Proyecto: Replicación Activa

Fecha: 18/12/2024

Autores:
- Mihail Cojocaru Chitoroaga
- Oier Layana Medrano
- Oier Alduncin Azcárate

Índice

Pruebas de rendimiento

{clientes, réplicas, manejadores}

Aumentando los manejadores:

{1, 1, 1} → 80.84 ± 9.5 op/s
{1, 1, 3} → 80.65 ± 5.13 op/s
{1, 1, 5} → 64.11 ± 17.58 op/s

{1, 3, 1} → 74.9 ± 7.7 op/s
{1, 3, 3} → 71.62 ± 12.14 op/s
{1, 3, 5} → 68.77 ± 4.5 op/s

{1, 5, 1} → 79.77 ± 1.66 op/s
{1, 5, 3} → 72.42 ± 5.26 op/s
{1, 5, 5} → 67.46 ± 6.28 op/s

{3, 1, 1} → 121.0 ± 6.5 op/s
{3, 1, 3} → 146.5 ± 17.11 op/s
{3, 1, 5} → 113.67 ± 10.1 op/s

{3, 3, 1} → 151.11 ± 9.44 op/s
{3, 3, 3} → 150.44 ± 21.47 op/s
{3, 3, 5} → 126.2 ± 42.6 op/s

{3, 5, 1} → 137.5 ± 24.4 op/s
{3, 5, 3} → 131.2 ± 5.9 op/s
{3, 5, 5} → 96.3 ± 24.5 op/s

Parece que aumentando solo el número de manejadores el rendimiento baja. Esto puede deberse a que hace falta enviar los mensajes del secuenciador a más manejadores.

Aumentando las réplicas:

{1, 1, 1} → 80.84 ± 9.5 op/s
{1, 3, 1} → 74.9 ± 7.7 op/s
{1, 5, 1} → 79.77 ± 1.66 op/s

{1, 1, 3} → 80.65 ± 5.13 op/s
{1, 3, 3} → 71.62 ± 12.14 op/s
{1, 5, 3} → 72.42 ± 5.26 op/s

{1, 1, 5} → 64.11 ± 17.58 op/s
{1, 3, 5} → 68.77 ± 4.5 op/s
{1, 5, 5} → 67.46 ± 6.28 op/s

{3, 1, 1} → 121.0 ± 6.5 op/s
{3, 3, 1} → 151.11 ± 9.44 op/s
{3, 5, 1} → 137.5 ± 24.4 op/s

{3, 1, 3} → 146.5 ± 17.11 op/s
{3, 3, 3} → 150.44 ± 21.47 op/s
{3, 5, 3} → 131.2 ± 5.9 op/s

{3, 1, 5} → 113.67 ± 10.1 op/s
{3, 3, 5} → 126.2 ± 42.6 op/s
{3, 5, 5} → 96.3 ± 24.5 op/s

Aumentando únicamente el número de réplicas no hay un patrón tan claro, aunque en la mayoría el que mejor rendimiento obtiene es el que utiliza 3 réplicas. Harían falta pruebas más exhaustivas para poder afirmarlo con certeza.

Aumentando los clientes:

{1, 1, 1} → 80.84 ± 9.5 op/s
{3, 1, 1} → 121.0 ± 6.5 op/s

{1, 1, 3} → 80.65 ± 5.13 op/s
{3, 1, 3} → 146.5 ± 17.11 op/s

{1, 1, 5} → 64.11 ± 17.58 op/s
{3, 1, 5} → 113.67 ± 10.1 op/s

{1, 3, 1} → 74.9 ± 7.7 op/s
{3, 3, 1} → 151.11 ± 9.44 op/s

{1, 3, 3} → 71.62 ± 12.14 op/s
{3, 3, 3} → 150.44 ± 21.47 op/s

{1, 3, 5} → 68.77 ± 4.5 op/s
{3, 3, 5} → 126.2 ± 42.6 op/s

{1, 5, 1} → 79.77 ± 1.66 op/s
{3, 5, 1} → 137.5 ± 24.4 op/s

{1, 5, 3} → 72.42 ± 5.26 op/s
{3, 5, 3} → 131.2 ± 5.9 op/s

{1, 5, 5} → 67.46 ± 6.28 op/s
{3, 5, 5} → 96.3 ± 24.5 op/s

Aumentando los clientes de 1 a 3 aumenta el rendimiento del sistema. Esto puede deberse a que el sistema permite más operaciones concurrentes que las que generan 3 clientes. Haría falta un estudio más en detalle para buscar el límite.

