PROYECTO DEL CURSO - SEGUNDA ENTREGA

Maria Paula Rodríguez Ruiz

Daniel Felipe Castro Moreno

Juan Enrique Rozo Tarache

Eliana Katherine Cepeda González

Introducción a Sistemas Distribuidos

Dpto. de Ingeniería de Sistemas

Pontificia Universidad Javeriana

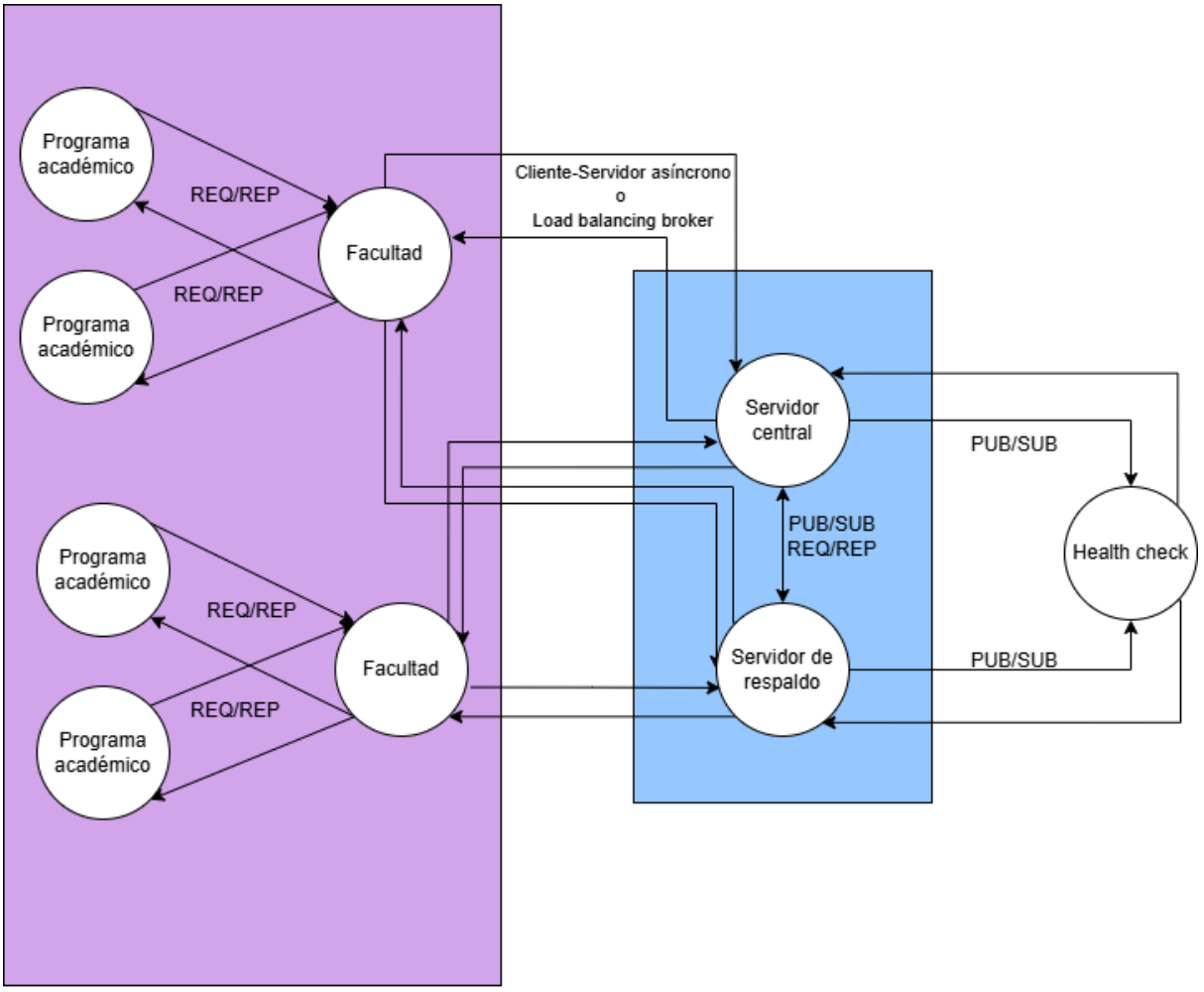
Bogotá, Colombia

2025

Diseño del sistema

Modelos del sistema

Modelo Arquitectónico



En este modelo arquitectónico podemos evidenciar una simplificación de los componentes de este proyecto. Este proyecto consta de 10 facultades, con 5 programas cada una, donde cada facultad se comunica con el servidor activo para hacer las peticiones de salones que hayan solicitado dichos programas. El servidor enviará una propuesta de salones disponibles y la facultad debe confirmar o rechazar, remitiendo su respuesta al programa también. A continuación, se explicará a detalle que hace cada componente en el sistema y cómo se comunican entre sí:

Componente: Programa Académico (academic_program.py)
Se encuentra en la máquina 1 con dirección IP 10.43.103.51, en la máquina 2 con dirección IP 10.43.103.58 y en la máquina 3 con dirección IP 10.43.96.50

Se encarga de representar a un programa académico específico, generar solicitudes de un número variable aleatorio de aulas para un semestre específico (asignado como parámetro de entrada) y mostrar la respuesta final recibida por Facultad.

- Se comunica con:
    - Facultad: A través del protocolo ZeroMQ, utilizando un patrón Request-Reply Síncrono (socket zmq.REQ en academic_program.py se conecta al zmq.REP de la facultad). Envía la solicitud y espera una respuesta.

Componente: Facultad (faculty.py y facultylbb.py)
Se encuentra en la máquina 2 con dirección IP 10.43.103.58

Este componente representa a una de las 10 facultades del sistema distribuido y actúa como intermediario entre sus 5 programas académicos asociados y el servidor activo (central o de respaldo).

- Posee 2 versiones:
    - **Patrón Asynchronous Client/Server:** En su versión estándar (faculty.py), escucha solicitudes entrantes de los programas académicos (tipo SOL) a través de un socket REP, y transforma dichas solicitudes para reenviarlas al servidor activo mediante un socket DEALER. De forma asincrónica, gestiona la recepción de respuestas del servidor (mensajes tipo PROP, ACK, RES), y retransmite el resultado final al programa solicitante. Simultáneamente, monitorea la disponibilidad del Servidor Primario y del Servidor de Respaldo usando heartbeats con un socket SUB, reaccionando automáticamente ante fallos y cambiando de servidor cuando es necesario (failover automático). Además, registra métricas detalladas de latencia y procesamiento a través de datastore.py, y mantiene una base persistente de datos sobre la facultad y sus programas.
    - **Patrón Load Balancing Broker:** La versión extendida (facultylbb.py) mantiene toda la funcionalidad anterior, pero con mejoras clave:
        - Selección dinámica del servidor: A diferencia de faculty.py, no mantiene una conexión persistente con el servidor. En cambio, elige de forma dinámica el endpoint del servidor activo (Primario o Respaldo) con base en los últimos heartbeats válidos recibidos. Esta selección se realiza dentro de una hebra dedicada que actualiza una variable compartida protegida por un lock.
        - Socket REQ por transacción: Para cada transacción con el servidor (por cada SOL recibido), se crea un nuevo socket REQ temporal que se conecta solo si hay un servidor activo disponible. Esta estrategia permite mayor flexibilidad ante fallos temporales de red o reinicios.
        - Tiempos detallados: Mide el tiempo exacto entre los eventos SOL → PROP, y ACK → RES, almacenando estas métricas en datastore.py para propósitos de monitoreo y análisis del sistema.
- Se comunica con:
    - Programas académicos: Utiliza el patrón Request-Reply (REQ/REP) de ZeroMQ. Tanto faculty.py como facultylbb.py escuchan mediante un socket zmq.REP en tcp://*:6000 y reciben solicitudes SOL y responden con mensajes RES.
    - Servidor Central (o de Respaldo):
        - En el **Patrón Asynchronous Client/Server** se realiza mediante un socket zmq.DEALER, que mantiene conexión persistente con el servidor activo (zmq.ROUTER).
        - En el **Patrón Load Balancing Broker**, se crea un nuevo socket REQ por cada solicitud, conectándose dinámicamente al servidor activo, sin conexión permanente.

