UNIVERSIDAD VERACRUZANA

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA

INTEGRANTES

Raúl Hernández Huerta

MATRICULAS

zS21020227

EXPERIENCIA EDUCATIVA

Programación de Redes

FACILITADOR

Yuliana Berumen Diaz

PROYECTO FINAL

TicketHub/Servicio Técnico

Entrega:

Boca del Rio, Veracruz

15 de diciembre 2025

Contenido

PROYECTO TICKETHUB-SERVICIO TÉCNICO

1. INTRODUCCION

El proyecto TicketHub nace como una continuación natural del trabajo que realicé el semestre pasado en la materia de Diseño de Aplicaciones Web, donde desarrollé una aplicación orientada a la gestión de servicios técnicos. En aquel proyecto descubrí que me sentía muy cómodo trabajando con flujos de atención de clientes, registro de equipos y seguimiento de problemas, por lo que decidí tomar esa misma línea para iniciar el desarrollo del sistema solicitado en Programación de Redes.

Elegí el tema de soporte técnico porque es algo que realmente me gusta y me resulta muy cercano a lo que hago en la vida real: revisar computadoras, limpiarlas por dentro y por fuera, formatear discos, reinstalar sistemas operativos, diagnosticar problemas de hardware y software, y dejar los equipos listos para que los usuarios puedan trabajar sin inconvenientes. Actualmente también estoy aprendiendo a reparar impresoras, lo que amplía el rango de fallas y dispositivos que puedo atender.

Por todo esto, un sistema de tickets de soporte no solo cumple perfectamente con los requisitos técnico-académicos de la materia—manejo de sockets TCP y UDP, servicios REST, base de datos relacional y arquitectura cliente-servidor—, sino que además me permite practicar la solución de un problema que realmente existe en entornos profesionales y que yo mismo podría llegar a necesitar en mi trabajo como técnico.

El nombre TicketHub fue elegido pensando en que representa un "hub" o centro neuralgico donde convergen todas las solicitudes de soporte tecnico que llegan de distintas partes. Cada usuario reporta un problema creando un ticket, y desde alli, un equipo de tecnicos puede acceder, trabajar y resolver esos tickets de manera coordinada.

2. OBJETIVOS Y ALCANCE

2.1 Objetivo General

Desarrollar una aplicación cliente-servidor que implemente una arquitectura de red multicapa utilizando sockets TCP/UDP y servicios REST para la gestión de tickets de soporte técnico, permitiendo la comunicación simultánea de múltiples dispositivos desde diferentes ubicaciones.

2.2 Objetivos Específicos

1. Implementar un servidor HTTP con API REST para la gestión transaccional de datos (CRUD).
2. Crear un servidor TCP multihilo para operaciones de cambio de estado en tiempo real.
3. Desarrollar un sistema de notificaciones UDP para informar cambios inmediatos.
4. Diseñar una base de datos relacional con múltiples tablas y relaciones.
5. Crear una interfaz web responsive que funcione en desktop, tablet y móviles.
6. Asegurar la integridad y confidencialidad de las credenciales mediante algoritmos de hashing criptográfico (SHA-256).
7. Proveer interfaces multiplataforma (Web Responsiva y Escritorio Java) que operen concurrentemente.
8. Permitir que múltiples usuarios accedan simultaneamente desde diferentes dispositivos.
9. Asegurar la escalabilidad del sistema mediante programación concurrente (multithreading).

2.3 Alcance

El sistema TicketHub incluye:

Un servidor central que corre en un puerto específico (8081 para HTTP, 9090 para TCP, 9091 para UDP).

El sistema abarca desde la capa de persistencia (Base de Datos PostgreSQL con 6 tablas) hasta la capa de presentación (Clientes), incluyendo un servidor intermedio ("Middleware") desarrollado nativamente en Java que orquesta la lógica de negocio, la seguridad y las comunicaciones de red.

El proyecto no incluye (como mejoras futuras):

- Autenticación multi-factor.
- Sistema de permisos granulares (roles y privilegios avanzados).
- Historial completo de cambios (audit log).
- Aplicación nativa para iOS/Android (web responsive es suficiente).

3. REQUISITOS FUNCIONALES Y NO FUNCIONALES

3.1 Requisitos Funcionales

RF1: Autenticacion de usuarios

- El sistema debe permitir que los tecnicos inicien sesión con usuario y contraseña.
- Cada tecnico tiene asignado un ID, nombre y especialidad.
- El login debe retornar los datos del tecnico autenticado.

RF2: Gestión de tickets

- Los usuarios pueden crear nuevos tickets con título y descripción.
- Los tickets tienen estados: ABIERTO, EN_PROCESO, CERRADO.
- Los tickets tienen prioridades: BAJA, MEDIA, ALTA.
- Los tickets se almacenan en una base de datos relacional.

RF3: Asignación de tickets

- El Super Admin puede asignar un ticket disponible a los técnicos.
- Solo el técnico asignado puede modificar el estado.
- Los tickets sin asignar están disponibles para cualquier tecnico.

RF4: Cambio de estado

- Los tecnicos pueden cambiar el estado de un ticket (ABIERTO → EN_PROCESO → CERRADO).
- Los cambios son inmediatos y se reflejan en todos los dispositivos conectados.

RF5: Visualización de datos

- La aplicación web muestra una tabla con todos los tickets.
- Cada tecnico puede ver solo sus tickets o todos los tickets según prefiera.
- La tabla incluye: ID, Título, Estado, Prioridad, Tecnico Asignado, Fecha de Creación.

RF6: Acceso multidispositivo

- El sistema debe ser accesible desde navegadores web en PC, Tablet y móviles.

