2025

SISTEMAS DISTRIBUIDOS ESCALARES

ESTUDIO SOBRE DIFERENTES TECNOLOGÍAS Y ARQUITECTURAS EN SISTEMAS DISTRIBUIDOS

IVÁN GARCÍA PALLARÉS

UNIVERSITAT ROVIRA I VIRGILI

ÍNDICE

GITHUB

https://github.com/IvanGP7/Scalable-Distributed-System.git

OBJETIVO

En el desarrollo de sistemas distribuidos modernos, la escalabilidad, eficiencia y adaptabilidad son aspectos clave para satisfacer cargas de trabajo variables, garantizar rendimiento constante y facilitar la administración de recursos. Este trabajo se enmarca en ese contexto, con el objetivo de analizar, comparar y optimizar diferentes arquitecturas distribuidas aplicadas a un mismo conjunto de tareas, bajo condiciones controladas de estrés y paralelismo.

Se han implementado y probado cuatro tecnologías de comunicación y procesamiento distribuido:

- **XML-RPC**, como modelo de llamada a procedimiento remoto basado en XML sobre HTTP, ideal por su simplicidad.
- **Pyro (Python Remote Objects)**, como solución orientada a objetos en Python, con enfoque en transparencia de invocación remota.
- **Redis**, usado como sistema de colas y almacenamiento compartido en memoria, al que se le ha añadido un sistema de escalado dinámico de workers.
- **RabbitMQ**, como broker de mensajes robusto basado en colas AMQP, ampliamente usado en arquitecturas orientadas a eventos.

Cada arquitectura ha sido implementada con capacidad de multi-nodo y ejecutada bajo distintas condiciones de carga (número de peticiones e hilos concurrentes). Además, se ha desarrollado un sistema automatizado de pruebas (montaje.sh) y recopilación de métricas, incluyendo tiempos de ejecución y gráficos de escalabilidad (speedup).

En conjunto, este trabajo no solo busca comparar el rendimiento bruto de cada arquitectura, sino también comprender sus limitaciones y comportamientos para desarrollar arquitecturas adaptadas a cada sistema y optimizar su funcionamiento.

TECNOLOGÍAS

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 MULTI-NODO

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 lo que dificultaba el uso de 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 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 practica 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.

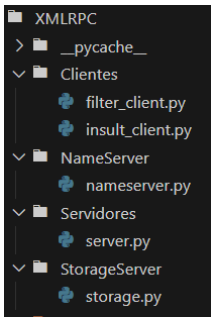


Figura 1.1 (Estructura archivos XMLRPC)

La Figura 1.2 muestra el diagrama de funcionamiento de la arquitectura propuesta a más detalle. Este diagrama muestra de manera visible la comunicación que hay entre las diferentes estructuras y su funcionamiento. El sistema se compone de cuatro elementos principales:

1. **Clientes (InsultClient y FilterClient):**
   Los clientes son responsables de emitir peticiones al sistema. InsultClient permite añadir insultos a la base de datos compartida, mientras que FilterClient solicita el filtrado de texto utilizando la lista de insultos almacenada. Estos clientes utilizan múltiples hilos (threads), uno por cada nodo servidor disponible, para enviar sus peticiones de forma concurrente. Cada cliente obtiene inicialmente la lista de servidores disponibles a través del NameServer.

2. **Worker de insultos (InsultServer1, InsultServer2, ...):**
   Actúan como workers que reciben y procesan las peticiones de los clientes. Implementan métodos como add_insult() para registrar nuevos insultos, y censor_text() para aplicar el filtrado de texto. No almacenan la información localmente, sino que sincronizan constantemente su lista de insultos con un nodo central: el StorageServer. A través de la petición sincronizar obteniendo y otorgando los insultos nuevos y recibiendo la lista actualizada para hacer operaciones localmente.

3. **StorageServer:**
   Es el nodo central de almacenamiento y sincronización. Su función principal es mantener la lista principal de insultos actualizada. Cuando un InsultServer recibe un nuevo insulto desde un cliente, este se reenvía al StorageServer, que actualiza su base de datos y posteriormente notifica al resto de servidores para que sincronicen su copia local. De esta forma, todos los servidores mantienen respuestas actualizadas en todo momento.

4. **NameServer:**
   Este componente funciona como un registro de servicios. Cada InsultServer, al iniciar, se registra automáticamente en el NameServer, proporcionando su URI y puerto. A su vez, los clientes contactan con el NameServer para recuperar la lista actual de servidores activos. Esto les permite distribuir las peticiones de manera dinámica según los nodos disponibles en cada momento.

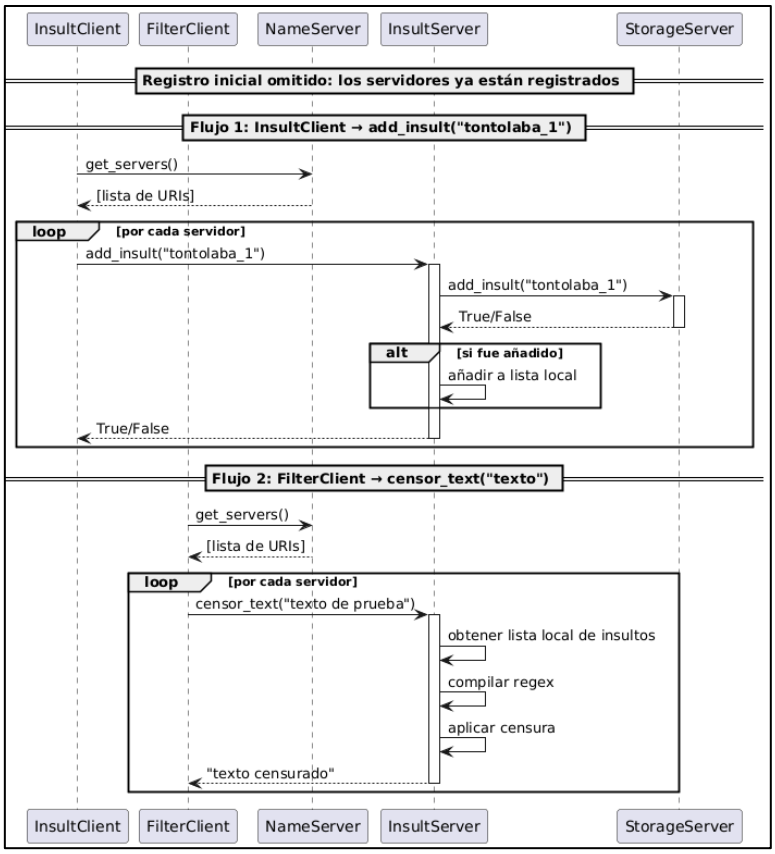


Figura 1.2 (Diagrama de funcionamiento de la arquitectura XMLRPC)

Pyro

En este caso, al tratarse de una arquitectura multi-nodo implementada con Pyro4, uno de los aspectos más importantes es definir cómo y dónde se almacena la información, asegurando que sea accesible desde varios nodos sin convertir el sistema en una arquitectura uni-nodo. Bastante similar al XML-RPC, pero en este caso se ha decidido por implementar una comunicación extra con un servicio llamado sync que vinculará constantemente la información del storage con los workers. En la figura 2.1 se puede ver que la estructura se mantiene como en el caso anterior, pero añadiendo una nueva estructura sync que se encargará de comunicar el storage como los workers. En la figura 2.1 se puede ver la estructura de este sistema con la nueva estructura sync.

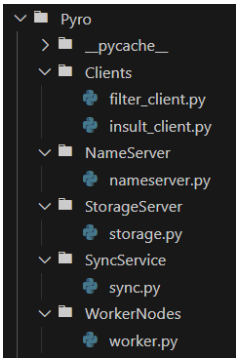


Figura 2.1 (estructura archivos Pyro)

La Figura 2.2 representa el funcionamiento interno de la arquitectura, donde se describen las relaciones y flujos de datos entre los componentes. El sistema se compone de cinco elementos principales:

1. **Clientes (insult_client.py y filter_client.py)**
   Los clientes del sistema son los encargados de generar las peticiones hacia el sistema. Por un lado, el cliente InsultClient tiene como objetivo insertar nuevos insultos en el sistema. Para ello, obtiene la lista de WorkerNodes disponibles desde el NameServer y envía peticiones add_insult() de forma concurrente a cada uno de ellos. Por otro lado, el cliente FilterClient se encarga de enviar textos que deben ser filtrados a través de la función censor_text(), utilizando también los WorkerNodes. Ambos clientes están diseñados para funcionar con múltiples hilos, uno por cada nodo disponible. Esto significa que la carga de trabajo se distribuye entre todos los WorkerNodes registrados en el sistema, optimizando el rendimiento general. Gracias a la comunicación dinámica con el NameServer, los clientes pueden adaptarse automáticamente a la disponibilidad de nodos.

2. **WorkerNodes (worker.py)**
   Los WorkerNodes representan los servidores de trabajo que ejecutan las peticiones de los clientes. Su función principal es recibir y procesar los insultos y textos que llegan desde los clientes. Cada WorkerNode implementa dos métodos clave: add_insult() y censor_text(). El primero se encarga de reenviar el insulto recibido al StorageServer, que actúa como almacenamiento persistente. Si la operación es exitosa, el nodo también actualiza su copia local del insulto. El segundo método, censor_text(), analiza el texto recibido y reemplaza todos los insultos encontrados con la palabra "CENSORED".

Es importante destacar que los WorkerNodes no almacenan permanentemente la información. En su lugar, mantienen una copia temporal sincronizada de la lista de insultos, la cual se actualiza periódicamente mediante el SyncService.

3. **StorageServer (storage.py)**
   El StorageServer es el nodo central encargado de almacenar de forma persistente todos los insultos registrados en el sistema. Su misión es recibir insultos desde los WorkerNodes mediante el método add_insult() y mantener una lista actualizada de estos. Además, expone el método get_insults(), que permite consultar la lista completa de insultos en cualquier momento.
   Este servidor no interactúa directamente con los clientes, sino que funciona como una base de datos accesible exclusivamente para otros componentes internos del sistema, como los WorkerNodes y el SyncService. De esta forma, se asegura la integridad de los datos y se evita que múltiples nodos escriban o lean información de forma descontrolada.

4. **SyncService (sync.py)**
   El SyncService es el componente responsable de mantener la coherencia de la información entre los distintos nodos del sistema. Su funcionamiento se basa en un ciclo continuo que, cada 0.1 segundos, realiza una sincronización global. Para ello, consulta al StorageServer mediante el método get_insults() y obtiene la lista actualizada de insultos. Luego, recorre todos los WorkerNodes registrados en el NameServer y les envía la lista completa a través del método update_insults().
   Este servicio garantiza que todos los nodos trabajen con la misma información, independientemente de cuál haya sido el nodo que recibió originalmente una nueva entrada. Además, al realizar la sincronización de forma periódica y separada de la lógica principal de los clientes y trabajadores, evitando bloquear operaciones el sistema.

5. **NameServer (nameserver.py)**
   El NameServer actúa como punto central de descubrimiento de servicios en el sistema. Cada WorkerNode, al iniciar su ejecución, se registra en el NameServer proporcionando su URI. Esta información queda disponible para cualquier cliente que desee interactuar con los nodos activos, permitiéndole obtener la lista actual de servidores mediante el método get_workers().
   Gracias a esta arquitectura dinámica de registro y descubrimiento, se evita la necesidad de configurar manualmente las direcciones de cada nodo en los clientes. Esto hace que el sistema sea mucho más flexible, permitiendo que se añadan o eliminen nodos sin afectar al funcionamiento general. Además, el NameServer también es consultado por el SyncService para obtener la lista completa de nodos a sincronizar.

