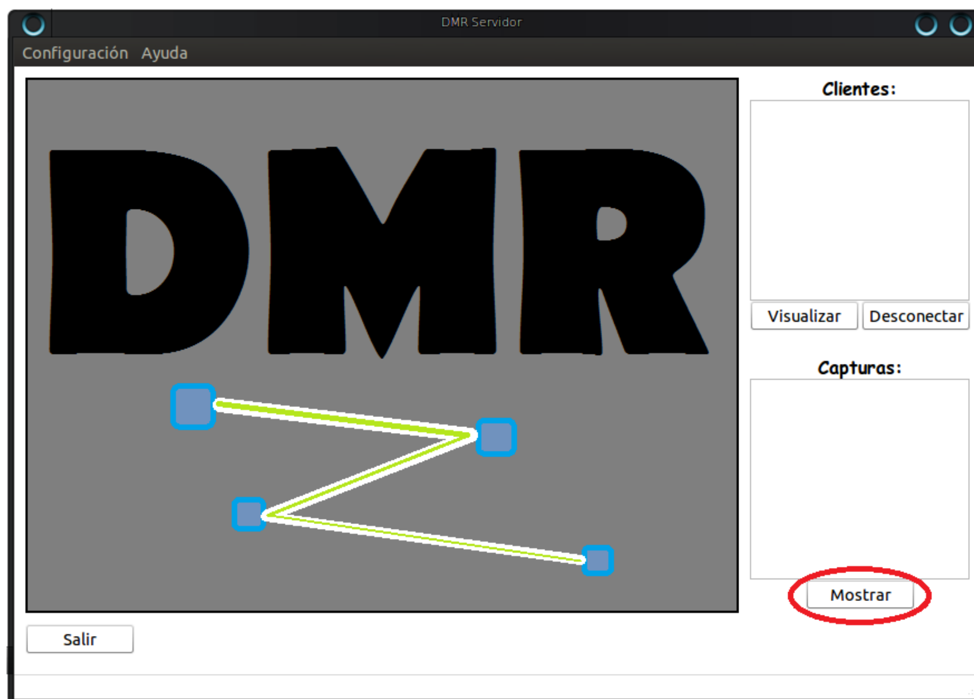
El botón *Visualizar* nos permitirá solicitar una imagen de la escena del cliente seleccionado, dentro de la lista clientes.



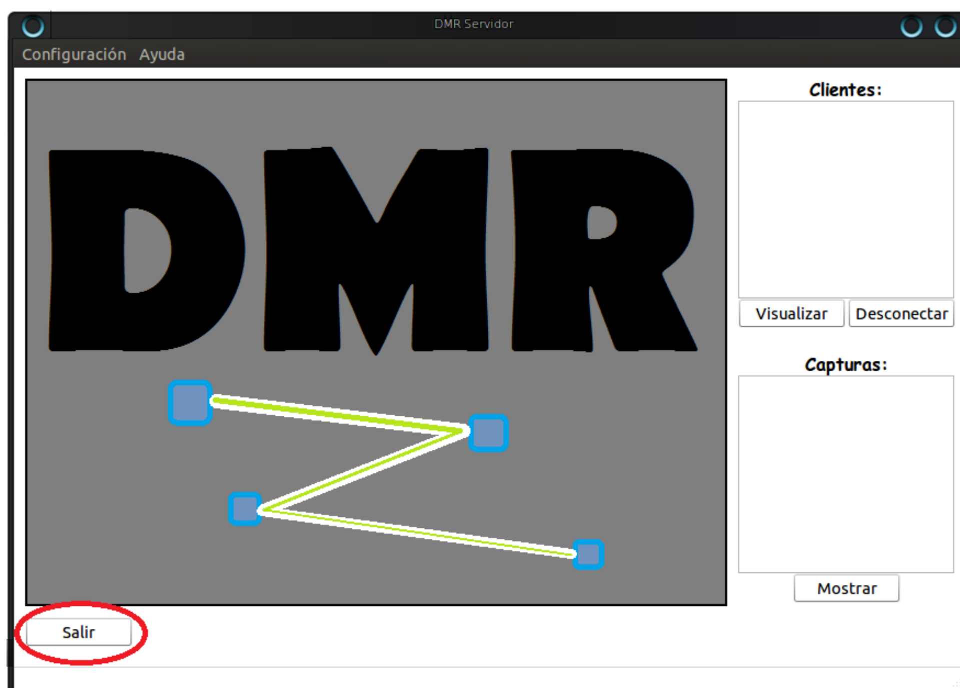
El botón de *Desconectar*, como su nombre lo indica, nos permitirá desconectar a un cliente seleccionado en la lista.



