Algoritmos de coordinación en sistemas distribuidos

Cristian Aguirre Janampa
Universidad Nacional de Ingeniería
Lima, Perú
caguirrej@uni.pe

Daniel Sarmiento Bolimbo
Universidad Nacional de Ingeniería
Lima, Perú
daniel.sarmiento.b@uni.pe

Espinoza Vicuña Carlos
Universidad Nacional de Ingeniería
Lima, Perú
cespinozav@uni.pe

***Resumen*—En el día de hoy donde es común los sistemas distribuidos, es necesario poder usarlos de una manera óptima para obtener el máximo rendimiento posible de una aplicación dada. En este proyecto tratamos de mostrar cómo es que los procesos trabajan en un sistema distribuido, particularmente cómo es que se comunican para ser procesados de manera eficiente en el sistema distribuido.**
Daremos una breve explicación de estos conceptos importantes para su correcto entendimiento de los algoritmos y ejemplos(codificaciónn) que se detallarán más adelante.
Los resultados que se obtendrán nos darán de un registro de tiempos para poder comparar la efectividad de cada algoritmo de coordinación para una tarea(número de procesos a sincronizar) dada y así darnos una idea de cual escoger.

Índice de Términos— procesos, sección crítica, líder, sincronizadores.

I. Artículos científicos relevantes

- Linial, Nathan (1992), "Locality in distributed graph algorithms", SIAM Journal on Computing https://www.cs.huji.ac.il/~nati/PAPERS/locality_dist_graph_algs.pdf.
- James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson http://verdi.uwplse.org/verdi.pdf

II. Propuesta de proyecto a realizar

- Estimar los tiempos de ejecucuión de los algoritmos para compararlos y entender e explicar el funcionamiento y comportamiento de procesos en sistemas distribuidos.

III. Objetivos

- Explicar el funcionamiento de los algoritmos distribuidos.
- Explicar como los porcesos en los sistemas distribuidos acceden a los recursos compartidos.
- Explicar el comportamiento de los procesos en sistemas distribuidos al interacturar con otros porcesos(parar,cambiar,abortar).

IV. Introducción

Los sistemas distribuidos han sido testigos de un crecimiento fenomenal en los últimos años. El costo decreciente del hardware, los avances en la tecnología de la comunicación, el crecimiento explosivo de Internet y nuestra dependencia cada vez mayor de las redes para una amplia gama de aplicaciones que van desde la comunicación social hasta las transacciones financieras han contribuido a este crecimiento. Los avances en sistemas embebidos, nanotecnología y comunicación inalámbrica han abierto nuevas fronteras de aplicaciones como redes de sensores y computadoras portátiles. El rápido crecimiento de la computación en la nube y la creciente importancia de los grandes datos han cambiado el panorama de la computación distribuida.

En este tema explicaremos algunos Algoritmos de coordinación en sistemas distribuidos y su construcción, notaremos que la asincronía es difícil de manejar en las aplicaciones de la vida real debido a la falta de garantías temporales: es más sencillo escribir algoritmos en el modelo de proceso sincrónico (donde los procesos ejecutan acciones en sincronía sincronizada) y más fácil demostrar su corrección.

V. Estado del Arte

V-A. Elección de Líder

La existencia de unn líder es importante en aplicaciones distribuidas. El líder (controlador) cotrola y gestiona todo el sistema. Por ejemplo, un administrador de base de datos centralizado es considerado como líder: los procesos del cliente en el sistema pueden acceder o actualizar los datos compartidos. Este administrador mantiene una cola de lecturas pendientes y escribe y procesa estas solicitudes en un orden apropiado. Si el líder falla o se vuelve inalcanzable, se elige un nuevo líder entre los procesos no defectuosos. Las fallas afectan la topología del sistema, incluso no se descarta la división del sistema en varios subsistemas disjuntos.

