



Departament d'Enginyeria  Informàtica i Matemàtiques  UNIVERSITAT ROVIRA I VIRGILI	Sistemas Distribuidos
	SD
	Curso 24/25
	Primera Convocatoria
	Práctica 1: Scaling-Distributed-Systems

Diseño, Arquitectura y Experimentación en Scaling-Distributed-Systems

Miriam Durán Galindo
URV 2024-25

Índice

Diseño y Arquitectura	3
InsultServer	3
Diseño del LoadBalancer	4
Solución XML-RPC	4
Solución Pyro	4
Solución Redis	5
Solución RabbitMQ	5
Escalabilidad con Round Robin Dinámico	5
Experimentación y Resultados	6
Análisis de los Resultados	6
Resumen del Análisis	8
Conclusión	9

Diseño y Arquitectura

El diseño del InsultServer sigue un enfoque basado en arquitectura distribuida y escalabilidad dinámica. Se han implementado cuatro soluciones diferentes para manejar peticiones de manera eficiente, todas utilizando el mismo InsultServer como una instancia de clase pero con distintos mecanismos de comunicación y balanceo de carga.

A continuación, se explican los principales atributos que componen la clase:

InsultServer

El InsultServer es un servicio que genera insultos de manera aleatoria y los distribuye a los clientes. Su funcionamiento principal incluye:

- Contiene una lista de 100 frases ofensivas y un mecanismo de filtrado de palabras inapropiadas.
- Permite que los clientes se suscriban para recibir insultos nuevos en tiempo real. Cada vez que un cliente demanda un insulto este es primero filtrado por la función de filtraje para después ser enviado al consumidor.
- Se comunica a través de protocolos como XML-RPC, Pyro, Redis y RabbitMQ.

Por último, justo después de la clase, contiene una función para inicializar el servidor, de esta manera podemos llamar remotamente a esta función para añadir un nuevo servidor cuando sea conveniente.

Función de escalado dinámico

Para la implementación del dinamismo en los diferentes sistemas se ha escogido la fórmula de escalado utilizando la tasa de llegada de mensajes.

$$N = [(\lambda * T)/C]$$

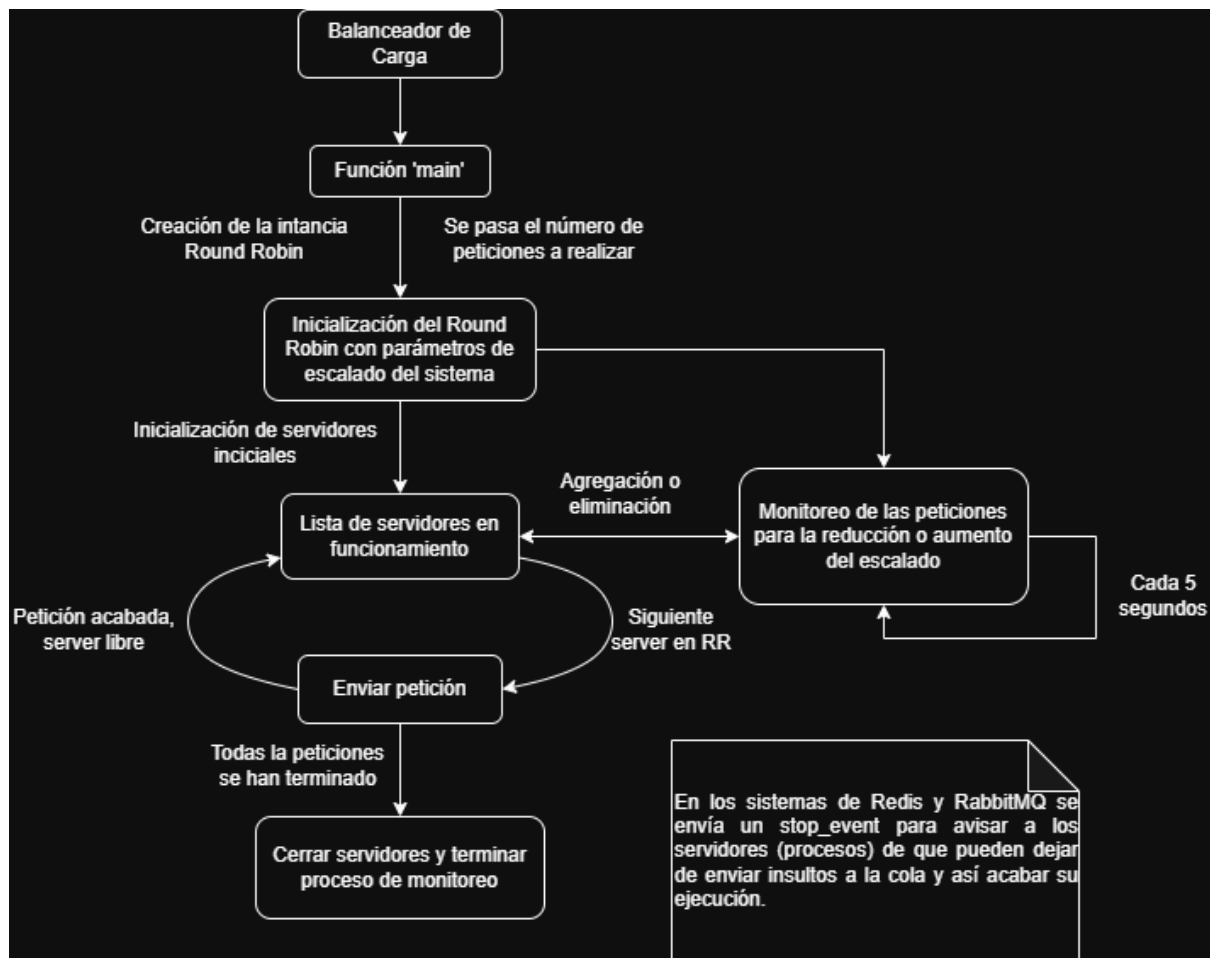
Donde:

- N = Número de trabajadores requeridos
- λ = Tasa de llegada de mensajes (mensajes por segundo)
- T = Tiempo promedio de procesamiento por mensaje (segundos por mensaje)
- C = Capacidad de un solo trabajador (mensajes por segundo)

La tasa de llegada se obtendrá a partir de la medición del tiempo que hacemos desde el último monitoreo. Se guardará en una variable el número de peticiones nuevas que llegan y se dividen por la medición del tiempo para obtener el valor de λ .

El tiempo promedio y la capacidad de un solo trabajador se ha obtenido a partir del análisis de los sistemas con un único nodo.

Diseño del LoadBalancer



Solución XML-RPC

Protocolo XML-RPC: XML-RPC es un protocolo de llamada a procedimiento remoto (RPC) que utiliza XML para codificar las solicitudes y respuestas, y HTTP como protocolo de transporte.

Funcionamiento: Permite la comunicación entre diferentes sistemas a través de una interfaz simple basada en XML, donde un cliente puede invocar métodos remotos en un servidor. A partir del LoadBalancer se realiza un threadpool con todas las peticiones y gracias a la cola Round Robin los servidores irán rotando para servirlos.

Limitaciones: Aunque es fácil de implementar, XML-RPC es menos eficiente en comparación a las otras soluciones debido a la sobrecarga del uso de XML y HTTP.

Solución Pyro

- **Protocolo Pyro:** Pyro permite exponer los métodos del servidor "InsultServer" de manera remota, facilitando la comunicación entre objetos distribuidos.

- **Servidor Pyro:** El servidor maneja las solicitudes de los clientes remotos, quienes acceden a él mediante un objeto proxy.
- **Escalabilidad:** Se implementa un balanceo de carga dinámico mediante el método "Round Robin", permitiendo que cada nuevo servidor agregado maneje una parte equitativa de las solicitudes.
- **Solicitudes:** Para enviar múltiples solicitudes a la vez se crea un threadpool para mantener las peticiones en espera hasta que un servidor se encargue de ellas.

Solución Redis

- **Protocolo Redis:** Se utiliza Redis como sistema de mensajería en un esquema productor-consumidor.
- **Solicitudes:** El InsultServer envía insultos hasta que un stop_event le avisa de que las peticiones han sido satisfechas. De esta forma, no sobrecargamos la cola y el servidor solamente trabaja lo justo y necesario.
- **Suscriptores:** Los clientes son suscriptores del servidor y reciben los insultos que se publican en tiempo real. En este diseño, LoadBalancer hace de suscriptor a la vez que maneja el balance de servidores. A través del main consume los insultos que los servidores van colocando en la cola.
- **Escalabilidad:** Redis permite un alto rendimiento en el procesamiento de solicitudes y se puede escalar horizontalmente.

