Los desafíos de los sistemas distribuidos

Jacob Gabrielson

Desde el momento en que agregamos el segundo servidor, los sistemas distribuidos se convirtieron en el estilo de vida de Amazon. Cuando comencé a trabajar en Amazon en 1999, teníamos tan pocos servidores que los apodábamos con nombres reconocibles como “fishy” u “online-01”. Sin embargo, aún en 1999, la informática distribuida no era sencilla. En aquel entonces, al igual que ahora, los desafíos de los sistemas distribuidos involucraban la latencia, la escalabilidad, la comprensión de las API de redes, la serialización y la deserialización de datos, y la complejidad de los algoritmos como Paxos. A medida que los sistemas crecían en tamaño y se distribuían con mayor rapidez, lo que habían sido casos teóricos extremos se convirtieron en sucesos regulares.

Es difícil desarrollar servicios informáticos distribuidos como servicios públicos; por ejemplo, redes telefónicas confiables de larga distancia, o servicios de Amazon Web Services (AWS). Además, la informática distribuida es *más peculiar* y *menos intuitiva* que los demás tipos de informática debido a dos problemas interrelacionados. Las **fallas independientes** y la **falta de determinación** generan los problemas más graves en los sistemas distribuidos. Además de las fallas informáticas típicas a las que la mayoría de los ingenieros están acostumbrados, las fallas en los sistemas distribuidos se pueden producir de muchas otras formas. Lo peor es que no siempre es posible *saber* si algo deja de funcionar.

En toda la Biblioteca de creadores de Amazon, abordamos la forma en que AWS lidia con los problemas complejos de desarrollo y operaciones que surgen de los sistemas distribuidos. Antes de detallar estas técnicas en otros artículos, vale la pena repasar los conceptos que hacen de la informática distribuida un modelo tan peculiar. Primero, repasemos cuáles son los tipos de sistemas distribuidos.

Tipos de sistemas distribuidos

Los sistemas distribuidos varían en cuanto a la dificultad de implementación. Por un lado, encontramos los sistemas distribuidos *desconectados*. Entre ellos, se incluyen los sistemas de procesamiento por lotes, los clústeres de análisis de big data, las granjas de renderización de escenas de películas y los clústeres de plegamiento de proteínas, entre otros. Si bien su implementación no es asunto menor, los sistemas distribuidos desconectados presentan casi todos los beneficios de la informática distribuida (como la escalabilidad y la tolerancia a errores) y prácticamente ninguna de sus desventajas (los modos de falla complejos y la falta de determinación).

Luego, están los sistemas distribuidos *flexibles de tiempo real*. Se trata de sistemas críticos que deben generar o actualizar resultados de forma permanente, pero que disponen de un período relativamente largo para hacerlo. Algunos ejemplos incluyen creadores de índices de búsqueda, sistemas que detectan servidores en mal estado, roles para Amazon Elastic Compute Cloud (Amazon EC2), etc. Según la aplicación, un indexador de búsqueda puede permanecer desconectado durante un período de 10 minutos a varias horas sin perjudicar a los clientes. Los roles para Amazon EC2 deben insertar las credenciales actualizadas en prácticamente todas las instancias EC2, pero disponen de horas para hacerlo, ya que las credenciales anteriores permanecen vigentes durante un tiempo.

Por último, están los sistemas distribuidos *de tiempo real crítico* que resultan ser los más complejos. Generalmente, se los denomina servicios de solicitud y respuesta. En Amazon, cuando pensamos en crear un sistema distribuido, lo primero que se nos viene a la mente son los sistemas de tiempo real crítico. Desafortunadamente, es más difícil lograr que este tipo de sistemas funcione correctamente. Su complejidad se debe a que las solicitudes son impredecibles y se deben brindar respuestas con mayor rapidez (por ejemplo, porque el cliente la está esperando). Algunos ejemplos incluyen servidores web front-end, la canalización de compras, las transacciones con tarjeta de crédito, todas las API de AWS, la telefonía, etc. En este artículo, nos enfocamos principalmente en los sistemas distribuidos de tiempo real críticos.

Los sistemas de tiempo real críticos son peculiares

En una de las historias de los cómics de *Superman*, el superhéroe se encuentra con su alter ego llamado *Bizarro*, quien vive en un planeta (*Bizarro World*) donde todo está al revés. Bizarro *se parece* a Supermán, pero en realidad es malvado. Los sistemas distribuidos de tiempo real críticos son iguales. Se parecen a la informática común, pero son diferentes y, en realidad, tienen un poco de maldad.

El desarrollo de este tipo de sistemas es peculiar por un motivo: las redes de solicitud y respuesta. *No* nos referimos a los detalles esenciales de TCP/IP, DNS, sockets u otros protocolos similares. Esos temas son muy difíciles de comprender, pero se asemejan a otros problemas complejos de la informática.