- Múltiples usuarios pueden conectarse simultáneamente desde diferentes IPs.
- Los datos se sincronizan en tiempo real (refresh cada 3 segundos).

RF7: Panel de super administrador

- Interfaz exclusiva para el registro de técnicos nuevos y usuarios del sistema.

3.2 Requisitos No Funcionales

RNF1: Rendimiento

- El servidor debe responder a solicitudes en menos de 1 segundo bajo carga normal.
- Soporte para al menos 20 conexiones concurrentes simultáneas.

RNF2: Disponibilidad

- El servidor debe estar disponible 24/7 cuando se ejecuta.
- Las bases de datos deben recuperarse automáticamente ante desconexiones.

RNF3: Seguridad

- Las contraseñas se validarán contra la base de datos.
- Se habilitará CORS para permitir solicitudes cross-origin.
- En producción, se utilizaría HTTPS en lugar de HTTP.

RNF4: Usabilidad

- La interfaz debe ser intuitiva y requerir mínima capacitación.
- La aplicación debe ser responsive y funcionar en pantallas de 320px hasta 4K.
- El tiempo de carga debe ser menor a 3 segundos.

RNF5: Mantenibilidad

- El código debe tener comentarios explicativos.
- Debe seguir principios SOLID y DRY (Don't Repeat Yourself).
- Las clases deben tener responsabilidad única.

RNF6: Escalabilidad

- La arquitectura debe permitir agregar nuevos endpoints fácilmente.
- La BD debe soportar crecer sin cambios en la aplicación.
- El multithreading permite escalar el número de usuarios.

4. DISEÑO Y ARQUITECTURA

4.1 Arquitectura General

Para TicketHub se optó por una **Arquitectura Híbrida de N-Capas** que combina el modelo REST (Representational State Transfer) para operaciones de datos con un modelo de Sockets puros para señalización en tiempo real.

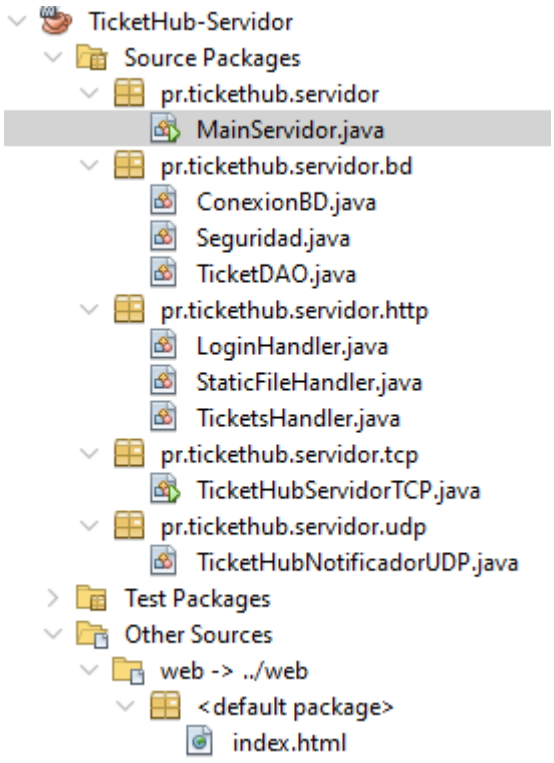
1. **Capa de presentación (Cliente)**: Interfaz web HTML5 + JavaScript responsive.
2. Capa de lógica de negocio (Servidor HTTP/TCP): Java con manejo de solicitudes REST y TCP.
3. **Capa de persistencia (Base de datos)**: PostgreSQL con 6 tablas relacionadas.

El sistema también utiliza UDP para notificaciones en tiempo real.

4.2 Componentes Principales

Servidor (TicketHub-Servidor)

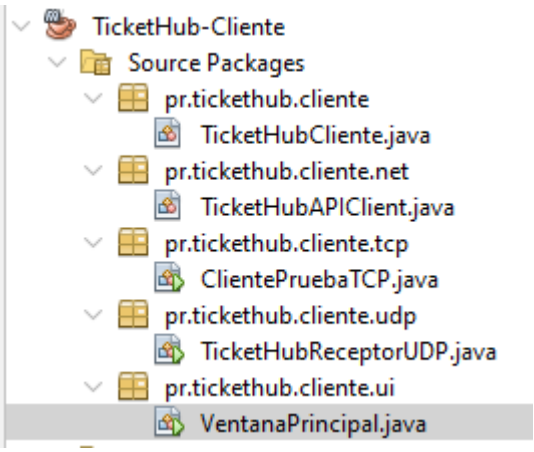
- **MainServidor**: Punto de entrada, inicia los servidores HTTP, TCP y UDP.
- **ConexionBD**: Administra conexiones a PostgreSQL.
- **Seguridad**: Maneja la encriptación de las contraseñas SHA 256.
- **TicketDAO**: Capa de acceso a datos para tickets.
- **LoginHandler**: Maneja autenticación de usuarios.
- **StaticFile**: para lectura de archivos html, css, etc.
- **TicketsHandler**: Maneja solicitudes GET, POST, PUT para tickets.



- **TicketHubServidorTCP**: Servidor multihilo en puerto 9090 para cambios de estado.
- **TicketHubNotificadorUDP**: Envia notificaciones UDP cuando hay cambios.
- **index.html**: Interfaz web única con HTML5 + CSS

Cliente (TicketHub-Cliente)

- **VentanaPrincipal**: Interfaz en JavaScript muy parecida (visualmente) a index.
- **TicketHubAPIClient** (antiguo): Ya integrado en JavaScript pero aun se manda a llamar.
- **TicketHubReceptorUDP**: Aplicación separada para escuchar notificaciones (opcional).
- **ClientePrueba**: solo se utilizo al inicio cuando se creaba el proyecto.



