Tolerancia a Fallos

30221 - Sistemas Distribuidos

Rafael Tolosana Calasanz

Dpto. Informática e Ing. de Sistemas

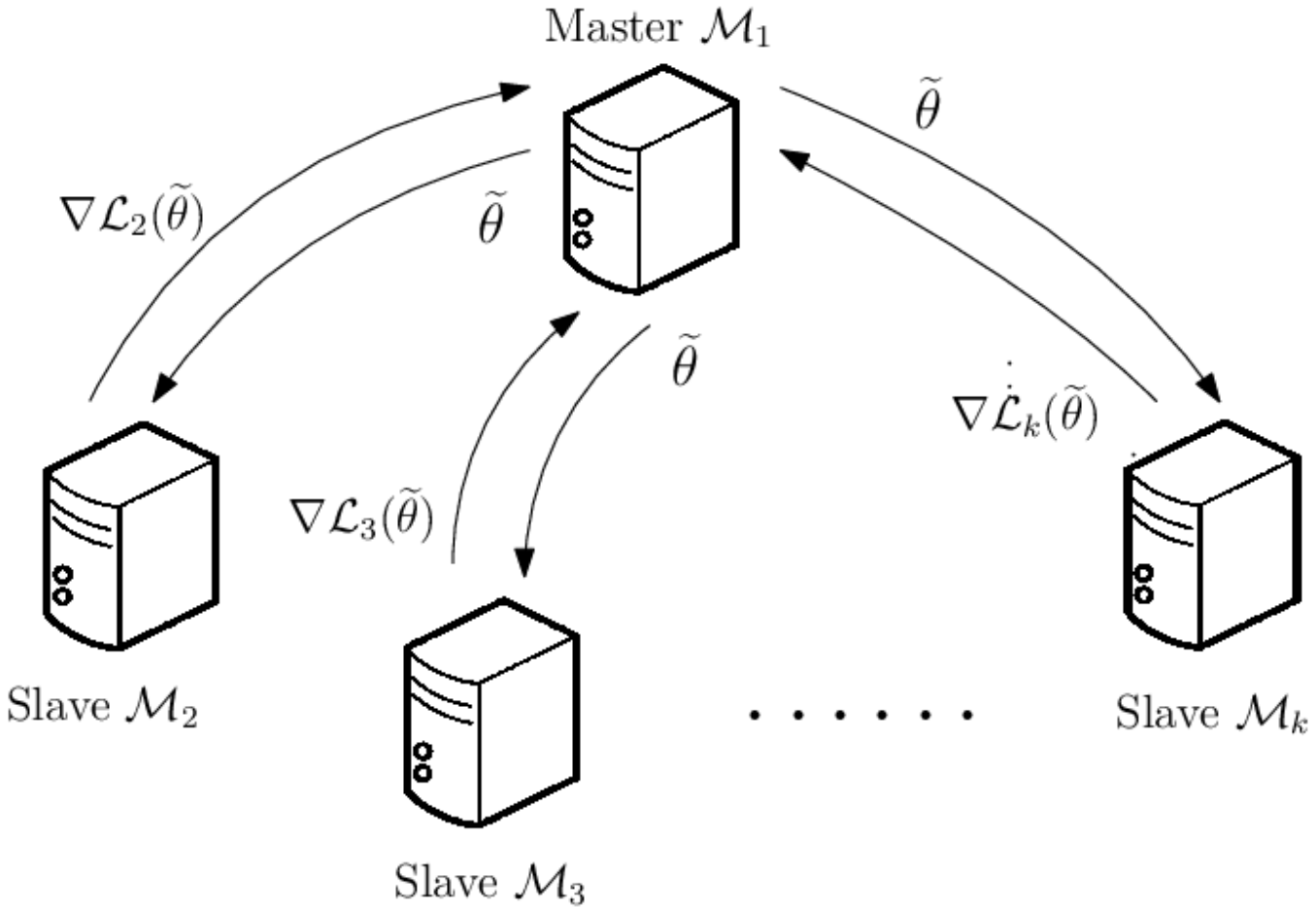
Lectura Recomendada

- Tanenbaum Van Steen, Distributed Systems: Principles and Paradigms, 3e, (c) 2017. **Chapter 8**

Motivación

Motivación

Fallos Parciales



Motivación

Fallos Parciales

- ¿Es el Algoritmo de Ricart-Agrawala tolerante a fallos?
- ¿Y el Algoritmo de Raymond?

operation acquire_mutex() **is**
(1) $cs\_state_i \leftarrow trying$;
(2) $\ell rd_i \leftarrow clock_i + 1$;
(3) $waiting\_from_i \leftarrow R_i$; % $R_i = \{1, \ldots, n\} \setminus \{i\}$
(4) **for each** $j \in R_i$ **do** send REQUEST$(\ell rd_i, i)$ to p_j **end for**;
(5) **wait** $(waiting\_from_i = \emptyset)$;
(6) $cs\_state_i \leftarrow in$.

operation release_mutex() **is**
(7) $cs\_state_i \leftarrow out$;
(8) **for each** $j \in perm\_delayed_i$ **do** send PERMISSION(i) to p_j **end for**;
(9) $perm\_delayed_i \leftarrow \emptyset$.

when REQUEST(k, j) **is received do**
(10) $clock_i \leftarrow \max(clock_i, k)$;
(11) $prio_i \leftarrow (cs\_state_i \neq out) \wedge (\langle \ell rd_i, i \rangle < \langle k, j \rangle)$;
(12) **if** $(prio_i)$ **then** $perm\_delayed_i \leftarrow perm\_delayed_i \cup \{j\}$
(13) **else** send PERMISSION(i) to p_j
(14) **end if**.

when PERMISSION(j) **is received do**
(15) $waiting\_from_i \leftarrow waiting\_from_i \setminus \{j\}$.