El botón *Mostrar* se utiliza para mostrar una imagen anteriormente recibida por un cliente, para ello basta con seleccionar la imagen dentro de la lista capturas y dar click en dicho botón.

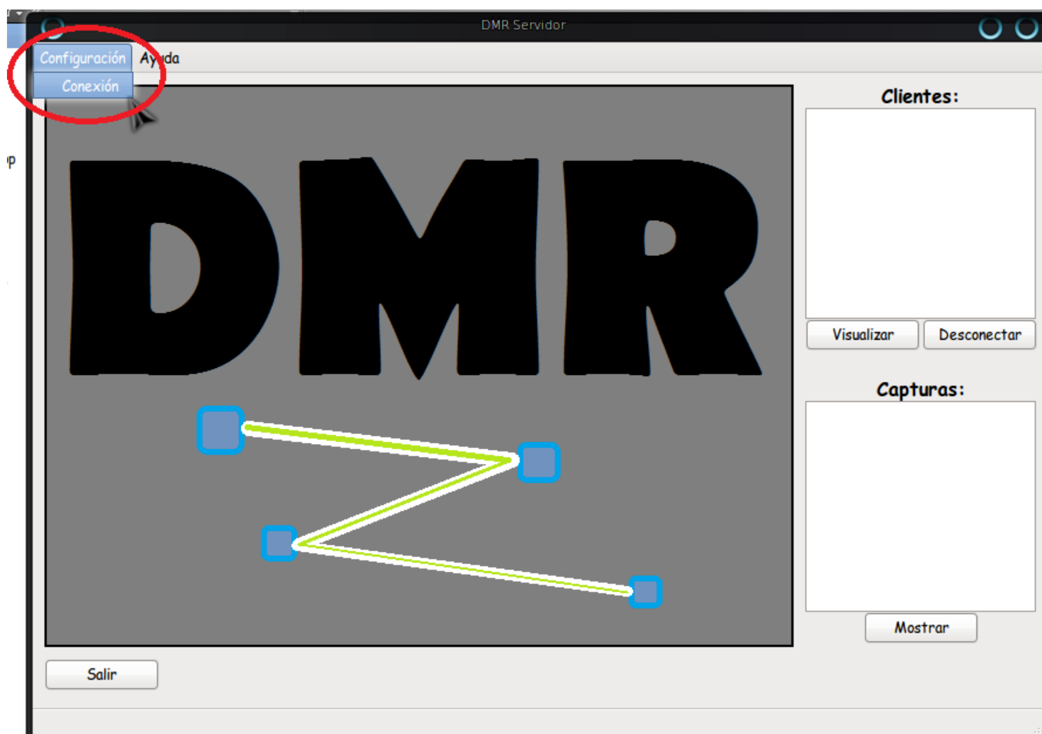


El botón *Salir*, cerrará la interfaz, terminando así con la conexión y ejecución del servidor. Al hacer esto todos los clientes se desconectarán automáticamente.



En la parte superior izquierda se observa el primer menú, que se trata de *Configuración*, al desplegar este menú se cuenta con la opción de conexión, que desplegará una nueva

ventana para establecer el puerto en el cual se iniciará nuestro Servidor. En la **Imagen B** se observa la nueva ventana de conexión.