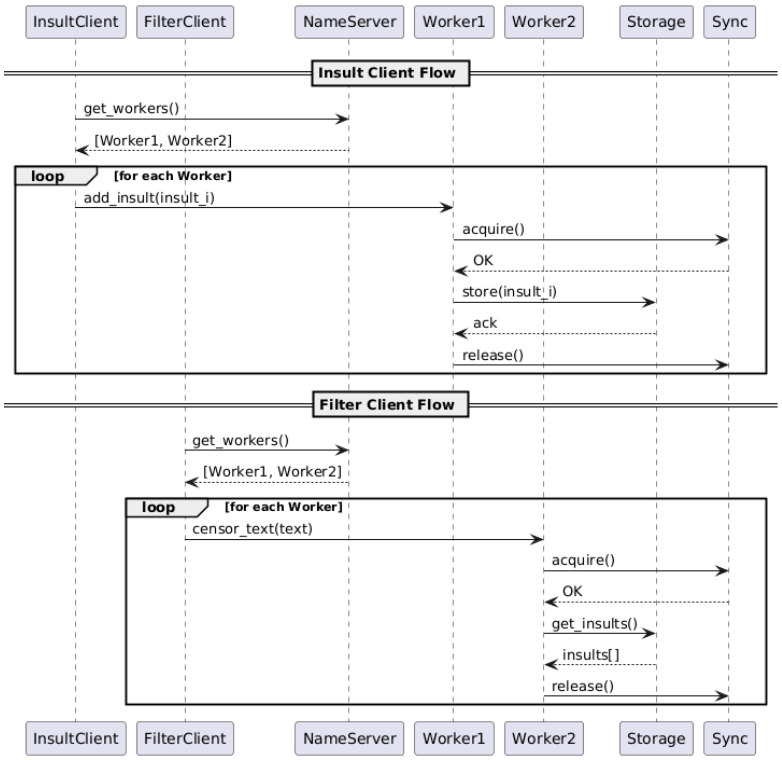


Figura 2.2 (Diagrama de funciones de la arquitectura Pyro)

Redis

En esta arquitectura se ha optado por utilizar Redis como sistema de comunicación y coordinación entre múltiples nodos de procesamiento distribuidos. A diferencia de Pyro y XML-RPC, donde los nodos comunican directamente mediante llamadas remotas (RPC) hacia objetos o servicios expuestos, en Redis el enfoque es completamente desacoplado: los clientes simplemente publican tareas en colas (insult_queue, filter_queue), y los workers las consumen de forma asíncrona.

Este modelo tipo *productor-consumidor* facilita una mayor independencia entre componentes y simplifica el reparto de carga, ya que cualquier Worker disponible puede recoger y procesar tareas sin necesidad de conocer a los clientes emisores. En la *Figura 3.1* se puede ver cómo los componentes del sistema interactúan con el servidor Redis central, mientras que la *Figura 3.2* detalla el funcionamiento interno.

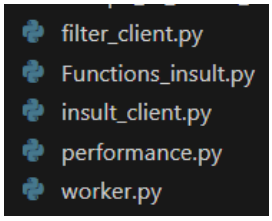


Figura 3.1 (Estructura de archivos Redis)

La arquitectura está compuesta por los siguientes elementos principales:

1. **Clientes (insult_client.py y filter_client.py)**
   Ambos clientes se encargan de generar peticiones concurrentes a través de múltiples hilos. El InsultClient genera insultos que se encolan en la insult_queue, mientras que el FilterClient envía textos para filtrar, los cuales se almacenan en la filter_queue de Redis.
   Cada cliente inicializa un contador (insult_done o filter_done) y espera a que el número de tareas procesadas coincida con las peticiones realizadas. Esto permite medir con precisión el tiempo de ejecución de cada experimento, sin terminar prematuramente.
   Gracias al uso de colas separadas y contadores individuales, se puede garantizar una correcta coordinación incluso con múltiples hilos y nodos trabajadores en paralelo.

2. **WorkerNodes (worker.py)**
   Cada Worker actúa como consumidor de las colas Redis. Utiliza una operación blpop() sobre las colas insult_queue y filter_queue, esperando nuevas tareas que pueda procesar.
   Cuando se recibe una tarea, el Worker la interpreta (como JSON), extrae los campos relevantes y llama a los métodos definidos en el InsultManager. Si la tarea es un insulto, se invoca add_insult(), y si es una frase para censurar, se llama a censor_text().
   Finalmente, cada Worker incrementa el contador correspondiente (insult_done o filter_done) en Redis, lo cual permite a los clientes saber cuándo finalizar su ejecución.

3. **Redis (broker central)**
   Redis es el centro de la arquitectura. Almacena tanto las colas de tareas (insult_queue, filter_queue) como los contadores de progreso.
   La elección de Redis permite una comunicación eficiente y baja latencia entre clientes y workers, además de una forma simple y directa de coordinación entre procesos.
   En esta arquitectura no se usa Redis como base de datos persistente, sino como canal de mensajes en tiempo real. Esto permite un sistema simple, fácilmente replicable, pero también con ciertas limitaciones si se requieren garantías fuertes de entrega o persistencia.

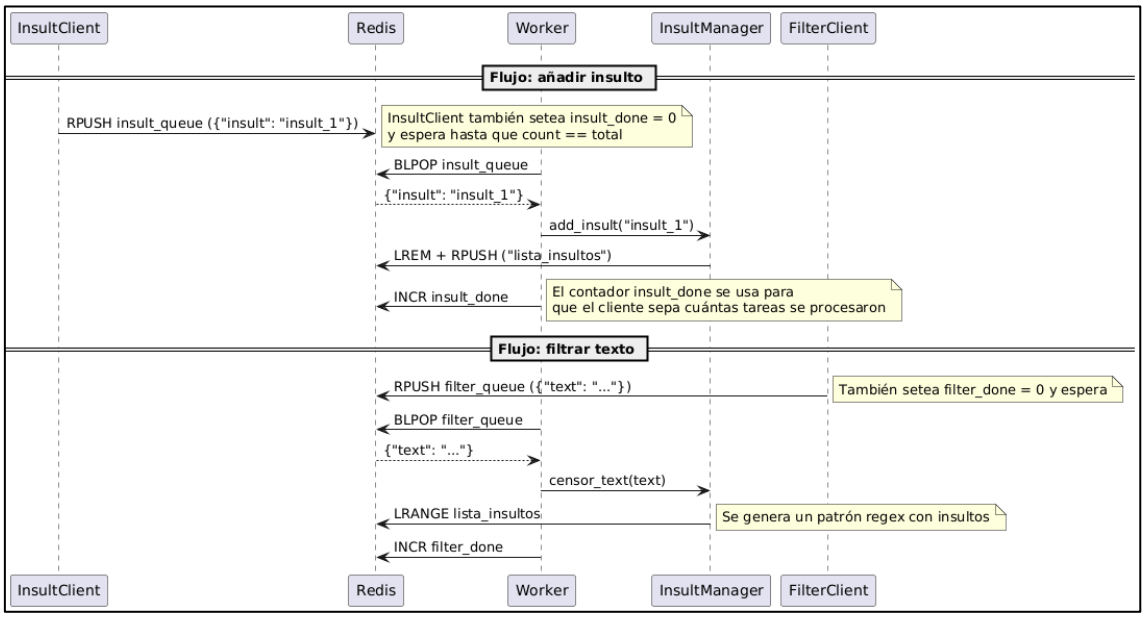


Figura 3.2 (Diagrama sobre el funcionamiento de la arquitectura Redis)

RabbitMQ

La implementación presentada se basa en RabbitMQ como sistema de comunicación principal entre clientes, trabajadores y almacenamiento. A diferencia de arquitecturas centradas en RPCs directas (como XML-RPC o Pyro) o almacenamiento compartido (como Redis), RabbitMQ proporciona una infraestructura orientada a colas de mensajes, donde los componentes se comunican de forma desacoplada y asincrónica. Esto facilita una arquitectura reactiva, escalable y tolerante a fallos.

RabbitMQ actúa como intermediario entre los distintos procesos, permitiendo que los clientes publiquen mensajes en colas específicas (add_insults o filter_text) sin conocer de antemano qué nodo lo procesará. Los WorkerNodes (trabajadores) se encargan de consumir estos mensajes, ejecutar la lógica correspondiente y, en caso necesario, enviar una respuesta al cliente mediante colas temporales. La sincronización de datos (en este caso, la lista de insultos) se gestiona mediante un StorageServer que transmite actualizaciones a todos los trabajadores. En la figura 4.1 podemos ver la estructura usada.

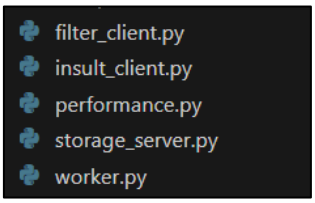


Figura 4.1 (Estructura de archivos RabbitMQ)

1. **Clientes (insult_client.py y filter_client.py)**
   Los clientes funcionan como productores de mensajes. Su rol es enviar peticiones al sistema sin necesidad de saber cuál será el nodo que las procese. Cada cliente lanza un número determinado de peticiones a través de RabbitMQ y espera las respuestas de manera asincrónica, utilizando una cola exclusiva creada para cada ejecución. Esto les permite recibir los resultados de forma segura y sin interferencias de otros procesos.
   El cliente InsultClient se encarga de insertar nuevos insultos. Para ello, publica mensajes en la cola add_insults, cada uno con un insulto individual. Estos mensajes llevan asociada una propiedad 'reply_to' para que el trabajador pueda enviar una respuesta de confirmación para tratar el tiempo de mensajes.
   Por su parte, el cliente FilterClient publica textos aleatorios en la cola filter_text, solicitando que los mismos sean procesados y censurados. Ambos clientes se bloquean a la espera de las respuestas y registran el tiempo total empleado en el archivo tiempos_clientes.log, facilitando la monitorización del rendimiento bajo diferentes volúmenes de carga.

2. **WorkerNodes (worker.py)**
   Los trabajadores (WorkerNodes) son consumidores de mensajes. Su tarea consiste en leer las peticiones de las colas add_insults y filter_text, procesarlas y, si corresponde, devolver una respuesta al cliente. Para ello, implementan dos métodos internos: _process_insult() y _process_filter().
   En _process_insult(), el trabajador analiza el insulto recibido y lo añade a su base local si aún no estaba presente. En _process_filter(), aplica una expresión regular sobre el texto recibido, sustituyendo cualquier insulto por la palabra

"CENSURADO". Esta lógica se ejecuta de forma protegida mediante bloqueos (locks) para evitar condiciones de carrera.

Cada trabajador mantiene una lista local de insultos, la cual se actualiza automáticamente cada vez que el StorageServer emite una sincronización. Para ello, los WorkerNodes se suscriben a un *exchange* de tipo fanout llamado insults_sync, desde donde reciben actualizaciones periódicas con la lista global de insultos. Esta arquitectura permite que los trabajadores mantengan una copia local consistente sin necesidad de consultar constantemente una base de datos central.

3. **StorageServer (storage_server.py)**

   El StorageServer actúa como base central para el almacenamiento y sincronización de insultos. Su principal función es recibir nuevas palabras desde los trabajadores y propagar la lista actualizada a todos los nodos activos. La lógica de este componente está diseñada para garantizar que cualquier cambio relevante en la base de insultos se transmita de forma eficiente a través del sistema.

   Cuando el StorageServer recibe una actualización (por ejemplo, nuevos insultos) en la cola storage_updates, verifica qué insultos no estaban previamente registrados y los añade a su conjunto local. A continuación, ejecuta el método broadcast_insults(), el cual publica la lista completa de insultos en el exchange insults_sync. Este mecanismo asegura que todos los WorkerNodes reciban la versión más reciente de la base de insultos de forma simultánea.

   Además, el servidor ejecuta un hilo de sincronización periódica, el cual transmite la lista actualizada cada cinco segundos, garantizando que incluso en ausencia de nuevas inserciones, todos los nodos se mantengan sincronizados.