Motivación

Introducción

- **Fallo (Fault):** se denomina fallo a lo que causa un error en un SD

Motivación

Introducción

- **Fallo (Fault):** se denomina fallo a lo que causa un error en un SD
- **Origen Fallo**
  - Avería física (Failure) hardware, red
  - Programación Incorrecta (bug)
  - Otros Fallos: fallos que generan otros fallos

Motivación

Introducción

- **Fallo (Fault):** se denomina fallo a lo que causa un error en un SD
- **Origen Fallo**
  - Avería física (Failure) hardware, red
  - Programación Incorrecta (bug)
  - Otros Fallos: fallos que generan otros fallos
- **Objetivo**:
  - Diseñar un SD que pueda operar en presencia de fallos
    - Estrategias Preventivas Error / Fallo
    - Estrategias Reactivas Error / Fallo
- Nosotros seguiremos estrategias correctivas de fallo:
  - **Tolerancia a Fallos**

Motivación

Introducción

- Estrategias Preventivas
  - Predicción mediante machine learning
  - Predicción estadística
- Estrategias Reactivas
  - El sistema tiene que enmascarar el fallo

Fallos

Fallos

Ciclo de Vida de un Fallo: Evento-Condición-Acción

$$e \xrightarrow{t_1} f \xrightarrow{t_2} a \xrightarrow{t_3}$$

- e es un error software / avería física en un SD
- t_1 es el tiempo en manifestarse, detectarse e identificarse el fallo
- f es el fallo que ha generado el error / avería
- t_2 es el tiempo para generar una acción correctiva
- a es una acción para corregir el fallo
- t_3 es el tiempo para ejecutar la acción a y que surta efecto

Detección de Fallos

Detección de Fallos

- Exclusivamente mediante paso de mensajes
- Expiración de un tiempo: **timeout**. Ejemplo:
  - Los procesos P, Q interactúan mediante request-reply
  - P fija un temporizador y envía la petición
  - si la respuesta de Q no llega antes de que expire el temporizador
  - P reenvía la petición

Detección de Fallos

¿Qué quiere decir que expira el timeout?

Detección de Fallos

¿Qué quiere decir que expira el timeout?

¿Con qué certeza se ha producido un fallo?

Detección de Fallos

¿Qué quiere decir que expira el timeout?

¿Con qué certeza se ha producido un fallo?

¿Cómo se fija el timeout?

- Tiempos de comunicación, ejecución, *overhead*
  - Si se conocen los tiempos, se puede ajustar
- Si **no** se conocen tiempos, se aumenta el *timeout* tras cada retransmisión de forma exponencial: *exponential backoff*

Detección de Fallos

Ejemplo de Detección: cliente-servidor FindPrimes

Identificación de Fallos

Vamos a extraer métricas sobre el fallo

- **Localización**: lugar donde se origina: no siempre se puede distinguir entre fallo máquina y de red
- **Origen**: componente que lo origina
- **Estampilla** temporal: cuándo se produce
- **Duración**: permanente, intermitente, temporal
- Nivel de **gravedad** (subjetivo): bajo, medio, alto
- **Comportamiento**: crash, omission, timing, response, byzantine

Se puede obtener un vector de métricas y con él se puede intentar identificar el problema

Identificación de Fallos

Comportamiento de un componente cuando falla:

- **Crash**: falla y deja de funcionar
- **Omission**: no se recibe respuesta tras la petición
- **Timing**: se recibe la respuesta con retardo
  - No se puede distinguir entre timing / omission: indeterminismo. Se pueden generar respuestas duplicadas
- **Response**: respuesta incorrecta, típicamente algún problema morfo-sintáctico en el mensaje
- **Byzantine**: respuesta maliciosa semánticamente, intencionadamente incorrecta

Acciones Correctoras

Algunas Ideas

- Redundancia temporal: reintentar
- Redundancia física: replicación
- Ignorar el fallo
- Pedir ayuda humana

Redundancia Cliente

Redundancia Cliente

Redundancia Temporal Cliente (arq. cliente-servidor)

El comportamiento de la gestión de los fallos se añade en el proxy

- Algoritmo 1
- Algoritmo 2

Redundancia Cliente

Algoritmo 1: redundancia temporal

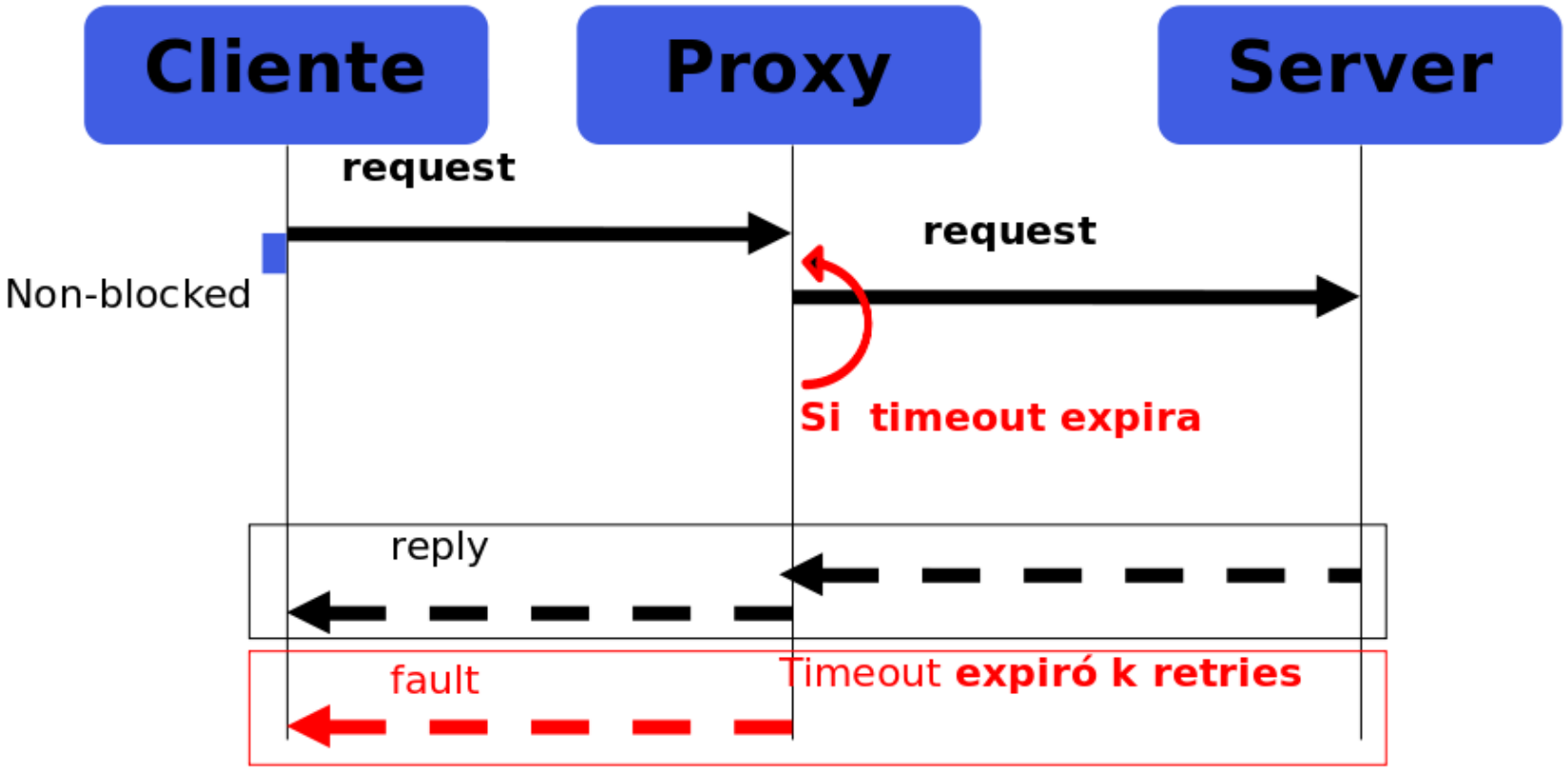
- El proxy gestiona los fallos
- El proxy fija un timeout y reintenta k veces si este expira
- Se detectan fallos de tipo:
  - Crash
  - Omission
  - Timing

¿Qué sucede si el timeout es...