Solución RabbitMQ

- **Protocolo RabbitMQ:** RabbitMQ permite gestionar la comunicación entre "InsultServer" y los clientes a través de un sistema basado en colas.
- **Servidor RabbitMQ:** Los mensajes se publican en una cola de RabbitMQ, de la que los clientes pueden suscribirse para recibir los insultos. Se colocan tantos mensajes hasta que los suscriptores reciben un stop_event a través del callback para avisarles de que todas las peticiones han sido satisfechas.
- **Suscriptores:** Los clientes son suscriptores a esa cola y consumen los insultos en tiempo real. En este diseño, LoadBalancer hace de suscriptor a la vez que maneja el balance de servidores. A través del main consume los insultos que los servidores van colocando en la cola.
- **Escalabilidad:** RabbitMQ está diseñado para ser altamente escalable, distribuyendo mensajes entre múltiples servidores y optimizando la carga de trabajo.

Escalabilidad con Round Robin Dinámico

El balanceo de carga dinámico basado en Round Robin consiste en lo siguiente:

1. Se mide la tasa de llegada de solicitudes mediante un hilo de monitorización constante y se ajusta el número de servidores.

2. Se añaden o eliminan servidores dependiendo del tráfico.
3. Se usa `itertools.cycle` para repartir las peticiones de manera uniforme.

Experimentación y Resultados

Para evaluar el rendimiento del sistema, se realizaron experimentos con diferentes cargas de trabajo. Se han realizado los tres siguientes gráficos:

1. Single-node Performance Analysis
2. Multiple-node Static Scaling Performance
3. Speedup in Multiple-node Static Scaling

Análisis de los Resultados

1. Gráfico Single-node Performance Analysis:

En este gráfico se muestra el rendimiento de cada uno de los sistemas de comunicación al manejar solicitudes por segundo en un único nodo.

Resultados:

- RabbitMQ destaca significativamente con 437.03 solicitudes por segundo, mostrando una eficiencia impresionante para manejar solicitudes.
- Redis le sigue con 9.80 solicitudes por segundo, lo que indica que es también una solución eficiente para un solo nodo, aunque no tan eficiente como RabbitMQ.
- XMLRPC realiza 3.88 solicitudes por segundo, siendo el sistema menos eficiente entre los cuatro.
- Pyro es el sistema más lento con 0.97 solicitudes por segundo, lo que indica que tiene una sobrecarga significativa en comparación con las demás soluciones.

El rendimiento de cada sistema varía considerablemente. RabbitMQ es claramente el más eficiente para manejar solicitudes por segundo, seguido de Redis, mientras que XMLRPC y Pyro muestran un rendimiento mucho más bajo. Según los resultados vistos podemos decir que para sistemas de alta carga de trabajo, RabbitMQ y Redis son opciones mucho más robustas y escalables que XMLRPC y Pyro.

Sin embargo, se ha de tener en cuenta que la diferencia de rendimiento entre XML-RPC/Pyro y Redis/RabbitMQ puede deberse en parte al uso de Docker en los últimos ya que Docker proporciona un entorno optimizado y aislado, lo que permite una mejor gestión de los recursos.

2. Gráfico Multiple-node Static Scaling Performance:

Este gráfico muestra el tiempo que tarda cada sistema en procesar solicitudes a medida que se aumenta el número de nodos, es decir, cómo se comportan estos sistemas al añadir 1, 2 y 3 nodos.

Aclarar que el número de peticiones demandadas ha sido de 100 para cada servidor, de esta forma están en igualdad de condiciones.

Resultados:

- XMLRPC mantiene un tiempo constante de aproximadamente 18.46 segundos al añadir más nodos (1, 2 o 3). Esto sugiere que XMLRPC no está aprovechando de manera eficiente la escalabilidad vertical, lo que puede indicar cuellos de botella en la comunicación.
- Pyro también muestra tiempos constantes cerca de los 64.83 segundos, sin mejora significativa con el aumento de nodos. Ocurre algo parecido al XMLRPC.
- Redis muestra una mejora considerable en el tiempo de procesamiento al escalar. Con 1 nodo tarda 10.20 segundos, pero con 3 nodos se reduce a 3.63 segundos. Esto indica que Redis escala muy bien y logra reducir los tiempos de procesamiento con más nodos.
- RabbitMQ es el que muestra la menor cantidad de tiempo en comparación con los otros sistemas, comenzando con 0.23 segundos en 1 nodo y aumentando ligeramente a 0.29 segundos con 3 nodos. Este pequeño aumento podría ser debido a que el número de peticiones es demasiado bajo para RabbitMQ y por lo tanto el número de nodos podría estar siendo demasiado grande (desaprovechando recursos) y creando una pequeña congestión en el transcurso de la realización de requests.