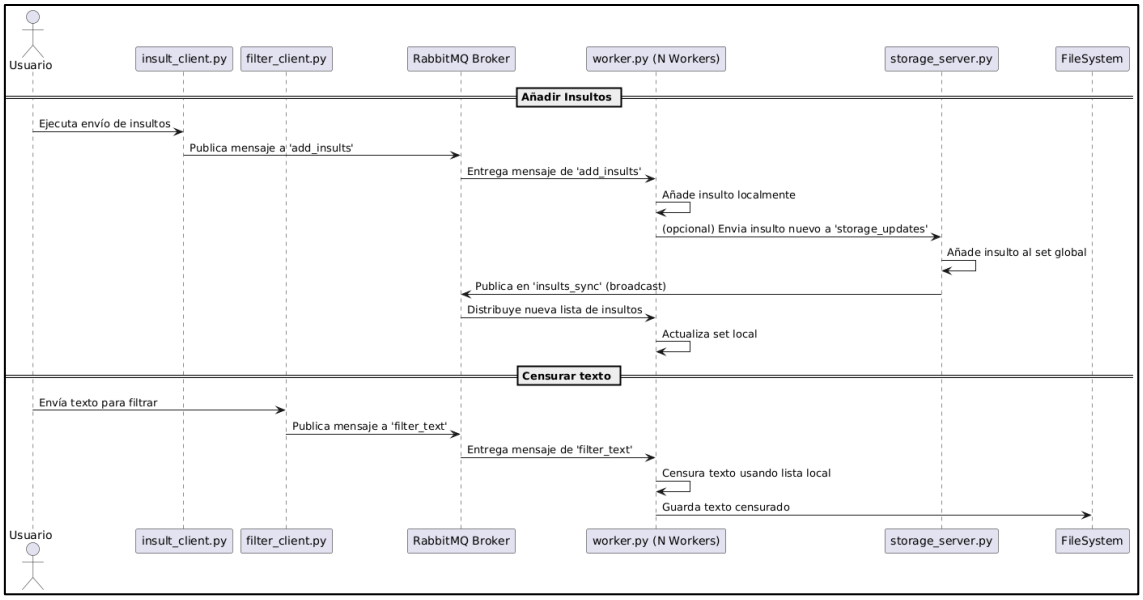


Figura 4.2 (Diagrama sobre el funcionamiento de la arquitectura RabbitMQ)

DESARROLLO NODOS DINAMICOS

Objetivo

El objetivo principal de esta implementación es adaptar dinámicamente el número de nodos de procesamiento (workers) según la carga real del sistema, medida como el volumen de peticiones en cola y su tasa de llegada. En lugar de iniciar una cantidad fija de nodos como en las arquitecturas multi-nodo previas, esta versión dinámica pretende optimizar el uso de recursos: crear nuevos workers cuando la demanda lo requiere y reducirlos cuando la carga disminuye.

Este enfoque permite que el sistema sea escalable de forma automática, reaccionando en tiempo real ante aumentos o descensos en el número de peticiones sin intervención manual, lo que resulta especialmente útil en entornos con carga variable o impredecible.

Redis dinámico

La arquitectura de Redis Dinámico se basa en los principios del Redis multinodo previamente implementado, pero introduce una lógica de escalado automático que monitoriza la longitud de la cola de tareas y estima la tasa de llegada de peticiones (λ) para decidir cuántos workers deben estar activos en cada momento.

Al igual que en el modelo multi-nodo, se emplea Redis como canal de comunicación centralizado entre los clientes (insult_client.py y filter_client.py) y los nodos de procesamiento (worker.py). Sin embargo, mientras que en el modelo estático el número de nodos era fijo y definido desde el inicio, en esta versión los nodos se gestionan dinámicamente mediante un módulo escalador (scaler) que lanza o detiene hilos Worker en función de la fórmula:

$$N = \left\lceil \frac{B + \lambda \cdot Tr}{C} \right\rceil$$

donde:

- B es la longitud actual de la cola de tareas,
- λ la tasa estimada de llegada de peticiones,
- T el tiempo de respuesta objetivo,
- C la capacidad de procesamiento de un worker.

Además, se han implementado límites de escalado para evitar generación extrema de nodos, garantizando una respuesta estable dentro de un entorno pequeño como puede ser un portátil. Este diseño busca equilibrar rendimiento, consumo de recursos y tiempo de respuesta, manteniendo el sistema funcional y escalable.

Componentes y Arquitectura

1. **Clientes (insult_client.py y filter_client.py)**

   Los clientes representan el punto de entrada de las peticiones al sistema. Son los encargados de generar la carga de trabajo, ya sea en forma de insultos a registrar o textos a filtrar.
   El cliente insult_client.py toma como entrada un número determinado de insultos, los codifica en texto plano y los envía a Redis a través de la cola insult_queue.
   Por otro lado, el cliente filter_client.py toma un texto base, lo convierte en múltiples peticiones codificadas en JSON, y las inserta en la cola filter_queue.
   Ambos clientes miden el tiempo que tardan en generar las peticiones para calcular una estimación de la tasa de llegada (λ). Este valor se almacena en Redis como lambda_estimate, y será consultado por el escalador para adaptar la capacidad del sistema. Finalmente, los clientes entran en espera activa hasta que todas sus peticiones hayan sido procesadas, validándolo mediante contadores que Redis mantiene (insult_done o filter_done).

2. **Servidor Redis**

   Redis actúa como bus de comunicación central y almacén temporal de tareas. Su papel es estrictamente pasivo: recibe mensajes y los mantiene en las colas insult_queue y filter_queue hasta que un worker los consuma.
   Además, mantiene claves de estado como lambda_estimate (la tasa de llegada estimada), y los contadores de tareas procesadas por tipo. Redis no realiza ningún procesamiento ni lógica, pero sin él, el sistema carecería de coordinación. Su diseño basado en memoria garantiza tiempos de acceso extremadamente bajos, permitiendo que múltiples threads de clientes y workers interactúen de forma simultánea sin cuellos de botella graves.

3. **Workers (worker.py)**

   Los nodos de procesamiento se representan mediante la clase Worker, que se lanza como un hilo en segundo plano. Cada instancia queda a la espera de tareas utilizando la instrucción brpop(), escuchando tanto en insult_queue como en filter_queue.
   Cuando recibe una tarea, la decodifica (ya sea en texto plano o JSON), y la procesa a través del módulo InsultManager, que se encarga de aplicar la censura de los insultos registrados. Una vez procesada la petición, el worker incrementa el contador correspondiente (insult_done o filter_done), marcando así que una unidad de trabajo ha sido completada.
   Los workers no tienen conocimiento del resto del sistema; su funcionamiento es autónomo y su ciclo de vida es gestionado por el escalador.

4. **Escalador (contenido también en worker.py)**

   El componente clave que permite la adaptabilidad del sistema es el escalador dinámico. Implementado como un hilo paralelo, este proceso consulta continuamente el estado de la cola de tareas (B) y la tasa de llegada (λ). Con estos valores, calcula el número óptimo de workers (N_formula) a partir de una fórmula derivada de principios de control de colas (similar a Little o M/M/1 adaptado).
   A partir de esta estimación, ajusta el número de workers activos:

   - Si hay más demanda, crea nuevos threads Worker y los lanza automáticamente.
   - Si la carga disminuye, detiene los workers innecesarios de forma ordenada.

   El escalador aplica bandas de decisión para evitar cambios bruscos (es decir, no escala o reduce por cada pequeña variación) y siempre respeta un número mínimo y máximo de nodos activos. Este comportamiento hace que el sistema mantenga un rendimiento aceptable bajo distintas condiciones de carga, sin intervención manual ni necesidad de reiniciar el servicio.

En la figura 5.1 Podremos ver la relación entre todos los componentes en su arquitectura.

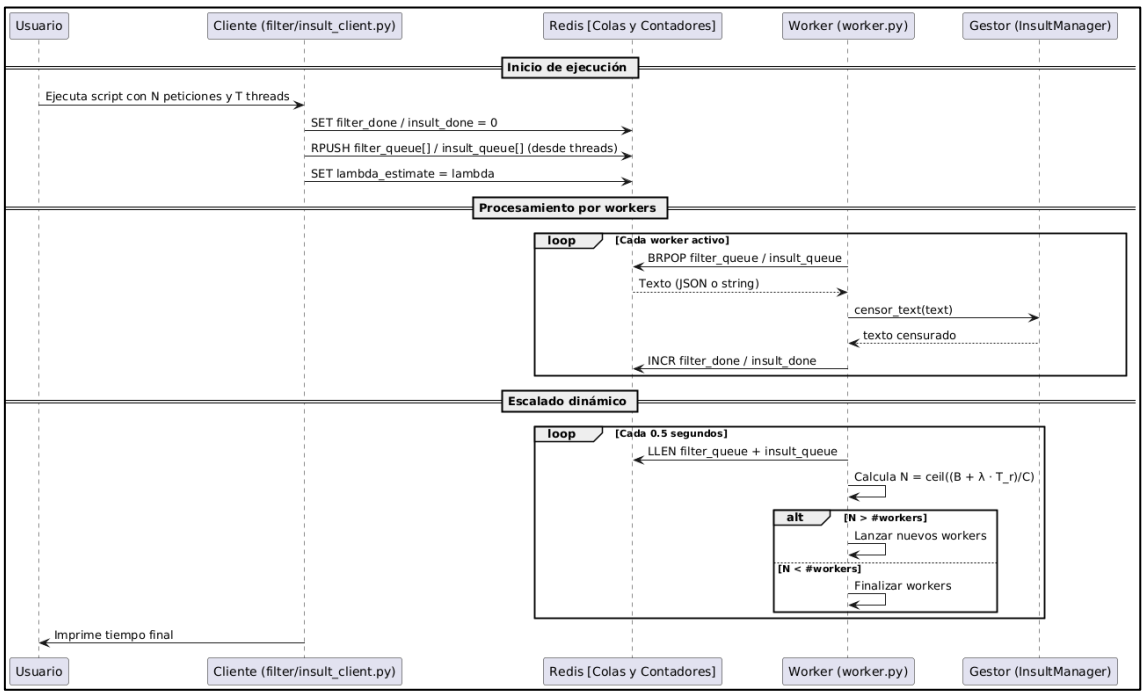


Figura 5.1 (Diagrama sobre el funcionamiento de la arquitectura Redis con nodos Dinámicos)

PRUEBAS Y RESULTADOS

Con el fin de evaluar de manera rigurosa el rendimiento de cada una de las arquitecturas desarrolladas, se ha diseñado un entorno de pruebas automatizado que permite ejecutar una batería de pruebas de forma controlada, precisa y homogénea entre todas las tecnologías implementadas.

Para ello, se han utilizado dos scripts fundamentales:

- **montaje.sh**: encargado de realizar una única ejecución del sistema completo, recibiendo como parámetros el número de nodos de procesamiento (o hilos cliente, dependiendo del contexto) y el número de iteraciones o peticiones que se desean lanzar. Este script inicializa los distintos componentes de la arquitectura correspondiente (clientes, workers, servidores, etc.), lanza la prueba y recoge los tiempos de ejecución.
- **test.sh**: actúa como lanzador de pruebas masivas. Llama iterativamente a montaje.sh con diferentes combinaciones de parámetros, generando así un conjunto amplio de resultados que permiten analizar la evolución del rendimiento en función de la carga de trabajo y los recursos asignados.
  Con una cantidad de:
  - Nodos = [1, 2, 3, 4, 5, 6, 7]
  - Iteraciones = [25, 100, 200, 700, 1500, 3000]

Aunque test.sh es compartido por todas las arquitecturas, el contenido de montaje.sh varía dependiendo de la tecnología empleada (Redis, Pyro, XML-RPC o RabbitMQ). Esto se debe a que cada sistema tiene una forma distinta de conexión, inicialización y relación entre sus componentes (bases de datos, brokers, etc.), lo que requiere configuraciones específicas para cada caso.

Durante la ejecución de las pruebas, cada cliente (ya sea InsultClient o FilterClient) genera automáticamente una entrada en el archivo tiempos_clientes.log. Este archivo contiene, en cada línea, el nombre del cliente, el número de hilos o nodos utilizados, el número de iteraciones enviadas y el tiempo total que ha tardado en completarse dicha ejecución.