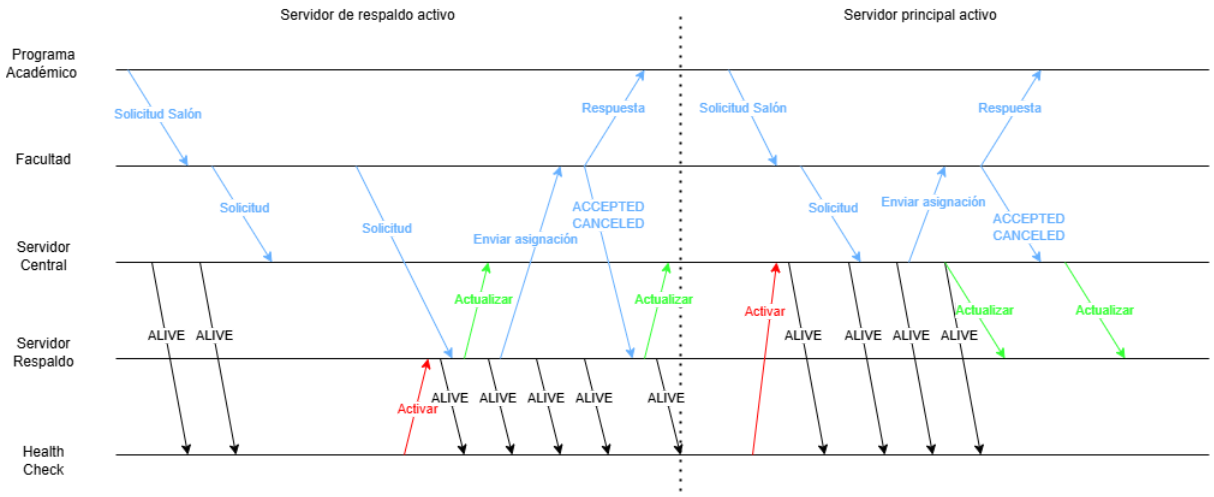
- Monitoreo de Heartbeats: Ambos utilizan el patrón Publish-Subscribe (PUB/SUB) para recibir heartbeats desde los servidores. Se conectan a tcp://10.43.96.50:7000 (Primario) y tcp://10.43.103.51:7000 (Respaldo), mediante un zmq.SUB suscrito a mensajes "HB" y evalúan la liveness del servidor según el tiempo transcurrido desde el último heartbeat recibido, aplicando los parámetros HB_INTERVAL y HB_LIVENESS. Si el servidor activo cambia, se informa por consola y se actualiza de inmediato la conexión usada.

Componente: Servidor Central y Backup (server.py y serverlbb.py)
Se encuentra en la máquina 3 con dirección IP 10.43.96.50

Se encarga de gestionar el inventario total de aulas (380 salones, 60 laboratorios); recibir las solicitudes de todas las facultades; procesar las solicitudes de forma concurrente usando hilos; asignar salones y laboratorios según disponibilidad; decidir si adaptar salones como aulas móviles si se agotan los laboratorios; generar alertas si no se puede cumplir una solicitud; almacenar un registro de solicitudes y asignaciones; y publicar su estado para sincronización con el backup y emitir "heartbeats" para el Health Check y las Facultades.

- Posee 2 versiones:
  - **Patrón Asynchronous Client/Server:**
    - Usa ROUTER/DEALER para comunicación asíncrona.
    - Implementa workers concurrentes para procesamiento paralelo.
    - Maneja transacciones con timeout para ACKs.
    - Sistema de heartbeats con PUB/SUB para failover.
    - Las facultades envían solicitudes (SOL) al ROUTER (puerto 5555), el servidor responde con propuestas (PROP) a través del mismo canal y espera confirmaciones (ACK) con timeout configurable. El servidor de respaldo se sincroniza mediante PUB/SUB (puerto 7000) para heartbeats y failover. Las respuestas finales (RES) se envían tras recibir ACK o por timeout.
  - **Patrón Load Balancing Broker:**
    - Utiliza proxy ROUTER/DEALER para balanceo de carga.
    - Workers independientes conectados al backend.
    - Mecanismo de confirmación/reserva simplificado.
    - Binary Star para alta disponibilidad.
    - Las facultades conectan vía REQ al ROUTER (puerto 5555), recibiendo propuestas (PROP) directamente de los workers. Las confirmaciones (ACK) se procesan en el mismo worker que originó la propuesta. La sincronización con el backup usa PUB/SUB (puerto 7000) con lógica Binary Star para heartbeats. Las respuestas finales (RES) son inmediatas tras el ACK.
  - Ambas implementaciones comparten la comunicación con Health Check mediante PUB (puerto 7000) para monitoreo de estado, y el almacenamiento en SQLite para persistencia de reservas e inventario. La IP de operación (10.43.96.50) y capacidad de gestión (380 salones/60 laboratorios) son idénticas en ambas versiones.

Modelo de Interacción



En el modelo de interacción se muestran los 3 procesos más importantes dentro del proyecto, en azul se muestra el proceso de envío y respuesta de una solicitud de un salón, estos procesos se inician por los programas académicos y terminan con la respuesta a la solicitud, como se puede ver la solicitud de los salones puede ser resuelta tanto por el servidor de respaldo como por el servidor principal, esto dependerá de cuál de los dos esté activo al momento de hacer la solicitud.

El segundo proceso importante que estamos mostrando acá es el proceso de cómo se comunican los servidores con el health check que se muestra en rojo, este se encarga de que cuando deje de recibir respuestas de vida de su servidor activo, mandará a activar el otro servidor que esté dando respuestas de vida.

Por último, en verde, mostramos como cada vez que hay un cambio en los datos del servidor activo, este envía actualizaciones al otro servidor para que ambos tengan la información actualizada, como se comunican a través de un patrón Publish-Subscribe, apenas el otro servidor reviva empezará a consumir las actualizaciones.

Modelo de Fallos

Todos y cada uno de los sistemas son susceptibles a fallas que no se pueden predecir y que pueden hacer que el sistema falle, ya sea por caídas de servidores, fallos en la recepción de mensajes, el tiempo en que se envía cada mensaje, entre otros. Por eso especificamos las fallas que tenemos previstas que pueden pasar y cómo las estamos solucionando para poder tener un sistema distribuido resiliente a los fallos.

Fallo: Caída del Servidor Central (server.py)

- **Detección:**
    - Ambos patrones (faculty.py y facultylbb.py) escuchan los heartbeats (HB) mediante zmq.SUB al puerto 7000 desde el servidor primario (10.43.96.50) y el respaldo (10.43.103.51).

    - Si no se recibe un HB dentro de un tiempo límite (HB_LIVENESS * HB_INTERVAL), se considera que el servidor ha fallado.
- **Manejo (Failover):**
    - En faculty.py: se ejecuta _reconnect() para redirigir las solicitudes al servidor de respaldo, manteniendo la transacción con attempt().
    - En facultylbb.py: el hilo dedicado cambia dinámicamente la IP del servidor activo en base al último heartbeat válido. El siguiente socket REQ que se cree se conectará al servidor actualizado.
    - En ambos servidores: el servidor de respaldo pasa a modo activo y comienza a escuchar en tcp://*:5555 y a emitir sus propios heartbeats.

Fallo: Fallos de Comunicación (Timeouts, Mensajes Perdidos)

- **Detección:**
    - **Timeouts de Socket:** Todos los sockets ZMQ (REQ/REP y DEALER/ROUTER) están configurados con SNDTIMEO y RCVTIMEO. Si no se recibe o envía el mensaje en el tiempo definido, se lanza una excepción zmq.Again.
    - **Timeout de Protocolo (ACK):** El servidor espera un ACK por cada PROP mediante un event.wait(ACK_TIMEOUT).
- **Manejo:**
    - **Reintentos (Facultad):** Al detectar zmq.Again, intentan reconectar y reintentar la operación (attempt()).
    - **Expiración de Reservas (Servidor):** Si el servidor no recibe un ACK (quizás por fallo de la facultad o pérdida del ACK), la reserva temporal asociada a esa transacción (reservations[fac]) tiene una marca de tiempo expiry. Un hilo en el servidor (expiration loop) revisa periódicamente estas marcas y si time.time() > expiry, cancela la reserva y libera los recursos (manager.expire()). Esto evita bloqueos indefinidos.
    - **Programa Académico:** Si academic_program.py no recibe respuesta de la facultad, imprime un error de timeout.

Fallo: Agotamiento de Recursos (Aulas Insuficientes)

- **Detección:**
    - El servidor verifica antes de cada PROP si los salones o laboratorios están disponibles (self.classrooms, self.labs).
    - Si no es posible atender la solicitud (ni siquiera con aulas móviles), se detecta agotamiento.
- **Manejo:**
    - Si no hay recursos suficientes para la reserva inicial o para la propuesta, el servidor registra el rechazo en la base de datos.
    - Genera una alerta por pantalla.
    - Envía un mensaje RES a la facultad con status: "DENIED" o status: "CANCELED" y una razón apropiada.
    - El sistema continúa operando y atendiendo otras solicitudes.

Modelo de seguridad

El modelo de seguridad para el sistema de Gestión de Aulas busca proteger sus activos y operaciones, asegurando tres pilares fundamentales:

- **Confidencialidad:** Garantizar que la información sobre solicitudes y asignaciones de aulas solo sea accesible para los componentes autorizados.
- **Integridad:** Asegurar que los datos y mensajes intercambiados no sean modificados de forma no autorizada durante la comunicación o el almacenamiento.
- **Disponibilidad:** Garantizar que el servicio de asignación de aulas esté operativo y accesible para las facultades y programas académicos cuando lo necesiten, incluso ante fallos parciales.