Para escoger un líder, se usan algoritmos de exclusión mutua. Los procesos que ingresan a la denominada sección crítica, tienen la posibilidad a ser escogidos como líder. Es necesarrio tomar en cuenta los siguientes puntos:

1. El fallo no es una parte inherente de los algoritmos de exclusión mutua. De hecho, el fallo dentro de la sección crítica generalmente se descarta.
2. No es necesario que los procesos cambiar de líder. El sistema puede funcionar exitosamente por un período indefinido con su líder original, siempre que no haya fallas.
3. Si la elección del líder se ve desde la perspectiva de la exclusión mutua, entonces la salida de la sección crítica es innecesaria. Por otro lado, el líder debe informar a cada proceso activo sobre su identidad.

Haciendo una explicación Formal. Sea $G = (V, E)$ la topología del sistema. Los procesos $i \in V$ tiene un identificador único. Cada proceso i tiene una variable $L(i)$ que representa el identificador de su líder. Además, $ok(i)$ denota que el proceso i no es defectuoso. Entonces, se cumple que:

1. $\forall i, j \in V : ok(i) \wedge ok(j) => L(i) = L(j)$
2. $L(i) \in V$
3. $ok(L(i)) = true$

V-A1. Bully Algorithm: Funciona en una red de procesos completamente conectada. Se supone que:

1. Los enlaces de comunicación están libres de fallas.
2. Los procesos solo pueden fallar deteniéndose.
3. Las fallas pueden detectarse correctamente utilizando algún mecanismo como el tiempo de espera.

Una vez que se detecta una falla del líder actual, este algoritmo permite que el proceso no defectuoso con la identificación más grande finalmente se elija como líder.

El algoritmo utiliza tres tipos diferentes de mensajes: elección, respuesta y líder. el procedimiento es el siguiente:

1. **Paso 1:** Cualquier proceso, después de detectar el fracaso del líder, hace una oferta para ser el nuevo líder enviando un *mensaje de elección* a cada proceso con un identificador más alto.
2. **Paso 2:** Si algún proceso con una identificación más alta responde con un *mensaje de respuesta*, el proceso de solicitud renuncia a su intento de convertirse en el líder. posteriormente, espera recibir un *mensaje del líder* (Yo soy el líder) de algún proceso con un identificador más alto.
3. **Paso 3:** Si ningún proceso de mayor número responde al mensaje de elección enviado por el nodo i dentro de un período de tiempo de espera, el nodo i se elige a sí mismo como líder y envía un mensaje de líder a cada proceso en el sistema.
4. **Paso 4:** Si el proceso i no recibe ningún *mensaje del líder* dentro de un período de tiempo de espera después de recibir una respuesta a su mensaje electoral, entonces el proceso sospecha que el ganador de la elección falló mientras tanto y reinicia la elección.

V-A2. Algoritmo Chang–Roberts: Es un algoritmo de elección de líder para un anillo unidireccional. Suponga que un proceso puede tener uno de dos colores: rojo o negro. Inicialmente, cada proceso es rojo, lo que implica que cada proceso es un candidato potencial para ser el líder. Entonces, el algoritmo funciona de la siguiente manera: Un token iniciado por un proceso rojo j se eliminará cuando lo reciba un proceso i > j. Entonces, en última instancia, el token de un proceso con la identificación más grande prevalecerá y volverá al iniciador. Por lo tanto, el proceso con la identificación más grande se elige como líder. Se requerirá otra ronda de mensajes del líder para informar la identidad del líder a cualquier otro proceso. Para analizar la complejidad del algoritmo, ver la Figura 1. Suponga que cada proceso es un iniciador, y sus fichas se envían en sentido antihorario alrededor del anillo. Antes de que el token del proceso (n - 1) llegue al siguiente proceso (n - 2), los tokens de cualquier otro proceso alcanzan el nodo (n - 1) en el siguiente orden: Token $< 0 >$ alcanza (n - 1), token $< 1 >$ alcanza (n - 1), el token $< 2 >$ alcanza (n - 1) y, finalmente, el token $< n-2 >$ alcanza (n - 1), y todos estos tokens se eliminan. La complejidad del mensaje en el peor de los casos es, por lo tanto, $n(n-1)/2$. El algoritmo puede extenderse naturalmente a una topología gráfica arbitraria para la cual existe un ciclo hamiltoniano.