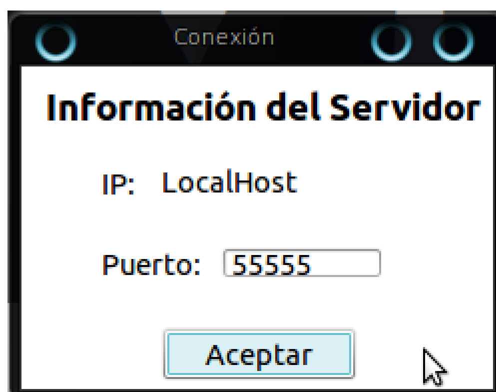
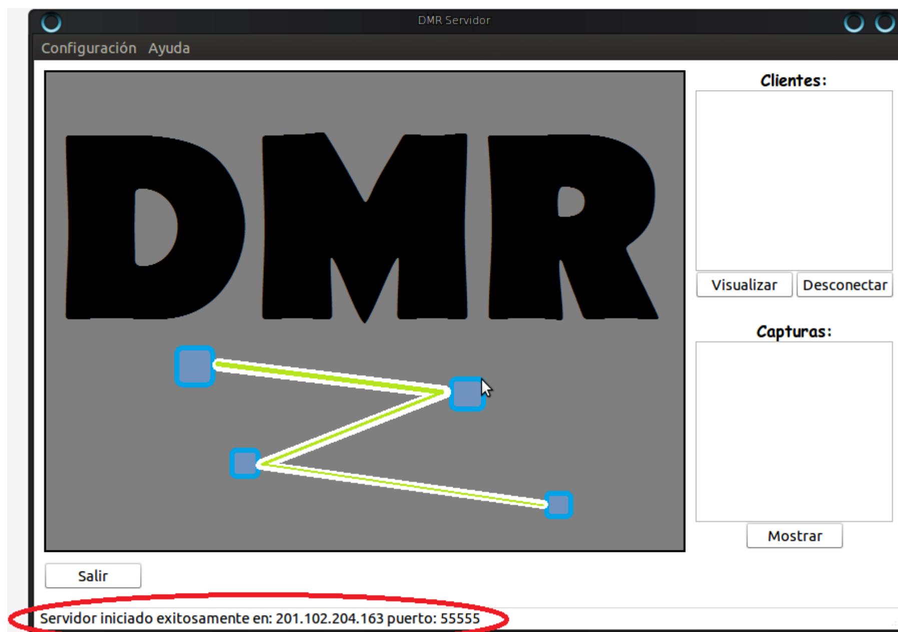


Imagen B. Ventana *Conexión* de Servidor.

En la ventana *Conexión*, el usuario puede asignar el puerto en el cual el servidor se iniciará para recibir las peticiones de conexión de los clientes. Cabe señalar que la dirección de la máquina en donde se ejecute el Servidor, necesariamente debe ser del tipo pública para que los clientes puedan verla mediante Internet.

Al hacer click en aceptar de la ventana *Conexión,* se informará en la interfaz principal en la barra de estado, el estado del servidor, así como el posible error que pueda ocurrir en el momento de la conexión.



Una vez iniciado nuestro servidor. Se ejecuta la aplicación Cliente en una computadora diferente, ubicada en el lugar a vigilar. En la **Imagen C** se puede observar la interfaz del Cliente.