Lo que dificulta los sistemas distribuidos de tiempo real críticos es el hecho de que la red permite enviar mensajes desde un *dominio de error* a otro. El envío de un mensaje puede parecer algo inofensivo. Pero, en realidad, es allí donde todo comienza a complicarse más de lo normal.

Para mostrar un ejemplo simple, observemos el siguiente fragmento de código de la implementación de Pac-Man. Dado que está diseñado para ejecutarse en un solo equipo, no envía ningún mensaje por la red.

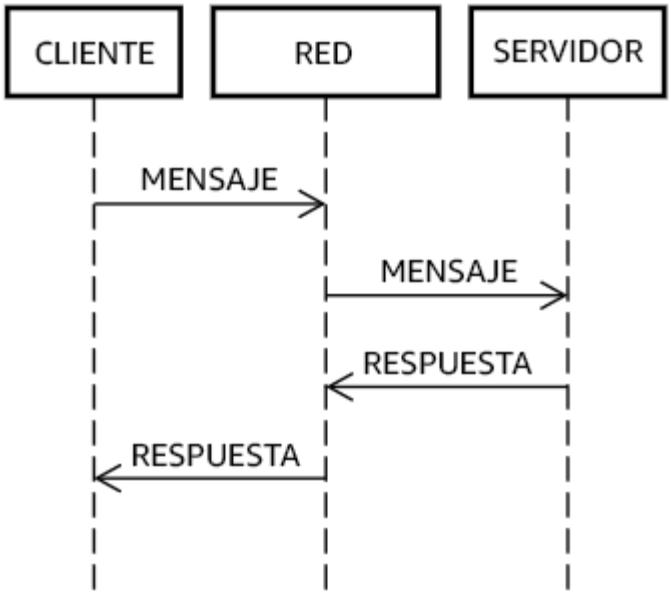
```
board.move(pacman, user.joystickDirection())
ghosts = board.findAll(":ghost")
for (ghost in ghosts)
  if board.overlaps(pacman, ghost)
    user.slayBy(":ghost")
    board.remove(pacman)
    return
```

Imaginemos que debemos desarrollar una versión en red de este código, donde el estado del objeto de la placa permanece en un servidor diferente. Cada llamada al objeto de la placa, como findAll(), resulta en el envío y la recepción de mensajes entre dos servidores.

Siempre que se envía un mensaje de solicitud y respuesta entre dos servidores, se repite el mismo conjunto de ocho pasos, como mínimo. Para comprender el código de Pac-Man en la red, repasemos los conceptos básicos de la mensajería de solicitud y respuesta.

Mensajería de solicitud y respuesta a través de la red

Una acción completa de solicitud y respuesta siempre incluye los mismos pasos. Tal como se muestra en el siguiente diagrama, el equipo del cliente (CLIENTE) envía una solicitud (MENSAJE) a través de la red (RED) al equipo del servidor (SERVIDOR), el cual responde con un mensaje (RESPUESTA), también a través de la red.



En caso de que todo salga bien, se seguirán los siguientes pasos:

1. **PUBLICACIÓN DE LA SOLICITUD**: el CLIENTE envía un MENSAJE de solicitud a través de la RED.
2. **ENTREGA DE LA SOLICITUD**: la RED entrega el MENSAJE al SERVIDOR.
3. **VALIDACIÓN DE LA SOLICITUD**: el SERVIDOR valida el MENSAJE.
4. **ACTUALIZACIÓN DEL ESTADO DEL SERVIDOR**: si es necesario, el SERVIDOR actualiza su estado en función del MENSAJE.
5. **PUBLICACIÓN DE LA RESPUESTA**: el SERVIDOR envía la RESPUESTA a través de la RED.
6. **ENTREGA DE LA RESPUESTA**: la RED entrega la RESPUESTA al CLIENTE.
7. **VALIDACIÓN DE LA RESPUESTA**: el CLIENTE valida la RESPUESTA.
8. **ACTUALIZACIÓN DEL ESTADO DEL CLIENTE**: si es necesario, el CLIENTE actualiza su estado en función de la RESPUESTA.

Son muchos pasos para un insignificante viaje de ida y vuelta. Sin embargo, estos pasos *definen* la comunicación de solicitud y respuesta a través de la red; son inevitables. Por ejemplo, es imposible saltarse el paso 1. El cliente debe enviar el MENSAJE a través de la RED de algún modo. En términos físicos, esto significa que se envían paquetes a través de un adaptador de red, que transmite señales eléctricas mediante cables en una serie de enrutadores que forman la red entre el CLIENTE y el SERVIDOR. Este paso es distinto del número 2, ya que el paso 2 podría fallar debido a razones independientes, como la pérdida repentina de energía del SERVIDOR y la imposibilidad de aceptar los paquetes entrantes. Esta misma lógica puede aplicarse al resto de los pasos.