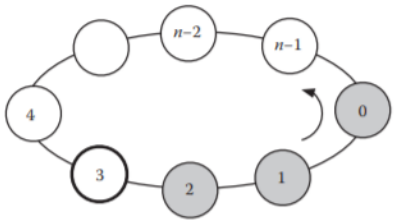

Figura 1. Algoritmo Chang-Robert: El token de el proceso 3 alcanzó el proceso (n - 1) y los procesos 2 y 1 se volvieron negros.

V-A3. Algoritmo de Franklin: Funciona en un anillo que permite la comunicación bidireccional. El objetivo es asignar liderazgo al proceso con el mayor pid. Para hacerlo, cada proceso envía su propio pid a ambos vecinos, recibe los pids de su vecino izquierdo y derecho, y los almacena en los registros $r1$ y $r2$, respectivamente (transiciones t1,..., T4). Si un proceso es un máximo local, es decir, $r1 < id$ y $r2 < id$, todavía está en la carrera por el liderazgo y permanece en estado activo. De lo contrario, tiene que tomar t2 o t3 y pasa al estado pasivo. En pasivo, un proceso solo reenviará cualquier pid que reciba y almacenará el mensaje que viene de la izquierda en r (transición t5). Tenga en cuenta que, dentro de la misma ronda, un mensaje puede reenviarse (y almacenarse) mediante varios procesos pasivos consecutivos, hasta que llegue a uno activo. Cuando un proceso activo recibe su propio pid (transición t4), sabe que es el único proceso activo restante. Copia su propio pid en r, que en adelante se refiere al líder. Podemos decir que una

ejecución está aceptando (o finalizando) cuando todos los procesos finalizan en pasivo o encontrado. Ver Figura 2

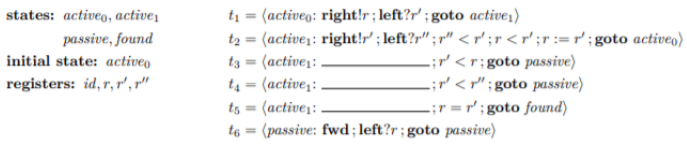

Figura 2. Algoritmo de Franklin: Proceso de Elección de Lider.

V-A4. Algoritmo de Peterson: Este algoritmo funciona en una topología de anillo y opera en ciclos sincrónicos. Curiosamente, elige un líder que usa solo mensajes O ($nlogn$) a pesar de que se ejecuta en un anillo unidireccional. Supongamos que los procesos pueden tener dos colores: rojo o negro. Inicialmente, cada proceso es rojo, potencial líder. Un proceso negro (pasivo), solo actúa como un enrutador y reenvía los mensajes entrantes a su vecino. Suponga que el anillo está orientado en el sentido horario. Cualquier proceso designará a su vecino en sentido antihorario como el predecesor y su vecino en sentido horario como el sucesor.

Se designa el predecesor rojo de i por N (i) y el predecesor rojo de N (i) por NN (i). Hasta que se elija un líder, en cada ronda, cada proceso rojo recibirá dos mensajes: uno de N (i) y el otro de NN (i); estos mensajes contienen alias de los remitentes. Los canales son FIFO. Dependiendo de los valores relativos de los alias, un proceso rojo decide continuar con un nuevo alias para la siguiente ronda o abandona la carrera volviéndose negro. Denote el alias de un proceso i por alias (i). Inicialmente, alias (i) = i. La idea es comparable a la del algoritmo de Franklin, pero a diferencia del algoritmo de Franklin, un proceso no puede recibir un mensaje de ambos vecinos. Entonces, cada proceso determina los máximos locales comparando el alias (N) con su propio alias y alias (NN). Si el alias (N) resulta ser más grande que los otros dos, entonces el proceso continúa su carrera de liderazgo asumiendo el alias (N) como su nuevo alias. De lo contrario, se vuelve negro y se cierra.