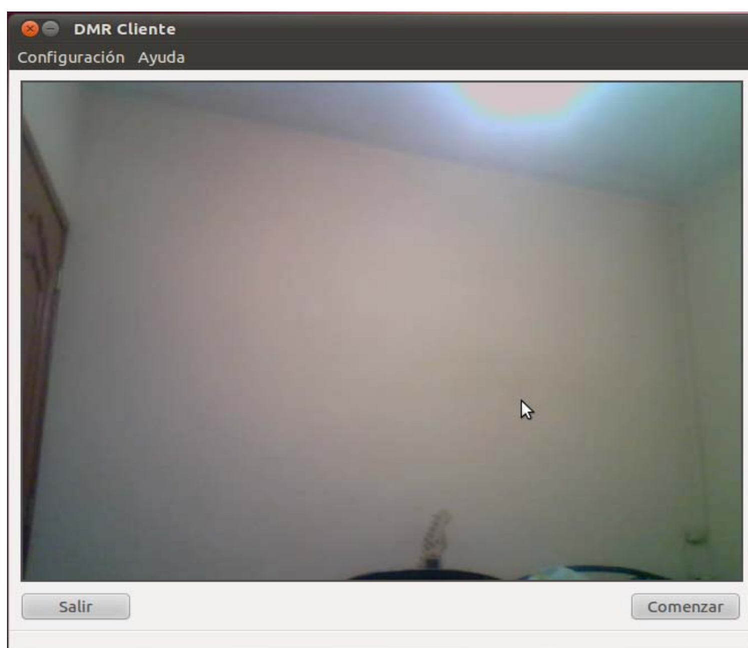
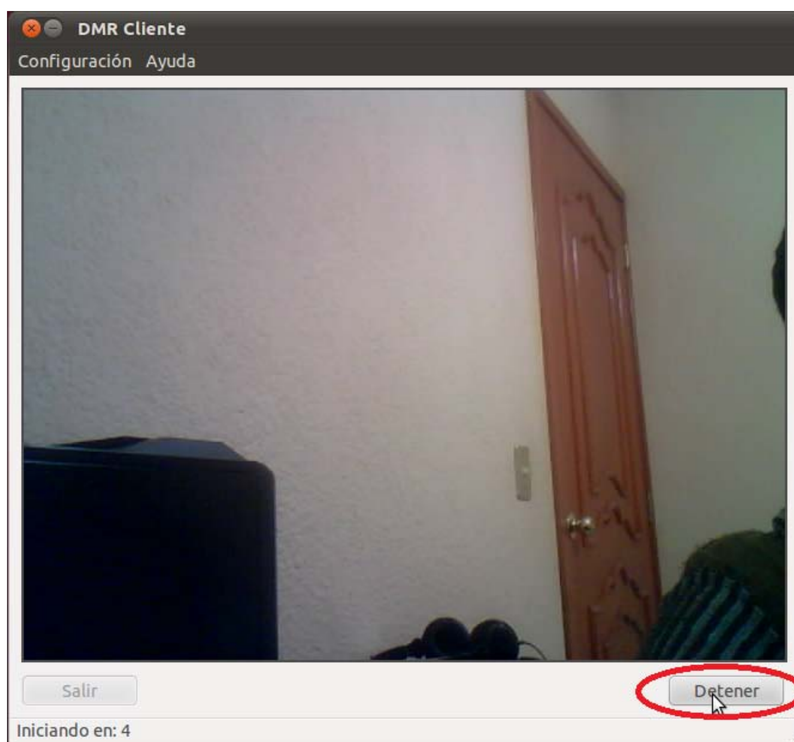


Imagen C. Interfaz del Cliente DMR.

Al igual que en la interfaz del Servidor, el Cliente cuenta con dos botones de acción y dos menús en la parte superior de la ventana. También se puede apreciar la captura de la escena.

Al igual que en la interfaz del servidor, el botón *Salir* del Cliente nos va a permitir cerrar la ventana de la aplicación y en caso de estar conectado al Servidor, de igual manera cerrará la conexión.