Pruebas de tolerancia a fallos

Para simular fallos hemos realizado varias pruebas simulando fallos de manejadores y de réplicas. Las pruebas se han realizado con 2 clientes, 2 manejadores y 2 réplicas. Los clientes eligen manejadores de forma aleatoria.

Fallo en manejador

Hemos puesto que, tras pasar 0.3 segundos ejecutándose, falle el primer manejador. Hemos observado que la ejecución general no se detiene, aunque se ralentiza porque los clientes pueden seleccionar al primer manejador para el envío de mensajes y, tras no recibir respuesta y saltar el timeout, seleccionan al segundo manejador. Esto lo hemos comprobado reduciendo el tiempo Δ de timeout. Hemos visto que reduciendo este tiempo ha aumentado el número de mensajes entregados en el mismo tiempo, lo cual verifica nuestra hipótesis anterior.

En consecuencia, hemos observado que no se detiene la ejecución y la consistencia sigue siendo atómica. Solamente se ralentiza la ejecución general al tener que esperar al timeout los clientes que seleccionan el primer manejador. Por lo tanto, hemos verificado que nuestra implementación es tolerante a fallos de manejadores.

Fallo en réplica

Al igual que en la prueba anterior, hemos puesto que, tras una ejecución de 0.3 segundos, falle en este caso la primera réplica. Hemos observado que la ejecución ha seguido su curso de forma correcta. Se han entregado un poco más de 320 mensajes en la prueba con fallo y algo más de 360 mensajes en la prueba correcta. Esta diferencia de mensajes entre la prueba con el fallo de réplica y la prueba de ejecución sin fallo se puede achacar perfectamente a la naturaleza del sistema de replicación activa.

En consecuencia, hemos visto que un fallo en una réplica no es terminal y que nuestra implementación continúa con su ejecución. Por lo tanto, podemos concluir que nuestra implementación es tolerante también a fallos en réplicas.

Verificación de consistencia

Para poder verificar adecuadamente la consistencia del sistema que hemos implementado, hemos hecho uso del verificador que implementamos en la práctica H1. Para ello, al verificador se le ha de pasar un log con los eventos de invocación y respuesta que se hayan generado durante la ejecución. Estos eventos, en nuestro sistema, son generados por cada cliente, ya que cada uno loguea todos los eventos realizados; los mensajes generados (invocación) y los mensajes recibidos (respuestas). Para poder verificar que todos los clientes al unísono mantienen una consistencia atómica, hemos agrupado todos los logs derivados de cada uno de los clientes y ordenado por su marca temporal.

Una vez unificados todos los logs en uno solo, al verificador de consistencia se le pasa ese log para poder garantizar que mantienen una consistencia atómica.

Cuestiones

Cliente

Responda a las siguientes cuestiones:

1. **Cada cliente se asocia a un único proceso de la aplicación. Asumiendo que el proceso de la aplicación se comporta secuencialmente, comprueba que por cada acción ReqCommand(OP) sólo se produce una única respuesta dada por el efecto Deliver_ResCommand(OP, RES).**

   Es imposible que se entregue más de una respuesta por cada acción ReqCommand(OP) ya que cuando se entrega un mensaje aumenta el valor de la variable *opnum*, por lo que al hacer la comprobación del siguiente mensaje repetido no se cumple la condición de msg.cmd =⟨CLTid, opnum, op⟩. En consecuencia, no es posible que se entreguen mensajes repetidos aunque lleguen varias respuestas al mismo mensaje al cliente.

2. **Describe el papel que juega la variable *running* y su efecto desde el proceso de la aplicación.**

   La variable *running* hace que cada cliente solo pueda procesar una petición de aplicación a la vez. Esto es así porque, cuando llega una petición a un cliente por primera vez, running vale false. En consecuencia, la ejecución hará que *running* se ponga a true inmediatamente y acabará enviando un mensaje a algún manejador. Como el único momento para que *running* se ponga a false de nuevo es a la hora de entregar el mensaje, el cliente no aceptará peticiones nuevas de la aplicación hasta que termine con la actual.

