Arquitectura ADSOA: Aplicación a calculadora descentralizada.

Javier Vázquez Gurrola
Universidad Panamericana
Facultad de Ingeniería

Resumen

Cuando nuestro software es el encargado de manejar un sistema como podría ser el manejador de cobro de la tienda de abarrotes de nuestro barrio, seguramente si llegase a presentar algún fallo no pasaría mayor inconveniente más allá de la incomodidad del empleado encargado de cobrarle al cliente en turno sus productos y no saber cuánto le tiene que dar de cambio. Sin embargo, esto no es lo mismo cuando el software encargado de activar los frenos del tren bala que va a 285 km/h, falla. Las arquitecturas centralizadas tienen un altísimo riesgo de sufrir algún error que hará que toda la aplicación caiga, y en sistemas críticos, unos cuantos segundos de fallos pueden significar daños y pérdidas irreparables.

Así nace la idea de implementar un sistema distribuido para asegurarse de que aunque una «célula» del sistema cayera, el resto del sistema se encargará de relevarla y tomar sus funciones sin dejar al descubierto y sin atención ninguno de los aspectos importantes para el correcto funcionamiento del sistema.

La arquitectura ADSOA busca combinar dos conceptos muy interesantes como lo son la arquitectura SOA y el concepto de ADS. Esto para ofrecer aplicaciones con capacidades empresariales y que presenten las características de ADS como la tolerancia a fallos y alta disponibilidad, dando así la posibilidad de tener sistemas críticos como pueden ser aquellos de entidades bancarias, sin el riesgo de sufrir pérdidas por fallos del sistema.

Para este trabajo se va a implementar la arquitectura ADSOA a una calculadora distribuida que implementará un modelo de tres eslabones (cliente-nodo-servidor) y que busca explorar a lo largo de su desarrollo y etapa por etapa cada uno de los conceptos más importantes de ambas de las arquitecturas antes ya mencionadas y algunas soluciones para su implementación en lenguaje Java.

I. INTRODUCCIÓN

A. Background

El software es una parte vital de todas las industrias: es una de las herramientas más importantes en el desarrollo futuro de la tecnología. El software juega un papel vital en todos los campos, desde los negocios hasta el gobierno y la educación. Sin embargo esa dependencia que hemos generado cada vez más hacía el uso de software también nos ha llevado a necesitar cada vez más software más preciso, capaz de soportar errores y que sea capaz de manejar grandes cantidades de información sobre una gran demanda. Cuando nuestro software es el encargado de manejar un sistema como podría ser el manejador de cobro de la tienda de abarrotes de nuestro barrio, seguramente si llegase a presentar algún fallo no pasaría mayor inconveniente más allá que la incomodidad del empleado encargado de cobrarle al cliente en turno sus productos y no saber cuánto le tiene que dar de cambio. Sin embargo, conforme escalamos en los eslabones de la cadena de la industria y nos vamos moviendo de sectores o de escalas, nos encontramos con escenarios donde los sistemas se vuelven críticos. En estos sistemas de alta criticidad un fallo no significa un simple inconveniente temporal para el usuario, sino que puede traducirse en pérdidas multimillonarias de dinero o incluso en un accidente catastrófico que cobre la vida de cientos o miles de personas.

Los fallos en el software tradicional ocurren por lo siguiente: la arquitectura tradicionalmente utilizada puede ser conocida como arquitectura centralizada o monolítica, este tipo de arquitecturas son centralizadas, es decir: todo el servicio se encuentra en un solo lugar.. Dicha arquitectura es un modelo tradicional de un software que se compila como una entidad unificada y que es autónoma e independiente de otras aplicaciones, pero centralizada. La palabra "Monolito" hace referencia a algo grande, cosa que no está alejada de la realidad de las arquitecturas monolíticas para el diseño de software. En dichas

arquitecturas todos los procesos están estrechamente ligados y se ejecutan como un solo servicio. Esto significa que si un proceso de una aplicación con arquitectura centralizada experimenta un fallo o un pico en la demanda que niegue el servicio, toda la aplicación se verá afectada y el sistema al que soporta Las arquitecturas centralizadas aumentan el riesgo de la pérdida de disponibilidad en la aplicación ya que debido a la gran cantidad de procesos dependientes y estrechamente vinculados, el impacto de un error en algún proceso se ve significativamente aumentado. Debido a estas desventajas que las arquitecturas de este tipo presentan es que se tuvo la necesidad de enfocar esfuerzos en el desarrollo de nuevas arquitecturas que no comprometan los recursos y disponibilidad de aplicaciones de sistemas en especial aquellos de alto nivel de criticidad.

