# Computación Distribuida

*UNAM, Facultad De Ciencias.*

**Profesor.** Fernando Michel Tavera
**Ayudante.** Mauricio Riva Palacio Orozco
**Ayudante.** Yael Antonio Calzada Martín

**Tarea 3**

**Alumnas:**
Góngora Ramírez Dania Paula (318128274)

Villafán Flores María Fernanda (318211767)

11 de abril de 2023
Ciudad de México.

**Ejercicio 1.** Demuestra que el siguiente protocolo resuelve el consenso en un sistema síncrono donde hasta $f < n$ procesos que pueden fallar.

```
Function Consensus(v_i)

V_i ← v_i; prev_i ← ⊥;
when r = 1, 2, ... , f + 1 do % r: round number %
begin_round
    if (prev_i ≠ V_i) then foreach j ≠ i: send (V_i) to p_j endif;
    let rec_from be the set of values received during r;
    prev_i ← V_i;
    if (rec_from ≠ ∅) then V_i ←min({V_i} ∪ rec_from) endif
end_round;
return (V_i)
```

Figura 1: Protocolo de Consenso mejorado

## Respuesta.

***Demostración:***
Mostraremos las siguientes propiedades:

- *Terminación*

  El algoritmo proporciona el número de rondas $r$, el cual llega hasta $f + 1$, donde $f$ es el número de fallos esperados. Ya que el protocolo no tiene bucles en los que pueda quedarse "atrapado" el algoritmo de forma que se quede sin terminar, entonces podemos garantizar que el protocolo finalizará despúes de las $f + 1$ rondas.
  Por lo tanto, se cumple la condición de *terminación.*

- *Validez*

  Notemos que el valor elegido en el consenso $V_i$ se actualizará de acuerdo al protocolo elegido, que en este caso podemos ver que es el mínimo de todos los mensajes recibidos. Estos mensajes recibidos están guardados en la variable $rec\_from$, que provienen solo de $p_j$. De esta forma, el valor elegido por todos los procesos fue propuesto por algún $p_j$.
  Por lo tanto, se cumple la condición de *validez.*

- *Acuerdo*

  Sea $V_i = v_k$ al finalizar la última ronda.
  Veamos que el proceso $p_i$ recibió a $v_k$ por última vez en alguna ronda y lo tomó como el valor mínimo.
  Los procesos $p_i$ y $p_k$ ejecutan todas las rondas ya que no hay una condición que los permita terminar antes.
  Tenemos los siguientes casos en función de $r$ que es la ronda en la que $p_i$ recibe a $v_k$.

  - ***Caso 1.*** $r < f + 1$

Aquí $p_i$ ya recibió a $v_k$ en la ronda $r$ y lo agregó a su $V_i$ ya que ese fue el valor mínimo. Posteriormente en la ronda $r+1$, $p_i$ envía $V_i$ a $p_j$ y así $p_j$ recibe $V_i$ de $p_i$ en la ronda $r+1 \leq f+1$.

- ***Caso 2.*** $r = f+1$

  Aquí $v_k$ ya fue reenviado varias veces, donde en un inicio $p_k$ lo envió a algún $p_j$ y eventualmente de alguna manera $p_i$ recibió a $v_k$. Para este caso, existe una cadena de procesos $p_k, p_l, p_m, ..., p_i$ en la que ninguno de los procesos se repite.
  Entonces, $v_k$ se fue reenviando ya que cada uno de los procesos en la cadena vió que $v_k$ era el valor mínimo y lo hizo su $V_*$. Como a lo más $f$ procesos pueden fallar, hay algún $p_l$ que recibió a $v_k$ en la ronda $r' < f+1$ y este $p_l$ lo reenvió en la ronda $r'+1 \leq f+1$.
  Por lo que $p_j$ debió recibir a $v_k$ en la ronda $r'+1 \leq f+1$. Así, para que $p_j$ haya recibido a $v_k$ todos los demás procesos fueron eligiendo a $v_k$ como su mínimo y reenviándolo en la cadena.