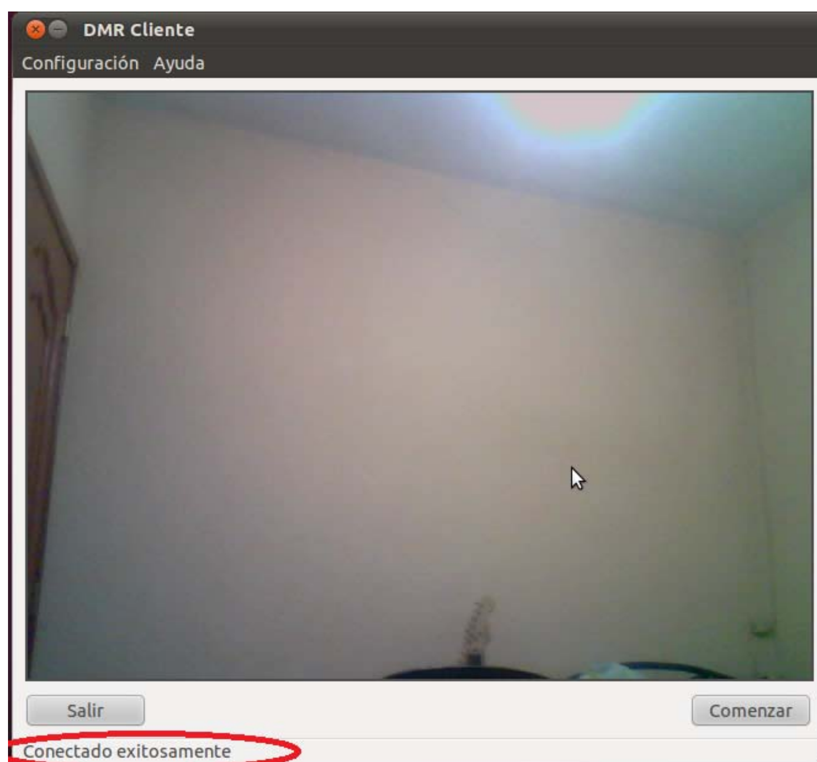
El segundo botón de acción *comenzar,* tiene la tarea de iniciar y detener la detección de movimiento. Al accionar este botón e iniciar con la detección de movimiento se puede observar que su texto cambiará a *Detener,* con lo cual significará que el detector esta activo. Al accionarlo por segunda vez detendrá el detector y el texto del botón cambiará a su estado normal.