Una vez finalizado todo el proceso de recopilación, se emplea el script performance.py, que toma como entrada el archivo tiempos_clientes.log y genera cuatro gráficos clave para el análisis de rendimiento:

- Speed-up de InsultClient
- Speed-up de FilterClient
- Tiempo en función del número de iteraciones
- Tiempo en función del número de nodos

Estos gráficos permiten visualizar de forma clara el comportamiento del sistema bajo distintas condiciones de carga, así como la escalabilidad y eficiencia de cada tecnología empleada. Las conclusiones derivadas de estos resultados se expondrán en una sección posterior, contrastando entre arquitecturas y destacando tanto los puntos fuertes como las limitaciones observadas.

XML-RPC

Más adelante, se presentan los resultados obtenidos al evaluar el comportamiento de la arquitectura basada en XML-RPC bajo distintas configuraciones de carga y número de nodos mencionados anteriormente. Para ello, se han realizado pruebas sistemáticas con diferentes cantidades de iteraciones y workers, analizando el rendimiento tanto del cliente InsultClient como de FilterClient.

Las Figuras 10.1 y 10.2 muestran la relación entre el número de nodos y el tiempo de ejecución para distintas cargas de trabajo, permitiendo observar cómo evoluciona el rendimiento al paralelizar el procesamiento. Por otro lado, las Figuras 10.3 y 10.4 ilustran el speedup obtenido en cada cliente al aumentar el número de nodos, comparando cada ejecución con el tiempo base correspondiente al uso de un solo nodo.

Estas gráficas permiten extraer conclusiones relevantes sobre la eficiencia y escalabilidad del modelo XML-RPC, las cuales se analizan en el apartado siguiente.

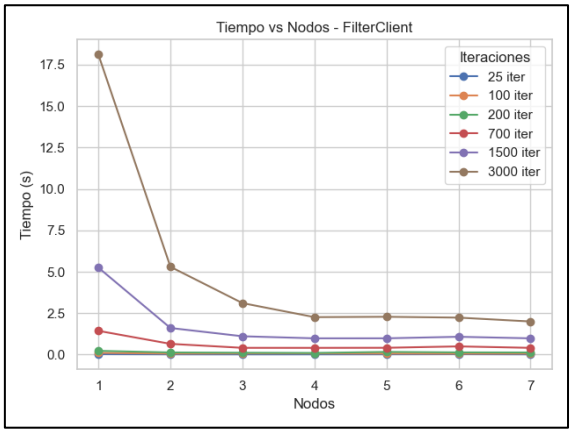


Figura 10.1 (grafica del rendimiento con el FilterClient)

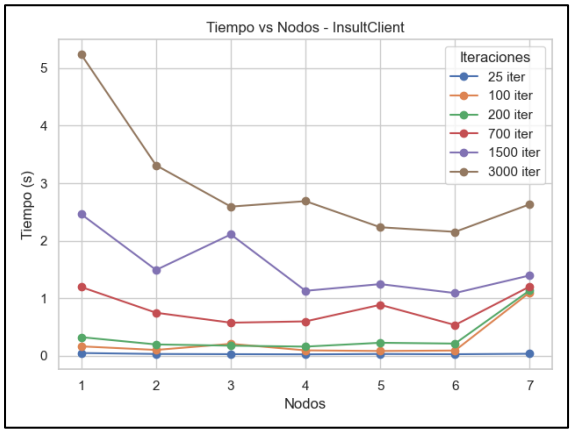


Figura 10.2 (grafica del rendimiento con el InsultClient)

Rendimiento de FilterClient (Figura 10.1)

El primer gráfico muestra una evolución muy clara y predecible del tiempo de ejecución en función del número de nodos utilizados. Para todas las cantidades de iteraciones, se observa una reducción significativa del tiempo de ejecución a medida que se incrementa el número de nodos, especialmente entre 1 y 4. Esta caída es más pronunciada en cargas altas (por ejemplo, con 3000 iteraciones), donde pasar de 1 a 2 nodos reduce el tiempo de más de 17 segundos a unos 5 segundos, y seguir aumentando los nodos hasta 4 baja el tiempo por debajo de 3 segundos.

A partir del cuarto nodo, el tiempo tiende a estabilizarse, lo que sugiere que el sistema alcanza su punto de saturación óptima y que añadir más nodos no aporta mejoras significativas. En cargas bajas (25, 100, 200 iteraciones), el tiempo apenas se ve afectado por el número de nodos, lo cual es esperable, ya que el coste de coordinación entre hilos supera al beneficio de paralelizar tan pocas tareas.

Este comportamiento demuestra que la arquitectura XML-RPC escala razonablemente bien en el cliente de filtrado, siempre que la carga sea lo suficientemente alta como para justificar la distribución del trabajo entre varios nodos.

Rendimiento de InsultClient (Figura 10.2)

El segundo gráfico, correspondiente a InsultClient, muestra una evolución más inestable en comparación con FilterClient. Aunque en líneas generales se aprecia una reducción del tiempo de ejecución cuando se pasa de 1 a 2 nodos, el comportamiento posterior es más irregular. En varios puntos (por ejemplo, con 1500 o 3000 iteraciones), el tiempo oscila incluso al aumentar los nodos, lo que sugiere problemas de sobrecarga o sincronización dentro del sistema XML-RPC para este tipo de tarea.

Una posible explicación de esta variabilidad es que InsultClient lanza muchas operaciones de escritura hacia los workers (cada insulto individual implica una llamada remota a add_insult()), lo cual puede aumentar la carga sobre el NameServer o sobre los hilos de red, especialmente si los workers se encuentran sincronizando con el almacenamiento en paralelo. Además, al no haber una cola intermedia que desacople la producción de la escritura (como ocurre en arquitecturas con colas como Redis o RabbitMQ), el sistema puede volverse menos eficiente en cargas altas debido a cuellos de botella en la comunicación directa.

Pese a esta variabilidad, el tiempo total en cargas pequeñas sigue siendo bajo, y el sistema responde correctamente a nivel funcional. No obstante, la eficiencia decrece conforme se incrementa la cantidad de nodos y la complejidad de sincronización.

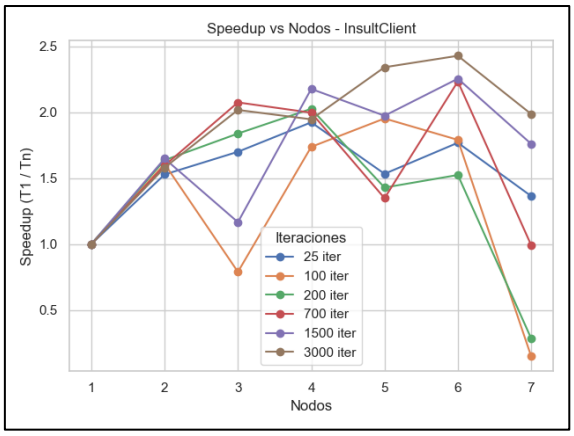


Figura 10.3 (grafica del Speed-Up con el InsultClient)

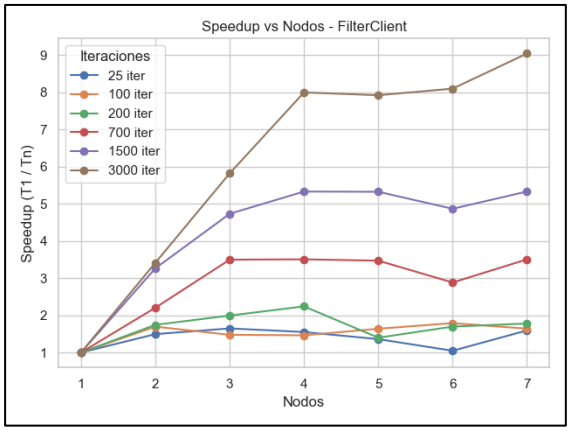


Figura 10.4 (grafica del Speed-Up con el FilterClient)

Speedup del FilterClient (Figura 10.4)

El gráfico correspondiente a FilterClient muestra una tendencia clara de mejora progresiva y sostenida del rendimiento a medida que se incrementa el número de nodos, especialmente en escenarios de carga media y alta.

Para 3000 iteraciones, el sistema alcanza un speedup superior a 9 con 7 nodos, lo que sugiere que el trabajo se distribuye eficientemente entre los WorkerNodes y que los costes de comunicación no son un factor limitante en este caso. También se observa un comportamiento bastante estable para 1500 y 700 iteraciones, donde el speedup se mantiene en torno a 4–6 con 4 o más nodos.

En cargas más bajas (25, 100 y 200 iteraciones), el beneficio de paralelizar es más limitado, lo cual es esperable: el coste de coordinación supera al trabajo efectivo realizado, y por tanto el speedup se estabiliza o incluso fluctúa levemente.

En conjunto, el sistema demuestra una alta escalabilidad para FilterClient, siendo capaz de aprovechar eficazmente los recursos adicionales conforme aumenta la carga.

Speedup del InsultClient (Figura 10.3)

En contraste con el comportamiento anterior, el gráfico de InsultClient muestra un speedup más irregular y menos predecible. Aunque en general se observa una mejora cuando se pasa de 1 a 3–4 nodos, a partir de ese punto los valores de speedup comienzan a oscilar, e incluso a decrecer en algunos casos.

Esto se puede explicar por el hecho de que InsultClient genera una gran cantidad de llamadas remotas a los nodos (add_insult()), lo cual implica un mayor coste de comunicación y sincronización. Además, en XML-RPC cada llamada bloquea al cliente mientras espera la respuesta, lo que limita el paralelismo efectivo, especialmente si los workers están ocupados procesando o sincronizando con el almacenamiento.

En el mejor de los casos (3000 iteraciones con 6 nodos), se alcanza un speedup de aproximadamente 2.5, lo que indica una mejora moderada. Sin embargo, este valor está lejos de lo que sería un escalado lineal. En otros escenarios (100 o 200 iteraciones), el speedup llega a decrecer con más nodos, revelando la presencia de cuellos de botella internos o saturación por exceso de peticiones simultáneas.

PYRO

Como resultado, se generaron datos registrados en tiempos_clientes.log, posteriormente analizados mediante performance.py, que produjo los gráficos de comportamiento por cliente. En las Figuras 11.1 y 11.2 se observa la relación entre el número de nodos y el tiempo de ejecución para FilterClient e InsultClient respectivamente. Mientras que las Figuras 11.3 y 11.4 muestran el speedup obtenido en cada caso con respecto a la ejecución base con un único nodo.

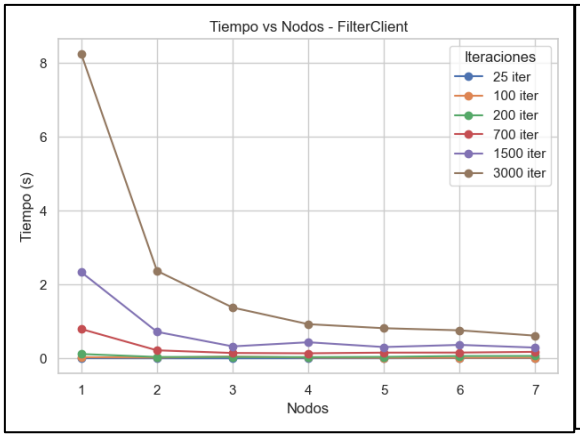


Figura 11.1 (grafica del rendimiento con el FilterClient)

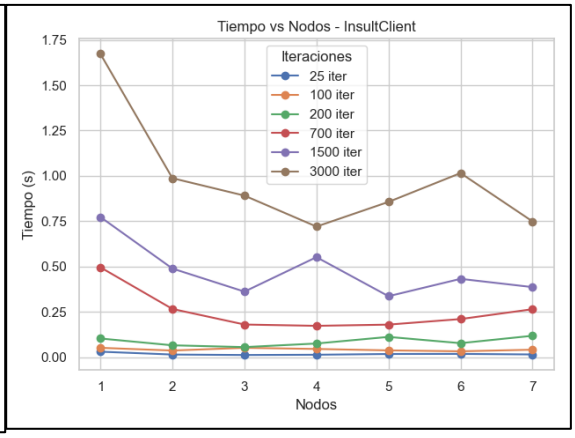


Figura 11.2 (grafica del rendimiento con el InsultClient)

Rendimiento de FilterClient (Figura 11.1)

En las pruebas realizadas con FilterClient, se observa un descenso marcado en el tiempo de ejecución conforme se incrementa el número de nodos. Esta tendencia es especialmente notable cuando se parte de una carga alta, como las 3000 iteraciones: el tiempo se reduce de más de 8 segundos a menos de 1.5 con tan solo 3 o 4 nodos. A partir de ese punto, la curva comienza a estabilizarse, mostrando que el sistema alcanza un equilibrio entre carga y recursos disponibles, es decir, añadir más nodos sigue beneficiando, pero en menor proporción. Este comportamiento se mantiene en cargas intermedias, como 700 o 1500 iteraciones, donde el sistema logra buenos tiempos con entre 3 y 5 nodos, lo que evidencia que la arquitectura es capaz de distribuir correctamente las peticiones entre los WorkerNodes.