■

**Ejercicio 2.** Argumenta por qué es necesario ejecutar $f+1$ rondas para llegar a un acuerdo en el protocolo de consenso que ejecuta $f+1$ rondas.

```
Function Consensus(v_i)

V_i ← [⊥, ... , v_i, ... , ⊥]; New_i ← {(v_i, i)};
when r = 1, 2, ... , f + 1 do % r: round number %
begin_round
(α)  if (New_i ≠ ∅) then foreach j ≠ i: send (New_i) to p_j endif;
     let rec_from[j] = set received from p_j during r (∅ if no msg);
     New_i ← ∅;
     foreach j ≠ i: foreach (v, k) ∈ rec_from[j]:
(β)      if (V_i[k] = ⊥) then
                      V_i[k] ← v; New_i ← New_i ∪ {(v, k)} endif
end_round;
let v = first non-⊥ value of V_i;
return (v)
```

Figura 2: Protocolo de Consenso que ejecuta $f+1$ rondas

## Respuesta.

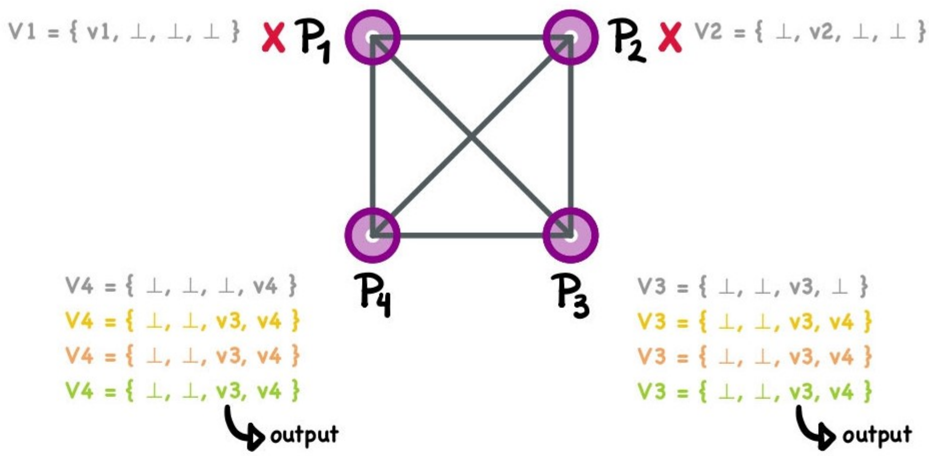
Sabemos que $f < n$, donde $n$ es el número de procesos en el sistema, por lo que cuando llegamos a la última ronda $r$ en la que todavía hay procesos que pueden fallar, si ejecutamos una ronda extra $r+1$, no tendremos procesos que puedan fallar, y entonces podemos garantizar que en esta ronda adicional $r+1$, los procesos que no hayan fallado intercambiarán información y podrán cumplir con el consenso (en particular con la propiedad de *acuerdo*).
Por lo tanto, se cumple que $r+1 \leq f+1$, y además que en $f+1$ rondas los procesos podrán llegar a un acuerdo en el protocolo de consenso que ejecuta $f+1$ rondas.

$\blacksquare$

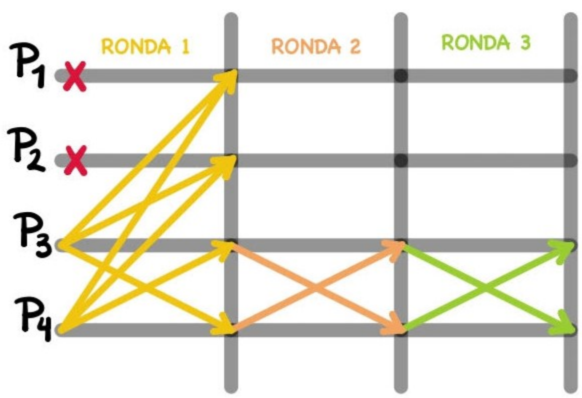
**Ejercicio 3.** Muestra la ejecución del protocolo de la figura 2 en un sistema con $n = 4$ procesos donde $f = 2$ y todos los procesos que fallan lo hacen en la primera ronda sin enviar ningún mensaje antes de fallar.