V-B. Sincronizadores

Los sincronizadores proporcionan una técnica alternativa para diseñar algoritmos en sistemas distribuidos asincrónicos. Suponga que cada nodo tiene un generador de pulso de reloj y estos relojes funcionan al unísono. Las acciones se programan con estos tics de reloj. La implementación de un sincronizador debe garantizar la condición de que se genere un nuevo pulso en un nodo solo después de que reciba todos los mensajes del algoritmo síncrono, enviados por sus vecinos en el pulso anterior. Sin embargo, la dificultad de proporcionar esta garantía es que ningún nodo sabe qué mensajes le fueron enviados por sus vecinos, y los retrasos en la propagación del mensaje pueden ser arbitrariamente grandes.

V-B1. Sincronizador asíncrono de retardo limitado (ABD): Puede implementarse en una red donde cada proceso tiene un reloj físico y los retrasos de propagación del mensaje tienen un límite superior conocido δ. Se asume que la diferencia entre un par de relojes físicos no cambia.

Inicialización: Sea C un reloj físico de un proceso. Uno o más procesos inician espontáneamente las acciones del sincronizador asignando $C := 0$, ejecutando las acciones para $tick(0)$ y enviando una señal $< start >$ a sus vecinos. Por supuesto, las acciones toman tiempo cero. Cada vecino no iniciador j se despierta cuando recibe la señal $< start >$ de un vecino, inicializa su reloj C a 0 y ejecuta las acciones para la marca 0.

Simulación Previmente un proceso p simule las acciones de $tick(i+1)$, p junto con sus vecinos deben enviar y recibir todos los mensajes correspondientes a $ticki$. Si p envía el mensaje $< start >$ a q, q se despierta y envía un mensaje que es parte de las acciones de inicialización del $tick0$, entonces p lo recibirá en el momento $<= 2\delta$. Por lo tanto, el proceso p comenzará la simulación del siguiente pulso ($marca1$) en el tiempo 2δ. Eventualmente, el proceso p simulará la marca k del algoritmo síncrono a la hora del reloj local $2k\delta$. El permiso para iniciar la simulación de una marca depende completamente del valor del reloj local.

V-B2. Sinconizador α: El sincronizador α, Implementa un paso donde pide a cada proceso que envíe un mensaje $< seguro >$ después de haber enviado y recibido todos los mensajes para el tic del reloj actual. En cada ($ticki$),

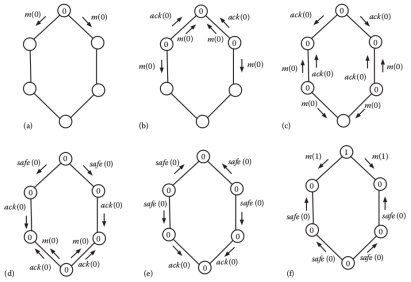

Figura 3. Rastreo parcial: Los números dentro de los círculos indican el número de tick que están simulando. El proceso en la parte superior comienza el cálculo. El mensaje $m(0)$ enviado despierta a sus vecinos.

se sigue estos pasos, ver Figura 3:

- Enviar y recibir mensajes $< m, i >$ para $ticki$.
- Envia $< ack, i >$ por cada mensaje entrante y recibe $< ack, i >$ por cada mensaje saliente para el $ticki$.
- Envia $< safe, i >$ a cada vecino.

Cuando un proceso recibe mensajes $< safe, i >$ de cada vecino para $ticki$, incrementa su tick a $(i+1)$ y comienza la simulación de tick $(i+1)$.

V-B3. Sinconizador β: **Inicialización:** Implica construir un árbol de expansión de la red, cuya raiz es el iniciador designado. El iniciador comienza la simulación enviando a sus hijos un siguiente mensaje que les indica

que inicien la simulación para el $tick0$. **Simulación:** Es similar al sincronizador α, excepto que los mensajes de control (siguiente, seguro y ack) se envían solo a lo largo de los bordes del árbol. Un proceso seguro envía un mensaje $< safe, i >$ a su padre para indicar que todo el subárbol debajo de él es seguro para $tickt$. Si la raíz recibe un mensaje $< safe, i >$ de cada hijo, entonces sabe que cada nodo en el árbol de expansión es seguro para el $ticki$, por lo que envía un siguiente mensaje para comenzar la simulación del siguiente $tick(i+1)$.