4.3 Diagrama de Arquitectura

```
                    Internet / LAN
                          |
      ____________________________________
     |                                    |
PC/Navegador                        Tablet/Móvil
(192.168.1.50:8081)                (192.168.1.50:8081)
     |                                    |
     |____________________________________|
                    |
        (HTTP REST, HTTPS en producción)
                    |
      ______________________________
     |                              |
     v                              v
TicketHub-Servidor           UDP Notificaciones
(localhost:8081)             (localhost:9091)
- LoginHandler               (Broadcast de cambios)
- TicketsHandler
- TicketHubServidorTCP
(localhost:9090)
     |
     v
PostgreSQL DB
(localhost:5432)
- usuarios_login
- tecnicos
- tickets
- clientes
- categorias
- comentarios_ticket
```

4.4 Flujo de Comunicación

Flujo de Login:

1. Usuario entra credenciales en web.
2. LoginHandler valida contra BD.
3. Si OK, retorna id_tecnico, nombre, especialidad.
4. JavaScript guarda datos en memoria y muestra dashboard.

Flujo de Creación de Ticket:

1. Usuario hace click en "+ Nuevo Ticket".
2. Ingresa título y descripción en dialog.
3. JavaScript envia POST a /tickets con JSON.
4. TicketsHandler inserta en BD y retorna id.
5. Ticket aparece en tabla (próximo refresh).

Flujo de Cambio de Estado:

1. Usuario selecciona ticket y hace click en "En Proceso".
2. JavaScript envia PUT a /tickets/:id/estado con JSON.
3. TicketsHandler actualiza BD.
4. (Opcional) TicketHubNotificadorUDP envia notificación UDP.
5. Tabla se refresca cada 3 segundos mostrando nuevo estado.

4.5 IMPLEMENTACION TECNICA

4.5.1 Tecnologías Utilizadas

Backend:

- Java 17
- Apache Maven para gestión de dependencias
- PostgreSQL para base de datos
- JDK HttpServer (nativo, sin Spring Boot)

- org.json para manejo de JSON
- JDBC para conexión a BD

Frontend:

- HTML5
- CSS3 con media queries (responsive)
- JavaScript vanilla (sin frameworks)
- Fetch API para comunicación HTTP

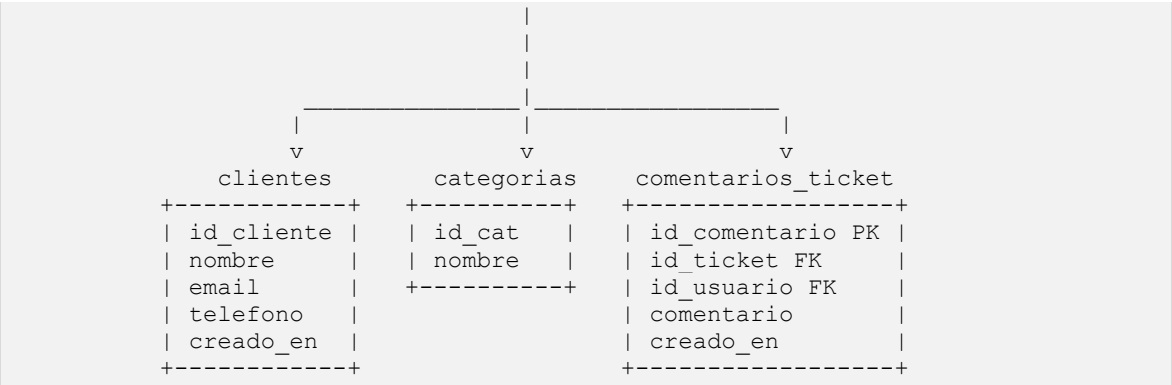
Herramientas de Desarrollo:

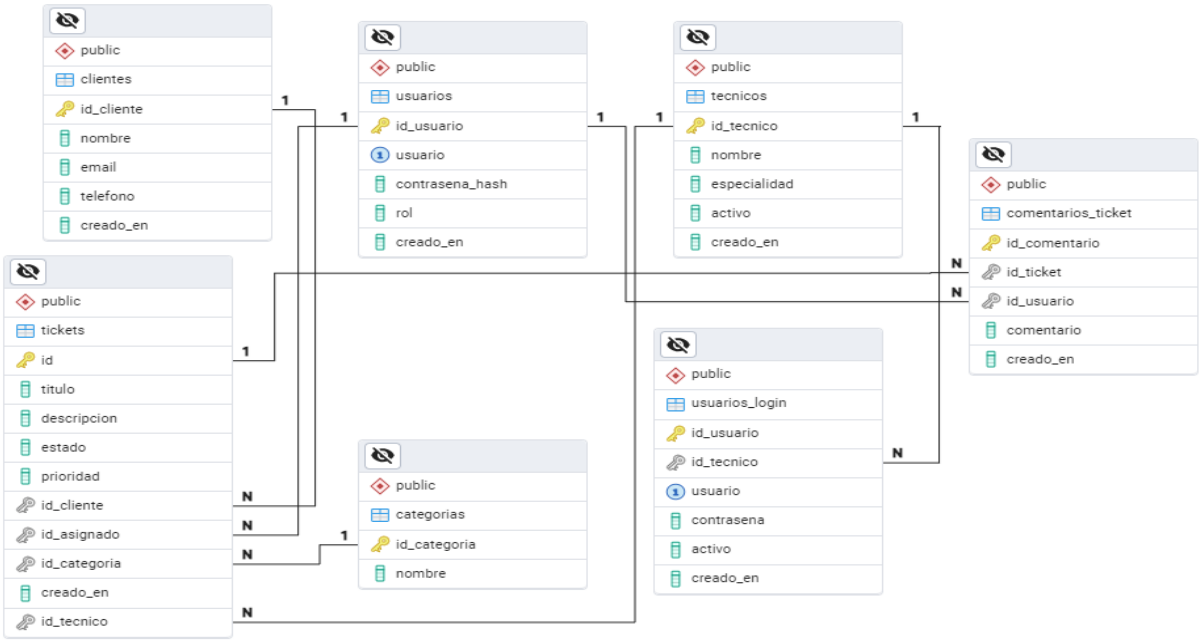
- NetBeans IDE
- pgAdmin para administración de BD