- demasiado corto?
- demasiado largo?

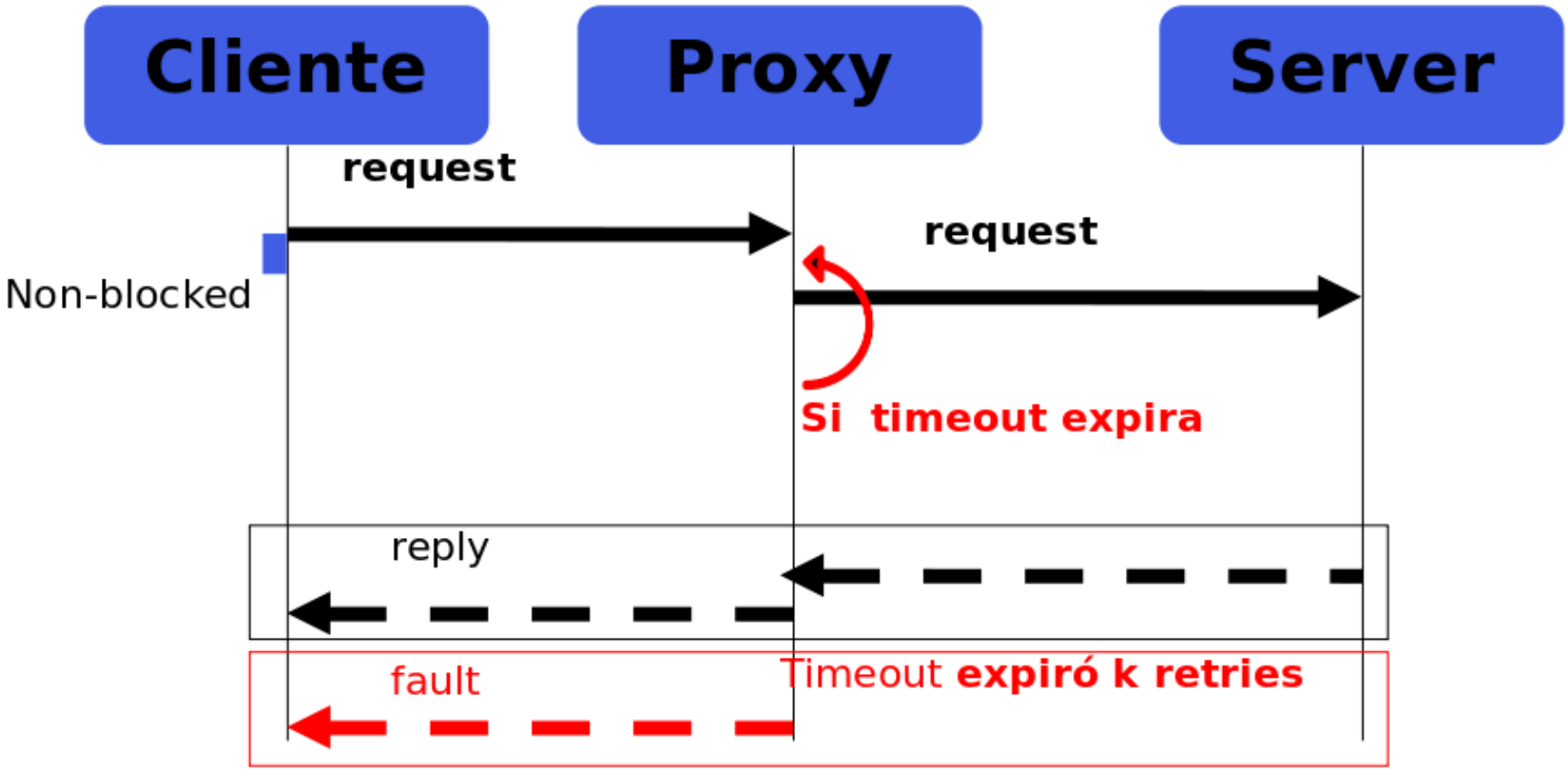
Redundancia Temporal Cliente

Algoritmo 1 con reintento



Redundancia Temporal Cliente

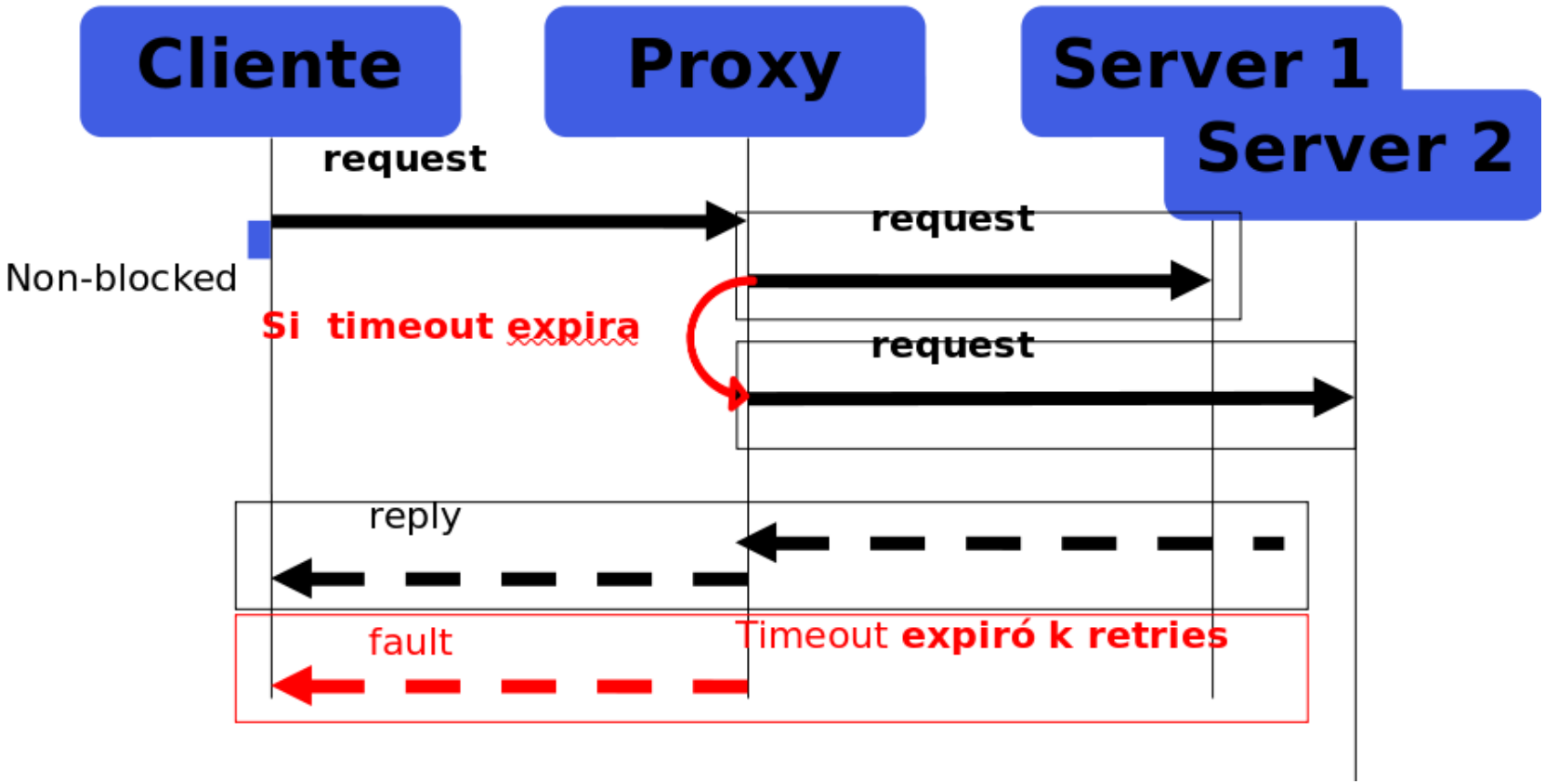
Algoritmo 1 con reintento



¿Cómo se fija el timeout?

Redundancia Temporal Cliente

Algoritmo 1 con alternativa



Redundancia Física y Temporal Cliente

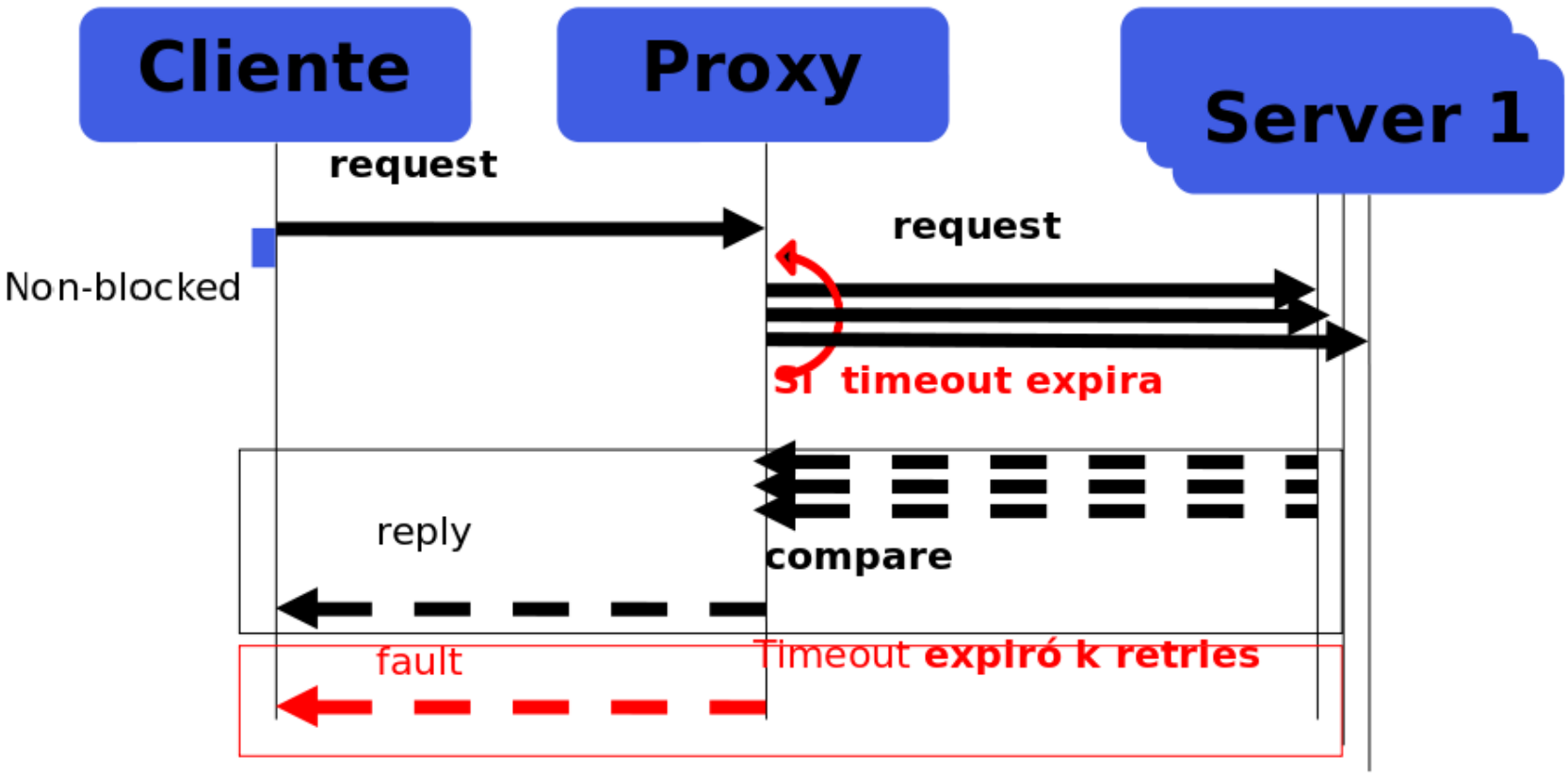
Algoritmo 2 – Redundancia física y temporal

- El proxy gestiona los fallos
- El proxy fija un timeout, envía la petición a N servidores a la vez y si no llega la respuesta reintenta k veces. En cuanto llega una respuesta, se la redirecciona al cliente
- Se detectan fallos de tipo:
  - Crash
  - Omission
  - Timing

La redundancia incrementa el coste económico

Redundancia Física y Temporal Cliente

Algoritmo 2 con reintento



Redundancia Física y Temporal Cliente

Posible Fragmento de Implementación del Proxy

```
req := ms.receive()
responseChannel := make(chan Response)
for i := 1; i < numReplicas; i++ {
    go Interaction(server[i], req, timeout, retry, responseChannel)
}
select{
    case reply <- responseChannel
        responses = append(responses, reply)
    case <- time.After(timeout^retry * 1.2 * time.Second)
        error! ??? ¿Qué hacemos?
}
response := compare(responses)
ms.send(req.sender, response)
```

Redundancia Física y Temporal Cliente

¿Qué soporte da Go para Reemplazar un código por otro?