3. **Describe cual es el efecto del Timeout. Realiza una propuesta para ajustar el valor de Δ.**

   Timeout se usa como seguro en caso de que no haya llegado respuesta al mensaje enviado. Al haber pasado un tiempo Δ sin recibir respuesta, el cliente asume que el manejador al que ha enviado el mensaje se ha caído, por lo que envía el mensaje de nuevo a otro manejador. Este proceso se repite hasta obtener respuesta.

   A la hora de realizar una propuesta para ajustar el valor de Δ se pueden estudiar varias ideas. Si se conoce la latencia media a la hora de recibir respuestas, se puede usar esta latencia para proponer el valor de Δ. Por ejemplo, Δ puede tomar un valor que sea el triple de la latencia media. Así, nos aseguramos que dejamos margen de error a cualquier ralentización de la red.

4. **Indica si es posible o no eliminar el Timeout. Indique si esto supone alguna ventaja.**

El Timeout solamente sirve para reenviar un mensaje si el manejador al que se le ha enviado el mensaje por primera vez se encuentra caído. Si tenemos la certeza de que el manejador elegido nunca va a fallar, se puede eliminar el Timeout directamente. Sin embargo, como en la realidad esto no es así, el Timeout es necesario para asegurar que todos los mensajes enviados por un cliente correcto se entregan siempre que haya al menos un manejador y una réplica correctas.

5. **La productividad del servicio por cliente en cada momento, se mide como el número de operaciones completadas en el cliente hasta ese momento entre el tiempo que ha tardado el servicio en realizar dichas operaciones. Modifique el código para obtener dicha información en el log del cliente.**

Modificar código. Para calcular la cantidad de mensajes entregados se podría usar la variable *opnum*. Tan solo hay que medir los tiempos que se desean y calcular la productividad directamente para escribirla después en el log de cada cliente (por ejemplo cada segundo). Esta solución tiene un problema, y es que si el cliente falla antes de recibir la respuesta de la última solicitud contaría una operación más para la productividad. Para solucionar esto podemos crear otra variable que almacene el número de operaciones terminadas y trabajar con ella.

Manejador

Responda a las siguientes cuestiones:

1. **Realice una demostración de la propiedad anterior.**

   Primero vamos a analizar lo que quiere decir la expresión.

   Para cualquier ejecución del sistema de replicación y para cualquier estado alcanzable S se cumple que:

   ∀ RHid, RHid'∈ RHIDs:
   if, for any j, S[RHid].sequenced[j] ≠ null ∧ S[RHid'].sequenced[j] ≠ null then
   S[RHid].sequenced[j] = S[RHid'].sequenced[j]

   La expresión se lee de la siguiente forma: para todo par RHid y RHid' pertenecientes al conjunto de identificadores de manejadores RHIDs tal que si, para cualquier j tal que en cualquier estado alcanzable S, el vector de comandos *sequenced* del manejador RHid en la posición j no es null y el vector de comandos *sequenced* del manejador RHid' en la posición j tampoco es null, se cumple que en ese estado alcanzable S el comando del vector de comandos *sequenced* del manejador RHid en la posición j y el comando del vector de comandos *sequenced* del manejador RHid' en la posición j son iguales.

   Es decir, que todos los manejadores tienen los mismos comandos en las mismas posiciones, cumpliendo así con el consenso entre todos los manejadores dado por el orden total del sistema de comunicación empleado TO-URB.

   Ahora vamos a intentar hacer la demostración:

   Un mensaje de tipo REQUEST llega a un manejador RH. Como ha llegado por primera vez, no está secuenciado y se hace TOBroadcast. Cuando el secuenciador asigna orden al mensaje, entrega este mensaje a todos los manejadores. Como los manejadores no tienen el comando de este mensaje en cada respectivo *sequenced*, lo secuencian. Así, todos los manejadores obtienen el mismo orden en los comandos de los mensajes.

   Esto es posible gracias a las propiedades del TOBroadcast y a los canales de comunicación *quasi-fiables*. Las propiedades del TOBroadcast garantizan el orden total en los mensajes y los canales *quasi-fiables* garantizan que no se pierden mensajes (mientras los procesos de cada extremo de los canales sean correctos).