4.6 Diseño de la Base de Datos (ER)

4.6.1 Diagrama Entidad-Relación

El esquema relacional está normalizado para evitar redundancia. Las tablas principales son `tickets`, `tecnicos` y `usuarios_login`.





4.6.2 Esquema SQL Completo

```
-- Tabla de usuarios de login
CREATE TABLE usuarios_login (
  id_usuario SERIAL PRIMARY KEY,
  id_tecnico INT REFERENCES tecnicos(id_tecnico) ON DELETE CASCADE,
  usuario VARCHAR(50) UNIQUE NOT NULL,
  contrasena VARCHAR(255) NOT NULL,
  activo BOOLEAN DEFAULT true,
  creado_en TIMESTAMP DEFAULT NOW()
);

-- Tabla de tecnicos
CREATE TABLE tecnicos (
  id_tecnico SERIAL PRIMARY KEY,
  nombre VARCHAR(100) NOT NULL,
```

```
  especialidad VARCHAR(100),
  activo BOOLEAN DEFAULT true,
  creado_en TIMESTAMP DEFAULT NOW()
);

-- Tabla de clientes
CREATE TABLE clientes (
  id_cliente SERIAL PRIMARY KEY,
  nombre VARCHAR(100) NOT NULL,
  email VARCHAR(120),
  telefono VARCHAR(30),
  creado_en TIMESTAMP DEFAULT NOW()
);

-- Tabla de categorias
CREATE TABLE categorias (
  id_categoria SERIAL PRIMARY KEY,
  nombre VARCHAR(60) NOT NULL
);

-- Tabla principal: tickets
CREATE TABLE tickets (
  id SERIAL PRIMARY KEY,
  titulo VARCHAR(140) NOT NULL,
  descripcion TEXT,
  estado VARCHAR(20) NOT NULL DEFAULT 'ABIERTO'
         CHECK (estado IN ('ABIERTO','EN_PROCESO','CERRADO')),
  prioridad VARCHAR(10) NOT NULL DEFAULT 'MEDIA'
            CHECK (prioridad IN ('BAJA','MEDIA','ALTA')),
  id_cliente INT REFERENCES clientes(id_cliente)
             ON UPDATE CASCADE ON DELETE RESTRICT,
  id_asignado INT REFERENCES usuarios_login(id_usuario)
              ON UPDATE CASCADE ON DELETE SET NULL,
  id_tecnico INT REFERENCES tecnicos(id_tecnico)
             ON DELETE SET NULL,
  id_categoria INT REFERENCES categorias(id_categoria)
               ON UPDATE CASCADE ON DELETE SET NULL,
  creado_en TIMESTAMP DEFAULT NOW(),
  actualizado_en TIMESTAMP DEFAULT NOW()
);

-- Tabla de comentarios
CREATE TABLE comentarios_ticket (
  id_comentario SERIAL PRIMARY KEY,
  id_ticket INT NOT NULL REFERENCES tickets(id)
            ON UPDATE CASCADE ON DELETE CASCADE,
  id_usuario INT NOT NULL REFERENCES usuarios_login(id_usuario)
             ON UPDATE CASCADE ON DELETE RESTRICT,
  comentario TEXT NOT NULL,
  creado_en TIMESTAMP DEFAULT NOW()
);

-- Insercion de tecnicos
INSERT INTO tecnicos (nombre, especialidad) VALUES
('Raul Admin', 'Sistemas'),
('Juan Perez', 'Redes'),
('Maria Lopez', 'Hardware'),
('Carlos Ruiz', 'Software');

-- Insercion de usuarios con contrasenas hasheadas (SHA-256)
INSERT INTO usuarios_login (id_tecnico, usuario, contrasena, activo) VALUES
(1, 'raul',
'e3d78166623f1225b72d50490495147a62058bb9c9354276f830ef4b57797921', true),
```

```
(2, 'juan',
'03ac674216f3e15c761ee1a5e255f067953623c8b388b4459e13f978d7c846f4', true),
(3, 'maria',
'03ac674216f3e15c761ee1a5e255f067953623c8b388b4459e13f978d7c846f4', true),
(4, 'carlos',
'03ac674216f3e15c761ee1a5e255f067953623c8b388b4459e13f978d7c846f4', true);

INSERT INTO categorias(nombre) VALUES
('General'),
('Hardware'),
('Software'),
('Impresora'),
('Red');

INSERT INTO clientes(nombre, email) VALUES
('Cliente Demo', 'demo@empresa.com');
```

Como parte de la mejora en seguridad del sistema, las contraseñas no se almacenan en texto plano dentro de la base de datos. En su lugar, se utilizan **hashes SHA-256**, los cuales son generados desde la aplicación Java antes de persistir la información. Estas inserciones garantizan que **ninguna contraseña sea visible ni recuperable en texto plano**, cumpliendo con buenas prácticas básicas de seguridad de la información.