Por lo tanto, un solo par de solicitud y respuesta en la red desglosa *una* sola acción (llamar a un método) en *ocho* pasos diferentes. Lo peor es que, como se indicó anteriormente, el CLIENTE, el SERVIDOR y la RED pueden fallar *independientemente* unos de otros. El código de los ingenieros debe encargarse de los errores que pudieren surgir en *cualquiera* de los pasos descritos. Pocas veces ocurre esto en la ingeniería típica. Para comprender el porqué, veamos la siguiente expresión de la versión del código de un solo equipo.

```
board.find("pacman")
```

Técnicamente, hay algunas formas peculiares en las que el código podría fallar durante el tiempo de ejecución, incluso si la implementación de la acción board.find no contiene ningún error. Por ejemplo, la

CPU podría sobrecalentarse durante el tiempo de ejecución de forma inesperada. El suministro de energía del equipo también podría fallar espontáneamente. Podría generarse un conflicto en el kernel. La memoria podría llenarse, lo cual evitaría la creación de algunos objetos por parte de la acción board.find. O bien, el disco del equipo en el que se está ejecutando podría llenarse, y la acción board.find no podría actualizar algún archivo de estadísticas y arrojaría un error, aunque probablemente no debería. Un rayo gama podría afectar el servidor y cambiar un bit en la memoria RAM. Pero, la mayor parte del tiempo, los ingenieros no se preocupan por esos problemas. Por ejemplo, las pruebas de las unidades nunca abarcan los casos en los que la CPU podría fallar, y muy pocas veces aquellos en los que se acaba la memoria.

En la ingeniería típica, este tipo de fallas ocurre en un solo equipo, es decir, un solo *dominio de error*. Por ejemplo, si el método board.file falla porque se quemó la CPU de repente, podemos asumir que todo el equipo dejó de funcionar. Ni siquiera es técnicamente posible resolver ese error. Lo mismo ocurre con los demás errores que se mencionaron anteriormente. Se puede intentar escribir pruebas para algunos de estos casos, pero no tiene mucho sentido en la ingeniería típica. Si se producen estos errores, damos por sentado que el resto del sistema también fallará. Se podría decir que todos los elementos *comparten el mismo destino*. Esta realidad reduce en gran medida los diferentes modos de falla que debe gestionar un ingeniero.

Gestión de modos de falla en los sistemas distribuidos de tiempo real críticos

Los ingenieros que trabajan con sistemas distribuidos de tiempo real críticos deben realizar pruebas para cada aspecto de falla de la red, ya que los servidores y la red *no* comparten el mismo destino. A diferencia del caso de un solo equipo, si la red falla, el equipo del cliente seguirá funcionando. Si el equipo remoto falla, el del cliente no dejará de funcionar, y así con cada elemento del sistema.

Para ejecutar una prueba completa de los casos de falla de los pasos de solicitud y respuesta descritos anteriormente, los ingenieros deben asumir que cada uno de los pasos podría fallar. Y deben asegurarse de que el código (tanto en el equipo del cliente como del servidor) siempre se comporte correctamente ante esos errores.

Observemos un ejemplo de una acción completa de solicitud y respuesta donde se producen errores:

1. **Error en la PUBLICACIÓN DE LA SOLICITUD**: la RED no pudo entregar el mensaje (por ejemplo, debido a una avería inesperada del enrutador intermedio) o el SERVIDOR lo rechazó de forma explícita.
2. **Error en la ENTREGA DE LA SOLICITUD**: la RED entrega el MENSAJE correctamente al SERVIDOR, pero este se avería justo después de recibir el MENSAJE.
3. **Error en la VALIDACIÓN DE LA SOLICITUD**: el SERVIDOR decide que el MENSAJE no es válido. Puede haber muchas causas. Por ejemplo, paquetes dañados, versiones de software incompatibles o errores en el cliente o el servidor.
4. **Error en la ACTUALIZACIÓN DEL ESTADO DEL SERVIDOR**: el SERVIDOR intenta actualizar su estado, pero no lo consigue.
5. **Error en la PUBLICACIÓN DE LA RESPUESTA**: independientemente de si el SERVIDOR logra emitir una respuesta o no, se podría producir un error al publicarla. Por ejemplo, su tarjeta de red podría quemarse justo en el momento equivocado.
6. **Error en la ENTREGA DE LA RESPUESTA**: se puede producir un error en la RED al entregar la RESPUESTA al CLIENTE como ya se describió, incluso si la RED funcionaba durante los pasos anteriores.
7. **Error en la VALIDACIÓN DE LA RESPUESTA**: el CLIENTE decide que la RESPUESTA no es válida.