En el caso de las cargas más pequeñas, como 25 o 100 iteraciones, los tiempos se reducen, pero de forma mucho más suave. En estos casos el coste de creación de hilos, comunicación y sincronización no llega a amortizarse frente al trabajo real a realizar. Aun así, el comportamiento es estable, sin fluctuaciones bruscas, lo que demuestra que el sistema mantiene la eficiencia incluso en condiciones de baja carga. En conjunto, los resultados confirman que Pyro ofrece un rendimiento adecuado y escalable para tareas de lectura como el filtrado de texto, especialmente cuando la carga de trabajo es suficiente como para justificar la distribución entre nodos.

El sistema no presenta caídas ni anomalías relevantes bajo ninguna de las configuraciones probadas, lo cual es un buen indicador de estabilidad. Esto sugiere que la estructura de comunicación con los WorkerNodes, basada en llamadas RPC a objetos remotos, se adapta bien a flujos de trabajo donde la concurrencia no implica escritura centralizada, sino acceso compartido a datos sincronizados por el servicio de sincronización.

Rendimiento de InsultClient (Figura 11.2)

En cambio, los resultados obtenidos con InsultClient presentan un comportamiento más irregular. Aunque se aprecia una mejora inicial del rendimiento al pasar de 1 a 3 nodos, a partir de ese punto la curva comienza a presentar oscilaciones. En el caso de 3000 iteraciones, por ejemplo, el tiempo disminuye hasta unos 0.7 segundos con 4 nodos, pero luego vuelve a subir por encima de 1 segundo al usar 6 o 7. Estas fluctuaciones sugieren que el sistema comienza a experimentar cuellos de botella o interferencias internas cuando la cantidad de workers activos supera cierto umbral operativo.

Este patrón se repite en otras cargas como 700 o 1500 iteraciones, donde el mejor rendimiento se alcanza con un número intermedio de nodos, y luego los tiempos empiezan a incrementarse nuevamente. Esto puede deberse a la forma en que los WorkerNodes comunican con el StorageServer, cada add_insult() implica no solo una llamada RPC, sino una sincronización explícita con el almacén central, lo que introduce esperas si varios workers intentan escribir al mismo tiempo. El efecto de estas colisiones es más notorio cuando se incrementa la concurrencia, ya que todos los workers están accediendo al mismo punto de sincronización.

En cargas bajas, como 25 o 100 iteraciones, el comportamiento es más plano y constante, el sistema responde rápidamente y no se aprecian mejoras significativas al añadir nodos, como era de esperar. Sin embargo, es importante destacar que incluso con cargas altas, el sistema mantiene tiempos razonables de ejecución y cumple correctamente su función, aunque no con la misma eficiencia que el cliente de filtrado.

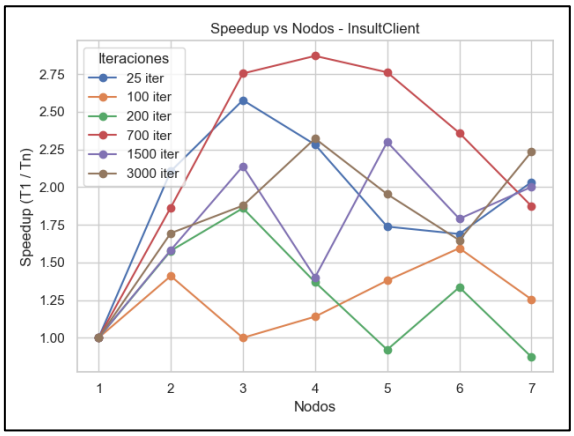


Figura 11.3 (grafica del Speed-Up con el InsultClient)

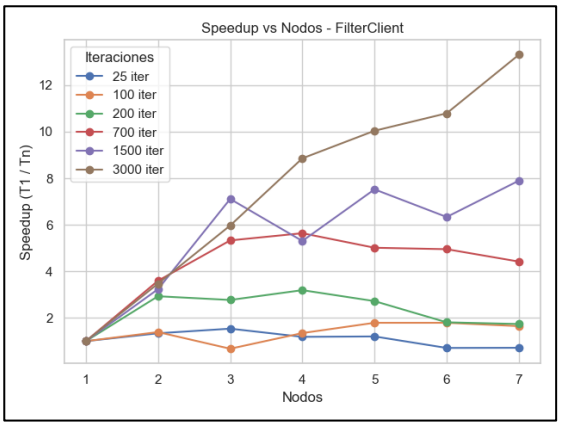


Figura 11.4 (grafica del Speed-Up con el FilterClient)

Speedup del FilterClient (Figura 11.3)
El gráfico de speedup para FilterClient muestra una evolución notablemente progresiva y estable a lo largo de los distintos niveles de carga. Se observa que con cargas altas, como 3000 iteraciones, el sistema es capaz de escalar de forma casi lineal... alcanzando un speedup superior a 13 al utilizar 7 nodos. Esta tendencia se mantiene firme en todas las configuraciones de carga elevada, como también ocurre en el caso de 1500 y 700 iteraciones, donde los valores de speedup superan 6 o 7 de forma consistente desde los 4 nodos en adelante. Esto confirma que la arquitectura Pyro, pese a su simplicidad, es perfectamente capaz de distribuir eficientemente tareas que no requieren acceso constante a datos centralizados ni bloqueos de sincronización.

En cargas intermedias como 200 y 100 iteraciones, el incremento del speedup es más modesto, pero sigue presente. Aunque no se alcanzan valores tan altos como en cargas pesadas, se mantiene una evolución estable y sin retrocesos marcados. El sistema distribuye el trabajo correctamente sin degradar el rendimiento, lo cual valida su robustez en distintos escenarios de uso.

Las pruebas con solo 25 iteraciones arrojan los resultados más limitados. El speedup oscila entre 1.1 y 1.5, lo cual es esperable, ya que el coste asociado a la creación de hilos y comunicación RPC no se ve compensado por el escaso volumen de trabajo. Aun así, la ausencia de caídas o comportamientos erráticos refuerza la estabilidad del sistema y su adaptabilidad a distintas magnitudes de carga.

Speedup del InsultClient (Figura 11.4)
A diferencia del caso anterior, el gráfico de speedup para InsultClient presenta una evolución más inestable y fluctuante. Si bien el sistema logra mejoras importantes en los primeros nodos —alcanzando picos de hasta 2.8 con 3 o 4 nodos en algunas cargas como 700 iteraciones— a partir de cierto punto el rendimiento tiende a degradarse o estancarse. Este patrón sugiere que el sistema comienza a sufrir penalizaciones por la saturación en las comunicaciones y la dependencia de acceso sincronizado al StorageServer.

Con cargas medias y bajas (100 y 200 iteraciones), se observan oscilaciones que indican que el coste de las llamadas remotas empieza a superar el beneficio de paralelizar el trabajo. En algunas configuraciones, incluso se producen retrocesos en el speedup al añadir más nodos, lo que evidencia un punto de inflexión en la escalabilidad. Esto puede explicarse por la necesidad de sincronizar la escritura de cada insulto hacia el componente de almacenamiento, lo cual introduce colisiones entre nodos que intentan acceder simultáneamente al mismo recurso.

En pruebas con muy poca carga (25 iteraciones), el sistema también mejora en los primeros pasos, pero los beneficios se atenúan rápidamente. El comportamiento general del gráfico demuestra que, si bien el modelo Pyro permite escalar en tareas de escritura moderada, **no mantiene la misma eficiencia al aumentar la concurrencia**, debido al acoplamiento directo con el almacenamiento y la naturaleza sincrónica de las operaciones RPC.

REDIS

Como resultado, se generaron datos registrados en tiempos_clientes.log, posteriormente analizados mediante performance.py, que produjo los gráficos de comportamiento por cliente. En las Figuras 12.1 y 12.2 se observa la relación entre el número de nodos y el tiempo de ejecución para FilterClient e InsultClient respectivamente. Mientras que las Figuras 12.3 y 12.4 muestran el speedup obtenido en cada caso con respecto a la ejecución base con un único nodo.

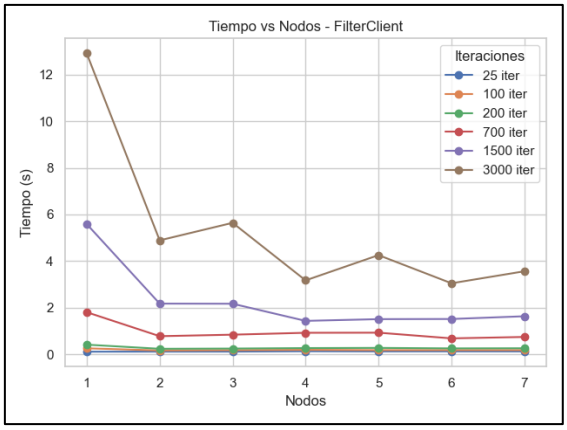


Figura 12.1 (grafica del rendimiento con el FilterClient)

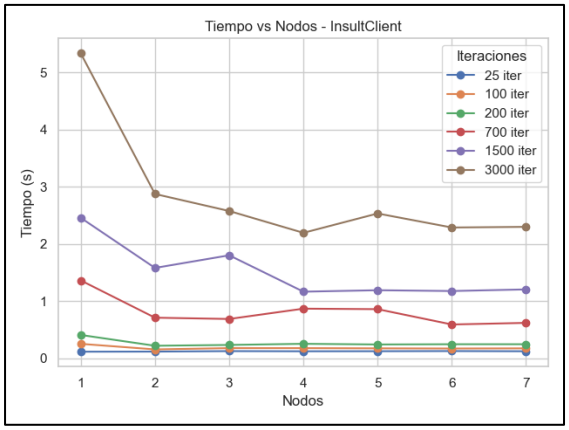


Figura 12.2 (grafica del rendimiento con el InsultClient)

Rendimiento de FilterClient (Figura 12.1)

El gráfico correspondiente a FilterClient en Redis presenta una caída significativa del tiempo de ejecución al pasar de 1 a 2 nodos, especialmente evidente en cargas altas como 3000 y 1500 iteraciones. En esos casos, el tiempo se reduce de forma drástica (por ejemplo, de más de 12 segundos a 5), lo cual indica que el sistema comienza a aprovechar la paralelización de forma efectiva.

Sin embargo, a partir del tercer nodo, se observa una tendencia más irregular. Aunque los tiempos tienden a mejorar globalmente, hay oscilaciones en cargas medias y altas que podrían deberse a condiciones de concurrencia internas, o a la forma en que Redis gestiona las colas cuando múltiples workers están extrayendo simultáneamente. Por ejemplo, el caso de 3000 iteraciones muestra una caída hasta los 3 segundos con 4 nodos, pero luego el tiempo vuelve a subir y bajar, lo que sugiere que el sistema no escala de forma completamente lineal.