- Higher Order Functions: funciones como parámetro formal y como resultado
  - Idea: invoco a una función que me pasan como parámetro
- Reflection
- Exception Handling simplificado de Go

Redundancia Física y Temporal Cliente

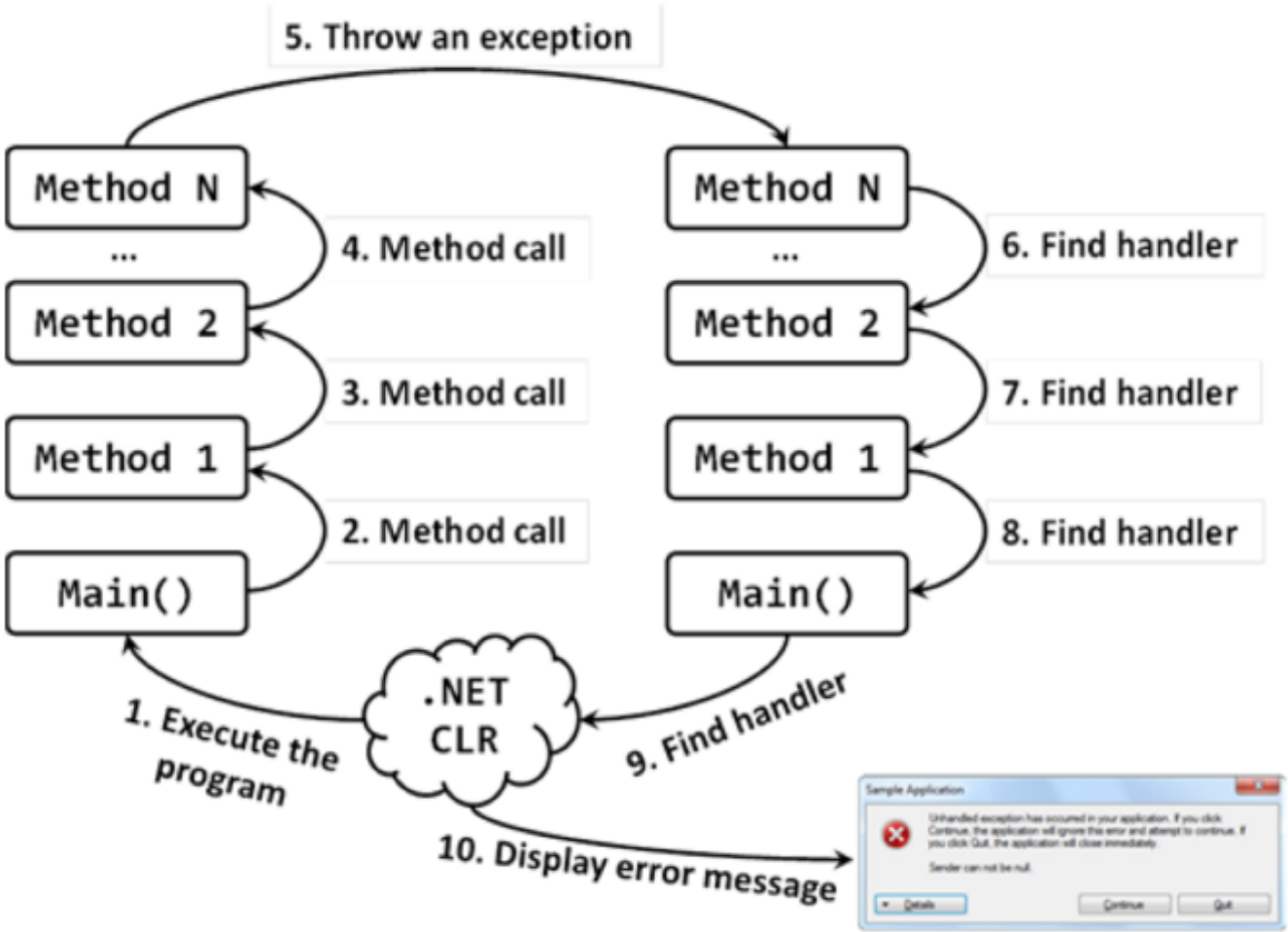
Exception Handling

- una **excepción** es un evento (error, fault, bug) anómalo que impide la normal ejecución de la instrucción que lo generó
- Objetivo: **separar** código normal del excepcional

```
double read_input()
{
    double input;
    bool valid = true;
    cout << "Enter number: " ;
    while(valid){
        cin >> input;
        if(cin.fail())
        {
            valid = false;
        }
    }
    return input;
}
```

Redundancia Física y Temporal Cliente

Exception Handling



Redundancia Física y Temporal Cliente

Errores en Golang [1]

Why didn't Golang utilize exceptions, a conventional way to handle errors? Keep these two hey points in mind while Golang error handling is:

- Keep it simple
- Plan where it can go wrong

Here are the advantages of Golang new error handling:

- No interruption of sudden uncaught exceptions
- Simple syntax
- You have complete control over the errors
- Transparent control-flow
- Easy implementation of error chains to take action on the error

[1] https://dev.to/amelias26018837/prepare-yourself-to-deal-with-golang-error-handling-3jj5

Redundancia Física y Temporal Cliente

Algoritmos de Detección de Fallos Bizantinos -Cliente-Servidor (cliente)

- Interacción Síncrona, Lamport
  - Comparación de resultados (consenso) entre k + 1 réplicas
- Interacción Asíncrona, Castro-Liskov
  - Comparación de resultados (consenso) entre 3*k + 1 réplicas
  - K es el número de procesos que fallan "a la bizantina"

Redundancia Servidor

Redundancia Servidor

Detector Fallos: Latido (Heartbeat)