8. **Error en la ACTUALIZACIÓN DEL ESTADO DEL CLIENTE**: el CLIENTE recibió la RESPUESTA, pero no pudo actualizar su estado, comprender el mensaje (debido a un problema de compatibilidad) o se produjo algún otro error.

Estos modos de falla dificultan en gran medida la informática distribuida. Yo los llamo los *ocho modos de falla apocalípticos*. En vista de estos modos de falla, volvamos a analizar la expresión del código de Pac-Man.

```
board.find("pacman")
```

Esta expresión se expande a las siguientes actividades del lado del cliente:

1. Se publica un mensaje, como {action: "find", name: "pacman", userId: "8765309"}, en la red, dirigido al equipo de la Placa.
2. Si la red no está disponible, o se rechaza explícitamente la conexión con el equipo de la Placa, se produce un error. Este caso es especial porque el cliente sabe que, en definitiva, no es posible que el equipo del servidor haya recibido la solicitud.
3. Se espera una respuesta.
4. Si la respuesta nunca llega, se agota el tiempo de espera. En este paso, que se agote el tiempo de espera implica que el resultado de la solicitud es DESCONOCIDO. Se puede haber efectuado o no. El cliente debe lidiar con este estado de forma correcta.
5. Si se recibe una respuesta, se determina si es de operación exitosa, de error o una respuesta incomprensible o dañada.
6. Si se trata de un error, se deserializa la respuesta y se la convierte en un objeto comprensible para el código.
7. Si es una respuesta incomprensible o de error, se genera una excepción.
8. Al momento de lidiar con la excepción, se debe determinar si se *vuelve a enviar* la solicitud o si se detiene el juego.

La expresión también inicia las siguientes actividades del lado del servidor:

1. Recibe la solicitud (lo cual podría no suceder).
2. Valida la solicitud.
3. Busca al usuario para verificar que aún esté activo. (Es posible que el servidor abandonara al usuario debido a que no recibió mensajes de su parte durante demasiado tiempo).
4. Actualiza la tabla persistente del usuario para que el servidor sepa que es posible que siga activo.
5. Busca la ubicación del usuario.
6. Publica una respuesta que contiene una expresión como esta {xPos: 23, yPos: 92, clock: 23481984134}.
7. Cualquier otra lógica del servidor debe lidiar adecuadamente con los efectos futuros del cliente. Por ejemplo, no puede recibir el mensaje, lo recibe pero no lo comprende, lo recibe pero se daña, o lo gestiona correctamente.

En resumen, *una expresión* en código normal se convierte en *quince* pasos adicionales en código de sistemas distribuido de tiempo real críticos. Esta expansión se debe a los ocho puntos distintos en los que puede fallar cada comunicación completa entre el cliente y el servidor. Cada expresión que representa un viaje de ida y vuelta a través de la red, como board.find("pacman"), resulta en la siguiente.

```
(error, reply) = network.send(remote, actionData)
switch error
  case POST_FAILED:
    // handle case where you know server didn't get it
  case RETRYABLE:
    // handle case where server got it but reported transient failure
  case FATAL:
    // handle case where server got it and definitely doesn't like it
  case UNKNOWN: // i.e., time out
    // handle case where the *only* thing you know is that the server received
    // the message; it may have been trying to report SUCCESS, FATAL, or RETRYABLE
  case SUCCESS:
    if validate(reply)
      // do something with reply object
    else
      // handle case where reply is corrupt/incompatible
```

La complejidad es inevitable. Si el código no gestiona todos los casos correctamente, el servicio acabará por fallar de formas extrañas. La idea de escribir pruebas para todos los modos de falla que podría tener un sistema de cliente y servidor como el ejemplo de Pac-Man es inconcebible.

Pruebas de los sistemas distribuidos de tiempo real críticos

La prueba de la versión del fragmento de código de Pac-Man de una sola máquina es comparativamente directa. Cree algunos objetos de Placa, colóquelos en diferentes estados, cree objetos de Usuario en diferentes estados, etc. Los ingenieros pensarían más en las condiciones del borde, y tal vez usarían pruebas generativas o un fuzzer.