Un ejemplo de estos sistemas de alto nivel de criticidad es el tren bala de Japón. Este medio de transporte desde el año 2015 alcanza como máximo los 285 kilómetros por hora, sin embargo se tiene registrado que la mayor velocidad alcanzada por el tren fue de 600 kilómetros por hora, incluso rompiendo un récord. A los 285 kilómetros por hora, es difícil para el ojo humano recibir y procesar la información de manera adecuada a tiempo, por ello es que un conductor humano encargado de controlar sistemas de frenado no es viable en un sistema como el tren bala. Sin duda en un país como Japón quedó muy claro que todo el sistema debía ser controlado por computadora, sin embargo surge la pregunta que nos trae a este punto. ¿Qué sucede si el sistema sufre un fallo y cae? Sin duda alguna esto no era un lujo que se pudieran dar. Si el sistema cae, todo lo dependiente como frenos, acelerador, sistemas de soporte y monitoreo, entre otros, estarían fuera, esto podría provocar un accidente de una escala inimaginable. Allí nace la idea de implementar un sistema distribuido para asegurarse que aunque una “célula” del sistema cayera, el resto del sistema se encargará de relevarla y tomar sus funciones sin dejar al descubierto y sin atención ninguno de los aspectos importantes para el correcto funcionamiento del tren.
Este es uno de los casos más fáciles de visualizar, en cuanto a importancia, el uso e implementación de un sistema distribuido, sin embargo este tipo de sistemas se encuentran en elementos como las populares criptomonedas, las cuales funcionan gracias a la tecnología blockchain, cuyo fundamento es un sistema distribuido de información.

SOA: Arquitectura Orientada a servicios

La Arquitectura Orientada a Servicios, también conocida como SOA, es un concepto de arquitectura de software que define el uso de servicios (programas o rutinas que realizan ciertas funciones) para respaldar los requisitos comerciales.
Una arquitectura orientada a servicios crea sistemas de información escalables, compartibles y flexibles que pueden ayudar a las organizaciones a mejorar el rendimiento al mismo tiempo que reducen los costos de TI y aumentan la flexibilidad de los procesos comerciales. Además, proporcionan una forma bien definida de exponer e invocar servicios (por lo general, pero no exclusivamente, servicios web) que facilitan la interacción entre diferentes sistemas propietarios o de terceros.
SOA proporciona un método y un marco para documentar las capacidades comerciales y respalda las actividades de integración y consolidación de datos en cualquier organización. Una arquitectura orientada a servicios es un poderoso aliado cuando se manejan grandes cantidades de datos (datos y jerarquías de datos en la nube).

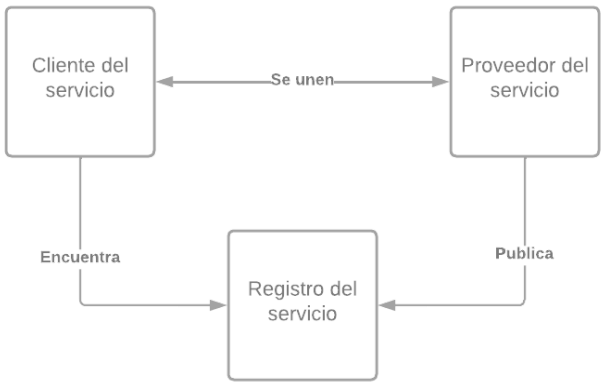


Figura 1: Arquitectura SOA

ADS: Sistema autónomo descentralizado

Los sistemas autónomos descentralizados (ADS) son sistemas informáticos que se ejecutan mediante algoritmos que no están controlados por una autoridad central. El sistema toma decisiones y administra cada componente y recurso que posee el mismo sistema, tal como lo hace un sistema viviente. El sistema ADS, es un concepto de TI que se basa en el sistema inmune del cuerpo humano y que trata de imitar el

comportamiento inteligente y autónomo de nuestras células. Esto con el fin de que el sistema sea capaz de seguir funcionando a pesar de los posibles fallos que puedan ocurrir durante la ejecución del software. En un sistema ADS se busca que las células del programa sean capaces de seguir cumpliendo con las funciones de alguna célula que "muera" tras algún fallo, consiguiendo con esto una alta tolerancia a fallos. De igual manera, se busca que el sistema identifique cuando una célula ha muerto y sea replicada para que el sistema no se quede sin los componentes y recursos necesarios para su correcto funcionamiento. Para ello un ADS es una entidad autónoma con su propio proceso de toma de decisiones. Tiene un conjunto de reglas que definen su comportamiento.
ADS está diseñado para ser autosuficiente y libre de la necesidad de intervención humana Todos los procesos que ocurren dentro del sistema producen sus propios resultados sin ninguna intervención de otra fuente.

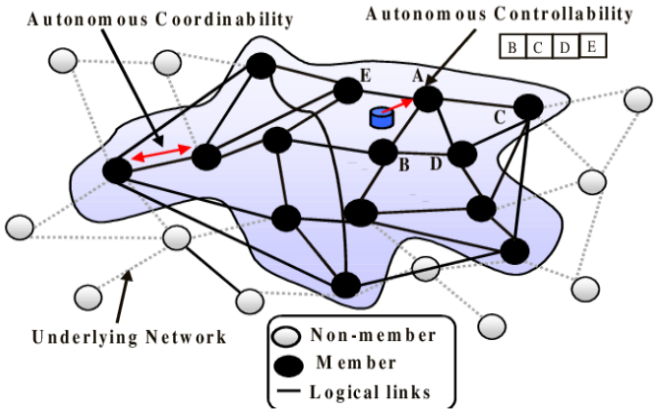


Figura 2: ADS

B. Problemática

ARQUITECTURA ADSOA A CALCULADORA

La arquitectura ADSOA busca combinar los dos conceptos anteriormente introducidos para ofrecer aplicaciones con capacidades empresariales y que presenten las características de ADS como la tolerancia a fallos y alta disponibilidad, dando así la posibilidad de tener sistemas críticos como pueden ser los entidades bancarias sin el riesgo de sufrir pérdidas por fallos del sistema.

Para este trabajo se va a implementar la arquitectura ADSOA a una calculadora.

