# Sistemas Distribuidos escalares

## Objetivo

El objetivo en esta práctica es trabajar con 4 tecnologías diferentes de sistemas distribuidos: XMLRPC, PYRO, Redis y RabbitMQ. Para trabajar diferentes niveles de peticiones a 2 servicios que tienen que proporcionar cada una: InsultService y InsultFilter. Adaptando las arquitecturas de cada tecnología a ambos servicios para poder ser eficientes y escalables a medida que se le añadan más peticiones y nodos de trabajo, para más tarde calcular su comportamiento a más peticiones y su speedup.

## Inicio

En el desarrollo de la práctica reciente varios sistemas distribuidos para gestión de tareas en tiempo real, se implementaron diversas tecnologías para abordar necesidades específicas de comunicación, escalabilidad y procesamiento asíncrono a través de diferentes arquitecturas. Pero cada tecnología tiene una integración o uso diferente.

**XML-RPC (Remote Procedure Call)**

XML-RPC es un protocolo que permite ejecutar funciones o métodos en un servidor remoto como si fueran locales, utilizando XML para codificar las solicitudes y respuestas, y HTTP como transporte. En Python, la implementamos con la biblioteca estándar xmlrpc, que abstrae la comunicación cliente-servidor.

Su funcionamiento se basa en un servidor que expone métodos públicos mediante un endpoint HTTP. Donde los clientes envías una solicitud XML con el nombre del método y sus parámetros. El servidor procesa la llamada, genera una respuesta XML y la devuelve al cliente.

Como características que nos interesan tendríamos, por ejemplo, que su funcionamiento es síncrono, por lo que el cliente espera bloqueado a recibir la respuesta correspondiente. Al ser multiplataforma permite integra varios sistemas heterogéneos, como Python y contendores Docker con otros servicios como Redis. Y, por último, tiene una arquitectura muy básica lo que la hace fácil de implementar, pero complicada de tratar con operaciones de alto rendimiento.

**Pyro (Python Remote Objects)**

Pyro es una biblioteca específica de Python diseñada para crear sistemas distribuidos transparentes, donde los objetos remotos se manipulan como si estuvieran en la misma máquina. Utiliza un mecanismo de serialización nativo de Python (Pickle) y TCP/IP para la comunicación.

El sistema opera mediante un servidor que registra objetos Python en un servicio de Pyro, el cual permanece a la escucha en un puerto designado. Los clientes establecen una conexión con este servicio, adquieren una referencia al objeto remoto y ejecutan sus métodos como si fueran locales. Pyro se encarga de forma transparente de la serialización de datos, la redirección de llamadas y la gestión de errores durante el proceso.

Está orientado a Python, por lo que no requiere configuración compleja ni adaptadores para otros lenguajes. Soporta comunicación bidireccional, permitiendo, por ejemplo, callbacks desde el servidor al cliente. Es ligero, pero no está diseñado para escalar en entornos con cargas masivas.

**Redis**

Redis es una base de datos en memoria que funciona como almacén clave-valor, pero con capacidades avanzadas para estructuras de datos (listas, conjuntos, streams) y mensajería pub/sub (publicación-suscripción). En Python, se integra mediante la biblioteca redis-py.

Redis actúa como un sistema versátil que opera en varios modos, pero principalmente son tres. Primero, como caché, almacenando datos temporales con tiempos de expiración definidos para optimizar el acceso frecuente. Segundo, como sistema de mensajería, donde los publicadores envían mensajes a canales específicos y los suscriptores los reciben en tiempo real, facilitando la comunicación asíncrona. Y tercero, con persistencia opcional, permitiendo guardar datos en disco mediante snapshots periódicos (RDB) o logs de operaciones (AOF) para garantizar durabilidad ante reinicios o fallos. Esta flexibilidad lo hace adecuado para escenarios que requieren velocidad, comunicación en tiempo real o robustez.

Redis destaca por ser ultra rápido, ya que al operar completamente en memoria es ideal para aplicaciones sensibles a la latencia que requieren respuestas en milisegundos. Además, es extremadamente versátil, pudiendo utilizarse como cola de mensajes simple, sistema de caché distribuido o incluso para notificaciones instantáneas gracias a su modelo de publicación/suscripción. Sin embargo, es importante tener en cuenta que no está diseñado para reemplazar sistemas de mensajería complejos, ya que carece de funciones avanzadas como garantías de entrega, particionado sofisticado o procesamiento por lotes a gran escala.

**RabbitMQ**

RabbitMQ es un broker de mensajería que implementa el protocolo AMQP (Advanced Message Queuing Protocol). Actúa como intermediario para enviar, recibir y almacenar mensajes entre aplicaciones, garantizando entrega confiable. En Python, se utiliza con bibliotecas como pika.