En el código de Pac-Man, el objeto de placa se utiliza en cuatro lugares. En el sistema distribuido de Pac-Man, hay cuatro puntos en ese código que tienen cinco resultados diferentes posibles, como se mostró anteriormente (ERROR EN LA PUBLICACIÓN, SE PUEDE VOLVER A INTENTAR, IRRECUPERABLE, DESCONOCIDO o CORRECTO). Estos amplían en gran medida el espacio de estado de las pruebas. Por ejemplo, los ingenieros de los sistemas distribuidos de tiempo real críticos deben gestionar muchas combinaciones. Digamos que falla la llamada board.find() con ERROR EN LA PUBLICACIÓN. Luego, usted debe probar qué sucederá si falla con el resultado SE PUEDE VOLVER A INTENTAR; luego, con IRRECUPERABLE, y así sucesivamente.

Pero incluso todas esas pruebas no son suficientes. En el código tradicional, los ingenieros pueden asumir que si board.find() funciona correctamente, la próxima llamada a la placa, board.move(), también lo hará. En cambio, en la ingeniería de sistemas distribuidos de tiempo real críticos, no existe esa garantía. El equipo del servidor podría fallar en cualquier momento e independientemente del resto. Por eso, los ingenieros deben escribir pruebas para los cinco casos por *cada* llamada a la placa. Supongamos que un ingeniero pensó en 10 escenarios de prueba para la versión de Pac-Man en un solo equipo. En la versión del sistema distribuido, deben probar cada uno de esos escenarios *20* veces. Lo que significa que la matriz de prueba se eleva de 10 a 200.

Pero hay más. El código del *servidor también* podría pertenecer al mismo ingeniero. Se deben probar todas las combinaciones de errores que puedan ocurrir en la red, el cliente y el servidor, de forma que no acaben con un estado dañado. El código del servidor puede ser similar al siguiente.

```
handleFind(channel, message)
  if !validate(message)
    channel.send(INVALID_MESSAGE)
    return
  if !userThrottle.ok(message.user())
    channel.send(RETRYABLE_ERROR)
    return
  location = database.lookup(message.user())
  if location.error()
    channel.send(USER_NOT_FOUND)
    return
  else
    channel.send(SUCCESS, location)

handleMove(...)
  ...

handleFindAll(...)
  ...

handleRemove(...)
  ...
```

Hay cuatro funciones del lado del servidor que se deben probar. Asumamos que cada función, en un solo equipo, tiene cinco pruebas. Son 20 pruebas. Dado que los clientes envían varios mensajes al mismo servidor, las pruebas deben simular secuencias de diferentes solicitudes para garantizar que el servidor permanecerá sólido. Algunos ejemplos de solicitudes son acciones como buscar, mover, quitar y buscar todo.

Digamos que cada construcción tiene 10 escenarios diferentes con un promedio de tres llamadas en cada uno. Se suman 30 pruebas más. Sin embargo, en cada escenario también se deben probar los casos de error. En cada una de esas pruebas, debe simular qué sucedería si el cliente recibe alguno de los cuatro tipos de falla (ERROR EN LA PUBLICACIÓN, SE PUEDE VOLVER A INTENTAR, IRRECUPERABLE o DESCONOCIDO) y envía una solicitud no válida al servidor. Por ejemplo, el cliente puede solicitar la acción "buscar" correctamente, pero recibe un error DESCONOCIDO cuando solicita la acción "mover". En ese caso, podría volver a solicitar la primera acción. ¿El servidor maneja este caso correctamente? Posiblemente, pero no lo sabremos hasta que realicemos la prueba. Al igual que con el código en el lado del cliente, la matriz de pruebas en el lado del servidor también reboza de complejidad.

Gestión del error DESCONOCIDO

Es abrumador considerar todas las combinaciones de fallas que se pueden producir en un sistema distribuido, especialmente en presencia de varias solicitudes. Uno de nuestros enfoques para abordar la ingeniería distribuida es desconfiar de *todo*. Cada línea de código podría dejar de cumplir su función, a menos que sea imposible que cree una comunicación de red.

Probablemente, el problema más difícil de solucionar es el tipo de error DESCONOCIDO descrito en la sección anterior. El cliente no siempre puede saber si la solicitud se ejecutó correctamente. Tal vez *sí* movió a Pac-Man (o bien, en un servicio bancario, retiró dinero de la cuenta bancaria de un usuario), o tal vez *no*. ¿Cómo deberían proceder los ingenieros? Es difícil porque los ingenieros son humanos, y los humanos solemos lidiar con la verdadera incertidumbre. Los seres humanos estamos acostumbrados a considerar

al código de la siguiente manera.

```
bool isEven(number)
  switch number % 2
    case 0
      return true
    case 1
      return false
```

Comprendemos este código porque hace lo que *parece* que hace. Y tenemos dificultades con la versión distribuida del código, que distribuye parte del trabajo a un servicio.

```
bool distributedIsEven(number)
  switch mathServer.mod(number, 2)
    case 0
      return true
    case 1
      return false
    case UNKNOWN
      return WHAT_THE_FARG?
```