Para comenzar debemos identificar las partes del código de una calculadora tradicional en arquitectura monolítica. En estas contamos con:

- Interfaz gráfica (números, signos, display)
- Código para obtener la información dada por el usuario
- Código de resolución de las operaciones
- Código para mostrar en la interfaz el resultado
- Código auxiliar para casos especiales (divisiones entre cero, dobles signos, puntos decimales, cadenas vacías)

Modelo Cliente-Nodo-Servidor

Teniendo en cuenta las arquitecturas a utilizar que son la arquitectura orientada a servicios y los conceptos de un sistema autónomo descentralizado, la nueva distribución inicial de la arquitectura general del programa de una calculadora distribuida debería quedar de la siguiente forma: Un modelo cliente-nodo-servidor.
Donde el cliente posea la interfaz de la calculadora que el usuario final usará como cualquier otra aplicación de calculadora y que hará las peticiones de operación al resto de la red y quedará en espera de una respuesta.
El nodo será un punto medio que funcionará como nuestro distribuidor de mensajes, este se conectará a los clientes y a los servidores, será en otras palabras el puente de la red.
El servidor será, por último, el programa encargado de resolver las operaciones y regresar el resultado por el mismo camino de vuelta hasta el cliente.
Es importante mencionar que para tener un sistema distribuido, cada uno de los eslabones de nuestra arquitectura necesitamos que sea un programa independiente, no una clase o una solución en el mismo proyecto, sino ejecutables independientes por sí solos que trabajarán en conjunto.
Para poder comunicar los tres eslabones de la arquitectura se hará uso de hilos y sockets. Los sockets son el medio de comunicación entre cada uno de ellos y los hilos nos ayudarán a poder tener envío de mensajes y recepción paralela sin interrupciones ni atascos entre los mensajes y procesos.

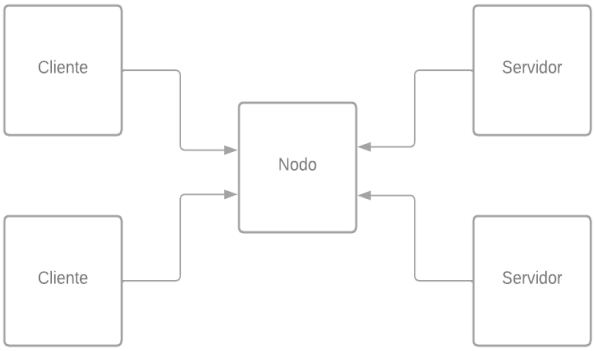


Figura 3: Modelo Cliente-Nodo-Servidor

En la figura 3 se puede observar cómo es que más de una célula, ya sea de cliente o servidor se puede conectar al nodo. Es un requisito necesario, no debe existir la limitación de que sólo una célula se pueda conectar a otra. Para lograr la conexión entre cada una de las células, cada socket creado se genera dentro de un Thread o hilo para ser capaces de tenerlos en paralelo.
En el caso de los servidores fue necesario que identificaran las operaciones de alguna manera, para ello se elaboró un código de contenido aprovechando lo ingresado por el usuario desde el comienzo de su petición. Las peticiones cuentan con tres partes las cuales son: operador1, operacion, operador2. Esto al salir del cliente en un sólo bloque de mensaje se concatena como un string que al llegar al servidor es necesario dividir mediante una función llamada ParseaOperación. Una vez identificada la operación y realizada, el servidor genera un nuevo mensaje con un código de contenido de cinco partes: operador1, operacion, operador2, igual, resultado. En la función ParseaOperación del servidor existe un if que discrimina entre los códigos de contenido de respuestas y peticiones para así ignorar los que son respuestas y sólo procesar las peticiones. De esta manera se puede evitar la redundancia y se garantiza que un mensaje no se cicle en la red generando tráfico indeseado que pueda afectar el desempeño del sistema o generar errores posteriormente conforme la demanda aumente y el sistema se haga más robusto y se vea en la necesidad de escalar en sus servicios.

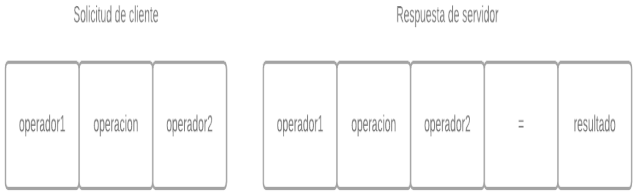


Figura 4: Prototipo de códigos de contenido

En esta primera instancia del desarrollo, el envío de mensajes se hará broadcast, esto implica que todas las células conectadas a la red recibirán todos los mensajes. Un ejemplo de esto es que la calculadora número 1 que realizó una petición de suma recibirá dos respuestas, una por cada servidor, sin embargo igual lo hará la calculadora número 2 aunque no haya realizado ninguna petición.