V-B4. Sinconizador γ: El sincronizador γ utiliza las mejores características de los sincronizadores β y α. **Inicialización** La red se divide en clústeres de procesos. Cada clúster es un subgrafo que contiene un subconjunto de procesos. El protocolo de sincronizador γ se usa para sincronizar los procesos dentro de cada clúster. **Simulación** Cada clúster identifica a un líder: actúa como la raíz de un árbol de expansión del mismo grupo. Los clústeres vecinos se comunican entre sí a través de un borde designado conocido como borde intercluster (ver Figura 4). Usando el protocolo del sincronizador β, cuando el líder de un clúster descubre que cada proceso en el clúster es seguro, transmite a todos los procesos en el clúster que el clúster es seguro. Los procesos que inciden en los bordes intercluster reenvían esta información a los clústeres vecinos mediante un mensaje $cluster.safe$. Los nodos que reciben el mensaje $cluster.safe$ de los clústeres vecinos transmiten a sus líderes que los clústeres vecinos están a salvo.

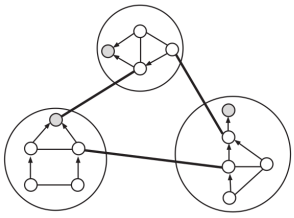

Figura 4. Clústeres en un sincronizador γ: los nodos sombreados son los líderes en los clústeres, y las líneas gruesas son los bordes entre clústeres entre los clústeres.

VI. Diseño del experimento

De la información recolectada se tienen varios algoritmos de coordinación para los cuales podemos simular un ejemplo simple(calcular el tiempo de duración del algoritmo) para comparar resultados.

Vamos a clasificar los algoritmos en 2 clases:

- **Algoritmo de elección del líder**, se comparan 3 algoritmos en una tabla para comparar sus resultados en simulacion para n procesos.
- **Sincronizadores**, se comparan 3 algoritmos (sincronizador alfa, beta y gamma) para comprobar su rendimiento y si es acorde con los datos recolectados de los arttículos.

Algoritmo	N° procesos-1	N° procesos-2	Tiempo
Bully	x	y	t seg
Chang-Robert	x	y	t seg
Franklin	x	y	t seg
Peterson	x	y	t seg

Cuadro I
Algoritmos de elección del líder

Sincronizador	N° procesos-1	N° procesos-2	Tiempo
$\alpha - synchronizer$	x	y	t seg
$\delta - synchronizer$	x	y	t seg
$\gamma - synchronizer$	x	y	t seg

Cuadro II
Algoritmos de sincronización

Los algoritmos serán aplicados en códigos donde calcularemos su tiempo de ejecución.
Tendrá un repositorio donde se subirán las avances hasta tener todos los algoritmos descritos en las tablas descritas arriba.

Link del repositorio :
https://github.com/Cristian-Aguirre/Coordination-Algorithms-for-Distributed-Systems

Referencias

[1] Linial, Nathan (1992), "Locality in distributedgraph algorithms", SIAM Journal on Computing. Disponible en .

[2] James R. Wilcox, Doug Woos, Pavel Panchekha, Za-chary Tatlock, Xi Wang, Michael D. Ernst, and Tho-mas Anderson. Disponible en .

[3] Sukumar Ghosh, Distributed Systems,An Algorithmic Approach, Distributed Systems, Second Edition. Disponible en https://mtjagtap.files.wordpress.com/2018/08/santosh-kumar-distributed-systems-algorithmic-approach-information-812.pdf

[4] Yao Chen, Weiguo Xia, Ming Cao, and Jinhu Lu, Random Asynchronous Iterations in Distributed Coordination Algorithms. Disponible en https://arxiv.org/pdf/1804.10554.pdf

[5] Alex Graves. Adaptive Computation Time for Recurrent Neural Networks (2017). Disponible en https://arxiv.org/abs/1603.08983.