Es casi imposible para un ser humano descubrir cómo manejar el error DESCONOCIDO correctamente. ¿Qué significa este tipo de falla realmente? ¿Debería el código volver a intentar ejecutar la operación? Si es así, ¿cuántas veces debería hacerlo? ¿Cuánto tiempo debería esperar entre los intentos? Esta situación empeora aún más cuando el código tiene efectos secundarios. Dentro de una aplicación de presupuestos que se ejecuta en una sola máquina, es fácil retirar dinero de una cuenta, como se muestra en el siguiente ejemplo.

```
class Teller
  bool doWithdraw(account, amount)
    switch account.withdraw(amount)
      case SUCCESS
        return true
      case INSUFFICIENT_FUNDS
        return false
```

Sin embargo, la versión distribuida de esa aplicación es peculiar debido a la posibilidad del error DESCONOCIDO.

```
class DistributedTeller
  bool doWithdraw(account, amount)
    switch this.accountService.withdraw(account, amount)
      case SUCCESS
        return true
      case INSUFFICIENT_FUNDS
        return false
      case UNKNOWN
        return WHAT_THE_FARG?
```

Buscar la manera de manejar el tipo de error DESCONOCIDO es una de las razones por las que las *apariencias engañan* en la ingeniería distribuida.

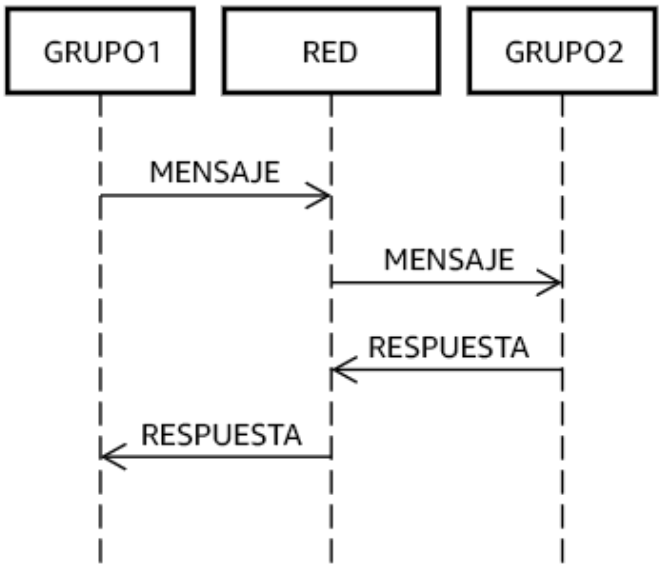
Conjuntos de sistemas distribuidos de tiempo real críticos

Los ocho modos de falla apocalípticos pueden ocurrir en cualquier nivel de abstracción dentro de un sistema distribuido. El ejemplo que detallamos antes se limitaba a un solo equipo de cliente, una red y un equipo de servidor. Incluso en ese caso tan simple, la matriz de estado de falla resultó ser muy compleja. Las matrices de estado de falla de los sistemas distribuidos reales son más complicadas que el ejemplo de un solo equipo de cliente. Los sistemas distribuidos reales constan de varios equipos que se pueden contemplar en distintos niveles de abstracción:

1. Equipos individuales
2. Grupos de equipos
3. Conjuntos de grupos de equipos
4. Entre otros posibles

Por ejemplo, un servicio creado en AWS puede agrupar equipos dedicados a la gestión de recursos dentro de una zona de disponibilidad en particular. Puede haber otros dos grupos de equipos que gestionan dos zonas de disponibilidad distintas. Luego, esos grupos pueden estar comprendidos en un conjunto de una región de AWS. Y este conjunto de región se puede comunicar de forma lógica con otros conjuntos de otras regiones. Desafortunadamente, incluso en este nivel superior, más lógico, *aparecen los mismos problemas*.

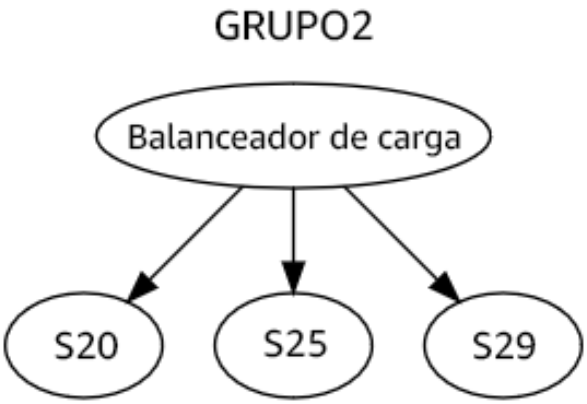
Supongamos que un servicio agrupó algunos servidores en un solo grupo lógico, GRUPO1. A veces, este GRUPO1 podría enviar mensajes a otro grupo de servidores, GRUPO2. Este es un ejemplo de *ingeniería distribuida recursiva*. Todos los modos de falla de red que se describieron anteriormente pueden ocurrir en este caso. Digamos que GRUPO1 desea enviar una solicitud a GRUPO2. Tal como se muestra en el siguiente diagrama, la interacción de solicitud y respuesta entre los dos equipos es exactamente igual a la que observamos en el ejemplo de un solo equipo.



