.UBAfiuba
FACULTAD DE INGENIERÍA

Técnicas de Programación Concurrente I

Concurrencia Distribuida - Parte II

Ing. Pablo A. Deymonnaz
pdeymon@fi.uba.ar

Facultad de Ingeniería
Universidad de Buenos Aires

.UBAfiuba FACULTAD DE INGENIERÍA

Índice

Introducción

- Varios algoritmos requieren de un coordinador con un rol especial (ej: algoritmos de exclusión mutua distribuida).
- En general, no es importante cuál es el proceso, sino que debe cubrirse el rol.
- Se asume: todos los procesos tienen un ID único, se ejecuta un proceso por máquina y conocen el número de los demás procesos.
- El objetivo: cuando la elección comienza, concluye con un elegido.

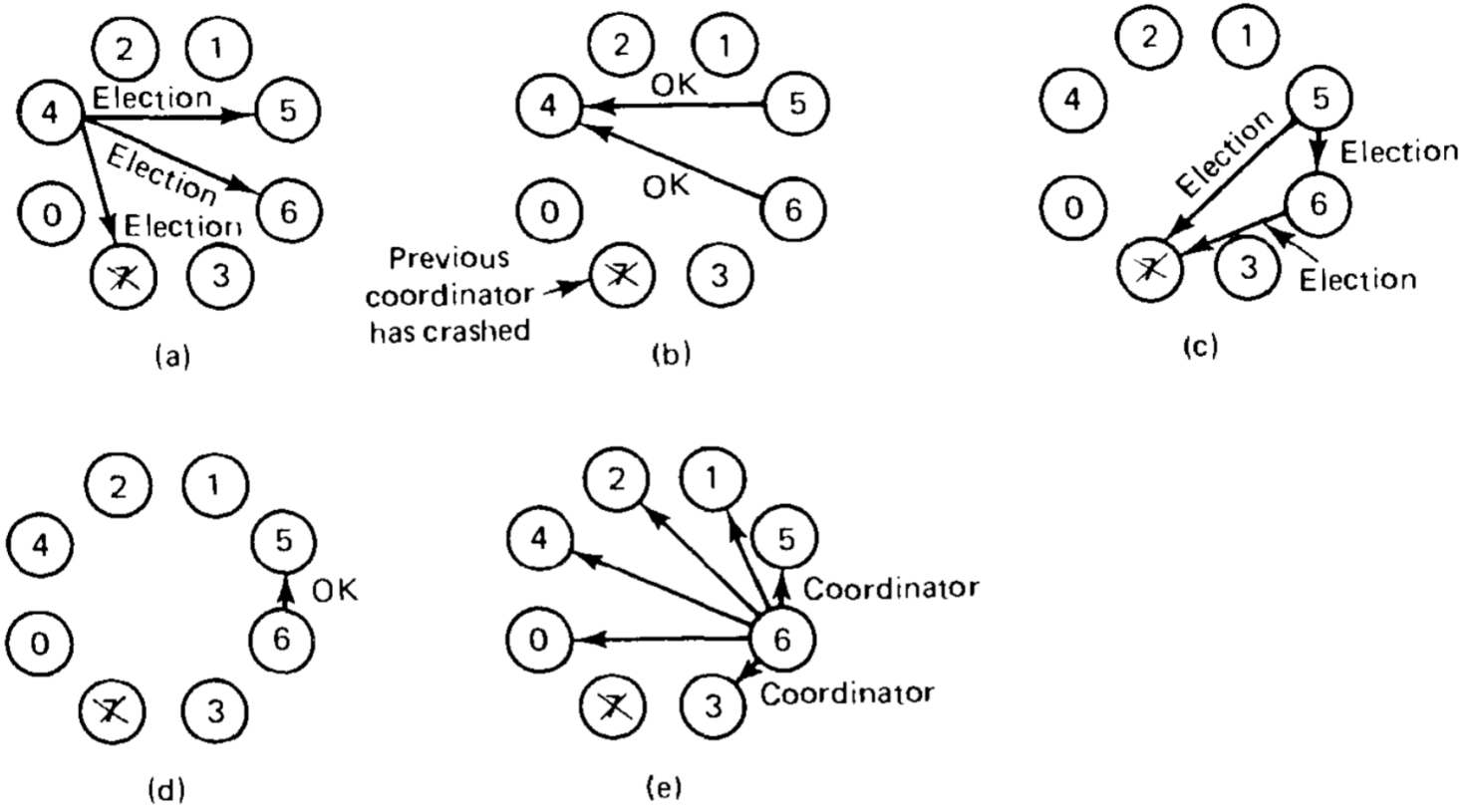
Elección: Algoritmo Bully

Cuando un proceso P nota que el coordinador no responde, inicia el proceso de elección:

1. P envía el mensaje ELECTION a todos los procesos que tengan número mayor
2. Si nadie responde, P gana la elección y es el nuevo coordinador
3. Si contesta algún proceso con número mayor, éste continúa con el proceso y P finaliza
4. El nuevo coordinador se anuncia con un mensaje COORDINATOR

Siempre gana el proceso con mayor número.

