Tarea 10 - Cómputo distribuido

Dalia Camacho, Gabriela Vargas

1 Describir lo visto en clase

1.1 Detectores de fallas

Un detector de fallas es una aplicación responsable de detectar las fallas o *crashes* de los nodos en un sistema distribuido. De igual forma, se puede considerar como un oráculo distribuido que entrega pistas sobre el estatus operacional de cada proceso.

Esta abstracción tiene un papel importante en el deiseño de sistemas con tolerancia a fallas y aplicaciones de manejo de *timeouts*. Los *timeouts* se pueden entender como una especie de tiempos límite es escenarios donde un proceso en una máquina invoca de forma remota una operación de un proceso que corre en otra máquina. Si la computadora que recibe la solicitud de operación falla, una excepción será lanzada cuando el tiempo límite *timeout* termine. Bajo este escenario, es díficil determinar cuánto tiempo hay que esperar, ya que al fijar *timeouts* muy largos, el desempeño del sistema baja, pero se fijan muy cortos entonces se corre el riesgo de generar falsas sospechas de falla (falsos posistivos).

El problema de asginación de *timeouts* corresponde a la capa de abstracción de *failure detector*, la cual funciona de la siguiente manera:

1. Las aplicaciones requieren hacer consultas a los procesos que pertenecen a una red. Por lo que en primer lugar le preguntan al *failure detector* si el proceso al que le quieren hacer una consulta está vivo.

2. El *failure detector* envía una solicitud de información a la red.

3. El *failure detector* determina si el proceso ha fallado dependiendo de si el *timeout* ha expirado y no ha obtenido respuesta.

4. El *failure detector* envía una respuesta de estatus operacional del proceso a la aplicación solicitante.

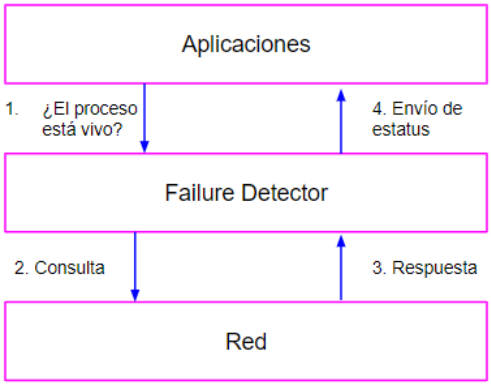

Figure 1: Actividades de un *Failure Detector*

En entornos puramente asíncronos, es imposible disitinguir con 100% de certeza si un proceso en la red realmente ha fallado, ya que existe la posisbilidad de que el proceso esté vivo pero operando muy lento.

Existen diferentes protocolos para tratar el problema de detectores de fallas.

1.1.1 Ping-ack Protocol

-
- El proceso P_i envía un mensaje *Ping* al proceso P_j cada T unidades de tiempo.
- Si P_j no responde en T unidades de tiempo, P_i marca a P_j como sospechoso.
- P_i vuelve a enviar un mensaje a P_j y marcará *timeout* si vuelven a pasar otras T unidades de tiempo sin recibir respuesta de P_j. El tiempo de detección de fallas en este protocolo es de $2T$ unidades de tiempo.

1.1.2 Heart-beating Protocol

- P_j envía permanentemente señales de vida o *heartbeats* al proceso P_i cada T unidades de tiempo
- Si P_i no ha recibido *heartbeats* de P_j en T unidades de tiempo, declarará a P_j como fallido. El tiempo de detección de fallas en este protocolo es de T unidades de tiempo.

Los detectores de fallas deben cumplir dos propiedades básicas: *completenes* y *accuracy*, cada una de las cuales puede tener intensidad débil o fuerte. Asimismo, estas propiedades pueden presentarse de forma perpetua o eventual.