5. IMPLEMENTACIÓN TÉCNICA

5.1 Configuración RMI/RESTful

En cumplimiento con la modernización tecnológica sugerida, se implementó una arquitectura **RESTful** utilizando el servidor HTTP nativo de Java. Esto reemplaza al obsoleto RMI, permitiendo una mayor interoperabilidad.

Fragmento de Código: Enrutamiento REST (`TicketsHandler.java`) Este código muestra cómo el servidor despacha las peticiones según el verbo HTTP, actuando como un controlador REST.

```
@Override
public void handle(HttpExchange ex) throws IOException {
    String method = ex.getRequestMethod();
    // Configuración de Cabeceras CORS para permitir acceso web cruzado
    ex.getResponseHeaders().add("Access-Control-Allow-Origin", "*");

    // Despacho según verbo HTTP (Mapeo REST)
    if ("GET".equalsIgnoreCase(method)) {
        handleGet(ex); // Recuperar recursos
    } else if ("POST".equalsIgnoreCase(method)) {
        handlePost(ex); // Crear recursos
    } else if ("PUT".equalsIgnoreCase(method)) {
        handlePut(ex, path); // Actualizar recursos
    }
}
```

5.2 Creación de Sockets (Cliente y Servidor)

Uno de los pilares del proyecto es el manejo explícito de sockets para comunicaciones de bajo nivel.

A. Sockets UDP (Notificaciones en Tiempo Real) El servidor actúa como emisor (Broadcaster) y el cliente de escritorio como receptor (Listener).

Código del Servidor (Emisor - `TicketHubNotificadorUDP.java`*):*

```
public void enviar(String mensaje) {
    try (DatagramSocket socket = new DatagramSocket()) {
        byte[] datos = mensaje.getBytes("UTF-8");
        InetAddress addr = InetAddress.getByName(host);
        // Empaquetado del datagrama con destino al puerto del cliente
        DatagramPacket packet = new DatagramPacket(datos, datos.length, addr,
puerto);
        socket.send(packet); // Envío asíncrono
    } catch (Exception e) { e.printStackTrace(); }
}
```

Código del Cliente (Receptor - `VentanaPrincipal.java`*):*

```
private void iniciarEscuchaUDP() {
    new Thread(() -> { // Hilo secundario para no bloquear la interfaz gráfica
        try (DatagramSocket socket = new DatagramSocket(9091)) {
            while (true) {
                // Bloqueo hasta recibir paquete
                socket.receive(packet);
                String msg = new String(packet.getData(), ...);
                if (msg.startsWith("NUEVO_TICKET")) {
                    // Actualización segura de la UI en el hilo de eventos
                    SwingUtilities.invokeLater(() -> cargarTickets());
                }
            }
        } catch (Exception e) { ... }
    }).start();
}
```

B. Sockets TCP (Servidor Concurrente) Se implementó un servidor TCP multihilo para demostrar conexiones persistentes y fiables.

Código del Servidor TCP (`TicketHubServidorTCP.java`*):*

```
public void iniciar() throws IOException {
    serverSocket = new ServerSocket(9090);
    pool = Executors.newFixedThreadPool(20); // Pool de hilos para concurrencia
    while (true) {
        Socket cliente = serverSocket.accept(); // Espera conexión
        // Delegación de la conexión a un hilo trabajador
        pool.submit(new ManejadorCliente(cliente, ...));
    }
}
```

5.3 Módulo de Seguridad (Cifrado)

Para proteger la información sensible, se implementó la clase `Seguridad.java` que utiliza el algoritmo SHA-256. Ninguna contraseña se almacena en texto plano.

```
public static String encriptar(String password) {
    MessageDigest digest = MessageDigest.getInstance("SHA-256");
    byte[] hash = digest.digest(password.getBytes(StandardCharsets.UTF_8));
    // Conversión de bytes a representación Hexadecimal
    return hexString.toString();
}
```

5.4 Instrucciones SQL Clave

El diseño de la base de datos incluye restricciones de integridad y valores por defecto para asegurar la consistencia.

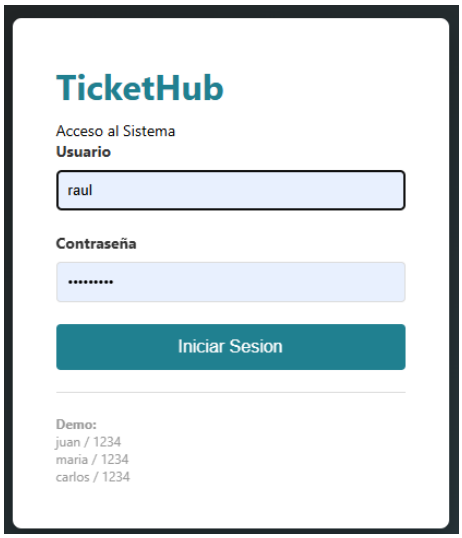
```
-- Tabla principal de Tickets con claves foráneas
CREATE TABLE tickets (
  id SERIAL PRIMARY KEY,
  titulo VARCHAR(140) NOT NULL,
  estado VARCHAR(20) DEFAULT 'ABIERTO' CHECK (estado IN
('ABIERTO','EN_PROCESO','CERRADO')),
  id_tecnico INT REFERENCES tecnicos(id_tecnico) ON DELETE SET NULL,
  creado_en TIMESTAMP DEFAULT NOW()
);

-- Inserción segura de usuario (El hash es generado por la App Java)
INSERT INTO usuarios_login(usuario, contrasena)
VALUES ('admin',
'03ac674216f3e15c761ee1a5e255f067953623c8b388b4459e13f978d7c846f4');
```