2. Activos a Proteger

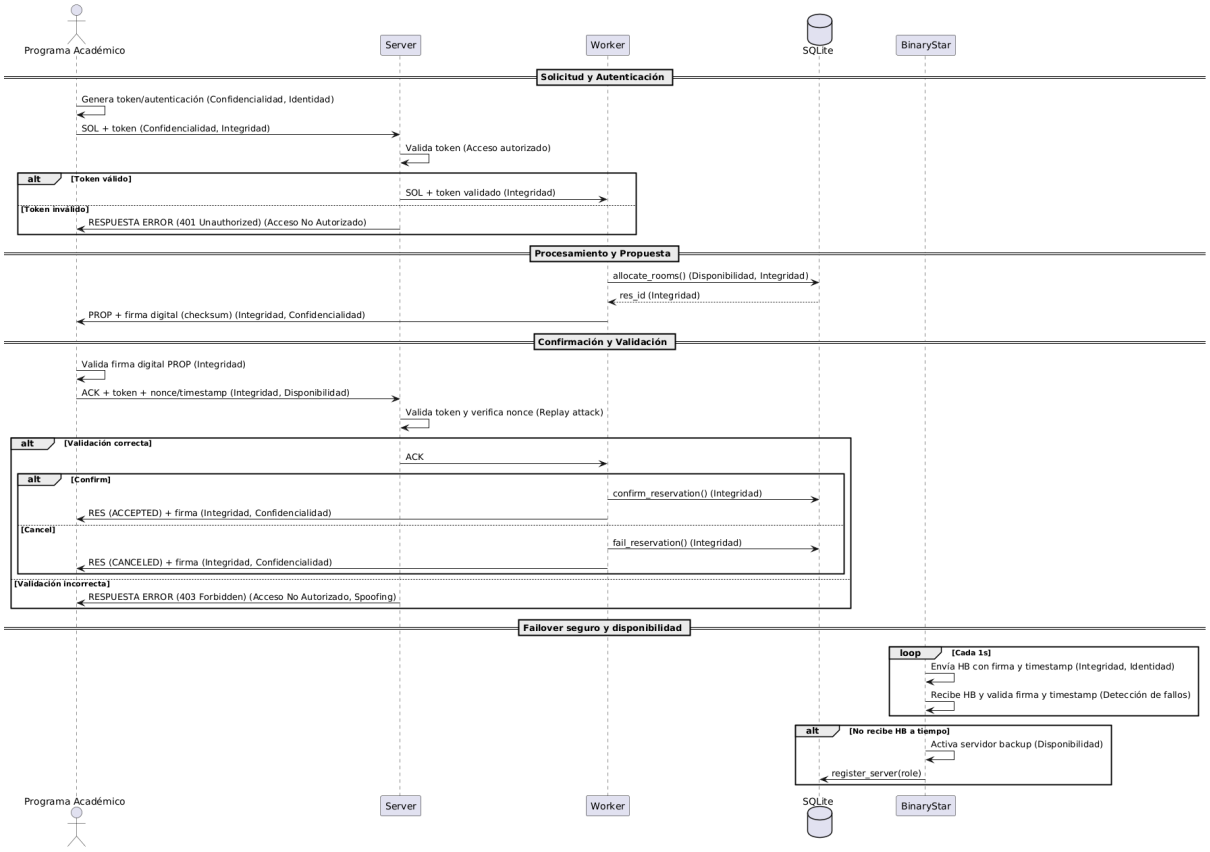
- La funcionalidad del servicio de asignación de aulas.
- La integridad y exactitud del estado de disponibilidad de aulas (salones, laboratorios).
- La confidencialidad de las solicitudes realizadas por los programas y facultades.
- Los canales de comunicación entre los componentes distribuidos.
- La identidad de los componentes legítimos del sistema.

3. Amenazas Consideradas

- **Acceso No Autorizado:** Intentos de conexión y operación por procesos no pertenecientes al sistema.
- **Eavesdropping (Escucha):** Intercepción de la comunicación en la red para obtener información sobre solicitudes o disponibilidad.
- **Message Tampering/Injection (Manipulación/Inyección):** Alteración de mensajes en tránsito o envío de mensajes falsos.
- **Identity Spoofing (Suplantación):** Procesos maliciosos intentando hacerse pasar por facultades, servidores u otros componentes legítimos.

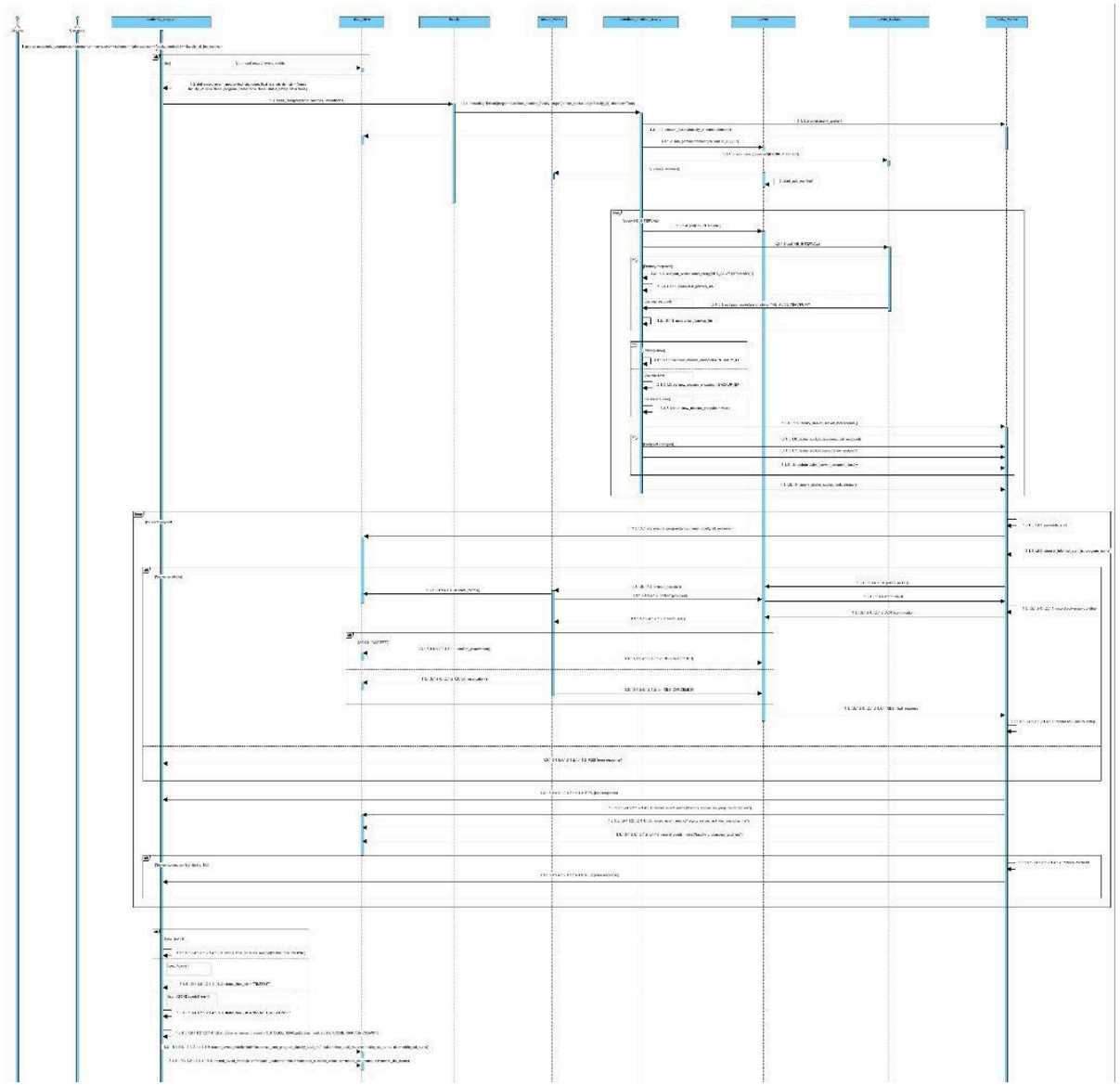
4. Representación gráfica

Lo anteriormente mencionado, se resume en el siguiente diagrama de secuencia:


Programa Académico
Server
Worker
SQLite
BinaryStar
Solicitud y Autenticación
Genera token/autenticación (Confidencialidad, Identidad)
SOL + token (Confidencialidad, Integridad)
Valida token (Acceso autorizado)
alt [Token válido]
SOL + token validado (Integridad)
[Token inválido]
RESPUESTA ERROR (401 Unauthorized) (Acceso No Autorizado)
Procesamiento y Propuesta
allocate_rooms() (Disponibilidad, Integridad)
res_id (Integridad)
PROP + firma digital (checksum) (Integridad, Confidencialidad)
Confirmación y Validación
Valida firma digital PROP (Integridad)
ACK + token + nonce/timestamp (Integridad, Disponibilidad)
Valida token y verifica nonce (Replay attack)
alt [Validación correcta]
ACK
alt [Confirm]
confirm_reservation() (Integridad)
RES (ACCEPTED) + firma (Integridad, Confidencialidad)
[Cancel]
fail_reservation() (Integridad)
RES (CANCELED) + firma (Integridad, Confidencialidad)
[Validación incorrecta]
RESPUESTA ERROR (403 Forbidden) (Acceso No Autorizado, Spoofing)
Failover seguro y disponibilidad
loop [Cada 1s]
Envía HB con firma y timestamp (Integridad, Identidad)
Recibe HB y valida firma y timestamp (Detección de fallos)
alt [No recibe HB a tiempo]
Activa servidor backup (Disponibilidad)
register_server(role)
Programa Académico
Server
Worker
SQLite
BinaryStar

Diagramas de secuencia

Los diagramas de secuencia para ambos patrones son los siguientes:



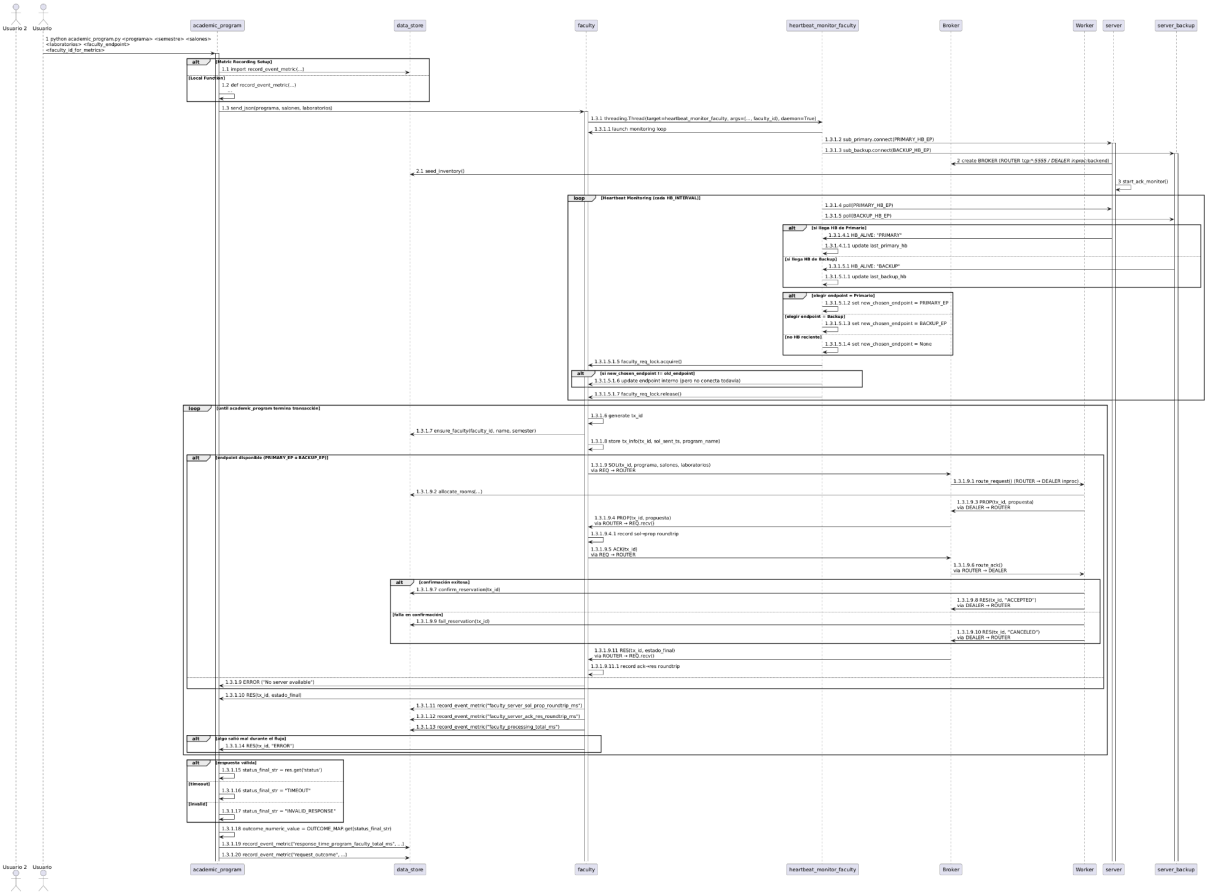
La implementación Async del sistema de gestión de recursos académicos sigue un flujo de comunicación asíncrona basado en el patrón Binary Star para alta disponibilidad. Cuando el sistema se inicia, los servidores (primario y backup) comienzan a emitir señales de heartbeat a través de un canal PUB/SUB, permitiendo que las facultades detecten cuál servidor está activo en cada momento. Las facultades mantienen conexiones persistentes con ambos servidores pero solo interactúan con el activo, cambiando automáticamente cuando detectan que el primario ha fallado.

El proceso comienza cuando un programa académico envía una solicitud a su facultad correspondiente mediante un socket REP/REQ tradicional. La facultad, que opera como un intermediario asíncrono, recibe esta solicitud y la reenvía al servidor activo usando un socket

DEALER, lo que permite comunicación bidireccional sin bloqueo. Este diseño permite que la facultad continúe atendiendo nuevas solicitudes mientras espera respuestas del servidor.

Los workers del servidor procesan las solicitudes en paralelo, cada uno en su propio hilo. Cuando un worker recibe una solicitud (SOL), calcula la propuesta de recursos y la envía de vuelta a la facultad. La facultad entonces responde con un ACK (aceptación) o rechazo, lo que desencadena la confirmación o cancelación de la reserva. Todo este intercambio se gestiona mediante transacciones identificadas únicamente, permitiendo que múltiples solicitudes se procesen concurrentemente sin mezclarse.

El sistema implementa un mecanismo de timeout robusto para gestionar casos donde las facultades no responden a las propuestas. Un monitor dedicado supervisa todas las transacciones pendientes y cancela automáticamente aquellas que exceden el tiempo de espera configurado, liberando los recursos reservados. Esta combinación de comunicación asíncrona, procesamiento paralelo y gestión de transacciones permite que el sistema maneje alta carga manteniendo la capacidad de respuesta incluso en escenarios de fallo parcial.



La implementación LBB sigue un enfoque basado en balanceo de carga para distribuir las solicitudes entre múltiples workers. El sistema inicia cuando un programa académico envía una solicitud a través del comando CLI, especificando los parámetros requeridos como nombre del programa, semestre, cantidad de aulas y laboratorios. Academic_program establece conexión con el endpoint de la facultad correspondiente y envía la solicitud en formato JSON, registrando simultáneamente las métricas iniciales en el datastore o usando una función local como fallback.

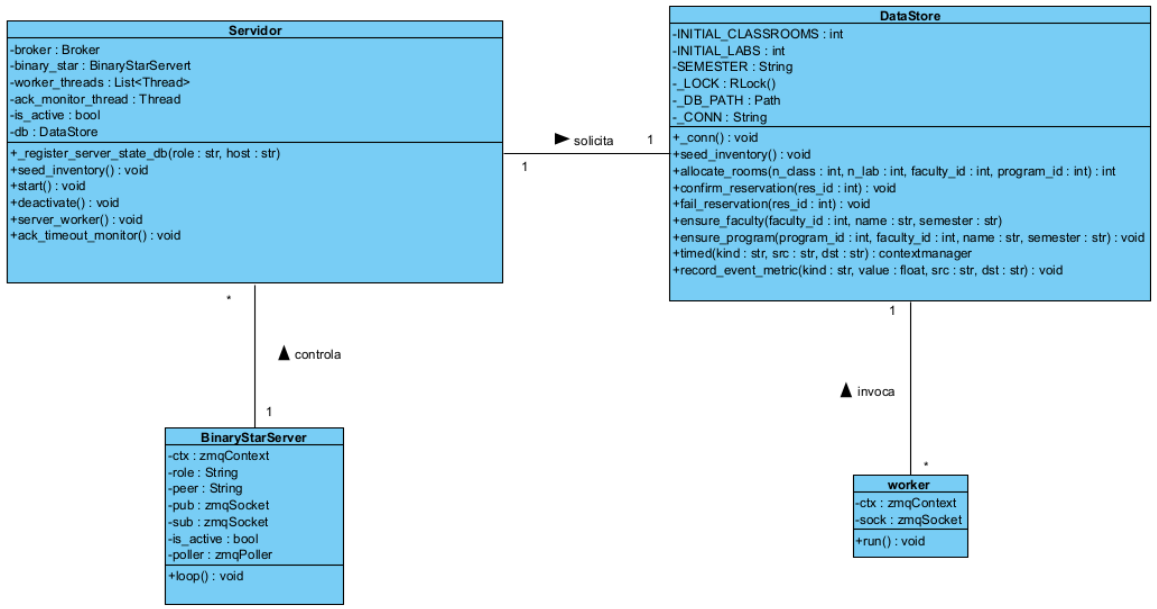
En paralelo, la facultad activa su monitor de heartbeat (HM_F) como un hilo demonio que mantiene conexiones SUB con los servidores primario y backup. Este monitor implementa un ciclo continuo que verifica periódicamente la disponibilidad de los servidores mediante sus señales de heartbeat. Cuando detecta un cambio en el servidor activo (por falla del primario o recuperación del mismo), actualiza internamente el endpoint objetivo pero mantiene la conexión existente hasta que sea necesario cambiar.