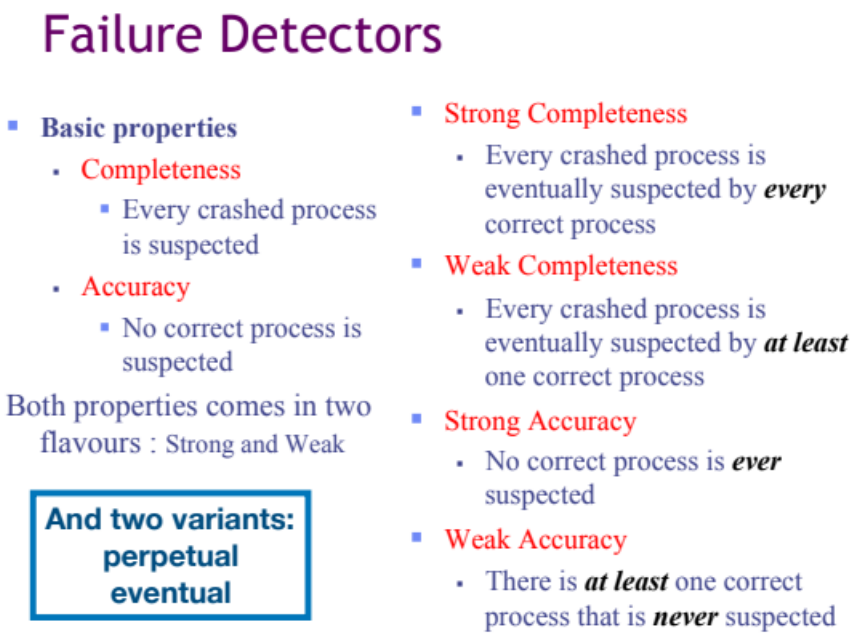

Figure 2: Propiedades de un *Failure Detector*

El que una propiedad fuerte tenga la caracterísa *eventual* significa que existe un tiempo t_0 tal que para todo $t > t_0$, todos los procesos fallidos están en la lista de sospechosos y ningún proceso correcto esta en esta lista.

En la siguiente tabla se observan los diferentes detectores de fallas en función de las combinaciones de las propiedades descritas anteriormente:

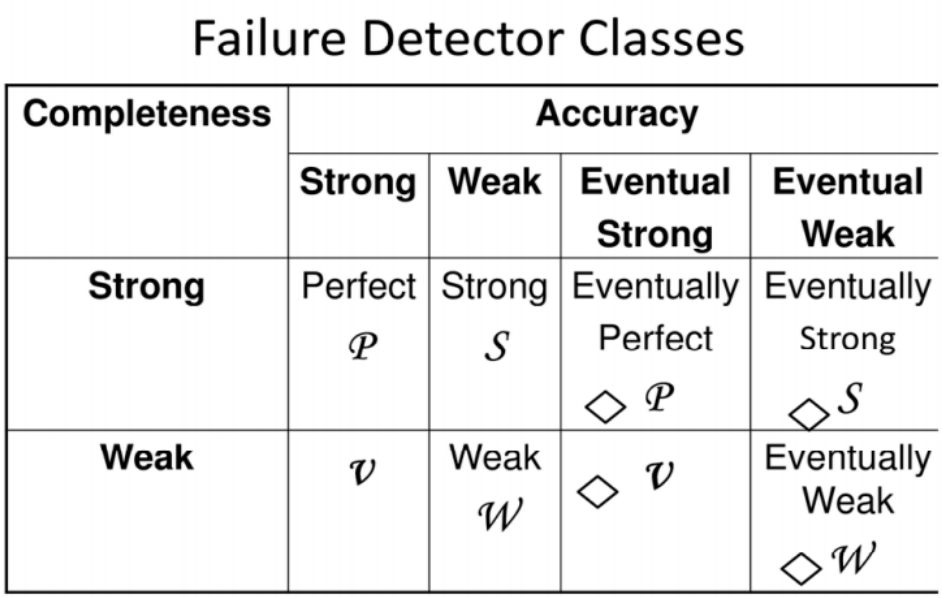

Figure 3: Tipos de *Failure Detector*

Bajo estas condiciones, el problema de consenso entre nodos consiste en encontrar aquellos nodos que verdaderamente han fallado sin sospechar de los nodos que operan correctamente. Considerando la tabla anterior, el resolver el problema para las cuatro clases de failure detector con *Strong completeness* automáticamente resuelve el problema para las cuatro clases restantes.

Existen diferentes algoritmos para resolver el problema de consenso para detección de fallas. Entre ellos se encuentran el *P-based consensus algorithm* y el *Consensus using S*. La idea del primer algoritmo es pasar por una serie de n con un líder en cada ronda. El líder de la ronda será el proceso con el id correspondiente al número de ronda que se esté ejecutando. El líder decide su propuesta y la envía a los demás procesos. El lo nodos que reciben la propuesta esperan a (a) recibir la propuesta del líder y aceptarla o (B) marcar al líder como sospechoso.