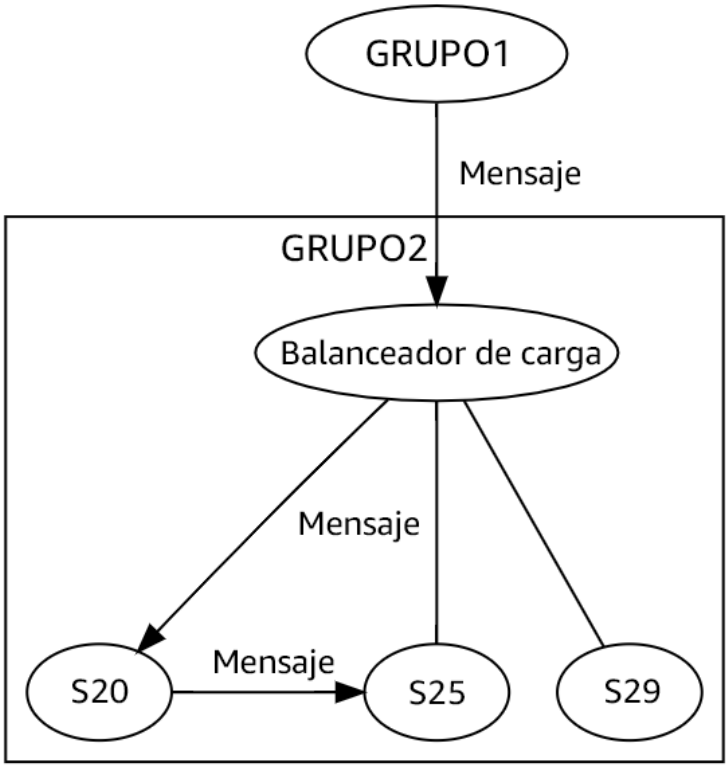
De una forma u otra, alguno de los equipos en GRUPO1 debe enviar un mensaje a través de la RED dirigido (lógicamente) a GRUPO2. Alguno de los equipos en GRUPO2 debe procesar la solicitud, y así

sucesivamente. El hecho de que ambos grupos (GRUPO 1 Y GRUPO 2) estén compuestos por varias máquinas no cambia los aspectos básicos. Aún se pueden producir fallas independientes en GRUPO1, GRUPO2 y la RED.

Sin embargo, esta es solo la perspectiva del nivel de grupo. También existen interacciones entre los equipos dentro de cada grupo. Por ejemplo, el siguiente diagrama muestra cómo podría ser la estructura de GRUPO2.



Primero, se envía un mensaje a un equipo (posiblemente S20) dentro de GRUPO2 a través del balanceador de carga. Los diseñadores del sistema saben que S20 podría fallar durante la etapa de ACTUALIZACIÓN DE ESTADO. Como resultado, S20 podría tener que enviar el mensaje a al menos un equipo diferente, ya sea dentro o fuera de su grupo. ¿Cómo lo logra S20? Puede hacerlo enviando un mensaje de solicitud y respuesta, por ejemplo, a S25; como se muestra en este diagrama.



Por lo tanto, S20 estaría ejecutando una conexión de red de forma recursiva. Nuevamente, se pueden producir las mismas ocho fallas de manera independiente. Así, la ingeniería distribuida es doble y no simple. El mensaje de GRUPO1 a GRUPO2, a nivel lógico, puede fallar de ocho maneras distintas. Ese mensaje genera otro mensaje, donde también se pueden producir las ocho fallas independientes. Entre las pruebas de este escenario se incluirían, al menos, las siguientes:

- Una prueba por cada una de las ocho fallas que se podrían producir en los mensajes a nivel de grupo entre GRUPO1 y GRUPO2.
- Una prueba por cada una de las ocho fallas que se podrían producir en los mensajes a nivel de servidor entre S20 y S25.

Este ejemplo de mensajería de solicitud y respuesta muestra por qué las pruebas de los sistemas distribuidos siguen siendo un problema especialmente fastidioso, incluso después de más de 20 años de experiencia. El proceso de prueba representa un desafío debido a la gran cantidad de casos extremos, pero es muy importante en estos sistemas. Los errores pueden demorar mucho tiempo en aparecer luego de que se implementen los sistemas. Y pueden afectar en gran medida al sistema en sí y a los que se encuentran alrededor.