## Respuesta.

Notemos que los procesos $P_1$ y $P_2$ son los procesos que fallan en la primera ronda, por lo que no envían ningún mensaje antes de fallar, y adicional a esto, ya no participan para diseminar la información por la red de comunicación.
Entonces, veamos una ejecución del protocolo:




Como f = 2, entonces el número total de rondas está dado por: f + 1 = 2 + 1 = 3.




■

> **Ejercicio 4.** Si no hay ninguna falla, ¿en cuántas rondas termina el protocolo con *early stoping*? Justifica tu respuesta

## Respuesta.

Podemos ver que con *Early Decision*, el número real de fallas $t$ es $t \leq f$, donde $f$ es el número de fallas esperadas en el sistema. El protocolo en buenas condiciones podrá ejecutarse en $min(t+2, f+1)$ rondas.
Lo anterior nos deja dos casos:

- ***Caso 1:*** Cuando el número de fallas esperadas es 0, es decir, $f = 0$. Aquí, el número de rondas estará dado por
$$min(0+2, 0+1) = min(2, 1) = 1$$
lo que significa que se ejecutará en solo una ronda.

- ***Caso 2:*** Cuando esperamos al menos una falla, es decir, $f > 0$. Aquí, el número de rondas estará dado por
$$min(0+2, 1+1) = min(2, 2) = 2$$
lo que significa que se ejecutará en dos rondas.

■

**EJERCICIO 5.** Considera el siguiente algoritmo distribuido para un sistema **síncrono** que calcula la la distancia entre la raíz de una gráfica y el nodo que se está visitando.

**Algorithm 1** Calcula distancia desde la raíz - Código para el proceso $p_i$

```
neighbors_i = { Conjunto de vecinos de p_i }
if p_s = p_i then
    distance_i = 0
    for each j ∈ neighbors_i do
        send distance_i + 1 to p_j
    end for
else
    distance_i = ∞
end if

when distance_j is received from p_j do
begin:
    if distance_j < distance_i then
        distance_i = distance_j
        for each k ∈ neighbors_i do
            send distance_i + 1 to p_k
        end for
    end if
end
```

Demuestra por **inducción** que si un proceso está a distancia $k$ de la raíz , entonces al finalizar la ronda $k$ debe tener $distance_i = k$.

## Respuesta.

***Demostración por inducción.***

- *Caso base.* Cada proceso $p_i$ distinto del proceso distinguido inicializa su distancia como infinito.

  En caso de ser el proceso distinguido, inicializa su distancia como 0, por lo que iniciando desde el proceso distinguido, está a distancia $k$ de la raíz, ya que al empezar $k = 0$.

- *Hipótesis de inducción.* Supongamos que al finalizar la ronda $k$, todos los procesos $p_i$ a distancia $k$ de la raíz tienen como distancia $distance_i = k$.

- *Paso inductivo.* Probemos para $k + 1$.

  En el caso donde los procesos $p_i$ a distancia $k$ no hayan recibido ningún mensaje en la ronda $k$, entonces aún conservarán esa distancia; pero en la ronda $k + 1$, recibirán el mensaje de sus vecinos a distancia $k+1$ y actualizarán su distancia a $k+1$ (esto gracias a las líneas 14 a 16).

Por *Hipótesis de inducción*, al finalizar la ronda $k$ todos los procesos $p_i$ a distancia $k$ de la raíz tienen como distancia $distance_i = k$. Durante la ronda $k+1$, cada proceso $p_i$ a distancia $k+1$ envía su distancia actual a sus vecinos (líneas 14 a 16) y le suman 1. Así, todos los procesos $p_i$ a distancia $k$ reciben la distancia y la actualizan.

Por lo tanto, se cumple que cualquier proceso que esté a distancia $k$ de la raíz, al finalizar la ronda $k$, tendrá $distance_i = k$.