2 El acuerdo bizantino

2.1 Objetivo del artículo

El artículo tiene dos objetivos principales el primero es proponer un algoritmo simple de *all-to-all broadcast* para valores binarios, llamado *Double Synchronized Binary Value broadcast* (DSBV). Este algoritmo acepta t procesos bizantinos, donde $t < n/3$. Además se cumple que el conjunto de valores que tienen los procesos correctos no contiene valores que hayan sido propuestos únicamente

por procesos bizantinos. Y si un proceso correcto tiene un único valor v en el conjunto final, entonces todo proceso correcto tiene a v en su conjunto final.

El segundo objetivo, corolario del primero, es definir un algoritmo para consenso binario con fallas bizantinas que hace uso de del DSBV *broadcast* y de una moneda común o *common coin*.

Este algoritmo puede tener t procesos bizantinos donde $t < n/3$, el número de pasos de comunicación en cada ronda es constante, también es constante el número de rondas necesarias para llegar a una decisión. La complejidad de transmisión de mensajes es de $O(n^2)$ mensajes por ronda. Cada mensaje contiene el número de ronda. Los procesos bizantinos pueden reordenar los mensajes. La moneda utilizada es una moneda débil, es decir hay una probabilidad constante que la moneda regrese valores distintos a distintos procesos. Finalmente se asume que no hay un adversario, por lo que no es necesario firmar los mensajes.

2.2 Resultado anterior que se mejora en este artículo

Previamente se creía que en algoritmos de consenso bizantino binario con complejidad cuadrática en el envío de mensajes entonces el número de procesos bizantinos estaba acotado por $t < n/5$ o era necesario usar firmas. Sin embargo este artículo muestra que se puede tener complejidad cuadrática, $t < n/3$ procesos bizantinos y no usar firmas.

El resultado se basa en el algoritmo presentado también por Mostefaoui en 2014, la diferencia es que ese algoritmo necesitaba una moneda común perfecta.

2.3 Resultado 1: Double Synchronized Binary Value Broadcast

El DSBV *broadcast* se construye sobre el *Synchronized Binary Value broadcast* (SBV) que a su vez se construye sobre el *Binary Value broadcast*(BV).

2.3.1 Binary Value broadcast

Las propiedades del BV *broadcast* son las siguientes:

- **BV-Termination:** Toda invocación de broadcast de un proceso correcto termina.
- **BV-Justification:** Si un proceso p_i es correcto y $v \in bin\_values_i$ entonces v fue enviado con un BV-*broadcast* por un proceso correcto.
- **BV-Uniformity:** Si $v \in bin\_values_i$, donde p_i es un proceso correcto, entonces eventualmente $v \in bin\_values_j$ para todo proceso correcto p_j.
- **BV-Obligation:** Eventualmente el conjunto bin_values_i de cada proceso correcto p_i es no vacío.

El algoritmo es el siguiente:

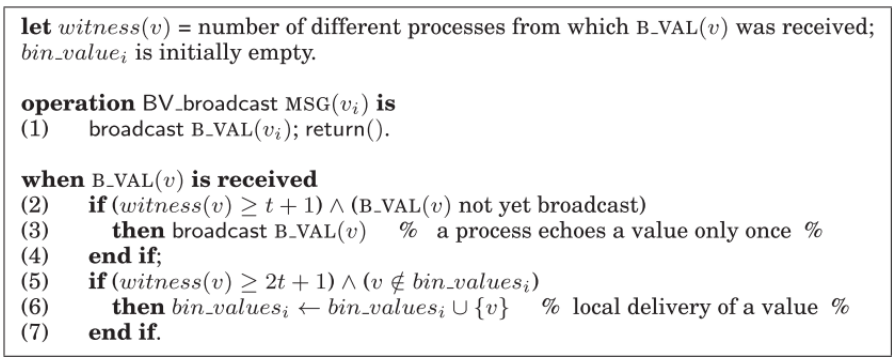

Figure 4: BV Broadcast

Se define una función $witness(v)$ que indica el número de procesos distintos de los cuáles se ha recibido v, e inicialmente bin_values_i es el conjunto vacío para todo proceso i. Se inicia el broadcast, es decir cada proceso manda un mensaje a los demás procesos. Cuando se recibe un valor se actualiza witness, si el valor se ha recibido mas de t veces y no se ha hecho broadcast con ese valor, entonces se hace un broadcast de este valor. Si el valor ha sido recibido por más de $2t$ procesos, entonces este valor se agrega al conjunto bin_values_i.