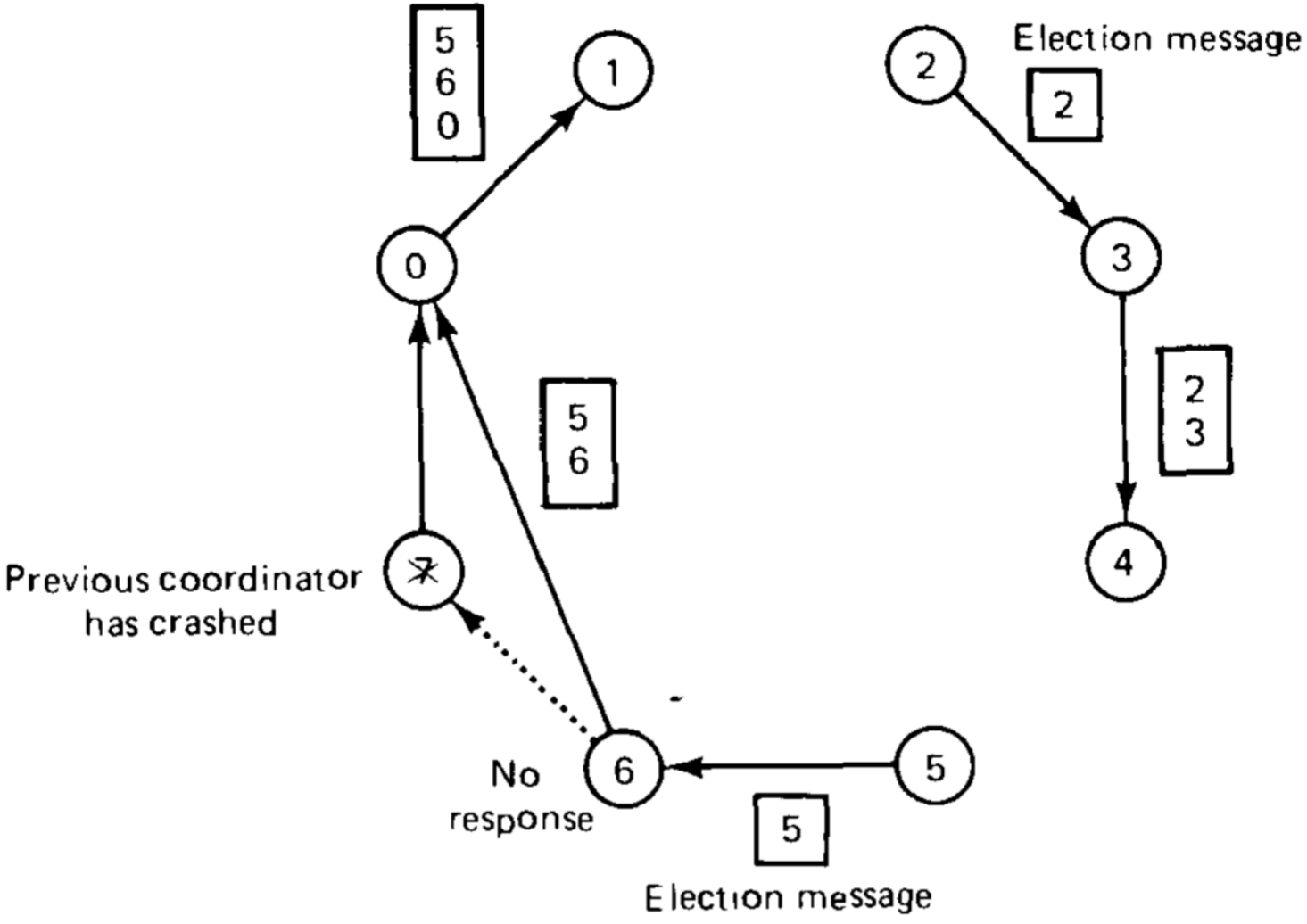
Elección: Algoritmo Bully



Elección: Algoritmo Ring

1. Los procesos están ordenados lógicamente; cada uno conoce a su sucesor
2. Cuando un proceso nota que el coordinador falló, arma un mensaje ELECTION que contiene su número de proceso y lo envía al sucesor
3. El proceso que recibe el mensaje, agrega su número de proceso a la lista dentro del mensaje y lo envía al sucesor
4. Cuando el proceso original recibe el mensaje, lo cambia a COORDINATOR y lo envía. El nuevo coordinador es el proceso de mayor número de la lista. La lista se mantiene para informar el nuevo anillo
5. Cuando este mensaje finaliza la circulación, se elimina del anillo

Elección: Algoritmo Bully



Índice

Servidor TCP

La biblioteca de networking de Rust se encuentra en el módulo **std::net**.
Para el **Servidor**.