En las cargas más pequeñas (25 a 200 iteraciones), los tiempos son ya muy bajos desde el primer nodo, y apenas se reducen con más workers. El tiempo se estabiliza por debajo de 0.2 segundos, lo que indica que el coste de encolar, consumir y procesar tareas es mínimo, y que la arquitectura Redis maneja adecuadamente este tipo de tareas distribuidas. En resumen, la arquitectura resulta eficiente, aunque algo sensible a ciertos niveles de carga intermedia, posiblemente debido a sincronización o a latencias de escritura en el contador de resultados.

Rendimiento de InsultClient (Figura 12.2)

En el caso de InsultClient, la evolución de los tiempos de ejecución también es positiva al pasar de 1 a 2 nodos, especialmente en cargas medias y altas como 700, 1500 y 3000 iteraciones. Los tiempos se reducen significativamente, por ejemplo de más de 5 segundos a menos de 3 en el caso de 3000 iteraciones. Sin embargo, a diferencia del FilterClient, el descenso posterior es más suave y se estabiliza a partir de los 4 nodos, con pequeñas fluctuaciones.

Estas oscilaciones pueden explicarse por la mayor frecuencia de operaciones de escritura que implica InsultClient, ya que cada petición requiere un acceso a Redis para almacenar un insulto y actualizar el contador insult_done. Aunque Redis maneja estos accesos eficientemente, la contienda entre múltiples threads que escriben simultáneamente puede generar pequeñas variaciones en el rendimiento, especialmente si las peticiones se distribuyen de forma no uniforme entre workers.

En cargas bajas (por debajo de 200 iteraciones), el tiempo de ejecución ya es bajo con un solo nodo (por debajo de 0.3 segundos), y apenas se reduce al aumentar el número de nodos. De hecho, en algunos casos el tiempo se incrementa levemente, lo cual es coherente con el coste añadido de tener múltiples hilos activos compitiendo por recursos compartidos en Redis.

En conjunto, los resultados muestran que Redis es una opción robusta y eficiente para tareas concurrentes, especialmente cuando el número de peticiones es elevado. El sistema escala razonablemente bien, aunque presenta ciertas limitaciones esperables cuando las tareas son pequeñas o las escrituras intensas.

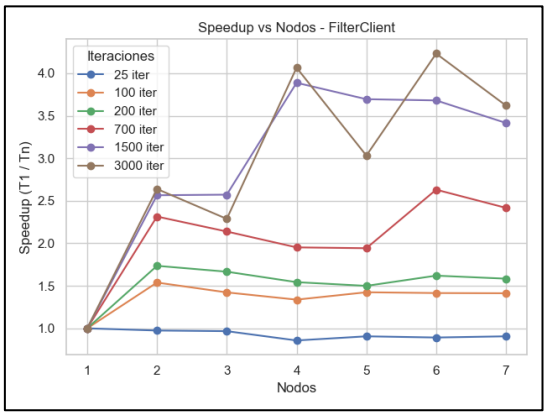


Figura 12.3 (grafica del Speed-Up con el FilterClient)

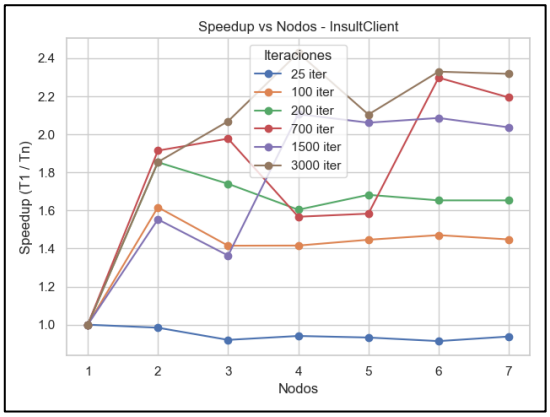


Figura 12.4 (grafica del Speed-Up con el InsultClient)

Speedup del FilterClient (Figura 12.3)
El comportamiento del sistema Redis en cuanto al speedup de FilterClient muestra una evolución coherente con los resultados anteriores. En cargas altas como 3000 y 1500 iteraciones, se alcanza un speedup de más de 4 al utilizar 6 o 7 nodos, lo cual indica que la arquitectura es capaz de distribuir eficientemente tareas de filtrado incluso con muchos nodos activos. El crecimiento es progresivo y sostenido hasta el cuarto nodo, y aunque existen ciertas oscilaciones a partir de ese punto, no se observan retrocesos drásticos. Esta estabilidad sugiere que Redis es capaz de mantener un rendimiento escalable en entornos de lectura intensiva sin saturarse.

En cargas intermedias (700 y 200 iteraciones), el speedup sigue mejorando hasta aproximadamente 2 o 2.5 veces respecto al nodo único, especialmente al utilizar entre 3 y 5 nodos. A partir de ahí, la mejora se estabiliza, lo cual es razonable considerando que el volumen de trabajo ya comienza a ser suficientemente absorbido por los nodos disponibles.

En el caso de las cargas más ligeras (25 y 100 iteraciones), la ganancia de rendimiento es más limitada, con valores de speedup que apenas superan 1.2. Este comportamiento refleja que, en escenarios con muy pocas tareas, el coste de comunicación y sincronización de múltiples workers no se compensa con la distribución de la carga. Aun así, la gráfica no presenta caídas abruptas ni inestabilidad, lo que refuerza la solidez del sistema Redis incluso en condiciones poco exigentes.

Speedup del InsultClient (Figura 12.4)
El speedup del cliente InsultClient es más moderado, pero también más estable que en arquitecturas como Pyro o XML-RPC. En cargas pesadas (3000 y 1500 iteraciones), el sistema alcanza valores de hasta 2.3 con 6 o 7 nodos, mostrando que Redis es capaz de escalar también en escenarios de escritura intensiva, aunque con un margen más limitado que en tareas de lectura.

Lo destacable es que, a pesar de no alcanzar una escalabilidad lineal, el sistema muestra una mejora constante sin pérdidas significativas de rendimiento. Incluso en cargas medias como 700 y 200 iteraciones, el speedup se mantiene entre 1.6 y 2.0, lo cual representa un rendimiento aceptable teniendo en cuenta que cada insulto implica acceso y modificación en Redis (actualización del contador, escritura en la cola, etc.).

En cargas bajas, el crecimiento del speedup es muy reducido, y en el caso de 25 iteraciones prácticamente se mantiene plano. Sin embargo, esto se considera normal, ya que el tiempo absoluto de ejecución es tan bajo que paralelizar no ofrece una mejora real. Aun así, no se observan degradaciones al aumentar los nodos, lo cual es un punto positivo para la estabilidad del sistema.

En conjunto, los resultados de speedup reflejan que la arquitectura Redis multi-nodo logra un buen equilibrio entre simplicidad, paralelismo y rendimiento, especialmente para cargas medias y altas. El desacoplamiento natural que ofrece Redis mediante colas compartidas permite que el sistema escale sin grandes penalizaciones por acoplamiento entre procesos.

RabbitMQ

Como resultado, se generaron datos registrados en tiempos_clientes.log, posteriormente analizados mediante performance.py, que produjo los gráficos de comportamiento por cliente. En las Figuras 13.1 y 13.2 se observa la relación entre el número de nodos y el tiempo de ejecución para FilterClient e InsultClient respectivamente. Mientras que las Figuras 13.3 y 13.4 muestran el speedup obtenido en cada caso con respecto a la ejecución base con un único nodo.

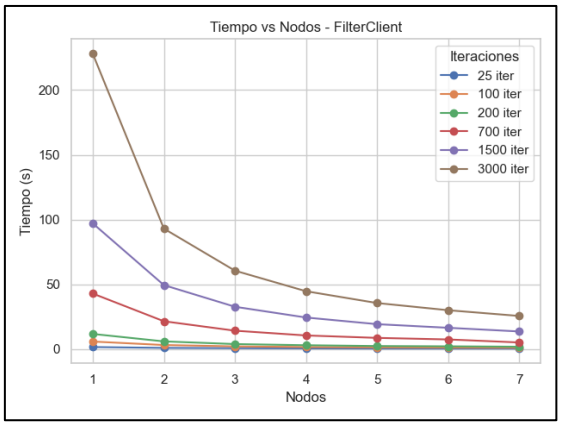


Figura 13.1 (grafica del rendimiento con el FilterClient)

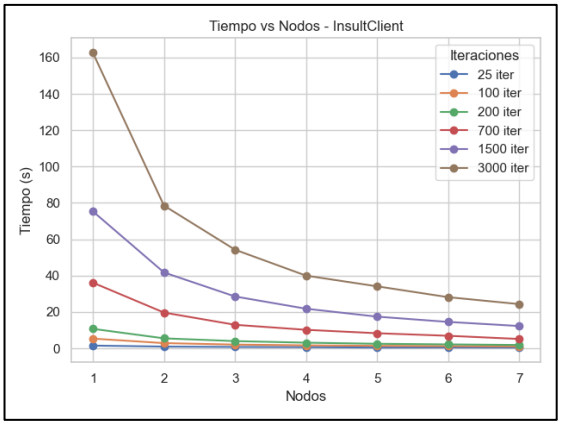


Figura 13.2 (grafica del rendimiento con el InsultClient)

Rendimiento de FilterClient (Figura 13.1)

El gráfico de tiempos para FilterClient en RabbitMQ evidencia un comportamiento altamente escalable y consistente a medida que se incrementa el número de nodos. En escenarios de carga alta, como con 3000 y 1500 iteraciones, se observa una reducción drástica del tiempo de ejecución entre 1 y 2 nodos, por ejemplo, en el caso de 3000 iteraciones, el tiempo desciende de más de 220 segundos a menos de 100 con solo duplicar los workers. Este patrón se mantiene con una caída progresiva a medida que se añaden más nodos, lo que demuestra una buena capacidad del sistema para repartir la carga de trabajo sin cuellos de botella significativos.

A partir del tercer nodo en adelante, aunque la curva comienza a estabilizarse, se sigue obteniendo mejora en todas las configuraciones. Incluso con 6 o 7 nodos, el sistema continúa reduciendo el tiempo, especialmente en las cargas más exigentes, lo que indica que RabbitMQ gestiona adecuadamente la distribución de tareas mediante sus colas. El desacoplamiento entre productores y consumidores permite que cada worker procese su lote de peticiones sin interferencias, aprovechando mejor la concurrencia.

En cargas más ligeras como 100 o 200 iteraciones, los tiempos también descienden, aunque de forma más discreta. El tiempo base ya es bajo, y la reducción progresiva con más nodos, aunque real, tiene un margen menor. Aun así, el sistema muestra estabilidad y eficiencia sin penalizaciones notables, lo que confirma que el modelo basado en colas de RabbitMQ es adecuado tanto para alta carga como para escenarios menos intensivos.

Rendimiento de InsultClient (Figura 13.2)

El comportamiento de InsultClient en RabbitMQ es también muy positivo y refleja una respuesta eficiente y estable al incremento de nodos, incluso bajo cargas intensivas. Para las 3000 iteraciones, el tiempo se reduce desde aproximadamente 165 segundos con un solo nodo hasta menos de 40 con siete workers... lo que representa una mejora considerable. Lo mismo ocurre con las demás cargas elevadas: 1500 y 700 iteraciones muestran una caída pronunciada y sostenida del tiempo de ejecución, sin oscilaciones preocupantes.

La clave de esta eficiencia está en la arquitectura asincrónica de RabbitMQ: los mensajes (insultos en este caso) son encolados y procesados de forma autónoma por cada worker, lo que evita la saturación por llamadas directas como ocurre en arquitecturas síncronas. Además, al desacoplar el flujo de datos, los trabajadores pueden escalar horizontalmente sin necesidad de coordinación explícita o bloqueo compartido.