Discriminación de mensajes

El siguiente problema a resolver es cómo hacer que el sistema sepa qué mensajes son de que cliente y cuáles no, puesto que en una aplicación funcional y en productivo no es posible que uno de los cliente muestre los resultados de peticiones que él jamás pidió, eso estaría quebrantando la confiabilidad y veracidad del sistema.
Para ello se implementó un método de identificación por medio de UUIDS, los cuales son números aleatorios generados a partir de una biblioteca del mismo nombre. Estos identificadores serían estáticos durante todo el proceso de envío (ida y vuelta). Se colocarían al inicio del código de contenido como un bloque más y serían parte del mensaje. Sin embargo para que el cliente conociera que mensajes eran suyos y cuáles no, este debe almacenar todos los uuids que genera al enviar las peticiones en un hashmap con el cual coteja al recibir un mensaje entrante para saber si ese es respuesta de una petición suya o no y procesar las adecuadas

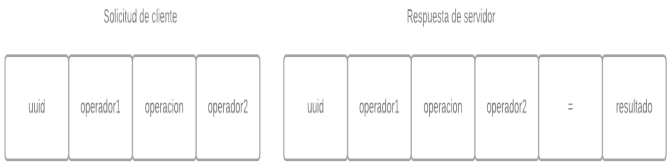


Figura 5: segundo prototipo de códigos de contenido.

Nodos en malla

En la arquitectura cliente-nodo-servidor, es evidente que en caso de que el nodo caiga toda la red fallará y dejará de servir. En ese caso, uno de los propósitos principales de esta implementación estaría siendo ignorado por completo: generar una aplicación con alta tolerancia a fallos y disponibilidad del sistema. Es por ello que es necesario interconectar todos los nodos existentes entre sí para que en el supuesto de que alguno de ellos falle, los demás sean capaces de seguir soportando el sistema.
Existe una aclaración, es evidente que todas aquellas células como servidores o clientes que estén conectadas a los nodos que caigan, caerán con ellos. Sin una conexión al punto de comunicación de toda la red, esas células quedan incomunicadas de toda la red, siendo capaces de interactuar (enviar o recibir mensajes) con otros nodos, clientes o servidores. Es por ello que en un sistema distribuido como este se busca que existan células de extremo como lo son clientes y servidores, conectadas a cada nodo existente, para así garantizar el continuo funcionamiento del sistema.

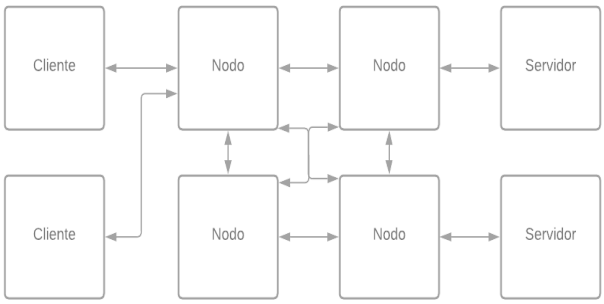


Figura 6: Nodos en malla

Para la interconexión de nodos fue necesario dotar al nodo de la más mínima memoria, en este caso por medio de la función main y sus parámetros de entrada, por los cuales el nodo recibirá cierta información como son los puertos a los que puede intentar conectarse. La conexión a estos puertos no necesariamente será exitosa, en algunos casos fallará, pero ese fallo se usará a favor del proceso de conexión en malla y se explicará a detalle más adelante.
Los nodos tendrán que recibir como parámetros “n” cantidad de puertos a los cuales podrán y deberán intentar conectarse sin importar si la conexión tiene éxito o falla . Una limitante de esta solución es que en la red generada podrá únicamente haber tantos nodos como puertos se les asignen como parámetros de entrada o como lista de parámetros. Es importante aclarar que la existencia de los nodos no limita la existencia de otro tipo de células como son los servidores y clientes, de ese tipo puede haber la cantidad que el usuario requiera debido a que un solo nodo puede soportar “n” cantidad de células extremo es decir, servidores y clientes.

La lógica de los nodos será la siguiente: Al iniciar irán al primer puerto en la lista de parámetros e intentarán conectarse, en caso de lograr establecer una conexión en ese puerto, darán por terminado ese proceso y continuarán con el siguiente que consiste en lanzar un hilo para permanecer escuchando/en espera, para en caso de que alguien más se quiera conectar a él, ya sean células extremo (clientes o servidores) u otros nodos.
En caso de no lograr una conexión exitosa al intentarlo con el primero de los puertos en la lista de puertos que se les asignó por parámetros, asumirá que existe un nodo ya conectado en ese puerto y procederá a intentar conectarse en el siguiente puerto. Así lo hará hasta que encuentre un puerto en el que la conexión sea exitosa y pueda pasar al siguiente paso. Ya que haya podido realizar la conexión y lanzado el socket necesario, tomará todos aquellos puertos en los que no pudo conectarse y lanzará la petición de conexión a ellos mediante hilos. Así los nodos que estaban ya conectados y en modo de espera/escucha aceptarán los sockets y se generarán conexiones entre todos. Una vez cada nodo ejecutado y levantado termine este proceso de conexión general, entonces todos entraran en modo de escucha/espera, en caso de que alguna otra célula quiera conectar a él, ya sean células extremo (clientes o servidores) u otros nodos. En caso de ser nodos, repetirá el proceso.

De esta forma cuantos sean los nodos que se establezcan necesarios para el sistema se interconectan sin mayor problema y se enviarán entre sí todo los mensajes provenientes de los clientes y servidores cumpliendo con la comunicación broadcast. Sin embargo debido a la misma comunicación broadcast que consiste en enviar el mensaje a todos los dispositivos finales sin la necesidad de conocer sus direcciones ni mayor información se produce un

problema, Al tener los nodos en malla enviando mensajes sin discriminar, estos mensajes se envían por la red infinitamente en bucle sin fin dejando al sistema inservible.