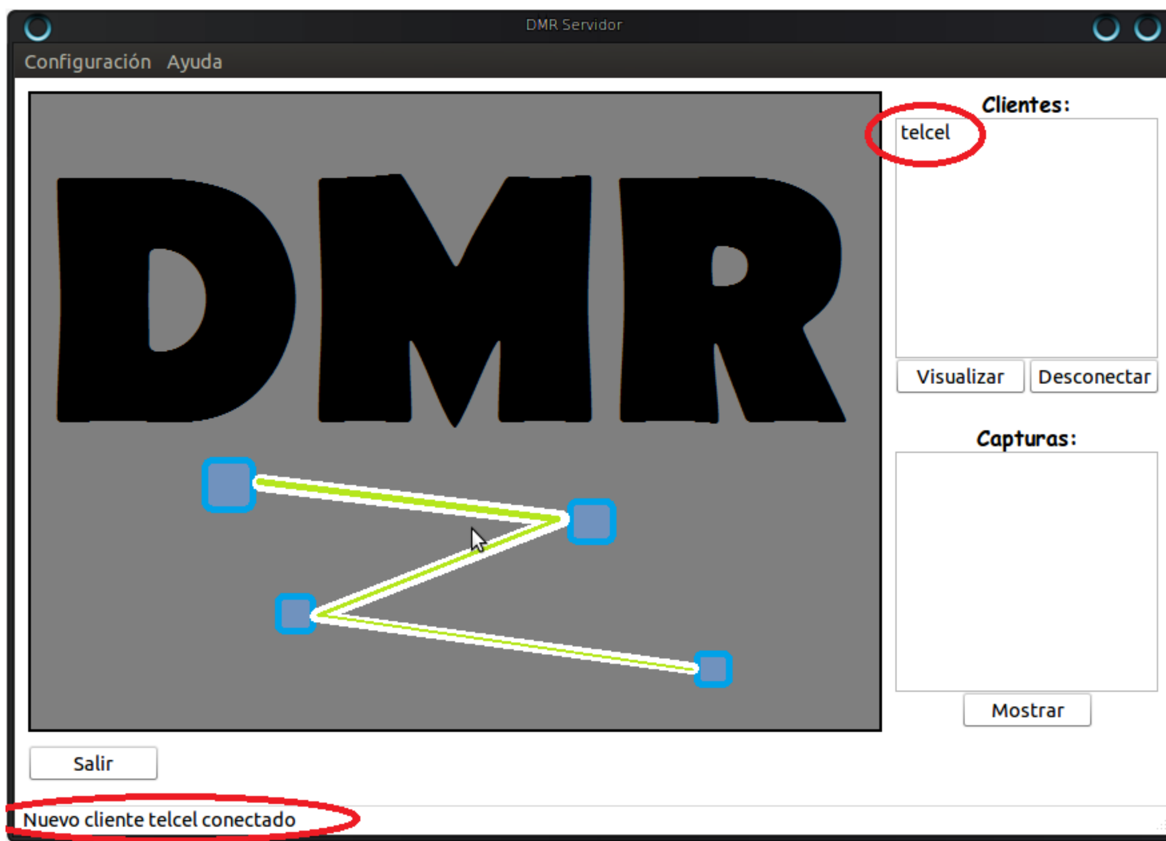


Para conectar el Cliente a nuestro Servidor, es necesario dar click al menú *Configuración* y elegir la opción *Conexión.* En seguida se mostrara una nueva ventana en donde se podrá ingresar la dirección IP del Servidor, el puerto ocupado y un Nick para identificar al Cliente. En la **Imagen D** se puede apreciar dicha ventana.

Imagen D. Ventana *Conexión* de Cliente.

Una vez llenados los campos de la ventana *Conexión,* se enlaza el Cliente con el Servidor y se informa del evento en ambas interfaces.





Como se puede observar en la imagen anterior, la interfaz Servidor se actualiza, agregando la nueva conexión en la lista clientes.

Con estos pasos, el Cliente y el Servidor quedan correctamente conectados para la detección de movimiento remoto.

GLOSARIO

Biblioteca: en ciencias de la computación, una biblioteca es un conjunto de subprogramas utilizados para desarrollar software, contienen código y datos, que proporcionan servicios a programas independientes**.**

Booleana: variable o contenedor en programación, en donde se pueden almacenar sólo dos valores, verdadero y falso.

BSD: (Berkeley Software Distribution) licencia de software libre. Está cerca del dominio público, permitiendo el uso del código fuente en software no libre.

Cámara web: cámara digital conectada a una computadora, la cual es capaz de capturar imágenes.

Comando: instrucción u orden que el usuario proporciona a un sistema informático, desde la línea de comandos o desde una llamada de programación.

Compilador: programa traductor que convierte un texto escrito en un lenguaje fuente de alto nivel en un programa objeto en código máquina.

Electrodo: conductor eléctrico utilizado para hacer contacto con una parte no metálica de un circuito.

Elemento Web: unidad modular de información que tiene contenido basado en la Internet.

GPL: (General Public License) licencia orientada principalmente a proteger la libre distribución, modificación y uso de software.

KDE: es un entorno de escritorio contemporáneo para estaciones de trabajo Unix.

Microcontrolador: circuito integrado o chip que incluye en su interior las tres unidades funcionales de una computadora: CPU, Memoria y Unidades de entrada y salida.

Multihilo: en sistemas operativos, es una característica que permite a una aplicación realizar varias tareas a la vez.

Multiplataforma: significa que el hardware o software tiene la característica de funcionar de forma similar en distintos sistemas operativos.

QPL: (Qt Public License) licencia creada por Trolltech para su edición gratuita de Qt.

Remoto: en tecnologías de la información, se define como sistema o elemento de sistema que se encuentran físicamente separados de una unidad central.

SDK: (Software Development Kit) Kit de desarrollo de software. Es un conjunto de herramientas y programas de desarrollo que permite al programador crear aplicaciones para un determinado paquete de software.

Sensor: dispositivo que detecta una acción o estimulo externo y genera una respuesta.

Socket: interfaz de programación de aplicaciones que permite la comunicación entre procesos.

Stream: término que hace referencia al hecho de transmitir un flujo de datos remotamente a través de una red en tiempo real.

Symbian: es un sistema operativo propietario, diseñado para dispositivos móviles, desarrollado por Symbian Ltd.

Trackbar: es también denominado en ocasiones control deslizante, se utiliza para navegar por grandes volúmenes de información o para ajustar visualmente una configuración numérica.

Transductor: dispositivo que convierte un tipo de energía en otra, transforma magnitudes físicas a eléctricaso magnitudes eléctricas a físicas.

UI: (Interfaz de Usuario) es el medio con que el usuario puede comunicarse con una máquina, un equipo o una computadora, y comprende todos los puntos de contacto entre el usuario y el equipo.