- un latido es una señal periódica (hardware / software) para comprobar si el funcionamiento de un sistema es normal o para sincronizar otras partes de un sistema computacional

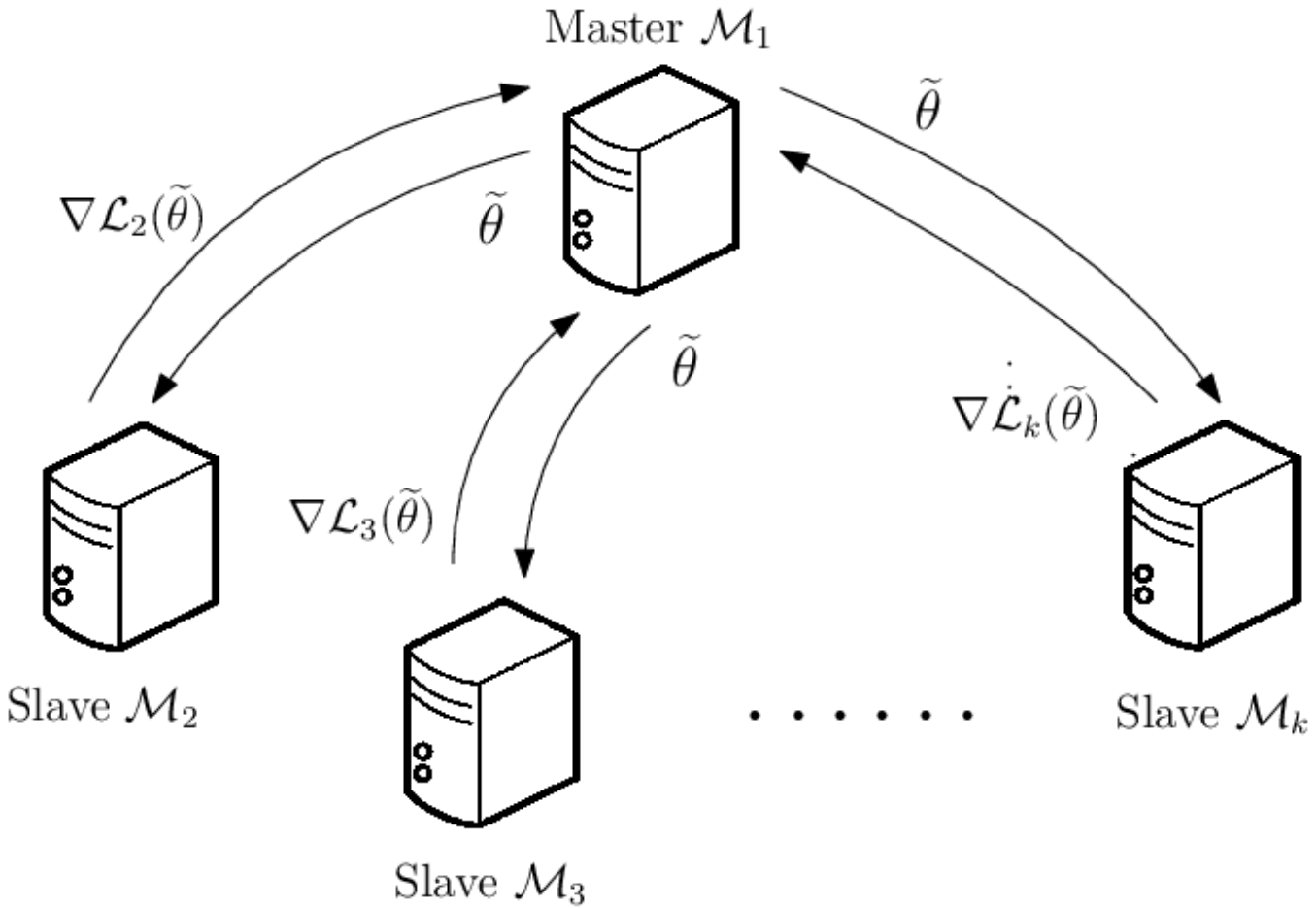
Redundancia Servidor

Detector Fallos: Latido (Heartbeat)

- un latido es una señal periódica (hardware / software) para comprobar si el funcionamiento de un sistema es normal o para sincronizar otras partes de un sistema computacional
- pueden ocurrir de forma periódica o bien solo en momentos puntuales (v.gr. en el arranque)
- En esencia, se puede articular mediante un protocolo de interacción request-reply.

Redundancia Servidor

Latido, Heartbeat



Redundancia Servidor

Servidor SIN estado

- No hay **estado** en el código del servidor
- Por ejemplo, en una estructura jerárquica como el master-worker de la práctica 1, no hay estado. Si el máster falla, los workers no pueden dar servicio. Si fallara un worker, el sistema aún funcionaría.
- Solución: **Algoritmos de Elección de Líder**

Servidor CON estado

- Hay un almacén de datos cuya integridad hay que mantener

Redundancia Servidor

Algoritmos de Elección de Líder

Objetivo: Seleccionar un proceso, dentro de un grupo distribuido, para tareas específicas, como la coordinación centralizada.

- Por ejemplo, falla un coordinador, entonces busco otro, v.gr. falla el máster en el máster worker.

También son útiles para designación automática de líder al arrancar sistema.

Consideraciones de prestaciones en sistemas distribuidos:

- Número de mensajes intercambiados
- Tiempo de ejecución del algoritmo de elección

Redundancia Servidor

Supuestos de algunas soluciones:

- Cualquier proceso del grupo electivo puede servir de coordinador y puede iniciar elección
- Cada proceso puede tener una sola elección ejecutándose al mismo tiempo
- En los algoritmos que proponemos, cada nodo tiene un único identificador (@red, pid, etc.)
- Elección: acuerdo sobre qué proceso vivo se convierte en coordinador.
- Al final del algoritmo se ha elegido un solo proceso y los demás conocen su identidad

Algoritmo del Matón

Algoritmo del Matón (Pág 644 Colouris)

- Cada proceso conoce el PID de los demás procesos y sabe, por tanto, cuáles son mayores
- Interacción **síncrona**: usa **timeouts** para detectar fallos [2]
- **Red sin fallos**
- Un proceso inicia una elección sólo si se da cuenta de que el **coordinador no responde**
- Los procesos con mayor identificador **"intimidan"** a los de menor para expulsarlos de la elección hasta que queda un solo proceso.
- Cuando un proceso, que ha fallado, rearranca, inicia un elección

[2] tienen que estar perfectamente ajustados para no tener falsos positivos

Algoritmo del Matón

4 tipos de mensajes

- **Elección**: anuncia elección.
- **Respuesta**: respuesta a elección
- **Coordinador**: anuncia coordinador elegido.
- **Latido**: mensaje que se envía al coordinador, si contesta está vivo

Un proceso lanza una nueva elección cuando

- se da cuenta de que el coordinador ha fallado mediante **timeout expirado** (varios procesos se pueden dar cuenta a la vez), o bien
- recibe un mensaje **Elección**.
  - Si ya había iniciado la Elección, y le llega mensaje de Elección, contesta si procede, pero no vuelve a lanzar la Elección.