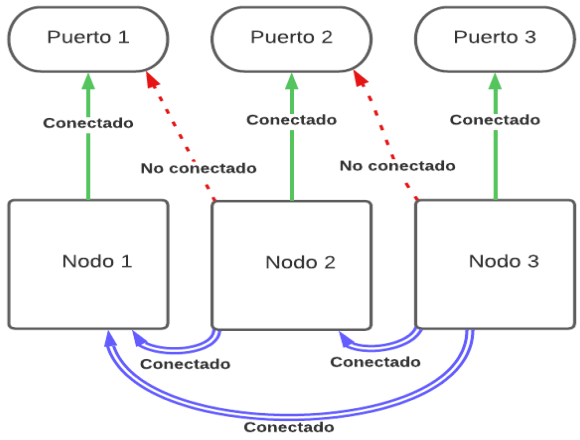


Figura 7: Lógica de conexión entre nodos

Mensajes ciclados

Cuando se interconectan los nodos en malla ocurre un problema que deja inservible al sistema.
Pongamos un ejemplo en una malla pequeña de una red de tan sólo dos nodos, al ser los nodos los encargados del envío de los mensajes por toda la red, supongamos que el nodo uno recibe una petición de operación de uno de sus clientes conectados, el cliente uno le envía la petición de operación al nodo uno y este la recibe, posteriormente envía el mensaje a toda la red, es decir lo envía de nuevo al cliente uno y al resto de clientes que tenga conectados, en este caso el cliente dos, el cliente tres y el cliente cuatro también reciben el mensaje de petición del cliente uno, pero al estos no comprenderlo no lo procesan y no sucede nada con ese mensaje, sin embargo el nodo uno también le envía el mensaje de petición al nodo 2 el cual lo recibe y lo envía a su vez a los servidores que tiene conectados.
Supongamos que el nodo dos envía el mensaje al servidor uno y al servidor dos. Estos recibirán el mensaje, lo procesarán y enviarán al nodo dos un mensaje de respuesta de vuelta. Sin embargo antes de eso regresemos al nodo dos, este aparte de enviarle el mensaje de petición al servidor uno y al servidor dos también se los envió de vuelta al nodo uno. este lo recibe y lo envía de nuevo al cliente uno y al resto de clientes que tenga conectados y no sólo eso sino que de igual forma al nodo dos. Si a este ciclo le agregamos los mensajes de respuesta provenientes de los servidores uno y dos respectivamente, terminamos con que en la red habrá un total de tres mensajes ciclados infinitamente entre todas las células de la red.

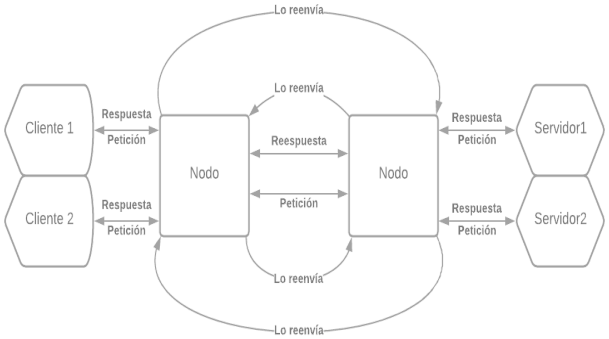


Figura 8: Ciclado de mensajes

Para solucionar dicho problema se buscó trabajar con el código de contenido. Si al nodo se le otorgaba un poco más de memoria de la que tenía entonces podría recordar que mensajes ya habría recibido y cuáles no. De esta forma el nodo podría discriminar los mensajes que ya hubieran pasado por él y detener el envío descontrolado, pero había dos posibles salidas a este razonamiento:

1. El nodo reconocía el mensaje por medio del uuid el cual es estático, por lo cual, al venir de regreso del servidor el mensaje ya como respuesta, el nodo reconocería ese uuid como ya recibido y no realizará el envío a toda la red.
2. El uuid no es estático, cambia dependiendo de si es una petición o una respuesta. Por lo tanto el nodo no lo reconocería como ya que ya lo había recibido y entonces lo enviaría a toda la red sin contratiempos. Sin embargo los clientes ya no serán capaces de distinguir si ellos fueron los remitentes de las peticiones y las respuestas serían recibidas e interpretadas por todos los clientes de la red.

Para la solución se decidió combinar los dos casos, así que se optó por modificar una vez más el código de contenido, esta vez agregando un uuid dinámico que conservaría ese nombre y agregando un uuid estático llamado id. Este id sería de interés especialmente para

las células extremo (clientes y servidores), mientras que el uuid que cambiaría dependiendo del sentido del mensaje en la red, sería de interés para los nodos.