Los productores publican mensajes en exchanges (rutas), los cuales enrutan estos mensajes a colas específicas según reglas predefinidas. Los consumidores reciben los mensajes de las colas y los procesan de manera asíncrona, permitiendo un flujo de trabajo desacoplado y escalable. Además, el sistema ofrece garantías avanzadas como persistencia de mensajes, confirmaciones de entrega (acknowledgments) para asegurar que los mensajes se procesen correctamente, y mecanismos de reintento en caso de fallos, lo que lo hace robusto y confiable para entornos empresariales.

Este sistema destaca por ser altamente escalable, ya que soporta clustering y réplicas para garantizar alta disponibilidad y tolerancia a fallos. Además, es notablemente flexible, permitiendo implementar diversos patrones como publicación/suscripción (pub/sub), colas prioritarias o balanceo de carga según las necesidades del proyecto. Sin embargo, su potencia viene acompañada de cierta complejidad, ya que requiere comprender conceptos avanzados como exchanges, bindings y políticas de alta disponibilidad (HA), lo que puede implicar una curva de aprendizaje más pronunciada para nuevos usuarios.

## Desarrollo

El desarrollo de las 4 tecnologías se ha basado principalmente en dos entornos diferentes primeramente el XMLRPC y el Pyro en un Cloud Computing llamado Replit, para facilitar y liberar carga de trabajo en el portátil el cual trabajaba y evitar que afectara al rendimiento del sistema. Por otra parte, tenemos el Redis y el RabbitMQ que sí se han ejecutado y desarrollado en mi portátil de estudiante ya que Replit trabaja con NixOS en vez de con Docker y dificultaba el hecho de implementar un servidor Redis o RabbitMQ, por lo que se ha mantenido la ejecución con el uso de contenedores.

Para el desarrollo Uni-Nodo y Multi-Nodo hemos usado una estructura de códigos donde se pasa por parámetro el número de peticiones por cliente y el número de nodos con los que trabajaremos que serán también el mismo número de threads con los que trabajarán los clientes para enviar información a los workers. Por lo que para el SpeedUp usaremos la del únicamente con un nodo en comparación al resto.

Y para el dinámico lo he aplicado al Redis con el servicio InsultFilter cargando las peticiones en una cola y en base a esta estructurar los workers que se ejecutarán.

**XMLRPC**

En este caso lo que tenemos a la hora de hacerlo multi-nodo es el dónde guardaremos la información para que sea accesible desde varios puntos de acceso sin acabar generando una arquitectura uni-nodo. Ya que aquí tenemos que crear cada estructura de manera que estén bien conectadas sabiendo que es una tecnología con una latencia de conexión y petición considerables.

Otro caso sería el cómo crear diferentes nodos ya que cada uno debe tener un puerto diferente y una uri diferente. Este aspecto nos interesa ya que el cliente hará peticiones directamente a la uri que le interesa ya que de alguna manera necesitaremos que esta información llegue de alguna manera u otra al cliente y este adapte sus peticiones a la cantidad de workers creados.

Como decisión de diseño previa a la arquitectura final haremos que los clientes envíen peticiones con threads, haciendo que este número sea el mismo que el de nodos (workers) para un mejor desempeño.

Para esta arquitectura tenemos el siguiente esquema de trabajo en la *figura 1.1* donde vemos que los principales archivos de la estructura son 2 clientes con peticiones constantes diferentes el servidor que en la práctica son los workers ya que tratan las peticiones. El storage que almacena la lista de insultos y sincroniza con el resto de workers para que tengan la lista actualizada ellos mismos. Y por último el nameserver que encarga de recibir peticiones de los servidores para poder pasar información a los clientes y que traten de la mejor manera las peticiones y sepan dónde enviarlas.

Interfaz de usuario gráfica, Texto, Aplicación

El contenido generado por IA puede ser incorrecto.

*Figura 1.1 (Estructura archivos XMLRPC)*

*Figura 1.1 (Estructura archivos XMLRPC)*

En la *figura 1.2* Podemos observar la arquitectura final aplicada al sistema XMLRPC, y como relacionamos las estructuras creadas para poder tratar tanto sistemas uni-nodo, como sistemas multi-nodo.

Para empezar para poder crear diferentes workers de manera continua con un único archivo se ha hecho que se pase por parámetro el puerto y generar tantos workers como queramos de manera rápida. Y los clientes

Diagrama

El contenido generado por IA puede ser incorrecto.

Figura 1.2 (Diagrama del funcionamiento de la arquitectura XMLRPC)

Gráfico

El contenido generado por IA puede ser incorrecto.

## Desarrollo