6. RESULTADOS

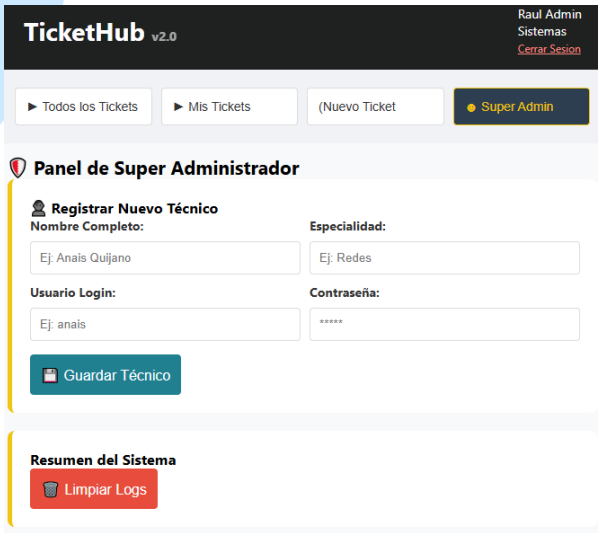
A continuación, se presentan las evidencias del funcionamiento del sistema.

6.1 Acceso y Seguridad



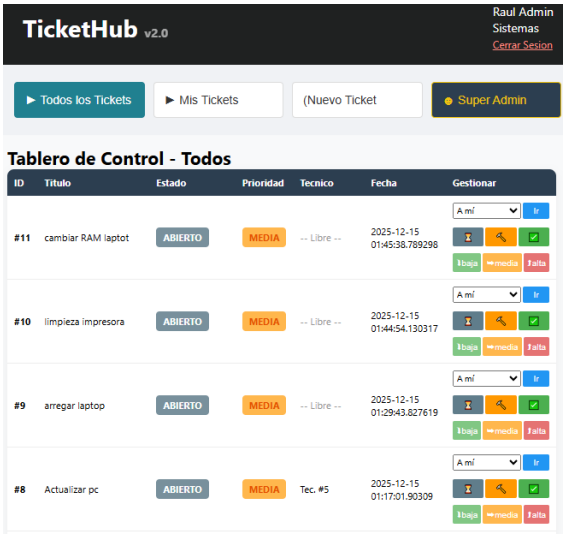
(Descripción: El sistema solicita credenciales. Al ingresar, valida el hash SHA-256 contra la base de datos).

6.2 Panel de Super Administrador



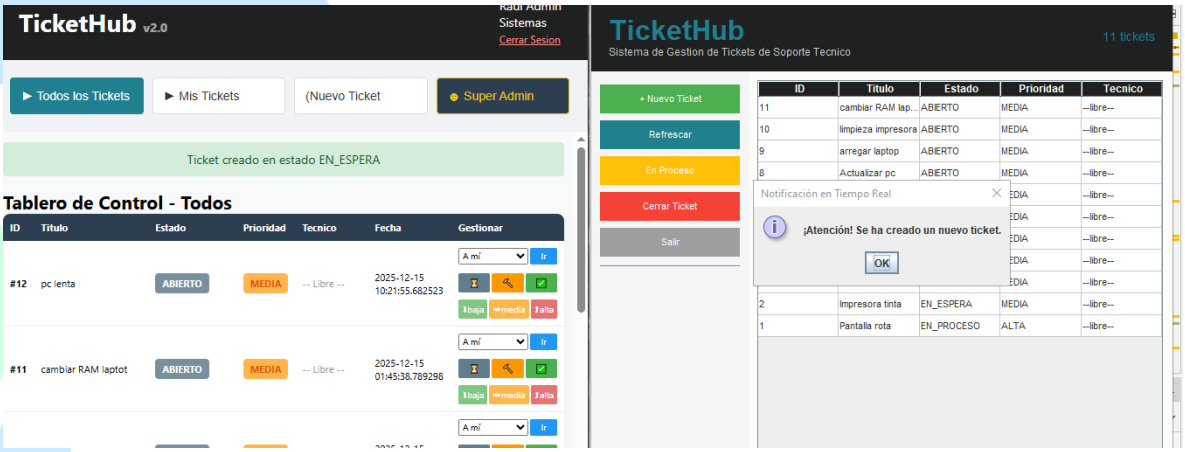
(Descripción: Formulario exclusivo para el alta de nuevos técnicos, demostrando la gestión de privilegios).

6.3 Gestión de Tickets (Web)



(Descripción: Vista principal responsiva donde se listan, asignan y gestionan los estados de los tickets).

6.4 Sincronización en Tiempo Real (Prueba UDP)



(Descripción: Evidencia crítica de la integración UDP. Al crear el ticket en la web, el cliente de escritorio recibe la notificación instantánea).

6.5 Resumen de Pruebas Realizadas

[x] Login funciona con usuarios creados

[x] Validación de credenciales incorrectas

[x] Creación de tickets exitosa

[x] Asignación de tickets a técnicos

[x] Cambio de estado en tiempo real

[x] Visualización sincronizada en múltiples clientes

[x] Responsive design en desktop, tablet y móvil

[x] CORS habilitado para acceso desde diferentes IPs

[x] Conexión a PostgreSQL estable

[x] Multithreading maneja múltiples solicitudes simultaneas

7. INSTRUCCIONES DE INSTALACION Y CONFIGURACION

7.1 Requisitos Previos

- Java JDK 17 o superior
- Maven 3.6 o superior
- PostgreSQL 13 o superior
- NetBeans IDE (recomendado) o Eclipse
- Un navegador web moderno (Chrome, Firefox, Safari, Edge)

7.2 Pasos de Instalación

Paso 1: Crear la base de datos PostgreSQL

Abre pgAdmin o psql y ejecuta:

```
CREATE DATABASE tickethub;
-- (Copiar todo el esquema SQL de la sección 4.6.2)
```

Paso 2: Descargar y abrir el proyecto

1. Descarga los archivos del proyecto TicketHub-Servidor y TicketHub-Cliente desde GitHub.
2. Abre NetBeans.
3. File → Open Project → selecciona TicketHub-Servidor.
4. Repite para TicketHub-Cliente.