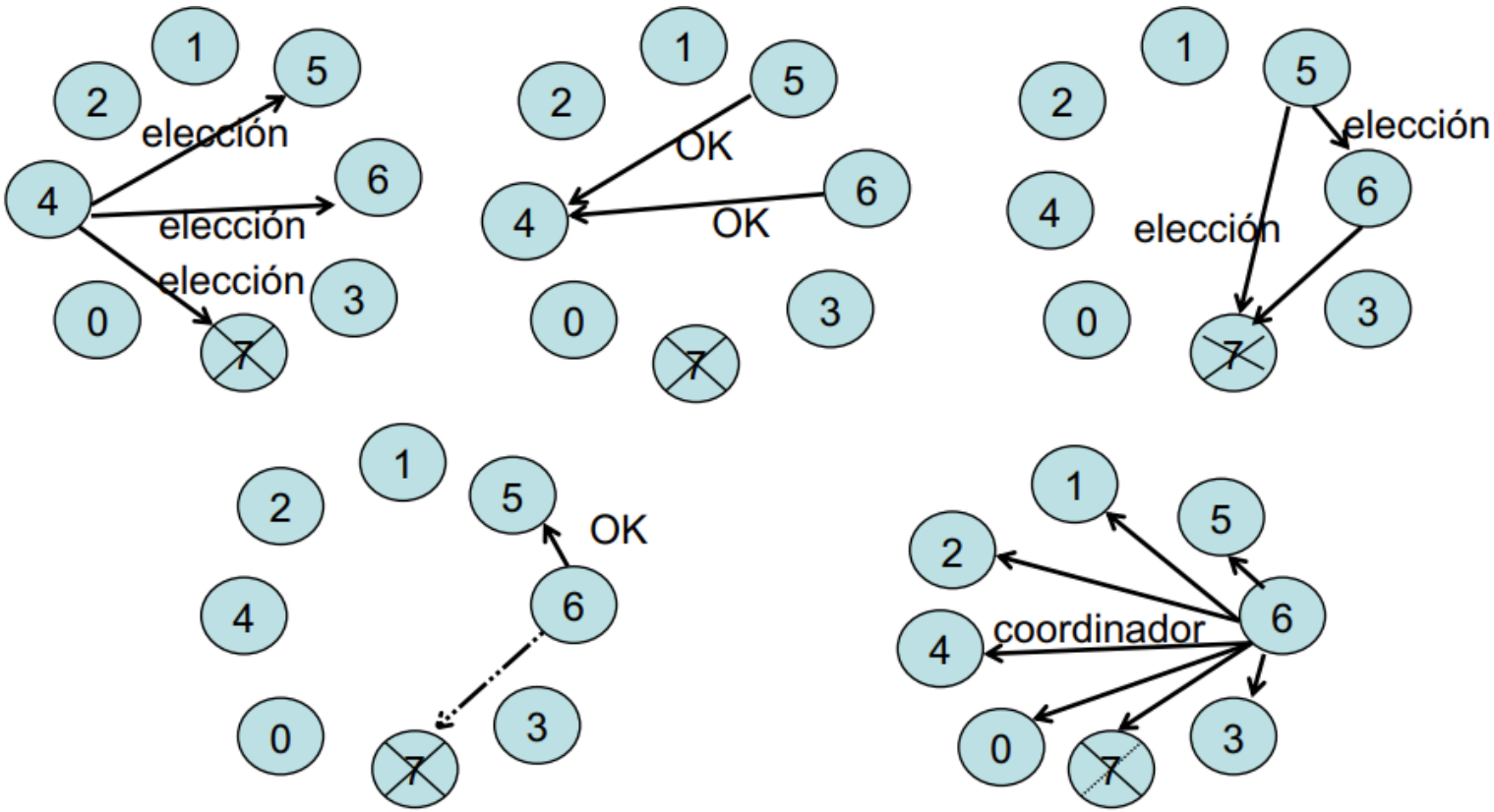
Algoritmo del Matón

Cuando un proceso empieza una Elección:

- Envía mensaje Elección a todos los procesos con PID más alto que el suyo.
- Si no se recibe mensaje Respuesta, el proceso que ha comenzado elección es el coordinador y envía mensaje Coordinador a todos los procesos vivos.
- Si llega Respuesta, espera un mensaje Coordinador durante un tiempo determinado
- Si no llega el mensaje Coordinador, reinicia la Elección

Si un proceso sabe que tiene el PID más alto, puede responder inmediatamente con mensaje Coordinador

Algoritmo del Matón



Algoritmo del Matón

Esquema del Algoritmo del Matón en Go

```
func (b* Bully) Reception() {
    for {
        msg := ms.Receive()
        if msg.type == ms.OK { // recibido un ok, hay procesos con ID mayor
            select {
            case b.Okchannel <- ok:
                continue
            case <- time.After(200*time.Millisecond):
                continue // si ya no hay nadie escuchando, descarto
            }
        } else if msg.type == ms.COORDINATOR{// nuevo coordinador
            b.coordinatorCh <- ms.sender
        } else if msg.type == ms.ELECTION { // nuevo proceso de elección
            b.electionChannel <- msg.sender
        } else if msg.type == ms.LEADERBEAT { // el leader contesta al latido
            b.leaderBeatChannel <- msg.sender
        }
}
```

Algoritmo del Matón

Análisis de las Prestaciones

- Se deben acotar las latencias de forma adecuada para ajustar los tiempos de expiración (2*RTT + Tprocesado) - Interacción Síncrona
- Si hay error durante la detección de fallo, la elección seguirá adelante. Tolera fallos durante la elección

¿Cuál es el mejor caso? ¿Cuál es el peor caso?

Algoritmo de Elección en Anillo

Asume procesos organizados en anillo (Pág 642 Colouris):

- Cada proceso conoce su posición en el anillo (ordenación por PID)
- Cada proceso, además, conoce al resto, de esta forma, es sencillo reconectar el anillo si hay procesos que fallan (prueba el siguiente, si no el siguiente, etc).
- Se asume que no hay fallos durante el proceso de elección
- El sistema es asíncrono
- El objetivo es elegir a un proceso como coordinador (líder), aquel proceso vivo cuyo ID es el mayor.