2. **Cuando se recibe el mensaje 〈m.rhid, m.cmd〉 tenemos ocho posibles casos en función de si m.rhid es igual o distinto a RHid, m.cmd pertenezca o no a mycommands, y que m.cmd pertenezca o no a sequenced. Estudie estos ocho casos para analizar si se puede sacar alguna ventaja de alguno de ellos en cuanto a reducción de número de mensajes a transmitir a las réplicas.**

Caso 1: m.rhid = RHid, m.cmd ∈ mycommands, m.cmd ∈ sequenced.

El mensaje ha sido emitido para secuenciar por este manejador, el cliente espera una respuesta de este manejador y el mensaje ya está secuenciado. Esto puede deberse a que el cliente ha vuelto a enviarle la petición que ya le había enviado previamente. Si el canal es fiable no es necesario reenviar la solicitud a la réplica, ya que probablemente aún la esté manejando o esté en proceso de volver.

Caso 2: m.rhid ≠ RHid, m.cmd ∈ mycommands, m.cmd ∈ sequenced.

El mensaje ha sido emitido para secuenciar por otro manejador, pero este ya lo tiene secuenciado y en sus comandos, es decir que es posible que al manejador le haya llegado la petición del cliente que tras no recibir respuesta de este manejador ha optado por probar otros. Si definimos algún método para que el manejador que ha enviado el mensaje sepa que este ya ha hecho la solicitud a las réplicas, podríamos hacer que ahorrara hacerla de nuevo. Esto supondría más comunicación entre manejadores, así que es posible sea peor en términos de eficiencia.

Caso 3: m.rhid = RHid, m.cmd ∉ mycommands, m.cmd ∈ sequenced.

El que ha emitido el mensaje es este manejador, pero no lo tiene en sus comandos y además ya está secuenciado. Esta situación es imposible, ya que si lo ha emitido para secuenciar, lo ha tenido que añadir a sus comandos, salvo que el secuenciador y el medio sean tan rápidos que aún no haya llegado a añadirlo. Además, si se da el caso, el hecho de que esté secuenciado significa que ya hay otro que ha solicitado la secuenciación del mensaje (pues si ya estuviera secuenciado de antes no lo hubiera mandado al secuenciador). Esto significa que este manejador es extremadamente lento comparado con los demás. Si se da esa situación sería un error en el sistema, ya que este manejador nunca manejaría ese comando.

Caso 4: m.rhid ≠ RHid, m.cmd ∉ mycommands, m.cmd ∈ sequenced.

El mensaje ha sido emitido por otro manejador, aunque este ya lo tiene secuenciado y no está en sus comandos. Esto quiere decir que hay un tercer manejador que ha solicitado la secuenciación del mensaje anteriormente. Es una situación parecida al caso 2, salvo que vista desde un tercer manejador.

Caso 5: m.rhid = RHid, m.cmd ∈ mycommands, m.cmd ∉ sequenced.

Este mensaje ha sido emitido para secuenciar por este manejador, el cual también lo ha añadido a sus comandos correctamente tras emitirlo. El hecho de que no esté secuenciado nos indica que es la primera vez que este manejador recibe este mensaje del secuenciador. El programa deberá seguir su flujo normal, solicitando respuesta a las réplicas.

Caso 6: m.rhid ≠ RHid, m.cmd ∈ mycommands, m.cmd ∉ sequenced.

Este mensaje ha sido emitido por otro manejador, pero este ya lo tiene en sus comandos y no lo tiene secuenciado. Esto nos indica que ya hay otro manejador que ha solicitado la secuenciación de este mensaje antes de que este reciba la respuesta del suyo. Puede deberse a un timeout en el cliente muy bajo (ya sea porque no ha recibido respuesta de este manejador o del otro o al revés) y envíe las solicitudes más rápido de lo que tardan siquiera en secuenciar los manejadores. Si establecemos un método de comunicación entre los manejadores se podría ahorrar una solicitud a las réplicas en el mejor caso, lo cual no parece muy provechoso respecto al coste.

Caso 7: m.rhid = RHid, m.cmd ∉ mycommands, m.cmd ∉ sequenced.

Es una situación muy parecida al caso 3. La única diferencia es que no ha habido una secuenciación por parte de otro manejador. Esta situación también supondría un error en el sistema.

Caso 8: m.rhid ≠ RHid, m.cmd ∉ mycommands, m.cmd ∉ sequenced.