■

**Ejercicio 6.** Escribe el protocolo de consenso para sistemas asíncronos sin fallas.

## Respuesta.

```
function asyncConsensus(v_i):
    V_i ← v_i; prev ← ⊥;
    when r = 1, 2, ..., f + 1 do:
    begin_round
        if (prev_i ≠ v_i) then for each j ≠ i: send(v_i) to p_j end if;
        wait(until all messages from all other processes have been received);
        let rec_from be the set of values received during r;
        prev_i ← v_i;
        if (rec_from ≠ 0) then v_i ← min(v_i ∪ rec_from);
    end_round;
    return(V_i)
end function
```

■

> **EJERCICIO 7.** Explica con tus propias palabras por qué el consenso no puede ser resuelto en sistemas asíncronos con al menos una falla.

## Respuesta.

El consenso no puede ser resuelto en sistemas asíncronos con al menos una falla porque si un proceso falla, entonces, el algoritmo se ciclaría infinitamente al esperar el mensaje del proceso que falló (esto a causa de la línea 5 del *Ejercicio anterior*, el “wait”). También, consideramos que los procesos no tienen manera de saber si otro de los procesos falló.

■

> **Ejercicio 8.** Explica con tus propias palabras por qué el problema del ataque coordinado no puede resolverse con $n \geq 2$.

## Respuesta.

Como sabemos, la primera condición de *validez* dice que si todos los procesos comienzan con 0, entonces 0 es la única decisión posible. La segunda condición de *validez* dice que si todos los procesos comienzan con 1, entonces la única decisión válida es 1. Éstas condiciones se cumplen siempre y cuando no se pierdan los mensajes, entonces no pueden ser satisfechas simultáneamente, por lo que no hay una sola decisión por entrada.
Adicional a esto, es posible perder mensajes ya que no podemos garantizar que todos los procesos reciban la información necesaria para llegar a una decisión (esto podría ocasionar que algunos procesos tomen una elección diferente a los demás), lo cuál impediría llegar a una decisión.
Por lo tanto, ya tenemos por qué el ataque coordinado no puede resolverse con $n \geq 2$, donde $n$ es el número de procesos.

■

**EJERCICIO 9.** Demuestra los siguientes teoremas:

**TEOREMA 1**

En un sistema con tres procesos y un proceso bizantino, no existe un algoritmo que resuelva el problema del consenso.

**TEOREMA 2**

En un sistema con $n$ procesos y $f$ procesos bizantinos, no existe un algoritmo que resuelve el problema del consenso si $n \leq 3f$.

## Respuesta.

- ***Teorema 1.***

  ***Demostración por contradicción.***

  Supongamos que existe un algoritmo que resuelve el problema del consenso en un sistema con tres procesos y un proceso bizantino.

  Sean $p_0$, $p_1$ y $p_2$ tres procesos conectados por una gráfica de comunicación completa.

  Veamos en un sistema síncrono de anillo con seis procesos en el que asignaremos un algoritmo local a cada sección de procesos de la siguiente forma:

  - $p_0$ y $p_3$ tienen $A$ como algoritmo local.
  - $p_1$ y $p_4$ tienen $B$ como algoritmo local.
  - $p_2$ y $p_5$ tienen $C$ como algoritmo local.

  Nombremos una ejecución $\beta$ en la que los valores de entrada son 1 para $p_0$, $p_1$, $p_2$ y 0 para $p_3$, $p_4$, $p_5$.

  Consideremos tres ejecuciones diferentes del algoritmo en las que algunos procesos son defectuosos y envían mensajes según la ejecución $\beta$:

  - Veamos la ejecución $\alpha_1$ en la que todos los procesos tienen como entrada 1 y $p_2$ es el proceso defectuoso. Siguiendo la ejecución de $\beta$, de esta forma tanto $p_0$ como $p_1$ deben decidir 1.
  - Veamos la ejecución $\alpha_2$ en la que todos los procesos tienen como entrada 0 y $p_0$ es el proceso defectuoso. Siguiendo la ejecución de $\beta$, de esta forma tanto $p_1$ como $p_2$ deben decidir 0.
  - Veamos la ejecución $\alpha_3$ en la que $p_1$ es el proceso defectuoso y envía mensajes según la ejecución $\beta$. Entonces, tenemos que $p_0$ debe decidir 1 y $p_2$ debe decidir 0.