RabbitMQ demuestra una escalabilidad eficiente, mostrando un aumento marginal en el tiempo de procesamiento con más nodos. Redis también muestra una mejora significativa con más nodos, lo que lo convierte en una excelente opción para aplicaciones que requieren escalabilidad. En cambio, XMLRPC y Pyro no aprovechan bien el aumento de nodos, lo que indica que no están optimizados para escalabilidad en arquitecturas distribuidas.

Al igual que la gráfica anterior, hay que tener en cuenta que RabbitMQ y Redis se simulan desde Docker y por lo tanto permite una mejor gestión de los recursos mientras que XMLRPC y Pyro desde los mismos recursos del ordenador lo que limita su rendimiento.

3. Gráfico Speedup in Multiple-node Static Scaling:

Este gráfico muestra el speedup o aceleración en el tiempo de procesamiento cuando se añaden más nodos, es decir, cuánto mejora el rendimiento al agregar recursos adicionales.

Resultados:

- XMLRPC no muestra aceleración, ya que su speedup es cercano a 1 en todos los casos. Esto significa que el sistema no se beneficia de la adición de nodos, lo que indica que los recursos no se están gestionando bien para la escalabilidad.
- Pyro también muestra un speedup cercano a 1, indicando que la adición de nodos no mejora significativamente el rendimiento.
- Redis muestra un speedup bastante significativo, con valores alrededor de 2.81 al agregar 3 nodos. Esto refleja su capacidad de escalado y optimización, lo que lo hace muy adecuado para manejar cargas de trabajo distribuidas.
- RabbitMQ muestra un speedup mucho menor, pero aún así existe una mejora con el incremento de nodos, aunque no tan pronunciada como en Redis. A pesar de que RabbitMQ es eficiente, su estructura de colas y el manejo de mensajes pueden generar un límite en la aceleración.

Redis es el sistema que mejor escala en términos de speedup. RabbitMQ tiene una escalabilidad moderada, mientras que XMLRPC y Pyro no muestran mejoras notables en rendimiento al añadir nodos, lo que implica que no están diseñados para escalar de manera eficiente y además no contienen los mismos recursos que los sistemas que están utilizando Docker..

Resumen del Análisis

RabbitMQ y Redis son las mejores soluciones para manejar grandes volúmenes de tráfico y escalar eficientemente, mientras que Pyro y XMLRPC no están diseñados para entornos distribuidos de alto rendimiento.

Conclusión

El sistema implementado demuestra ser capaz de manejar grandes volúmenes de tráfico de manera eficiente gracias a su enfoque dinámico. Se observa que el balanceo de carga basado en Round Robin y el ajuste automático de servidores permiten mantener un rendimiento óptimo sin intervención manual.

Haciendo un seguimiento del análisis de los resultados hemos visto que en cuanto a la capacidad de trabajo de un solo servidor RabbitMQ es claramente el más eficiente, seguido de Redis. Pyro y XMLRPC presentan un rendimiento significativamente más bajo.

Pasando a la escalabilidad, Redis y RabbitMQ son las mejores opciones para escalar en sistemas distribuidos. Redis muestra una mejora considerable con más nodos, mientras que RabbitMQ es altamente eficiente en la gestión de la carga, aunque con una aceleración menor. Sin embargo, Pyro y XMLRPC no muestran mejoras en el rendimiento al agregar más nodos, lo que los hace menos adecuados para aplicaciones que requieren escalabilidad dinámica.

En resumen, se ha observado que la implementación del balanceador de carga dinámica ha permitido reducir significativamente los tiempos de respuesta de las peticiones. En sistemas como Redis y RabbitMQ, donde se utilizan contenedores Docker, la diferencia en los tiempos de procesamiento es mucho más notable debido a la capacidad de escalar de manera eficiente en entornos aislados lo que permite una mejor gestión de los recursos y facilita la escalabilidad horizontal. En cambio, en sistemas como XMLRPC y Pyro, que hacen uso de los recursos locales del PC, la mejora no es tan pronunciada. Esto se debe a que los recursos del PC son limitados, lo que restringe la eficacia del escalado. A pesar de lanzar múltiples servidores, la sobrecarga adicional no se distribuye de manera tan eficiente, y por lo tanto, los servidores adicionales no logran aprovechar todo su potencial de mejora en el rendimiento.