Este mensaje ha sido emitido para secuenciar por otro, no está en los comandos de este manejador y además no lo tenía secuenciado previamente. Esto sirve solo para que el manejador esté al tanto del avance de la secuencia, solo debe registrar este mensaje y avanzar el localseq. No hay nada que se pudiera hacer para ahorrar mensajes en este caso, ya que este manejador no hará ninguna solicitud a las réplicas. Es meramente un mensaje informativo.

Réplica

Responda a las siguientes cuestiones:

1. **En los autómatas anteriores no nos hemos preocupado de la longitud que pueden alcanzar las variables sequenced, toexecute, o execeuted. Se debe ofrecer, en una implementación real, algún mecanismo de recolección de basura. Indique las posibilidades que tiene para mantener finitas estas variables.**

   Para implementar el recolector de basura nos basamos principalmente en la propiedad de los clientes de hacer las peticiones secuencialmente, es decir, solo piden un mensaje si han recibido la respuesta del anterior.

   sequenced -> Manejador; Se pueden borrar aquellos mensajes que ya no se van a utilizar más (cualquiera que no sea el último enviado por un cliente y que su número de secuencia sea < lastServed).

   toexecute -> Réplica; Se pueden borrar mensajes obsoletos (los cuales se acaban de ejecutar y no están pendientes de ejecutarse) cuando llegue una nueva petición del cliente.

   executed -> Réplica; Se pueden borrar mensajes obsoletos (ya ejecutados) cuando llegue una nueva petición del cliente.

2. **Por otro lado, si *seq* es un número entero indique cuantas operaciones se realizarán antes de que se desborde.**

   El tamaño máximo de un número entero depende del lenguaje de programación que se use. En nuestro caso, como estamos usando NodeJS, tenemos que todos los números, incluídos enteros y reales, son representados como *Number* usando el formato de punto flotante de doble precisión de IEEE 754 (64 bits). Por lo tanto, el valor máximo de *seq* antes de desbordarse será $2^{53} - 1$.

   Aunque seguramente no haga falta, si se desean usar números mayores se puede usar *BigInt*, que permite representar enteros de tamaño arbitrario. Sin embargo hay que tener en cuenta que para operar entre ellos hay que realizar conversiones de uno a otro.

3. **Sugiere algún cambio que mejore el rendimiento del sistema. En particular, estudie algún debilitamiento de la consistencia de las operaciones que pueda implementarse fácilmente.**

   Vamos a estudiar la debilitación de consistencia atómica a consistencia de prefijo con el objetivo de mejorar el rendimiento del sistema. Con la consistencia de prefijo, se garantiza que el lector observe una secuencia ordenada de escrituras empezando desde la primera de las escrituras. Por ejemplo, la lectura puede ser respondida por

una réplica que recibe escrituras en orden desde una réplica maestra pero que no ha recibido todavía un número de escrituras recientes. En otras palabras, el lector ve una versión de los datos que ha existido en la réplica maestra en algún momento en el pasado.

Como la consistencia de prefijo solo modifica las lecturas, las escrituras seguirán del mismo modo en el que están. Por otra parte, podemos modificar las lecturas para aumentar el rendimiento del sistema. Para obtener consistencia de prefijo no hace falta que las lecturas pasen por el secuenciador para garantizar el orden total.

Solamente con este cambio se logra la consistencia de prefijo. De esta forma, las lecturas devolverán valores que han estado en el registro en algún momento en el pasado. Haciendo este cambio el rendimiento del sistema debería mejorar, ya que las lecturas no tienen que pasar por el mecanismo del orden total, decreciendo así el número de mensajes y la carga del sistema.

Otra debilitación a una consistencia todavía menor sería pasar a consistencia eventual. Para obtener esta consistencia se puede quitar directamente el mecanismo de orden total. De esta forma, solamente se garantiza consistencia eventual y se logra disminuir en gran medida la cantidad de mensajes que se generan y la carga del sistema. De esta forma, aumenta el rendimiento y disminuye la latencia. Sin embargo, para obtener los beneficios deseados hay que ver hasta qué punto merece la pena perder consistencia para aumentar el rendimiento del sistema.

Horas de trabajo

	Cliente	Manejador + Secuenciador	Réplica
Oier L	3	2.5	7.5
Mihail	2.5	6.5	4
Oier A	8	2	3