El servidor principal, al iniciarse, crea el broker con sockets ROUTER (frontend) y DEALER (backend), inicializa el inventario en la base de datos y arranca el monitor de ACKs. Los workers se conectan al backend del broker y quedan en espera de solicitudes. Cuando llega una solicitud de faculty, el broker la enruta automáticamente a un worker disponible usando el patrón load-balancing.

Cada worker procesa la solicitud (SOL) verificando disponibilidad en el datastore y genera una propuesta (PROP) que envía de vuelta a faculty a través del broker. La facultad registra el tiempo de este intercambio y responde con un ACK que viaja nuevamente al mismo worker a través del broker. El worker entonces confirma o cancela la reserva en la base de datos según corresponda y genera la respuesta final (RES) que llega a faculty para ser reenviada al programa académico original.

Todo el flujo está instrumentado con métricas de tiempo (roundtrip SOL-PROP, ACK-RES y tiempo total de procesamiento) que se registran en el datastore. El sistema maneja automáticamente escenarios de fallo mediante timeouts y cambio entre servidores primario/backup, manteniendo la capacidad de respuesta incluso en casos de fallo parcial. La arquitectura LBB prioriza simplicidad y distribución equitativa de carga sobre la complejidad de la implementación asíncrona, usando conexiones temporales por transacción en lugar de conexiones persistentes.

Diagramas de clases



El servidor sigue una arquitectura modular centrada en tres componentes principales: el Broker, el BinaryStarServer y los Workers, coordinados a través de un DataStore compartido. El sistema inicia cuando el BinaryStarServer arranca su loop principal, estableciendo los sockets PUB/SUB para el monitoreo de heartbeats entre servidores primario y backup. El Broker actúa como intermediario,

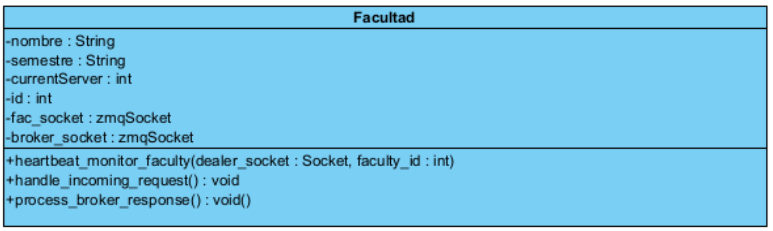
conectando el frontend (ROUTER) que recibe solicitudes externas con el backend (DEALER) que distribuye la carga entre workers.

El DataStore gestiona el estado persistente del sistema, incluyendo el inventario inicial de aulas (INITIAL_CLASSROOMS) y laboratorios (INITIAL_LABS), protegido por un Lock (LOOK_RLock) para acceso concurrente seguro. Proporciona operaciones críticas como allocate_rooms para reservas, confirm_reservation para comprometer recursos, y full_reservation para liberarlos, además de funciones auxiliares como ensure_faculty/program para mantener la integridad referencial.

Los Workers, ejecutándose en hilos independientes (worker_threads), procesan solicitudes recibidas a través del Broker. Cada Worker mantiene su propio socket DEALER conectado al backend e implementa la lógica para: calcular propuestas de asignación, interactuar con el DataStore, y generar respuestas. Un hilo especial (ack_monday_thread) monitorea timeouts en las confirmaciones.

El BinaryStarServer implementa el patrón de alta disponibilidad mediante su loop de monitoreo continuo. Usa zmq.Poller para detectar cambios en la disponibilidad de servidores pares, actualizando el estado is_active según corresponda. En caso de fallo del primario, activa automáticamente el backup y notifica al Broker para redirigir tráfico.

La métrica y temporización son transversales: el decorador timed (contextmanager) mide duraciones de operaciones, mientras que record_event_metric registra eventos significativos. El sistema mantiene coherencia mediante register_server_state_db, que actualiza periódicamente el rol activo (primary/backup) en el DataStore.



La Facultad actúa como intermediario inteligente entre los programas académicos y los servidores de recursos, implementando lógica de conmutación automática entre servidores primario y secundario. El componente mantiene un estado interno que incluye su identificación única (id), nombre descriptivo (nombre) y periodo académico actual (semestre), esenciales para el registro de métricas y validación de solicitudes.

El núcleo de operaciones gira alrededor de dos sockets ZMQ: fac_socket para comunicación REQ/REP con los programas académicos (manejado en handle_incoming_request), y broker_socket (DEALER) para conexión asíncrona con el broker de servidores. La variable currentServer almacena dinámicamente la referencia al servidor activo, permitiendo la reconexión transparente cuando el monitor de heartbeat detecta cambios.

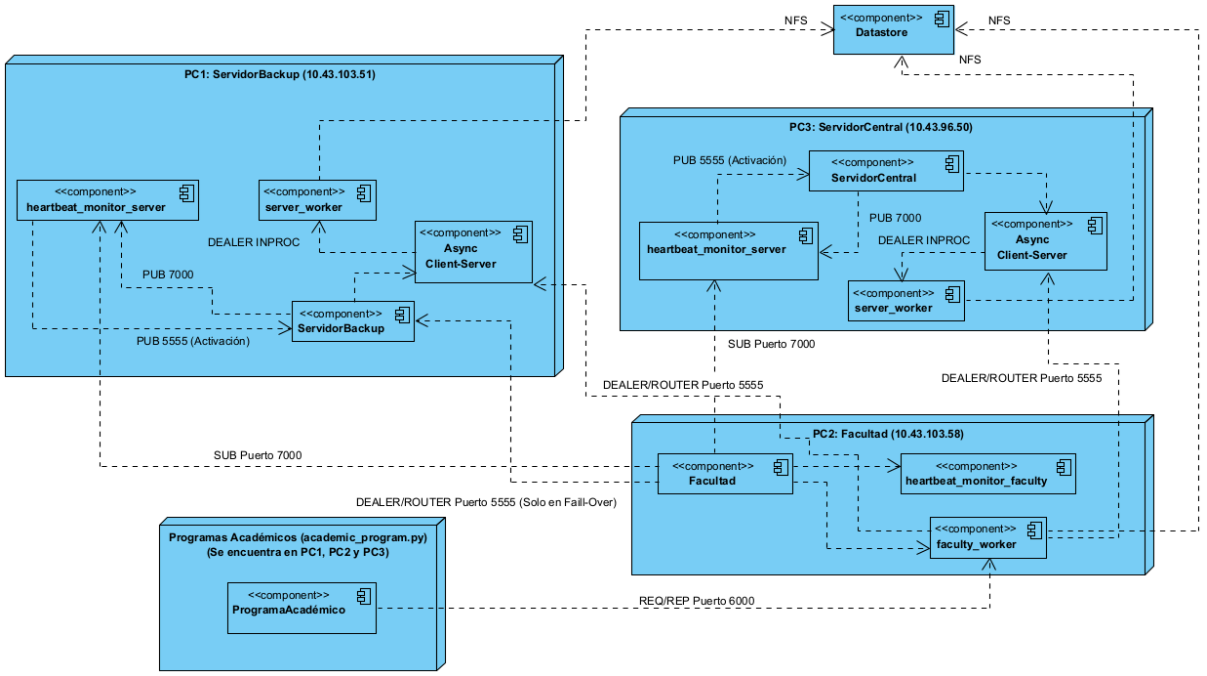
El heartbeat_monitor_faculty opera como hilo independiente, implementando el patrón Binary Star para supervisión continua.

El ProgramaAcadémico es el componente cliente del sistema de gestión de recursos educativos, diseñado para solicitar aulas y laboratorios de manera eficiente. Como punto de entrada principal, este módulo encapsula toda la lógica necesaria para interactuar con las facultades y registrar métricas operacionales. Sus atributos clave incluyen el nombre del programa y el semestre académico, que validan la pertinencia de la solicitud, junto con los campos salones y laboratorios que especifican los recursos requeridos. Además, mantiene identificadores únicos (program_id y faculty_id) que establecen las relaciones institucionales dentro del sistema.