En las cargas pequeñas (25 a 200 iteraciones), el comportamiento se mantiene estable. Aunque la ganancia en tiempo es menos significativa (porque ya se parte de ejecuciones cortas), no se observan incrementos de tiempo ni penalizaciones por tener más nodos. Esto refuerza la fiabilidad del sistema en entornos mixtos donde puede haber variabilidad en la carga.

En conjunto, RabbitMQ demuestra ser una de las arquitecturas más robustas y escalables en términos de rendimiento. Tanto FilterClient como InsultClient se benefician claramente del aumento de nodos, especialmente en contextos de alta concurrencia, sin que se presenten anomalías en los tiempos o saturaciones prematuras del sistema.

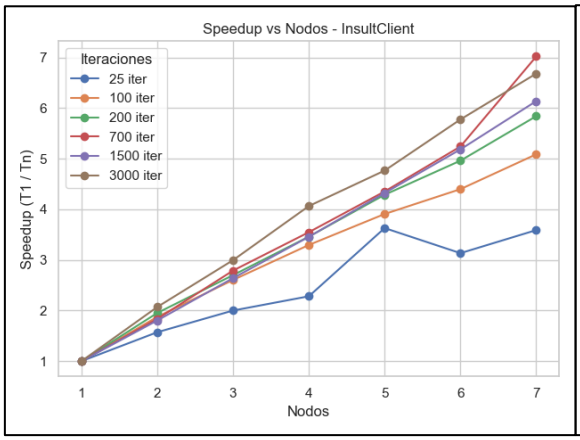


Figura 13.3 (grafica del Speed-Up con el InsultClient)

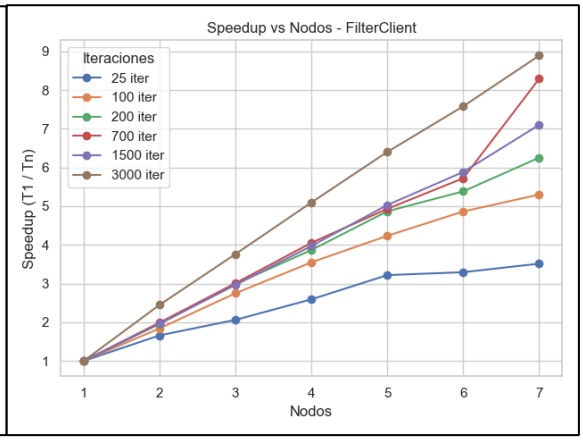


Figura 13.4 (grafica del Speed-Up con el FilterClient)

Speedup del InsultClient (Figura 13.3)
El comportamiento del speedup en InsultClient refuerza aún más la idea de que RabbitMQ es altamente eficiente para arquitecturas distribuidas. A diferencia de otras tecnologías donde la carga de escritura puede introducir cuellos de botella, aquí se observa una curva de crecimiento firme y sostenida en todas las configuraciones de carga, incluyendo aquellas que implican muchas operaciones de escritura como las de 1500 o 3000 insultos.

En el caso de 700 y 3000 iteraciones, el sistema supera los valores de speedup de 7 con 7 nodos, manteniendo una pendiente casi constante... lo que demuestra que incluso en operaciones que implican interacción frecuente con la cola de mensajes y escritura de resultados, el sistema continúa distribuyendo las tareas eficientemente. RabbitMQ lo logra gracias a su arquitectura basada en brokers y encolado, que permite desacoplar totalmente la producción de mensajes de su consumo.

Además, en cargas más ligeras, como 100 o 200 iteraciones, el sistema también presenta una mejora progresiva con cada nodo añadido, alcanzando speedup de entre 4 y 5 con 7 nodos. El comportamiento sigue siendo estable, sin caídas, lo que indica que el coste de mantener múltiples consumidores nunca supera el beneficio de paralelizar, incluso con volúmenes de trabajo relativamente bajos.

La carga más baja (25 iteraciones) es la única en la que el crecimiento del speedup se vuelve más discreto, aunque sigue siendo estable y ascendente. Este patrón era esperable y coherente con los resultados observados en otras tecnologías: en trabajos tan pequeños, el paralelismo aporta menos beneficio. No obstante, la ausencia de regresiones o picos refuerza la solidez y eficiencia del sistema RabbitMQ en todos los rangos de uso.

Speedup del FilterClient (Figura 13.4)
La gráfica de speedup para FilterClient en RabbitMQ muestra uno de los comportamientos más ideales y predecibles entre todas las tecnologías analizadas. La evolución es casi perfectamente lineal, especialmente en las cargas altas, como 3000 y 1500 iteraciones. En estos casos, el sistema alcanza un speedup cercano a 9 con 7 nodos, lo que indica que casi todo el trabajo adicional aportado por los workers se traduce en ganancia de rendimiento... sin cuellos de botella ni ineficiencias notables. Este comportamiento es una clara consecuencia del modelo asincrónico que proporciona RabbitMQ, donde los workers operan de forma independiente al ritmo que las colas de tareas les asignan.

También es destacable que las cargas medias (700 y 200 iteraciones) muestran curvas de speedup crecientes, regulares y sin oscilaciones. Aunque la pendiente es algo menos pronunciada que en las cargas máximas, el crecimiento es constante, alcanzando valores entre 6 y 7 con 7 nodos. En estos escenarios, el sistema aprovecha bien la paralelización, sin penalizaciones por el número de nodos activos.

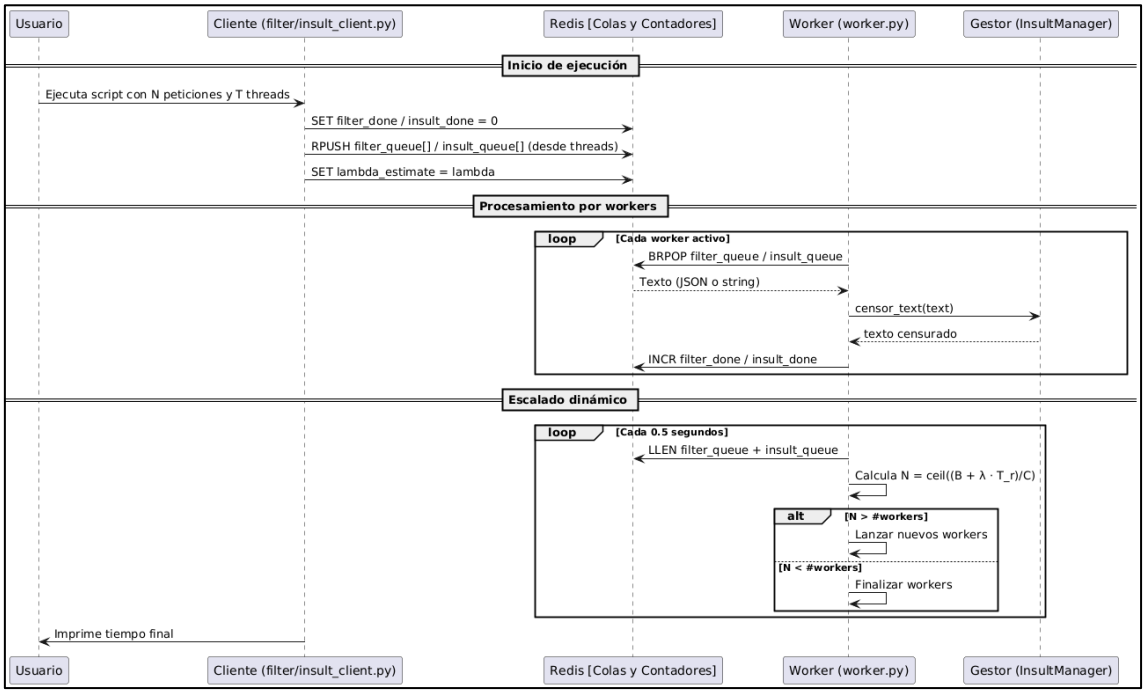
Incluso en los casos de cargas bajas, como 25 y 100 iteraciones, el comportamiento del sistema sigue siendo destacable. Aunque el speedup en estos casos no es lineal (algo previsible, debido al bajo volumen de trabajo), sí que mantiene un crecimiento constante, lo que sugiere que el overhead de comunicación y coordinación entre nodos es muy bajo. En conjunto, esta gráfica evidencia que RabbitMQ ofrece una escalabilidad excepcional para tareas de lectura distribuida.

SISTEMA DISTRIBUIDO DINÁMICO (REDIS)

En este apartado se presenta el desarrollo e implementación de un sistema distribuido dinámico, en el cual se ha optado por utilizar exclusivamente la tecnología Redis como base de comunicación y coordinación entre componentes. A diferencia de las arquitecturas anteriores (como XML-RPC, Pyro o RabbitMQ).

A lo largo de este apartado se detallará cómo se ha estructurado el sistema, qué componentes lo conforman, y cómo Redis ha permitido implementar una arquitectura capaz de crecer o reducirse en tiempo real, ajustando sus recursos a la demanda de forma eficiente y sin intervención manual. Posteriormente, se analizarán los resultados obtenidos y su impacto en el rendimiento y la escalabilidad.

Arquitectura Implementada



La arquitectura mostrada en el diagrama de secuencia representa el funcionamiento del sistema distribuido Redis Dinámico, diseñado para adaptar automáticamente el número de nodos trabajadores en función de la carga del sistema. Se trata de una estructura modular y completamente asíncrona, que se apoya en Redis como componente central de comunicación, coordinación y almacenamiento temporal. A continuación se explican sus tres bloques principales:

Inicio de ejecución

El flujo de funcionamiento del sistema comienza cuando el usuario ejecuta uno de los dos clientes disponibles, filter_client.py o insult_client.py, especificando un número concreto de peticiones (N) y un número de hilos (T). Esta simple acción desencadena una serie de pasos clave que preparan al sistema distribuido para gestionar la carga de trabajo entrante de forma eficiente y coordinada.

En primer lugar, el cliente realiza una operación SET en Redis para inicializar un contador: filter_done o insult_done, según el tipo de cliente. Este contador tiene como función monitorizar cuántas de las tareas han sido correctamente procesadas por los workers, permitiendo saber en qué momento se ha completado el total de peticiones. A continuación, el cliente realiza una operación de inserción masiva (RPUSH) a la cola correspondiente (filter_queue o insult_queue). Esta operación se realiza en paralelo desde cada hilo, lo que permite una distribución no bloqueante y altamente eficiente de las tareas hacia Redis.

Por último, el cliente calcula una estimación de la tasa de llegada de peticiones (λ, lambda) en base al tiempo que ha tardado en realizar el envío de todas las tareas. Esta estimación se almacena en Redis mediante otra operación SET sobre la clave lambda_estimate. Esta tasa será posteriormente utilizada por el sistema de escalado dinámico para calcular de forma adaptativa cuántos workers deberían estar activos en cada momento, en función del tamaño de la cola de tareas y del objetivo de tiempo de respuesta.

Gracias a estas operaciones iniciales, Redis queda completamente configurado como centro de coordinación del sistema: contiene todas las tareas pendientes encoladas y los contadores necesarios para supervisar el progreso, así como la estimación de la carga actual del sistema. Desde ese momento, los workers entran en acción y el sistema empieza a procesar las peticiones, escalando dinámicamente si es necesario para mantener el rendimiento.

Procesamiento por workers

Cada uno de los *workers* activos en el sistema entra en un bucle continuo de trabajo en el que se mantiene atento a nuevas tareas por procesar. En primer lugar, realiza una operación BRPOP (Figura 20.1) sobre ambas colas disponibles en Redis: filter_queue e insult_queue. Esta operación es bloqueante pero extremadamente eficiente, ya que permite al worker "dormir" sin consumir recursos de CPU hasta que Redis le entrega una nueva tarea. Gracias a ello, se evita el uso de bucles ocupados innecesarios y se garantiza una respuesta inmediata en cuanto hay datos disponibles.