Paso 3: Crear la carpeta web

1. En TicketHub-Servidor, crea una carpeta llamada `web` en la raiz del proyecto *(al lado de src, pom.xml).*
2. Dentro de `web`, copia el archivo `index.html`.

Paso 4: Compilar

1. Click derecho en TicketHub-Servidor → "Clean and Build".
2. Espera a que compile sin errores.

Paso 5: Ejecutar el servidor

1. Click derecho en MainServidor.java → "Run File" (F6).
2. En la consola deberías ver:

```
========================================
TicketHub servidor iniciado
URL: http://localhost:8081
Usuarios: juan, maria, carlos (pass: 1234)
========================================
```

Paso 6: Acceder a la aplicación

1. Desde tu PC: abre `http://localhost:8081` en el navegador.
2. Desde otro dispositivo en la misma red:
3. Abre PowerShell en la PC del servidor y ejecuta `ipconfig`.
4. Busca "IPv4 Address", por ejemplo `192.168.1.50.`
5. Desde el otro dispositivo, abre `http://192.168.1.50:8081`.
6. Tambien puedes iniciar el VentanaPricipal.java desde TicketHub-Cliente
7. Click derecho en VentanaPricipal.java → "Run File" (F6).

7.3 Configuración de Base de Datos

Archivo de conexión: `ConexionBD.java`

Por defecto esta configurado para:

- Host: localhost
- Puerto: 5432
- Base de datos: tickethub
- Usuario: postgres
- Contraseña: 1234

Si tu configuración es diferente, edita `ConexionBD.java`:

```
private static final String URL = "jdbc:postgresql://tu-host:5432/tickethub";
private static final String USUARIO = "tu-usuario";
private static final String CONTRASENA = "tu-contraseña";
```

7.4 Puertos Utilizados

- **8081**: Servidor HTTP (API REST + archivos estáticos)
- **9090**: Servidor TCP (cambios de estado en tiempo real)
- **9091**: UDP (notificaciones, opcional)
- **5432**: PostgreSQL

Asegúrate de que estos puertos no estén bloqueados por el firewall en tu pc.

8. CONCLUSIONES

8.1 Logros Alcanzados

1. **Arquitectura completa cliente-servidor** implementada con tecnologías estudiadas en la materia.
2. **Multiples protocolos de comunicación**:
    - HTTP REST para operaciones principales
    - TCP multihilo para cambios en tiempo real
    - UDP para notificaciones
3. **Base de datos relacional** con 6 tablas bien diseñadas y relaciones normalizadas.
4. **Interfaz responsive** que funciona perfectamente en dispositivos de cualquier tamaño.
5. **Autenticación de usuarios** con técnicos asignados a credenciales específicas.
6. **Escalabilidad** demostrada con manejo de múltiples conexiones concurrentes mediante ExecutorService.
7. **Aplicación funcional** lista para usar en un ambiente de soporte técnico real.

Se logró desarrollar un ecosistema completo de soporte técnico que cumple con los rigurosos estándares de la computación distribuida moderna. La implementación de **Hashing SHA-256** elevó la seguridad del sistema a un nivel profesional. La integración de **Sockets UDP** para notificaciones en tiempo real demostró ser una solución eficiente para el desacoplamiento de eventos entre el servidor y los clientes de escritorio.

8.2 Limitaciones

1. **Sin autenticación de tokens**: No hay JWT o sesiones persistentes. Se validé cada vez que se hace login.
2. **Sin cifrado de comunicación**: Usa HTTP. En producción, usar HTTPS/TLS.
3. **Sin historial de cambios**: No se registra quién cambió qué y cuándo (audit log).
4. **UI minimalista**: Sin iconos, imagenes ni animaciones avanzadas. Funcional pero podría ser más atractiva.

8.3 Mejoras Futuras

1. **HTTPS en producción**: Usar certificados SSL/TLS.
2. **Sistema de roles y permisos**: Agregar ADMIN, SUPERVISOR, TECNICO con permisos diferenciados.
3. **Historial de tickets**: Registrar todos los cambios con timestamp y usuario que los hizo.
4. **Notificaciones en tiempo real sin polling**: Usar WebSockets en lugar de refrescar cada 3 segundos.
5. **Busqueda y filtros avanzados**: Filtrar por estado, prioridad, fecha, técnico asignado.
6. **Exportación de reportes**: Generar PDF o Excel con tickets cerrados en un período.
7. **Aplicación móvil nativa**: Aunque la web es responsive, una app nativa mejoraría UX.
8. **Chat interno**: Comentarios y comunicación entre técnicos sobre tickets.
9. **Integración con email**: Notificaciones por correo cuando se crea un ticket o cambia de estado.