2.3.2 Synchronized Binary Value broadcast

El SBV broadcast cumple con las siguientes propiedades:

- **SBV-Termination:** La invocación de SBV *broadcast* iniciada por un proceso correcto termina.
- **SBV-Obligation:** El conjunto $view_i$ de un proceso correcto es no vacío.
- **SBV-Justification:** Si p_i es un proceso correcto y $v \in view_i$ entonces un proceso correcto hizo un SBV *broadcast* de v-
- **SBV-Inclusion:** Si p_i y p_j son dos procesos correctos y $view_i = \{v\}$ entonces $v \in view_j$.
- **SBV-Uniformity:** Si todos los procesos correctos hacen SBV *broadcast* de v, entonces $view_i = \{v\}$ para todo proceso correcto p_i.
- **SBV-Singleton:** Si p_i y p_j son dos procesos correctos y $[(view_i = \{v\}) \wedge (view_j = \{w\})] \Rightarrow (v = w)$.

El algoritmo es el siguiente:

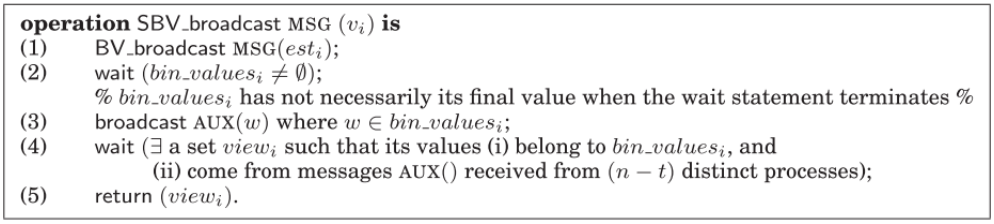

Figure 5: SBV Broadcast

Se inicia el BV *broadcast* visto en el algoritmo de la Figura 4 para generar sincronización se espera hasta que el conjunto bin_values_i es distinto del vacío. Se hace un broadcast de mensajes $AUX(w)$ donde $w \in bin\_values_i$. El proceso espera hasta tener una conjunto $view_i$ donde los valores de este conjunto pertenecen a bin_values_i y estos valores han sido recibidos de $n - t$ procesos distintos en mensajes $AUX()$. Una vez que se tien $view_i$ se regresa el conjunto $view_i$.

2.3.3 Double Synchronized Binary Value broadcast

El DSBV broadcast cumple con las siguientes propiedades:

- **DSBV-Termination:** La invocación de DSBV *broadcast* hecha por un proceso correcto termina.
- **DSBV-Obligation:** El conjunto $view_i$ regresado por un proceso correcto p_i es tal que $1 \leq |view_i| \leq 2$.
- **DSBV-value-Justification:** Si p_i es un proceso correcto y $v \in view_i$ y $v = \perp$ entonces un proceso correcto hizo DSBV *broadcast* de v.
- **DSBV-bottom-Justification:** Si p_i es un proceso correcto y $\perp \in view_i$, entonce tanto a como b han sido propuestos por procesos correctos.
- **DSBV-Inclusion:** Si p_i y p_j son procesos correctos y $view_i = \{v\}$ entonces $v \in view_j$. El valor v puede ser un valor propuesto por un proceso correcto o $\perp$.
- **DSBV-Uniformity:** Si todo proceso correcto hace DSBV *broadcast* de un mismo valor v, entonces $view_i = \{v\}$ para todo p_i.
- **DSBV-Mutex:** Si p_i y p_j son dos procesos correctos y $v \neq \perp$, entonces $view_i = \{\}$ y $view_j = \perp$ son mutuamente excluyentes. Es decir no puede ocurrir simultáneamente.
- **DSBV-Singleton:** Si p_i y p_j son dos procesos correctos y $[(view_i = \{v\}) \wedge (view_j = \{w\})] \Rightarrow (v = w)$.