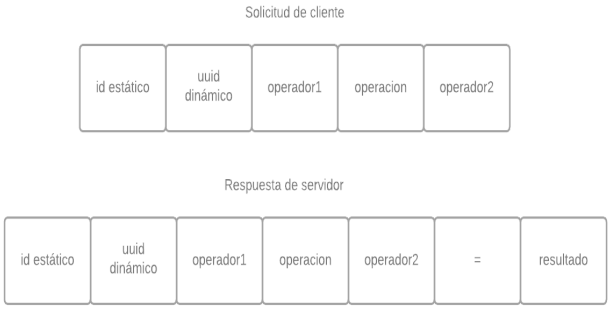


Figura 9: Modelo final de códigos de contenido

Para que esta solución funcionase fue necesario modificar una parte del código de los nodos. Los nodos hasta este punto eran bastante tontos, lo único "inteligente" que hacían era recordar los puertos a los que no tuvieron éxito al intentar conectarse, para después establecer conexiones con los nodos que ya estaban conectados ahí

A partir de aquí tienen un poco más de memoria puesto que cuentan con un hashmap en el cual van a almacenar todos los uuids de los mensajes que van a recibir, sin importar si los reciben de un cliente, de un servidor o de otro nodo, puesto los nodos no saben identificar con quienes están conectados. Este proceso de identificación lo realizan con el propósito de formar una colección de uuids de mensajes que los ayude a poder discriminar aquellos mensajes que necesitan envíar de aquellos que no necesitan envíar a la red. De esta forma nos aseguramos que los nodos distribuyan los mensajes de manera correcta y completa sin caer en el error del ciclo de mensajes que anteriormente explicamos.

Por otro lado, nos aseguramos que los clientes sean capaces de continuar diferenciando las respuestas de sus peticiones de las que ellos no solicitaron, garantizando la seguridad y confidencialidad de las peticiones de los diferentes usuarios de la aplicación de la calculadora. Así como los servidores, de igual forma sean capaces de distinguir qué peticiones ya han respondido y cuales les falta por responder.

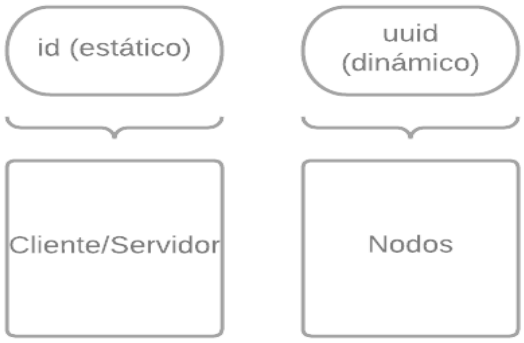


Figura 10: Identificadores utilizados por cada célula

Microservicios

Para una aplicación de alta criticidad como lo sería aquella en la que se tenga que implementar una arquitectura ADSOA, es indispensable utilizar todos aquellos conceptos que impulsen y sean una ventaja para el manejo del código en especial frente a errores o mantenimiento. Por ello al analizar el estado actual de nuestro sistema distribuido, nos podemos dar cuenta que el código perteneciente a la parte del programa del servidor es lo más parecido a una arquitectura monolítica. En una arquitectura de este tipo todos los procesos están estrechamente ligados y se ejecutan como un solo servicio. Esto significa que si un proceso de la aplicación experimenta un pico en la demanda, es necesario escalar toda la arquitectura antes de que éste falle.

Agregar, mejorar o modificar las características de una aplicación monolítica se vuelve mucho más complejo y caro a medida que crece el código fuente. Esta complejidad limita la experimentación y dificulta la implementación de nuevas propuestas al código. Para temas específicos de nuestra=o sistema distribuído y sus fines es de nuestro interés el hecho de que las arquitecturas monolíticas aumentan el riesgo de la pérdida de disponibilidad en la aplicación ya que debido a la gran cantidad de procesos dependientes y estrechamente vinculados, el impacto de un error en algún proceso se ve significativamente aumentado.

Debido a estas desventajas se recomienda utilizar algunas otras maneras o arquitecturas que no comprometan los recursos, disponibilidad, mantenimiento y mejora de las aplicaciones.

Es por eso que para nuestro caso es mucho más

conveniente implementar una arquitectura de microservicios debido a que en estas arquitectura de microservicios, el cliente puede solicitar cualquiera de las funcionalidades de la aplicación, sin embargo estas funcionalidades no van a estar compiladas dentro de un sólo “paquete de código”, sino que cada una de las características o servicios existirán en códigos independientes que serán solicitados dependiendo de las solicitudes de parte del cliente. De esta manera se puede trabajar en cada una de las funcionalidades sin la necesidad de alterar el código “principal”.

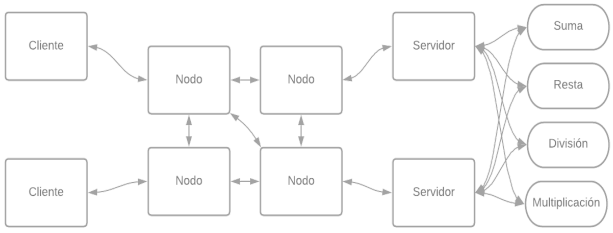


Figura 11: Modelo de microservicios

Para los microservicios implementados en la arquitectura de la calculadora decidimos dividir el código del servidor, que recordamos es el encargado de recibir las peticiones de los clientes y resolver las operaciones para posteriormente enviar las respuestas, y emular el comportamiento del mismo programa “madre”. Se generaron programas de “servidores dedicados” a cada operación. Dichos programas comparten la estructura, método y proceso de conexión. Sin embargo los microservicios por operación se diferencian en el comportamiento que tendrán al recibir mensajes con ciertos códigos de contenido. Como pudimos ver en el modelo de los códigos de contenido, en uno de los bloques, ambos modelos, tanto los de respuesta como los de petición cuentan con un bloque de operación, Dicho bloque de operaciones no es mas que el signo de la operación (+, -, x, /). Al recibir el mensaje y realizar la división en los diferentes bloques del código de contenido necesarios para que el código los interprete, los microservicios son capaces de leer y entender dicho bloque de operación. Dependiendo de dicha información podrán distinguir si su ejecución es necesaria o no.