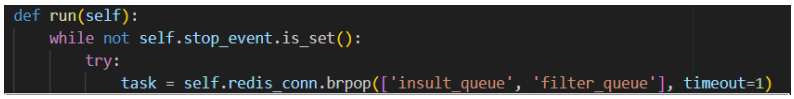


Figura 20.1 (BRPOP en cada Cola)

Una vez que el worker recibe una nueva tarea, esta llega como una cadena de texto que puede estar en formato plano o codificada en JSON. El worker se encarga de analizar y extraer el contenido útil, normalmente una cadena de texto a filtrar, y lo reenvía directamente al módulo InsultManager. Este módulo es responsable de aplicar el procesamiento requerido, que en el caso de los textos se concreta en la función censor_text(), sustituyendo los insultos detectados por la palabra "CENSORED".

Después de procesar correctamente la tarea, el worker ejecuta una operación INCR sobre el contador correspondiente en Redis (filter_done o insult_done). Este contador es esencial para que el cliente pueda conocer el estado del procesamiento y saber cuándo se han completado todas las peticiones enviadas. Gracias a este diseño basado en colas y contadores, los workers funcionan de manera completamente desacoplada: no necesitan conocerse entre sí ni compartir estado, lo que permite un paralelismo natural y escalable. Cada worker opera de forma autónoma, respondiendo a la demanda del sistema y procesando tareas conforme estas aparecen en las colas.

Escalado dinámico

Uno de los elementos más característicos y potentes de esta arquitectura basada en Redis es el componente de escalado dinámico. Este módulo se ejecuta en paralelo al resto del sistema y actúa de forma periódica, evaluando el estado del sistema cada 0.5 segundos. Su misión principal es adaptar en tiempo real el número de *workers* activos a la carga de trabajo actual, garantizando un equilibrio entre rendimiento y eficiencia de recursos.

Durante cada iteración, el escalador realiza dos acciones clave: primero, consulta Redis para obtener la longitud total de las colas de tareas pendientes (tanto filter_queue como insult_queue) mediante una operación LLEN, y recupera el valor de lambda_estimate, que representa la tasa estimada de llegada de peticiones. A partir de estos valores, se aplica la fórmula de cálculo del número óptimo de trabajadores necesarios:

$$N = \left\lceil \frac{B + (\lambda \cdot Tr)}{C} \right\rceil$$

donde:

- **B** es el tamaño actual de la cola,
- **λ** es la tasa estimada de llegada de tareas,
- **Tr** es el tiempo objetivo de respuesta (por defecto, 1 segundo),
- **C** es la capacidad promedio de procesamiento por nodo (por ejemplo, 4 tareas por segundo).

Con el resultado de N, el sistema evalúa el número actual de *workers*. Si N es mayor al número actual, se lanzan nuevos *workers* para satisfacer la demanda. Si es menor, se eliminan algunos *workers* activos para liberar recursos. Este ajuste se realiza de forma completamente autónoma, sin intervención externa, logrando una elasticidad que permite absorber picos de carga y reducir consumo en momentos de baja actividad. Gracias a este mecanismo, el sistema ofrece una solución eficiente, reactiva y autosuficiente en entornos distribuidos.

```python
def scaler(redis_conn, workers, stop_event, min_workers=1, max_workers=15, check_interval=0.5):
    T_r = 1.0  # objetivo de respuesta
    T = 0.25   # tiempo medio de procesamiento
    C = 1 / T

    while not stop_event.is_set():
        try:
            B = redis_conn.llen('insult_queue') + redis_conn.llen('filter_queue')
            lambda_est = estimate_lambda()
            if lambda_est < 0.01:
                lambda_est = 10
        except ConnectionError:
            print("[WARN] Redis desconectado. Escalador deteniéndose.")
            break

        # Escalado con mayor resistencia a partir de B > 3000
        if B < 100:
            N = 1
        elif B >= 30000:
            N = max_workers
        else:
            N = int((B - 100) / (30000 - 100) * (max_workers - 1)) + 1

        N = max(min(N, max_workers), min_workers)
        current_workers = len(workers)

        print(f"[STATE] B={B}, lambda={lambda_est:.2f}, N={N}, workers={current_workers}", flush=True)

        if N > current_workers:
            for _ in range(N - current_workers):
                worker_stop = threading.Event()
                worker = Worker(redis_conn, worker_stop)
                workers.append((worker, worker_stop))
                worker.start()
            print(f"[INFO] Escalando a {len(workers)} workers")
            with open("worker.log", "a") as f:
                f.write(f"Escalando a {len(workers)} workers\n")

        elif N < current_workers:
            for _ in range(current_workers - N):
                worker, worker_stop = workers.pop()
                worker_stop.set()
                worker.join()
            print(f"[INFO] Reduciendo a {len(workers)} workers")
            with open("worker.log", "a") as f:
                f.write(f"Reduciendo a {len(workers)} workers\n")

        time.sleep(check_interval)
```

Figura 20.1 (Código de escalado de workers)

El sistema de escalado dinámico implementado en la Figura 20.1 tiene como objetivo adaptar en tiempo real el número de *workers* activos en función de la carga de trabajo recibida por el sistema. Para ello, se calcula periódicamente (cada 0.5 segundos) un valor estimado de carga a partir de dos métricas principales: la longitud combinada de las colas (insult_queue + filter_queue) y la tasa estimada de llegada de peticiones (lambda_est). Esta última se obtiene mediante un análisis temporal de los eventos LPUSH realizados sobre Redis. Si la estimación de lambda es muy baja o inestable, se fuerza a un valor mínimo para evitar una infraestimación del trabajo y garantizar un mínimo de capacidad operativa.

Una vez obtenida la carga (B) y la tasa (λ), se calcula un número teórico óptimo de trabajadores activos (N) usando la fórmula clásica de dimensionamiento en la Figura 20.2.

```python
# Escalado con mayor resistencia a partir de B > 3000
if B < 100:
    N = 1
elif B >= 30000:
    N = max_workers
else:
    N = int((B - 100) / (30000 - 100) * (max_workers - 1)) + 1

N = max(min(N, max_workers), min_workers)
current_workers = len(workers)
```

Figura 20.2 (Formula de escalado)

El resultado de la fórmula se compara con el número actual de *workers* en ejecución, y si el sistema detecta que necesita más capacidad, lanza nuevos hilos Worker en paralelo. Del mismo modo, si detecta infrautilización, procede a cerrar ordenadamente los hilos sobrantes. Todo esto se realiza de forma autónoma y sin intervención externa, lo que convierte a este módulo en el núcleo inteligente del sistema distribuido, capaz de autoescalar su capacidad en función del tráfico real y mejorar tanto el rendimiento como el uso de recursos.

Pruebas y Resultados

Para generar las pruebas de escalado dinámico realizadas en Redis, se ha empleado un script test.sh (Figura 20.3) que lanza múltiples ejecuciones de montaje.sh variando dos parámetros clave: el número de iteraciones y la cantidad de hilos utilizados por los clientes. Este procedimiento permite simular diferentes escenarios de carga y observar cómo el sistema reacciona al crecimiento del trabajo encolado. El comportamiento del escalador ha sido registrado en el archivo worker.log, donde se almacena el número de workers activos en cada momento, mientras que los tiempos de procesamiento por cliente se guardan en tiempos_clientes.log.

```
#!/bin/bash

> tiempos_clientes.log
> worker.log

iteraciones=(25 200 700 1500 3000 7000 12000 20000 50000 100000)
max_nodos=(1 4 7 10)

for num in "${iteraciones[@]}"; do
    for i in "${max_nodos[@]}"; do
    echo "TEST Nodos: $i, Iteraciones: $num"
        ./montaje.sh $i $num
        sleep 2
    done
done
```

Figura 20.3 (Test del código nodos-dinámica)

En la gráfica generada (Figura 20.4), se observa claramente cómo el número máximo de workers escala de forma progresiva con el aumento del número de iteraciones. Para cargas pequeñas (por debajo de las 700 iteraciones), apenas se activan más de uno o dos workers, independientemente del número de hilos empleados, ya que la carga es fácilmente absorbida por un solo nodo. Sin embargo, conforme se incrementan las iteraciones, especialmente a partir de 3000, el sistema empieza a escalar con mayor agresividad hasta alcanzar el máximo de 15 workers en pruebas de alta demanda (como 50.000 y 100.000 iteraciones). Esta escalabilidad progresiva evidencia que el sistema responde correctamente a la saturación de la cola mediante el escalado proporcional previsto por la fórmula.

Además, al comparar las curvas por número de threads, se aprecia que una mayor concurrencia (por ejemplo, con 10 threads) acelera la acumulación de tareas en Redis, provocando un escalado más temprano. A pesar de ello, todos los escenarios convergen al mismo máximo de workers, lo que valida que la política de escalado es estable y no depende exclusivamente del número de hilos, sino del volumen real de trabajo. Esto confirma que la implementación dinámica con Redis consigue adaptar sus recursos de forma eficiente según la carga, manteniendo un equilibrio entre rendimiento y consumo.

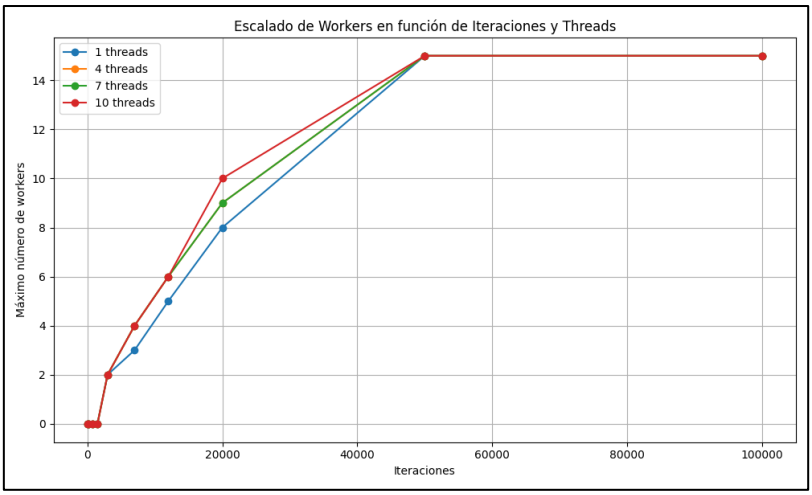


Figura 20.4 (Relación workers generados x iteraciones)

CONCLUSIONES

A lo largo de este trabajo se han explorado e implementado diversas arquitecturas de sistemas distribuidos utilizando tecnologías como XML-RPC, Pyro, Redis y RabbitMQ, con un enfoque particular en cómo gestionar tareas concurrentes de forma eficiente. Cada una de estas tecnologías ha presentado ventajas y limitaciones específicas en cuanto a facilidad de integración, rendimiento, paralelismo y adaptabilidad. Gracias a esta diversidad de pruebas, ha sido posible observar y comparar cómo se comportan diferentes arquitecturas ante cargas crecientes, tanto en contextos estáticos como en entornos con escalado dinámico.

Una de las principales conclusiones obtenidas es que, si bien arquitecturas como XML-RPC y Pyro resultan sencillas de implementar y permiten una distribución básica de tareas, su rendimiento no escala tan bien como el de soluciones orientadas a colas de mensajes como RabbitMQ o Redis. En particular, RabbitMQ ha demostrado ser muy eficaz para manejar volúmenes altos de peticiones, manteniendo una baja latencia y una elevada escalabilidad. Redis, por otro lado, ha sido especialmente interesante al implementar una versión dinámica del sistema, capaz de ajustar el número de workers activos en tiempo real según la carga del sistema y la tasa de llegada de peticiones.

Finalmente, la implementación del sistema distribuido dinámico con Redis ha demostrado que es posible construir arquitecturas adaptativas eficientes con recursos relativamente simples. La capacidad de escalar automáticamente el número de nodos ha permitido optimizar el uso de recursos y mantener tiempos de respuesta estables bajo cargas variables. Este enfoque representa un paso importante hacia soluciones distribuidas más inteligentes, capaces de equilibrar rendimiento y eficiencia sin requerir supervisión manual constante. En conjunto, el trabajo ha servido como una sólida base práctica y comparativa sobre la aplicación de sistemas distribuidos en entornos concurrentes.