El componente ofrece funcionalidades críticas para el correcto funcionamiento del sistema. Por un lado, garantiza la existencia y validez de los programas académicos a través de ensure_program(), que crea registros en la base de datos cuando es necesario. Por otro, gestiona todo el ciclo de vida de las solicitudes mediante send_request(), construyendo payloads JSON, estableciendo conexiones síncronas con las facultades y manejando timeouts para evitar bloqueos. El procesamiento de respuestas en process_response() interpreta diversos estados como aceptación, rechazo o cancelación, calculando además métricas de rendimiento y proporcionando retroalimentación al usuario final.

Un aspecto destacable del ProgramaAcadémico es su robusto sistema de telemetría, implementado en record_event_metric(). Esta función captura métricas valiosas como tiempos de respuesta, resultados de solicitudes y eventos excepcionales, permitiendo un monitoreo detallado del sistema. La implementación combina autonomía (funcionando incluso con conexiones intermitentes gracias al buffering de ZMQ) con capacidades avanzadas de validación y adaptabilidad, ajustando dinámicamente los timeouts según las condiciones de red. El flujo típico de operación, desde la ejecución inicial hasta la presentación de resultados, está optimizado para ofrecer simplicidad al usuario final mientras mantiene capacidades avanzadas de diagnóstico y registro para los administradores del sistema.

Diagramas de despliegue



Este diagrama de despliegue representa la arquitectura distribuida de un sistema de gestión de aulas con tolerancia a fallos, desplegado sobre tres PCs distintos. Cada nodo del sistema cumple un rol específico dentro de una red distribuida interconectada mediante ZeroMQ, y cuenta con varios componentes internos orientados al procesamiento, comunicación, monitoreo de estado y activación de backups.

PC1: Servidor de Respaldo (10.43.103.51)
Este nodo contiene el Servidor Réplica, que permanece en estado pasivo hasta que se detecta una falla en el servidor central. Incluye un componente heartbeat_monitor_server, encargado de suscribirse al canal de latidos (PUB 7000) para detectar la ausencia de latidos del servidor principal. En caso de fallo, publica un mensaje de activación por el puerto 5555 para activar el backup. El componente server_worker maneja la lógica de atención a solicitudes cuando el servidor está activo. La comunicación interna entre componentes se da mediante DEALER INPROC, y se emplea Async Client-Server para manejar las peticiones entrantes desde los programas académicos.

PC3: Servidor Central (10.43.96.50)
Este es el servidor activo por defecto. Contiene componentes similares a los del backup: Servidor Central, server_worker, heartbeat_monitor_server y el Async Client-Server. El Servidor Central publica latidos a través del puerto 7000 para que los demás nodos puedan verificar su disponibilidad. También publica en el puerto 5555 cuando está activo. Su canal de comunicación hacia otros nodos (como facultades) opera en modo DEALER/ROUTER sobre el puerto 5555, lo que permite escalabilidad y tolerancia a desconexiones temporales.

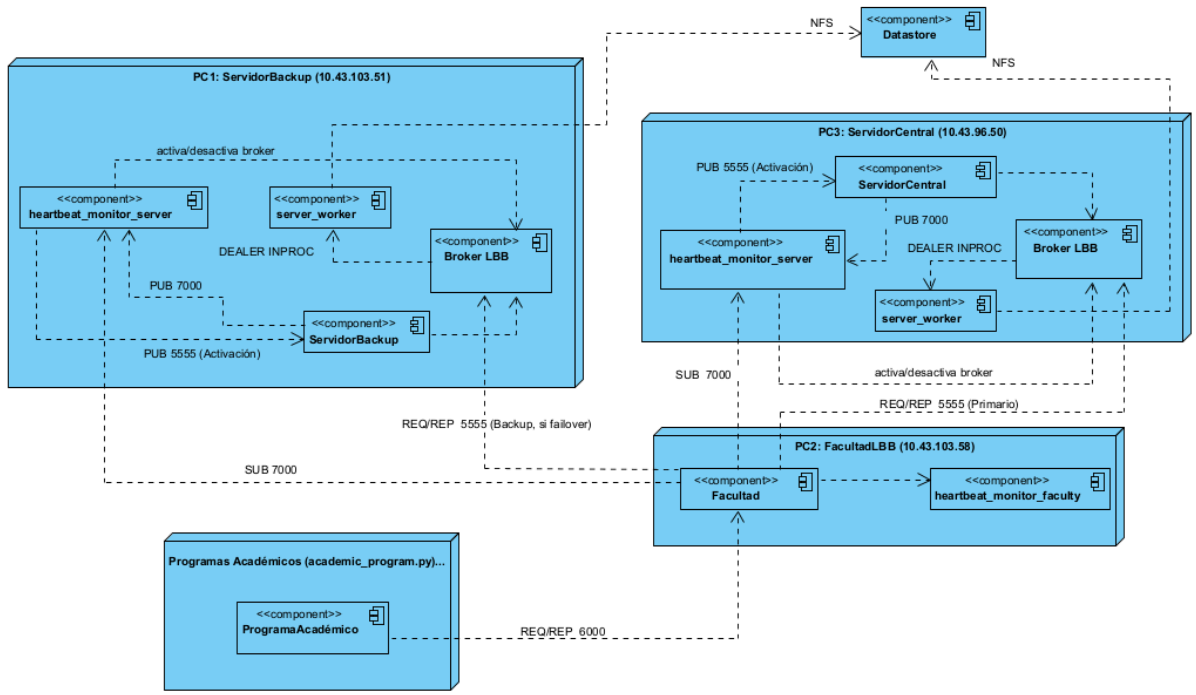
PC2: Facultad (10.43.103.58)
Este nodo representa la interfaz de entrada desde el lado de las facultades. Contiene el componente facultad, que actúa como intermediario entre los programas académicos y los servidores. Utiliza un canal REQ/REP sobre el puerto 6000 para recibir solicitudes desde los programas académicos. También se incluye un faculty_worker para manejar la lógica local de procesamiento, y un

heartbeat_monitor_faculty que monitorea la salud del servidor central mediante suscripción al puerto 7000.

Programas Académicos
Estos se despliegan en todas las PCs (PC1, PC2 y PC3) y están representados por el componente ProgramaAcadémico. Funcionan como clientes que envían solicitudes al componente facultad, el cual se comunica a su vez con el servidor activo (central o backup, según el estado del sistema).

Datastore
Es un componente compartido accedido mediante NFS por los servidores, lo que permite mantener consistencia en los datos, como el estado de las reservas, incluso si se produce un failover.

Red de Comunicación
La arquitectura emplea varios patrones de ZeroMQ: PUB/SUB para los heartbeats y monitoreo, DEALER/ROUTER para la comunicación asincrónica y balanceo de carga entre facultad y servidores, y REQ/REP entre los programas académicos y la facultad. Esto permite una arquitectura desacoplada y tolerante a fallos.



Este segundo diagrama representa una arquitectura distribuida más clara, modular y robusta que la anterior, incorporando tolerancia a fallos mediante failover automático y balanceo de carga centralizado a través de un componente dedicado: el Broker LBB. La infraestructura se compone de tres nodos principales: el Servidor Central (PC3), el Servidor Réplica (PC1) y la FacultadLBB (PC2), además de un nodo cliente que representa a los Programas Académicos. Cada nodo cumple un rol específico y todos están interconectados mediante ZeroMQ utilizando patrones de comunicación como PUB/SUB y REQ/REP.

En la **máquina PC3 (Servidor Central)** reside el servidor primario del sistema. Este servidor es responsable de enviar latidos periódicos por el canal PUB 7000 para indicar su disponibilidad. También publica su activación por PUB 5555 y tiene un componente Broker LBB encargado de