Pongamos un ejemplo, si un mensaje con un código de contenido [id uuid 8 + 7] llegase a un microservicio que contenga las instrucciones para realizar una suma, este microservicio será capaz de identificar que dicho mensaje requiere su ejecución. Si el mismo mensaje llegase a un microservicio de multiplicación que solamente contiene las instrucciones de dicha operación, lo recibiría, leería el código de contenido y sabría que ese mensaje no es para él, así que lo ignorará y no se ejecutará

Auto Reparación: Clonación de células

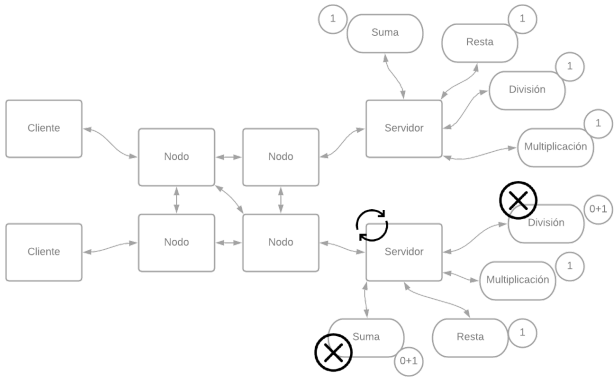
Se le denomina células del sistema a aquellos programas que están ejecutándose y corriendo los microservicios. Con la finalidad de que el sistema permanezca estable y disponible en cualquier momento a pesar de posibles caídas es que se busca que dichas células se clonen de ser necesario.

Esta idea proviene de querer imitar el comportamiento del cuerpo humano en el cual las células por sí solas saben cuándo es necesario que se multipliquen con las funcionalidades previamente existentes para que el sistema, es decir el cuerpo, se mantenga funcionando. Toda la arquitecta ADSOA se basa en elementos y conceptos de la naturaleza biológica. Se busca que el sistema sea un organismo vivo autorreparable.

En el caso de nuestro sistema se está implementando por medio de acuses de recibo. Cada cierto tiempo el nodo revisa que células existen en sus conexiones ya que estas le dan muestras de vida. En caso de notar que alguna de las células necesarias establecidas para el correcto funcionamiento del sistema ha caído, el nodo se encarga de identificar que tipo de célula se ha ido y por ende cuál es la que se requiere y con qué funcionamiento y se manda a ejecutar una de dichas características inmediatamente. Este proceso se podría dar casi en automático sin embargo se decidió dar un breve espacio de tiempo entre cada chequeo de acuses para no saturar el sistema.

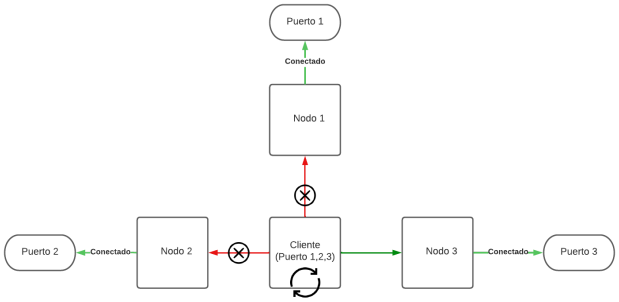
En la configuración del sistema se puede establecer a voluntad el tipo de requerimientos que necesite para funcionar correctamente, por ejemplo para ser capaz de responder solicitudes de suma se puede establecer, debido a la baja dificultad de la operación, la necesidad de dos células que ejecuten el microservicio, de no ser así será necesario que dichas células se clonen hasta llegar a las características deseadas de ejecución. Cada requisito por microservicios es independiente, así que es adaptable para cada grado de dificultad o demanda de la actividad o ejecución.

En el escenario de la clonación de células se puede dar el caso de un cáncer, el cual consiste en que las células solicitadas a clonación no sepan cuándo detenerse y lleguen a saturar el sistema, consumiendo sus recursos y en un final haciendo que toda la red caiga, dejando todo el desarrollo sin su propósito principal que es la disponibilidad.
Este es un problema primordial a tratar, el cual se le dio solución en este caso de la siguiente manera: se optó porque los que ejecuten la orden de clonación sean los nodos al igual que son ellos los que poseen la configuración de ejecución necesaria para el sistema. Por ende se necesito dotar a los nodos de cierta habilidad de discriminación, para que pueda conocer que tipo de celulas son las que están conectándose a él. De esta manera el nodo es capaz de llevar un conteo de las células con los microservicios necesarios. Dicho conteo no afecta en lo absoluto al funcionamiento habitual del sistema, únicamente facilita el tema de control del cáncer.