- **Primer Paso:** Asociar el socket a una dirección.
  El método **bind** crea un nuevo *TcpListener* y lo asocia a una dirección específica.

```
pub fn bind<A: ToSocketAddrs>(addr: A) -> Result<TcpListener
```

El *listener* retornado está listo para aceptar conexiones.

```
let listener = TcpListener::bind("127.0.0.1:80")?;
```

Servidor TCP

- **Segundo Paso:** Obtener conexiones establecidas.

Sobre la estructura *TcpListener* se obtienen conexiones establecidas. El método **incoming** retorna un iterador que devuelve una secuencia de streams de tipo *TcpStream*.

```
pub fn incoming(&self) -> Incoming<'_>
```

Cada stream representa una conexión abierta entre el cliente y el servidor.

```
for stream in listener.incoming() {
let stream = stream.unwrap();
println!("Conexion establecida!");
}
```

La iteración es sobre "intentos de conexiones". Puede retornar *Err*.

Servidor TCP

- **Segundo Paso (forma alternativa):** Obtener conexiones establecidas con **accept**.

EL método **accept** obtiene una conexión establecida de un listener.

```
pub fn accept(&self) -> Result<(TcpStream, SocketAddr)>
```

El hilo se bloquea hasta que haya una conexión establecida.

```
match listener.accept() {
Ok((_socket, addr)) => println!("nuevo cliente: {:?}", addr)
Err(e) => println!("error: {:?}", e),
}
```

Servidor TCP

- **Tercer Paso:** Leer datos del socket: **read**.

TcpStream implementa el método **read** (del trait *std::io::Read*).

```
fn read(&mut self, buf: &mut [u8]) -> Result<usize>
```

por ejemplo:

```
let mut buffer = [0; 1024];
stream.read(&mut buffer).unwrap();
```

Servidor TCP

- **Escribir una respuesta:** El servidor envía una respuesta a una petición del cliente.

TcpStream implementa el método **write** (del trait *std::io::Write*).

```
fn write(&mut self, buf: &[u8]) -> Result<usize>
```

por ejemplo:

```
let response = "Respuesta!\n";

stream.write(response.as_bytes()).unwrap();
stream.flush().unwrap();
```

El método *flush* realiza una espera, previniendo que el programa continúe sin haber escrito en la conexión todos los bytes.
Flush this output stream, ensuring that all intermediately buffered contents reach their destination.

Cliente TCP

El cliente debe establecer la **conexión** con el servidor.
Construir la dirección de destino

- **A partir de una dirección IP:**

```
use std::net::{IpAddr, Ipv4Addr, SocketAddr};
let socket =
  SocketAddr::new(IpAddr::V4(Ipv4Addr::new(127, 0, 0, 1)), 8
```

- **A partir de un nombre:** método **to_socket_addrs**

```
fn to_socket_addrs(&self) -> Result<Self::Iter>
```

por ejemplo:

```
let mut addrs_iter = "localhost:443".to_socket_addrs().unwra
```

Devuelve un iterador de direcciones.

Cliente TCP

Conectarse al servidor: el cliente ejecuta el método **connect**

```
pub fn connect<A: ToSocketAddrs>(addr: A) -> Result<TcpStrea
```

Este método abre una conexión al host remoto.
Si se le envía un array de direcciones, intenta conectarse a cada una, hasta lograrlo.

```
let addrs = [
SocketAddr::from(([127, 0, 0, 1], 8080)),
SocketAddr::from(([127, 0, 0, 1], 8081)),
];
if let Ok(stream) = TcpStream::connect(&addrs[..]) {
println!("Connectado al servidor!");
} else {
println!("No se pudo conectar...");
}
```

Cliente TCP

Para enviar y recibir datos, el cliente ejecuta los métodos **read** y **write** igual que el servidor.

Finalizar conexión

- El cierre de la conexión TCP puede ser realizado de forma individual.
- La conexión establecida con *TcpStream* se cierra cuando el valor ejecuta *drop*. Esto inicia el envío del mensaje *close* de TCP.
- El método shutdown pude cerrar el extremo de escritura, de lectura o ambos.

```
pub fn shutdown(&self, how: Shutdown) -> Result<()>
```

Bibliografía

- **Distributed Operating Systems**, Andrew S. Tanenbaum, capítulo 3
- **Computer Networks**, Andrew S. Tanenbaum y David J. Wetherall, quinta edición
- **The Rust Programming Language**, *https://doc.rust-lang.org/book/*
- **The Complete Rust Programming Reference Guide**, Rahul Sharma, Vesa Kaihlavirta, Claus Matzinger.