Algoritmo de Elección en Anillo

Funcionamiento básico:

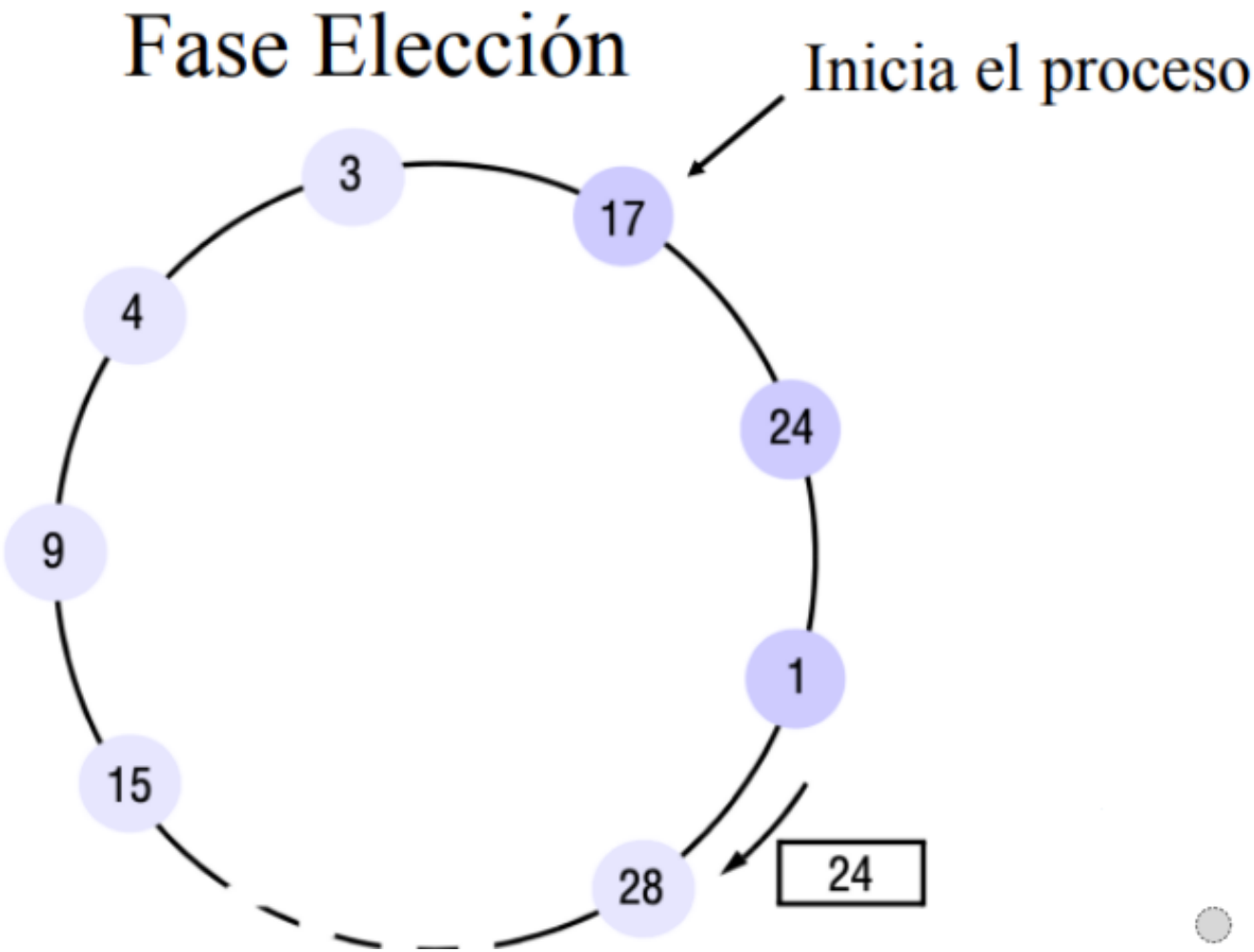
- Procesos ordenados en anillo lógico.
- 3 tipos de mensajes:
  - **Elección**: reenviar datos de elección
  - **Coordinador**: anuncio del coordinador elegido.
  - **Latido**: timeout para determinar si el coordinador funciona
- Cualquier proceso puede iniciar la elección

Algoritmo de Elección en Anillo

ALGORITMO:

- Un proceso P_i empieza una elección cuando se da cuenta del fallo del coordinador mediante un tiempo de expiración.
- P_i envía mensaje Elección junto con su Id al primer vecino que esta vivo.
- Cuando un proceso recibe un mensaje Elección, compara su Id con el recibido en el mensaje de Elección:
  - Si su ID es mayor, lo reemplaza y lo reenvía a lo largo del anillo.
  - Si es menor, reenvía el mensaje de Elección.
  - Si su ID es el que aparece en el mensaje de Elección, ese proceso es el nuevo líder.
- Finalmente, el proceso "reenvía" un mensaje Coordinador.

Algoritmo de Elección en Anillo



Algoritmo de Elección en Anillo

- Los procesos que han recibido un mensaje Elección no se les permite que inicien otra elección hasta que hayan recibido el mensaje Coordinador.
- ¿Y si falla el proceso p que ha iniciado el algoritmo antes de llegarle de vuelta el mensaje de elección?
- ¿Y si se originan 2 o más elecciones al mismo tiempo?
- ¿Otras optimizaciones?
- ¿Prestaciones?
- ¿Mejor y peor casos?

Ejercicios

Ejercicios

- En lugar de tener una arquitectura cliente-servidor mediante un middleware de paso de mensajes punto a punto, la interacción se realiza a través de un broker (Linda).
- Proponer una adaptación del Algoritmo de Detección de fallos 2 en una arquitectura cliente-servidor a través de un broker (Linda)

Resumen

Resumen

Fallos en SSDD

- Detección, Identificación, Corrección
- Tipos de Fallos
- Estrategias de corrección de fallos:
  - redundancia temporal
  - redundancia física
- Detección de fallo arquitectura cliente, servidor:
  - En el cliente: detección de fallos: $t_{tex} + t_{ton} + t_o$
  - En el servidor: t_{ton}, si interacción **síncrona**
- Estrategias de corrección:
  - Cliente: Reintentar, Alternativas, ...
  - Servidor: Algoritmos de elección de líder

Tolerancia a Fallos

30221 - Sistemas Distribuidos

Rafael Tolosana Calasanz

Dpto. Informática e Ing. de Sistemas