Reconexión a nodo:

Siguiendo con la idea de mantener la disponibilidad del sistema, cabe aclarar que hasta el momento en el supuesto caso de caída de un nodo, todo aquello conectado a este caería con él. Para los microservicios esto no implicaba gran pérdida, más con el sistema de clonación ya implementado. Sin embargo, si hablamos de un cliente, tenemos un problema grave. No se puede dejar al cliente sin acceso al sistema por más que el nodo principal al cual están conectado haya caído. Por ello es necesario implementar un protocolo que permita que los clientes que se queden sin nodo sean capaces de buscar algún otro nodo existente de la red y conectarse de vuelta a este para continuar con la capacidad de conexión y ejecución de las funcionalidades del sistema.
Para este caso se aprovechó la arquitectura pensada para la conexión en malla de los nodos. Como ya se explicó anteriormente, los nodos por naturaleza deben conocer al inicio como parámetros todos aquellos puertos a los que pueden establecer una conexión y realizar su protocolo de conexión por barrido.
Para los clientes se utilizó a grandes rasgos la misma idea, se introdujo el fragmento de la conexión en un bucle que en caso de caer regrese a la etapa de conexión pero esta vez con el puerto del nodo siguiente en la red. Como se está imitando el comportamiento de los nodos, en este caso es necesario dar como parámetros a los clientes al momento inicial de la ejecución los puertos a los cuales los nodos están conectados. Esta lista de puertos es exactamente la misma que la de los nodos. Actualmente este proceso se hace manual, pero podría automatizarse para que los nodos envíen sus listas de parámetros a los clientes conforme se conectan y estos la guarden.



Inyección de microservicios

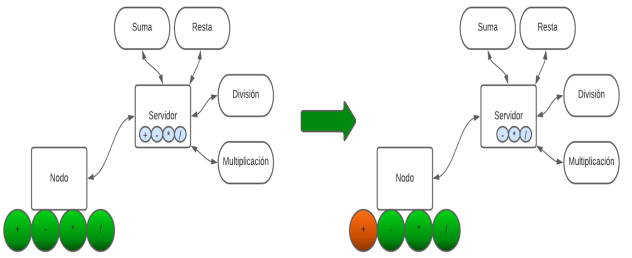
Como punto final del desarrollo se implementó un sistema básico para que el usuario sea capaz de insertar o retirar los microservicios deseados al sistema. Este sistema se añadió como un apartado extra a la misma interfaz del nodo, en el cual existe un botón por cada microservicio el cual cambia el estado de una simple bandera que habilita o inhabilita la ejecución de los microservicios. El nodo continuará realizando la verificación de los estados y acuses sin embargo de estas deshabilitado el microservicio no se clonará ninguna células con ese microservicio en específico.

Un punto a resaltar es que aunque el microservicio se retire del sistema, las células existentes anteriores a dicha acción continuarán existiendo y en pleno funcionamiento hasta que se suspendan manualmente o por fallos distintos.

Este agregado podría prestarse para eventos y acciones de mantenimiento, escalabilidad, mejora o chequeo. Por si solo, como ya hemos visto, la arquitectura orientada a microservicios presenta enormes ventajas respecto a otras arquitecturas como la monolítica o centralizada debido a que de esta manera es mucho más fácil poder hacer cambios, mejoras, añadir y hasta experimentar con el código sin poner en riesgo el sistema completo. Con esto en mente podemos considerar que el hecho de poder retirar el microservicio del sistema estando en productivo sin necesariamente retirar la funcionalidad del sistema por completo brinda aún un abanico mucho más amplio de posibilidades a la hora de actividades con el código.

Una vez que el microservicio sea alterado, mejorado o después de su mantenimiento este regresará al sistema y será cargado sin mayores problemas. Las células preexistentes habrán cubierto hasta donde pudiesen la demanda del servicio. Una vez aquí se puede buscar una solución para dichas células, para no tener que matar la ejecución manualmente. Sin embargo a partir de la nueva carga el resto de células nuevas contarán con las funcionalidades y características nuevas y seguirán respondiendo al resto de protocolos del sistema.

Bajo esta misma idea la implementación de nuevas operaciones como factoriales, raíces, porcentajes, logaritmos, etc, se vuelven posibilidades mucho más factibles y sencillas de implementar.



II. Conclusiones

La importancia del software hoy en día es innegable. En la actualidad los sistemas informáticos controlan más de nuestras vidas de lo que muchas veces imaginamos. Como ingenieros o desarrolladores es nuestro deber ético estar bien capacitados para saber distinguir cuales de esos sistemas son críticos y necesitan de ciertas características como las que hemos mencionado en este trabajo, tolerancia a fallos, disponibilidad, confiabilidad, etc.

Si bien una arquitectura como la implementada en el desarrollo de la calculadora distribuida (ADSOA) brinda una cantidad inmensa de ventajas frente a otras, su implementación en definitiva no es apta para todo tipo de proyectos, existen aquellos desarrollos que debido a su bajo nivel de criticidad, a su complejidad, o por el gasto que la implementación de una arquitectura de este tipo supondría, es mejor optar por otras opciones que son igualmente válidas y funcionales.

Como pudimos ver a lo largo del desarrollo del proyecto, la implementación de ADSOA no es para nada sencilla, este es un factor crucial que los ingenieros deben tomar en cuenta a la hora de valorar su implementación en un sistema, sobre todo al ser sistemas más elaborados que una calculadora.

III. Referencias

C. Perez-Leguizamo and J. S. G. Godinez-Borja, "Autonomous Decentralized Service Oriented Architecture: Concept, Technologies and Application," 2017 IEEE 13th International Symposium on Autonomous Decentralized System (ISADS), 2017, pp. 47-54, doi: 10.1109/ISADS.2017.27.

AWS. "¿Qué son los microservicios? | AWS". Amazon Web Services, Inc. https://aws.amazon.com/es/microservices/#:~:text=Los%20microservicios%20son%20un%20enfoque,servicios%20son%20equipos%20pequeños%20independientes. (accedido el 17 de octubre de 2022).