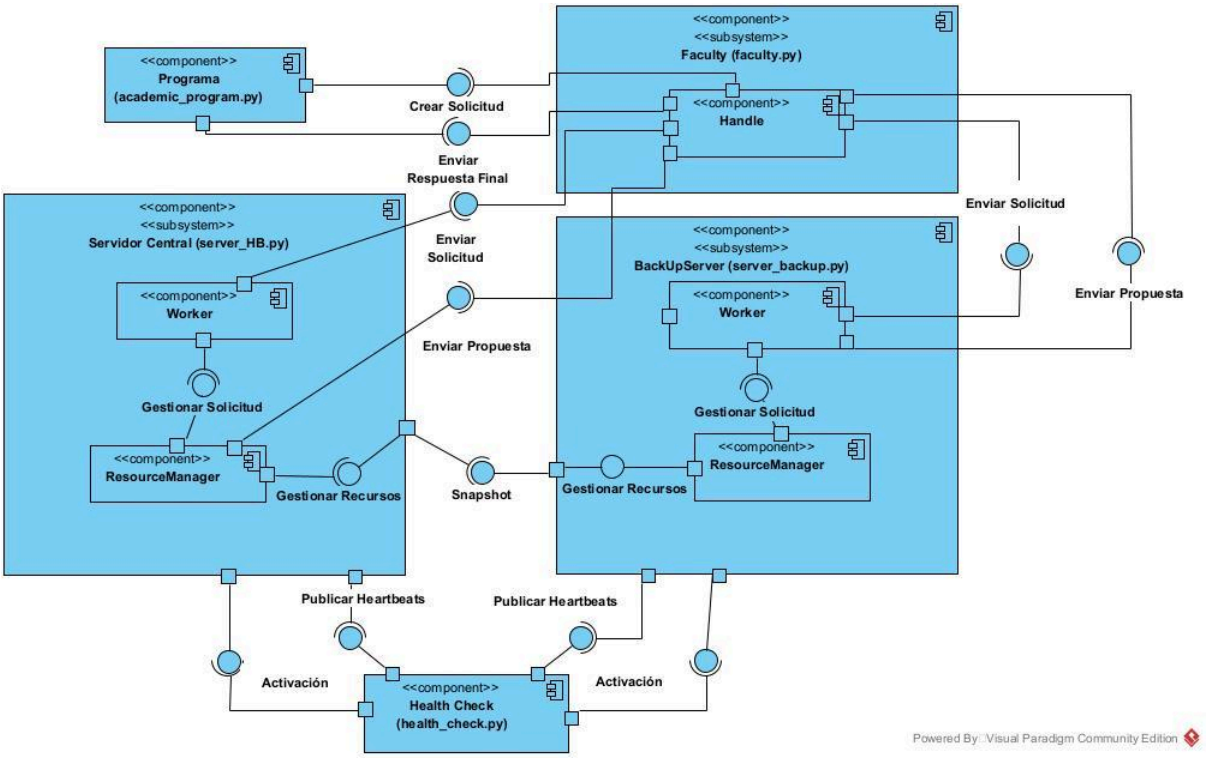
balancear las solicitudes que le llegan desde la Facultad. Estas solicitudes llegan a través del canal REQ/REP 5555, que es escuchado exclusivamente por el broker activo. El componente server_worker, conectado por DEALER INPROC al broker, procesa estas solicitudes.

Por su parte, PC1 (Servidor Réplica) mantiene una réplica pasiva de la lógica del servidor central. Su heartbeat_monitor_server escucha los latidos del servidor central por SUB 7000 y, en caso de detectar un fallo (es decir, ausencia de latidos), activa su propio Servidor Réplica, el cual empieza a emitir latidos (PUB 7000) y se convierte en el nuevo broker. También publica su activación por PUB 5555. Desde este punto, el broker y su server_worker asumen el control, escuchando las solicitudes entrantes desde la Facultad en el puerto 5555, igual que lo hacía el servidor central.

En PC2 (FacultadLBB) se encuentra el módulo de Facultad, que funciona como intermediario entre los programas académicos y el broker activo (ya sea el primario o el de respaldo). Este componente escucha los latidos del servidor activo (SUB 7000) mediante heartbeat_monitor_faculty y, según quién esté activo, redirige las solicitudes académicas al broker correspondiente en el puerto 5555. Esto permite que los programas académicos, conectados por REQ/REP 6000, no necesiten preocuparse por qué servidor está activo, ya que esta lógica está encapsulada en la Facultad.

Finalmente, existe un componente de Datastore compartido mediante NFS que asegura la consistencia entre el servidor central y el de respaldo. Esto permite que, al activarse el backup, continúe con el estado más reciente del sistema, evitando pérdida de datos o inconsistencias en las reservas de aulas o laboratorios.

Diagrama de componentes



Este diagrama de componentes es igual para ambas implementaciones y muestra cómo se orquesta el flujo de solicitudes, propuestas y failover entre cuatro piezas claves:

1. Programa (academic_program.py)
  - Componente ligero que crea la Solicitud y recibe la Respuesta Final.

2. Faculty (faculty.py)
  - Subsistema que engloba al componente Handle.
  - Recibe la solicitud del Programa, la reenvía al servidor activo, y devuelve la propuesta final al Programa.

3. Servidor Central y réplica (server.py)
  - Cada uno es un subsistema con dos componentes internos:
    - Worker: procesa las solicitudes entrantes y genera propuestas, confirma o rechaza la reserva.
    - DataStore: gestiona el inventario de salones y laboratorios.
  - Comparten la misma interfaz de mensajes:
    - Enviar Solicitud desde Faculty → Worker → DataStore → Worker → Enviar Propuesta de vuelta a Faculty y así con el flujo: Recibe SOL y ACK, envía PROP y RES.
  - El backup está normalmente pasivo, pero replica el estado del servidor central y asume la carga si se activa.

En conjunto, estos componentes garantizan que:
- Las facultades envían y reciben sus asignaciones a través de un único punto lógico (Faculty).
- El central atiende primero; el réplica y toma el relevo automáticamente.
- Se sigue el flujo: Recibe SOL y ACK, envía PROP y RES.

Pruebas

Protocolo de pruebas

A continuación, se establece una estrategia de verificación y validación del sistema distribuido, asegurando que cumpla con sus requerimientos funcionales y no funcionales a lo largo del desarrollo. El protocolo cubre pruebas de funcionamiento, comunicación, concurrencia, rendimiento, y resiliencia ante fallos. La siguiente tabla describe cada tipo de prueba y su propósito:

Tipo de Prueba	**Descripción**
Pruebas Funcionales	Verifican que cada componente cumpla correctamente su función según lo establecido en el enunciado.
Pruebas de Integración	Validan que los distintos módulos (Facultades, Servidor, Programas Académicos) se comuniquen e interactúen adecuadamente.

Pruebas de Concurrencia	Aseguran que el Servidor Central atienda múltiples solicitudes simultáneamente sin pérdida de datos o bloqueos.
Pruebas de Rendimiento	Miden tiempos de respuesta y uso de recursos para evaluar la eficiencia del sistema bajo carga normal y pico.
Pruebas de Resiliencia	Verifican la capacidad del sistema para recuperarse ante fallos, especialmente probando la activación del Servidor Backup.
Pruebas de Estrés	Se envían una gran cantidad de solicitudes para probar los límites del sistema y asegurar su estabilidad bajo condiciones extremas.
Pruebas de Seguridad	Validan la correcta configuración del mecanismo, asegurando autenticación, confidencialidad e integridad en las comunicaciones entre componentes del sistema distribuido.

Casos de prueba

A continuación, se presentan los casos de prueba diseñados para validar integralmente el sistema distribuido. Estos casos abarcan lo mencionado en la tabla presentada anteriormente.

Las pruebas serán ejecutadas y documentadas de manera completa en la fase final del proyecto, cuando el sistema se encuentre totalmente implementado, incluyendo la recolección de métricas de rendimiento conforme a lo establecido en el informe final.

Caso de Prueba	Descripción	Resultado Esperado
CP1 – Solicitud básica	Una facultad envía una solicitud con tres programas (ej.: Ing. Sistemas, Ing. Electrónica, Ing. Industrial) con cantidades dentro de los rangos mínimos (ej. 7 salones y 2 laboratorios).	El Servidor Central procesa la solicitud y responde asignando los recursos solicitados correctamente.
CP2 - Concurrencia	Se ejecutan dos (o más) instancias de facultades en paralelo (desde distintas VMs o terminales) enviando solicitudes simultáneas.	El Servidor Central atiende todas las solicitudes de forma concurrente sin errores y se registran las respuestas.
CP3 – Recursos justos	Se solicita exactamente el total de laboratorios disponibles	Asignación correcta sin error, sin aulas móviles
CP4 - Recursos Insuficientes	Una facultad envía solicitudes que superan la disponibilidad de laboratorios (por ejemplo, múltiples programas piden más laboratorios de los 60 disponibles).	El Servidor asigna los recursos disponibles, utiliza aulas móviles para suplir la demanda o genera alerta de "no disponible".
CP5 - Comunicación interrumpida	Simulación de caída del Servidor Central inmediatamente después de enviar una solicitud (error de red controlado).	El cliente (Facultad) gestiona el error mediante conexión al servidor backup y reintento.
CP6 - Persistencia y Registro	El Servidor Central registra el historial de solicitudes y asignaciones en archivos de la base de datos.	Los archivos se generan correctamente con los datos esperados, permitiendo la trazabilidad de la operación
CP7 - Prueba de Rendimiento	Se simula un entorno de alta carga enviando solicitudes con valores máximos (por ejemplo, 10 salones y 4 laboratorios por programa) desde varias facultades y programas, así como un entorno de carga normal con valores mínimos (ej. 7 salones y 2 laboratorios)	Se miden y registran tiempos de respuesta promedios, mínimos y máximos, cumpliendo con los criterios de rendimiento establecidos.

	desde varias facultades y programas.	
CP8 - Prueba de Resiliencia	Se simula una falla en el Servidor Central (PC3) durante el procesamiento de solicitudes para comprobar si el proceso HealthChecker (PC1) activa al Servidor Réplica, el cual asume la carga de manera automática.	El sistema detecta la falla, el Servidor Réplica se activa y continúa atendiendo las solicitudes sin reiniciar el proceso desde cero.
CP9 - Prueba de Estrés	Se envía un volumen muy alto de solicitudes (aumentando la concurrencia y la cantidad de solicitudes) para evaluar la estabilidad y la capacidad de recuperación del sistema.	El sistema se mantiene estable, o en caso de sobrecarga se generan alertas y se documentan los tiempos de respuesta y pérdidas de solicitudes.
CP10 - Caída y reintento	Se simula un fallo de red o pérdida de conexión durante la comunicación con el servidor, forzando el mecanismo de reintento automático por parte de la Facultad.	Reintento exitoso o recuperación automática
CP11- Validación de Seguridad	Se intenta establecer comunicación desde un componente no autorizado hacia cualquiera de los sockets protegidos (Programa-Facultad, Facultad-Servidor).	La conexión es rechazada automáticamente, sin permitir intercambio de mensajes. Se registra el intento fallido en los logs si aplica.