El algoritmo de DSBV es el siguiente:

```
operation DSBV_broadcast MSG(v_i) is
(1)   view_i[0] ← SBV_broadcast STAGE[0](v_i);
(2)   if (view_i[0] = {v}) then aux_i ← v else aux_i ← ⊥ end if;
(3)   view_i[1] ← SBV_broadcast STAGE[1](aux_i);
(4)   return (view_i[1]).
```

Figure 6: DSBV broadcast

Este algoritmo tiene dos etapas de SBV_broadcast. Se guarda en $view_i[0]$ el resultado de SBV *broadcast* cuando el proceso i hace *broadcast* de algún valor v_i. Si el conjunto $view_i[0] = \{v\}$, es decir que contiene un solo valor entonces $aux_i = v$, en otro caso $aux_i = \perp$. Se hace nuevamente un SBV broadcast, esta vez el proceso p_i emite el valor aux_i. El resultado final del algoritmo es el resultado de esta segunda etapa de broadcast.

2.4 Resultado 2: Acuerdo Bizantino

Las propiedades de consenso son las siguientes:

- **BC-Validity:** El valor decidido fue propuesto por un proceso correcto.
- **BC-Agreement:** No puede ocurrir que dos procesos correctos elijan valores distintos.
- **BC-Decision:** Todo proceso correcto toma una decisión.

Para el acuerdo bizantino de este artículo se utiliza un proceso aleatorio, éste se da haciendo uso de una moneda común. Una moneda común genera una secuencia de bits para cada proceso p_i. La probabilidad de que todos los procesos correctos tengan el valor 0 es $1/d$ y la probabilidad de que no todos tengan el valor 1 también es $1/d$, mientras que la probabilidad de que los procesos correctos tengan valores distintos es $(d-2)/d$. Si $d = 2$, entonces se tiene una moneda perfecta, pero si $d > 2$ entonces se tiene una moneda débil.

El algoritmo propuesto tolera $t < n/3$ procesos bizantinos y es el siguiente:

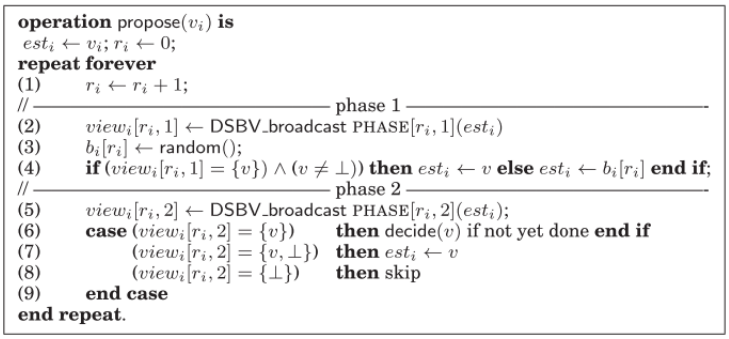

Figure 7: Consenso Bizantino

Se inicializan las variables $est_i = v_i$ y $r_i = 0$, donde est_i es el valor que se propone en cada ronda y r_i inidica el número de ronda. El algoritmo se va a repetir hasta que se logre llegar a un consenso. Se llama al algoritmo DSBV broadcast con el valor est_i. Si el resultado de DSBV es $view_i[r_i, 1] = \{v\}$ y $v \neq \perp$ entonces $est_i = v$, si no se elige un valor con la moneda común. Se hace DSBV broadcast con el nuevo valor de est_i y se guarda el resultado del broadcast en $view_i[r_i, 2]$. Si $view_i[r_i, 2] = \{v\}$ el proceso decide por v y termina el algoritmo. Si $view_i[r_i, 2] = \{v, \perp\}$ entonces $est_i = v$ y se repite el proceso. Si $view_i[r_i, 2] = \{\perp\}$ entonces se repite el proceso sin ninguna modificación adicional.

2.5 Idea intuitiva de la solución

Se hace uso dos veces del DSBV broadcast. Si en la primera fase se llega a un solo valor, entonces se propone ese valor en el broadcast, si no se propone un valor aleatorio, que si $d = 2$ entonces es igual para todos los procesos correctos que no hayan tenido un valor único. Después de este segundo broadcast si tienen un sólo valor deciden por éste. Si tienen un valor v y $\perp$, entonces eligen v para hacer broadcast en la ronda siguiente. Si no continúan y proponen lo que había salido de forma aleatoria en la ronda siguiente.

References