Los errores distribuidos suelen estar latentes

Si una falla va a suceder eventualmente, es mejor que ocurra lo antes posible. Por ejemplo, es preferible descubrir un problema de escalabilidad en un servicio, el cual se arreglará en seis meses, *al menos* seis meses antes de que el servicio alcance esa escala. Asimismo, es mejor encontrar los errores antes de que lleguen a la producción. Si logran afectar la producción, es preferible encontrarlos rápido, antes de que afecten a muchos clientes o generen otros efectos secundarios.

Los errores distribuidos, es decir, aquellos que resultan de una mala gestión de todas las combinaciones de los ocho modos de falla apocalípticos, suelen ser graves. Los ejemplos a lo largo del tiempo abundan en los sistemas distribuidos grandes, desde telecomunicaciones hasta sistemas de Internet principales. Estas interrupciones no solo son generalizadas y costosas, sino que pueden originarse a partir de errores que se implementaron en la producción meses atrás. Entonces, lleva cierto tiempo desencadenar la combinación de escenarios que realmente causan estos errores (y los extienden por todo el sistema).

Los errores distribuidos se propagan como una epidemia

A continuación, describiré otro problema fundamental en los errores distribuidos:

1. Los errores distribuidos involucran necesariamente el uso de la red.
2. Por lo tanto, es más probable que estos errores se propaguen a otros equipos (o grupos de equipos), ya que, por definición, *ya involucran al único elemento que vincula a los equipos*.

Amazon también ha enfrentado estos errores distribuidos. Un ejemplo antiguo, pero relevante, es la falla general en todo el sitio www.amazon.com. Esta se generó a partir de un error en un solo servidor dentro del servicio de catálogo remoto, cuyo disco se quedó sin espacio.

Debido a la mala administración de aquella condición, el servidor de catálogo remoto comenzó a enviar respuestas vacías a cada solicitud que recibía. Además, enviaba las respuestas en muy poco tiempo, porque es mucho más rápido enviar algo vacío que algo con contenido (al menos en este caso). Mientras tanto, el balanceador de carga entre el sitio web y el servicio de catálogo remoto no se percataba de que la longitud de todas las respuestas era cero. Sin embargo, *sí* notó que las respuestas llegaban mucho más rápido que

las del resto de los servidores de catálogo remoto. Por eso, envió una gran cantidad de tráfico de www.amazon.com al único servidor de catálogo remoto cuyo disco estaba lleno. Efectivamente, todo el sitio web dejó de funcionar porque un único servidor remoto no podía mostrar la información de ningún producto.

Encontramos el servidor dañado rápidamente y lo retiramos del servicio para restaurar el sitio web. Luego, procedimos con el proceso habitual de determinar cuál fue la causa raíz e identificar los problemas para evitar que vuelva a suceder. Compartimos esta información en Amazon para prevenir que otros sistemas experimenten el mismo problema. Además de darnos la lección específica acerca de este modo de falla, este incidente sirvió como un buen ejemplo de cómo los modos de falla se esparcen de forma rápida e impredecible en los sistemas distribuidos.

Resumen de los problemas en los sistemas distribuidos

En resumen, la complejidad de la ingeniería de los sistemas distribuidos se debe a los siguientes motivos:

- Los ingenieros no pueden combinar las condiciones de error. En su lugar, deben considerar muchas combinaciones de fallas. La mayoría de los errores pueden ocurrir en cualquier momento, independientemente de cualquier otra condición de error, por lo que podrían combinarse entre ellos.
- El resultado de cualquier operación de red puede ser DESCONOCIDO, en cuyo caso es posible que la solicitud haya fallado, se haya procesado correctamente o se haya recibido pero no procesado.
- Los problemas distribuidos se producen en *todos* los niveles lógicos de un sistema distribuido, no solo en los equipos físicos de nivel bajo.
- Los problemas distribuidos empeoran en los niveles superiores del sistema, debido a la recursividad.
- Los errores distribuidos suelen aparecer mucho después de su implementación en un sistema.
- Los errores distribuidos se pueden propagar en todo el sistema.
- Muchos de estos problemas provienen de las leyes físicas de las redes, que no se pueden cambiar.

El hecho de que la informática distribuida sea compleja y peculiar no significa que no existan formas de abordar estos problemas. En toda la Biblioteca de creadores de Amazon, analizamos cómo AWS administra los sistemas distribuidos. Esperamos que pueda aplicar lo que hemos aprendido cuando cree sistemas para sus clientes.