Cabe destacar que la CP7 - Prueba de Rendimiento estará directamente vinculada al Informe de Rendimiento requerido en el apartado "Rendimiento del Sistema" del documento del proyecto. Esta prueba no solo evaluará la eficiencia del sistema bajo condiciones de carga elevada, sino que también permitirá recolectar las métricas especificadas en la Tabla 3: Medidas de Rendimiento, incluyendo el tiempo de respuesta promedio, mínimo y máximo, así como el número de solicitudes atendidas y rechazadas. Los resultados obtenidos serán analizados y presentados en dicho informe, proporcionando una visión clara del comportamiento del sistema en términos de desempeño y capacidad de respuesta.

Tabla 3: Medidas de Rendimiento

	Patrón request-reply asíncrono entre Facultades y Servidor		**Patrón Load Balancing Broker entre Facultades y Servidor**	
	5 Facultades generando peticiones. 5 Programas académicos por Facultad. Cada programa pide los mínimos 7 Aulas y 2 laboratorios (o cómo máximo 2 y 7)	5 Facultades generando peticiones. 5 Programas académicos por Facultad. Cada programa pide los máximos 10 Aulas y 4 laboratorios (o cómo máximo 4 y 10)	5 Facultades generando peticiones. 5 Programas académicos por Facultad. Cada programa pide los mínimos 7 Aulas y 2 laboratorios (o cómo máximo 2 y 7)	5 Facultades generando peticiones. 5 Programas académicos por Facultad. Cada programa pide los máximos 10 Aulas y 4 laboratorios (o cómo máximo 4 y 10)
Tiempo de respuesta promedio (del servidor a las Facultades)				
Tiempo de respuesta mínimo y máximo (del servidor a las Facultades)				
Tiempo promedio desde que los programas hacen los requerimientos hasta que son atendidos				
Por Programa: número de requerimientos que son atendidos de forma satisfactoria				
Por Programas: número de requerimientos que no son atendidos por la Facultad				

Tomado de: Proyecto de Introducción a los Sistemas Distribuidos

Estrategias de ejecución

1. Preparación:
    - Configuración de los scripts en las VMs correspondientes (Servidor Central, Backup, Facultad y ProgramasAcadémicos).
    - Establecimiento del entorno de comunicación mediante ZeroMQ sobre TCP (puertos 5555, 5560, 5557, etc.).
2. Ejecución de Casos de Prueba:
    - Las pruebas se ejecutarán de forma individual y combinada, iniciando procesos concurrentes cuando sea necesario.
    - Registrar todos los resultados (tanto en consola como en los archivos generados) para cada caso de prueba.
3. Medición y Registro:
    - Utilizar time.time() para medir tiempos de respuesta en solicitudes.

    - Se verificará la correcta generación de logs (server_all.txt, faculty_requests.txt, etc)
    - Se documentarán alertas y eventos críticos (timeouts, activación del backup).
4. Análisis de Resultados:
    - Los resultados serán comparados con los valores esperados definidos en cada caso.
    - Se elaborarán tablas y gráficos para las pruebas de rendimiento y estrés.
5. Reporte Final:
    - Se consolidarán todos los resultados en un informe que incluya análisis detallado, métricas de desempeño, gestión de fallos y conclusiones sobre la estabilidad y eficiencia del sistema.

Métricas de desempeño

En este apartado, se define una estrategia para obtener indicadores cuantitativos del rendimiento del sistema distribuido de Gestión de Aulas en condiciones normales y de carga, para evaluar su eficiencia, capacidad de respuesta y robustez ante fallos. Las métricas se utilizarán en la entrega final para sustentar las pruebas de desempeño, resiliencia y estrés. Las métricas que vamos a usar son las siguientes:

Métricas a obtener

Métrica	**Descripción**	**Método de Obtención**
Tiempo de respuesta promedio Servidor-Facultad	Tiempo medio de procesamiento interno del servidor para varias solicitudes, desde que recibe la SOL hasta que envía la RES final	Instrumentación en server.py/server_backup.py: Registrar timestamps al recibir SOL y antes de enviar RES. Calcular diferencias. Almacenar en CSV/JSON. Procesar para obtener Promedio
Tiempo de respuesta mínimo Servidor-Facultad	Tiempo más rápido registrado en una solicitud	De los registros recibidos para el tiempo promedio se consigue el menor
Tiempo de respuesta máximo Servidor-Facultad	Tiempo más lento registrado	De los registros recibidos para el tiempo promedio se consigue el mayor
Tiempo de respuesta promedio Programa-RespuestaFinal	Latencia total experimentada por el programa académico, desde que envía su solicitud inicial hasta que recibe la respuesta final.	Instrumentación en academic_program.py: Registrar timestamps antes de enviar socket.send() y después de recibir socket.recv(). Calcular diferencias.

		Almacenar en CSV/JSON. Procesar para Promedio
Tiempo de respuesta mínimo Programa-RespuestaFinal	Tiempo más rápido registrado en una solicitud	De los registros recibidos para el tiempo promedio se consigue el menor
Tiempo de respuesta máximo Programa-RespuestaFinal	Tiempo más lento registrado	De los registros recibidos para el tiempo promedio se consigue el mayor
Número de Solicitudes Atendidas Exitosamente	Conteo de las solicitudes que finalizaron correctamente con una asignación de aulas confirmada (status: "ACCEPTED").	Instrumentación en faculty.py o análisis de logs/CSV/JSON: Registrar el estado final de cada transacción. Contar los éxitos (total y por programa).
Número de Solicitudes Denegadas/Fallidas	Conteo de las solicitudes que no pudieron ser completadas (status: "DENIED", "CANCELED", timeouts, etc.).	Instrumentación en faculty.py o análisis de logs/CSV/JSON: Registrar el estado final de cada transacción. Contar los fallos/denegaciones (total y por programa).
Cantidad de solicitudes por segundo	Tasa de procesamiento del sistema bajo carga	División entre cantidad total y tiempo total registrado por el sistema operativo.
Tiempo de recuperación	Tiempo desde la detección de la caída hasta la activación del servidor réplica	Medición con temporizadores en la base de datos.

Registro de resultados

- Los tiempos de respuesta individuales serán registrados mediante scripts auxiliares en archivos de formato .csv o .json, permitiendo un análisis estadístico de métricas como tiempos promedio, mínimos y máximos.
- Las alertas generadas cuando el servidor detecta insuficiencia de recursos (salones o laboratorios) se registran tanto en la consola como en la base de datos.
- Durante la prueba de resiliencia, el tiempo transcurrido entre la detección de la caída del Servidor Central y la activación efectiva del Servidor Réplica será documentado revisando los logs del sistema y las marcas de tiempo en consola y en especial la medición de temporizadores, asegurando así la trazabilidad del proceso de failover.

Herramientas a utilizar

- time.time():Utilizada en los clientes (Programas Académicos y/o Facultad) y en los servidores (Servidor Central y Servidor Réplica) para medir los tiempos de envío y recepción de solicitudes, permitiendo calcular los tiempos de respuesta del sistema.

- htop / top: Herramientas de monitoreo en tiempo real que serán utilizadas para la observación directa del uso de recursos (CPU, memoria) en las máquinas virtuales durante la ejecución de las pruebas de carga y estrés.

- Base de datos: Se utilizarán los registros en la base de datos para analizar el comportamiento ante eventos críticos, como la recuperación tras fallos o la gestión de solicitudes rechazadas.

Presentación de Resultados

Los datos recopilados serán analizados y representados en:

- Tablas estadísticas: con promedios, máximos y mínimos por caso de prueba.

- Gráficos: de tiempos de respuesta y uso de recursos durante las pruebas de estrés y resiliencia.

- Informe final: con interpretación de resultados y conclusiones