  Sin embargo, siguiendo la ejecución $\beta$ y ya que $p_0$ como proceso se comunica de $\alpha_1$ a $\alpha_3$, donde $p_0$ decide 0 en $\alpha_3$, como $p_2$ es un proceso que se comunica de $\alpha_2$ a $\alpha_3$, donde $p_2$ decide 1 en $\alpha_3$, entonces las decisiones entre $\alpha_1, \alpha_2, \alpha_3$ son mutuamente contradictorias.

  Por lo tanto, no es posible que exista un algoritmo que pueda resolver el problema del consenso en este sistema.

***NOTA:*** La demostración anterior es un resumen en nuestras palabras de lo que entendimos de la demostración sacada de la notas subidas a classroom CD_Cons.

- ***Teorema 2.***

  ***Demostración por contradicción.***

  Veamos que tenemos $n$ procesos y $f$ procesos bizantinos en el sistema tal que todos los procesos tienen diferentes valores iniciales.

  Supongamos que los $f$ procesos bizantinos están intentando sabotear el algoritmo de consenso, por lo que los $f$ procesos bizantinos pueden enviar diferentes valores a diferentes procesos.

  Por hipotesis, tenemos que $n \leq 3f$, esto significa que hay al menos un subconjunto $2f + 1$ de procesos no bizantinos, por lo que podemos suponer que $2f + 1$ procesos son capaces de llegar a un consenso; pero aún tenemos $f$ procesos bizantinos que pueden manipular los mensajes enviados a los procesos no bizantinos.

  Por lo tanto, los procesos no bizantinos son capaces de llegar a un consenso pero no tenemos certeza que estos valores sean los correctos (esto a causa de los $f$ procesos bizantinos).

  ■

**EJERCICIO 10.** Una versión alternativa del problema de consenso requiere que el valor de entrada de un proceso distinguido (llamado el general) se distribuya a todos los demás procesos (llamados los tenientes); este problema se conoce como consenso de fuente única. Las condiciones que deben cumplirse son:

- Terminación: Cada teniente no fallido debe eventualmente decidir.
- Acuerdo: Todos los tenientes no fallidos deben tener la misma decisión.
- Validez: Si el general es no fallido, entonces el valor de decisión común es la entrada del general.

La diferencia está en la condición de validez: si el general es defectuoso, el proceso no defectuoso no tiene por qué decidir sobre la entrada del general, pero deben estar de acuerdo entre sí. Consideremos el modelo síncrono de paso de mensajes sujeto a fallos bizantinos. Demuestre cómo transformar una solución al problema del consenso en una solución al problema del general y viceversa.

## Respuesta.

- ***Solución al problema del consenso $\longrightarrow$ Solución al problema del general***

  Para obtener ésta transformación, podemos asignar a uno de los procesos como el proceso distinguido, el cual tomará el rol de general. El general se encargará de enviar su valor de entrada a todos los demás procesos (tenientes) y de esta manera, tenemos que el problema del consenso se transforma en el problema del general, donde la validez necesita que los procesos acuerden sobre la entrada del proceso distinguido (general); aunque no necesariamente los procesos no fallidos deben decidir sobre la entrada del proceso distinguido si dicho proceso es defectuoso.

- ***Solución al problema del general $\longrightarrow$ Solución al problema del consenso***

  Para obtener ésta transformación, podemos asignar a uno de los procesos como el proceso distinguido, el cual tomará el rol de líder. El líder se encargará de enviar su valor de entrada a todos los demás procesos y de esta manera, cada proceso decide con base en la entrada del líder.
  Así, tenemos que:

  - Todos los procesos eventualmente deciden, es decir, tienen asignado un valor de salida, lo cual corresponde a la condición de *terminación* en el problema del consenso.
  - Los procesos no fallidos tienen el mismo valor de salida, lo cual corresponde a la condición de *acuerdo* en el problema del consenso.
  - La condición de *validez* se satisface ya que si el líder no es defectuoso, entonces su valor de entrada es el valor de entrada para todos. Pero si sucede que el líder es defectuoso, entonces la condición de *validez* no se cumple, sin embargo, los demás procesos deben tener un valor decidido (el cual fue propuesto por alguien).

■