8.4 Reflexión Personal

El desarrollo de TicketHub ha representado un hito significativo en mi formación como ingeniero, permitiéndome trasladar los conceptos teóricos vistos en el aula a una implementación práctica y funcional. A continuación, detallo los aprendizajes clave obtenidos durante este proceso:

- **Sockets TCP/UDP**: Uno de los mayores desafíos fue comprender no solo la teoría, sino la implementación práctica de estos protocolos. Al inicio, la diferencia conceptual era clara (fiabilidad vs. velocidad), pero al programar, enfrenté la complejidad de gestionar flujos de datos y datagramas. Logré consolidar este conocimiento al implementar TCP para la gestión crítica del estado de los tickets y UDP para las notificaciones en tiempo real, entendiendo finalmente en qué escenarios es indispensable cada uno.

- **Comunicación cliente-servidor**: Pasé de estudiar modelos teóricos a construir un ecosistema vivo donde múltiples clientes (Web y Escritorio) interactúan concurrentemente con un servidor centralizado. Esto me enseñó a manejar la sincronización de datos y la importancia de una "fuente de verdad" única en sistemas distribuidos.

- **Programación Concurrente (Multithreading):** La implementación del `ThreadPoolExecutor` en el servidor y de los `SwingWorkers` en el cliente fue reveladora. Comprendí la importancia vital de no bloquear el hilo principal de ejecución para mantener la fluidez de la aplicación, un concepto que eleva la calidad del software a un nivel profesional.
- **Diseño de API RESTful:** Diseñar los endpoints siguiendo convenciones estándar (uso correcto de verbos HTTP y códigos de estado) me permitió ver cómo se estructuran las comunicaciones en la industria moderna, facilitando la interoperabilidad entre distintas plataformas.

- **Integración de Base de Datos Relacional**: El diseño del esquema en PostgreSQL me permitió practicar la normalización y el manejo de integridad referencial con múltiples tablas, asegurando que la persistencia de los datos fuera robusta y escalable.

- **Desarrollo Frontend Responsivo**: Adquirí habilidades prácticas en CSS y Media Queries, logrando que una interfaz técnica compleja se adapte fluidamente a dispositivos móviles y de escritorio, mejorando significativamente la experiencia de usuario (UX).

En conclusión, el hecho de basar este proyecto en una actividad que me apasiona *el soporte técnico y reparación de computadoras* fue un factor motivante crucial. No solo cumplí con un requisito académico, sino que desarrollé una herramienta funcional que tiene el potencial de ser implementada en mi entorno laboral real, cerrando así la brecha entre la academia y la industria.

9. REFERENCIAS

9.1 Documentación Oficial

[1] Oracle. (2024). *Java 17 Documentation - HttpServer*. https://docs.oracle.com/en/java/javase/17/docs/api/com.sun.net.httpserver/module-summary.html

[2] PostgreSQL Global Development Group. (2024). *PostgreSQL 15 Documentation*. https://www.postgresql.org/docs/current/

[3] Mozilla Developer Network. (2024). *Fetch API Documentation*. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

[4] Berners-Lee, T., Fielding, R., & Masinter, L. (2005). *RFC 3986: Uniform Resource Identifier (URI) Generic Syntax*. Internet Engineering Task Force.

9.2 Recursos de Aprendizaje

[5] Tanenbaum, A. S., & Wetherall, D. J. (2010). *Computer Networks* (5th ed.). Pearson.

[6] Gorelick, M., & Ozsvald, I. (2020). *High Performance Python: Practical Performant Programming for Humans* (2nd ed.). O'Reilly Media.

[7] Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.

9.3 Herramientas Utilizadas

[8] Apache Software Foundation. (2024). *Apache Maven Project*. https://maven.apache.org/

[9] NetBeans Project. (2024). *Apache NetBeans IDE*. https://netbeans.apache.org/

[10] JSON.org. (2024). *Introducing JSON*. https://www.json.org/json-en.html

Pie de página:

***Documento generado**: diciembre 2025 **Autor**: Raul Hernández Huerta **Materia**: Programación de Redes **Profesor**: Yuliana Berumen Diaz **Institución**: Universidad Veracruzana*

APENDICE: GUIA RAPIDA DE USO

Arrancando el sistema

1. PostgreSQL debe estar corriendo
2. Ejecutar `MainServidor` en NetBeans
3. Abrir navegador en `http://localhost:8081`
4. Login: juan / 1234

Usuarios de prueba disponibles

Usuario	Contraseña	Especialidad
juan	1234	Hardware
maria	1234	Software
carlos	1234	Impresoras y Redes

Estructura de carpetas del proyecto

```
TicketHub-Servidor/
├── src/main/java/pr/tickethub/servidor/
│   ├── MainServidor.java
│   ├── http/
│   │   ├── LoginHandler.java
│   │   ├── TicketsHandler.java
│   │   └── StaticFileHandler.java
│   ├── bd/
│   │   ├── ConexionBD.java
│   │   └── TicketDAO.java
│   ├── tcp/
│   │   └── TicketHubServidorTCP.java
│   └── udp/
│       └── TicketHubNotificadorUDP.java
├── web/
│   └── index.html
├── pom.xml
└── ...
```

Comandos SQL útiles

```
-- Ver todos los tickets
SELECT * FROM tickets;

-- Ver usuarios
SELECT * FROM usuarios_login;

-- Ver tecnicos
SELECT * FROM tecnicos;

-- Crear un nuevo técnico
INSERT INTO tecnicos(nombre, especialidad) VALUES ('Pedro Torres', 'Software');

-- Crear usuario para ese técnico
INSERT INTO usuarios_login(id_tecnico, usuario, contrasena)
VALUES ((SELECT id_tecnico FROM tecnicos WHERE nombre='Pedro Torres'), 'pedro',
'1234');

-- Limpiar todos los tickets
DELETE FROM tickets;
```