Práctica 2 – Algoritmos en Computación Distribuida

Profesor: Mauricio Riva Palacio Orozco

Ayudantes: Adrián Felipe Fernández Romero y Alan Alexis Martínez López

Fecha de entrega: Lunes 29 de septiembre de 2025

1. Introducción

En esta práctica se busca que los alumnos implementen algunos de los algoritmos fundamentales en computación distribuida, utilizando Simpy para simular la comunicación entre procesos. El objetivo es comprender cómo los procesos (nodos) en una red pueden: 1) Conocer a los vecinos de sus vecinos, 2) Construir un árbol generador a partir de un nodo distinguido, 3) Difundir (broadcast) un mensaje desde un nodo raíz a toda la red, 4) (Extra) Realizar un convergecast.

2. Explicación de las clases base

Nodo.py: Clase general para representar un nodo. Contiene ID, lista de vecinos y canales de comunicación.

NodoVecinos.py: Implementa el algoritmo de vecinos de vecinos.

NodoGenerador.py: Implementa el algoritmo de construcción del árbol generador.

NodoBroadcast.py: Implementa el algoritmo de broadcast.

Canal.py: Interfaz general para modelar un canal.

CanalBroadcast.py: Especialización de canal para comunicación one-to-many.

3. Algoritmos

3.1 Algoritmo 1 – Conocer vecinos de vecinos

```
1: id_i = i
2: neighbors_i = { Conjunto de vecinos de p_i }
3: identifiers_i = {∅}

4: main()
5: begin:
6:     for each j ∈ neighbors_i do
7:         send MYNAME(neighbors_i) to j
8:     end for
9: end

10: when MYNAME(identifiers_j) is received from neighbor p_j
11: begin:
12:     identifiers_i = identifiers_i ∪ identifiers_j
13: end
```

Explicación

Cada nodo envía su lista de vecinos a todos sus vecinos. Cuando recibe la lista de un vecino, actualiza su conjunto de identificadores agregando esos valores.

3.2 Algoritmo 4 – Construcción del árbol generador

```
1: Initially do
2: begin:
3: if ps = pi then
4:     parent_i = i; expected_msg_i = |neighbors_i|
5:     for each j ∈ neighbors_i do send GO() to pj
6:     end for
7: else parent_i = ∅
8: end if
9: children_i = ∅
10: end

11: when GO() is received from pj do
12: begin:
13: if parent_i = ∅ then
14:     parent_i = j; expected_msg_i = |neighbors_i| - 1
15:     if expected_msg_i = 0 then send BACK(i) to pj
16:     else
17:         for each k ∈ neighbors_i \ {j} do send GO() to pk
18:         end for
19:     end if
20: else send BACK(∅) to pj
21: end if
22: end

23: when BACK(val set) is received from pj do
24: begin:
25: expected_msg_i = expected_msg_i - 1
26: if val set ≠ ∅ then children_i = children_i ∪ {j}
27: end if
28: if expected_msg_i = 0 then
29: if parent_i ≠ i then
30:     send BACK(i) to parent_i
31: end if
32: end if
33: end
```

Explicación

El nodo distinguido envía GO() a sus vecinos. Cada nodo que recibe un GO por primera vez establece a su padre y reenvía el mensaje. Cuando un nodo ha recibido todas las respuestas, envía un BACK a su padre.

3.3 Algoritmo 5 – Broadcast

```
1: Initially do
2: begin:
3: if ps = pi then
4:     data = mensaje que se quiere difundir
5:     for each j ∈ children_i do send GO(data) to pj
6:     end for
7: else data = ∅
8: end if
9: end

10: when GO(data) is received from pj do
11: begin:
12: for each k ∈ children_i do send GO(data) to pk
13: end for
14: end
```

Explicación

El nodo raíz inicializa el mensaje y lo envía a sus hijos. Cada hijo que recibe el mensaje lo reenvía a sus propios hijos, asegurando que toda la red reciba la información.

3.4 Algoritmo 6 – Convergecast (Extra)

```
1: Initially do
2: begin:
3: v_i . Los valores que se enviarán
4: if children_i = ∅ then
5:     send BACK((i, v_i)) to parent_i
6: end if
7: end

8: when BACK(data) is received from each pj such that j ∈ children_i do
9: begin:
10: val set_i = ■_{j ∈ children_i} val set_j ∪ {(i, v_i)}
11: if parent_i ≠ i then
12:     send BACK(val set_i) to p_k
13: else
14:     the root ps can compute f(val set_i)
15: end if
16: end
```

Explicación

Las hojas envían su valor a su padre. Cada nodo combina los valores recibidos de sus hijos y los reenvía. Finalmente, la raíz obtiene toda la información para calcular una función global.

4. Prerrequisitos

Para correr la práctica primero deberán instalar python y simpy con el siguiente comando: sudo apt update sudo apt install python3 python3 --version pip install simpy En caso de que estén trabajando con un subsistema de Linux en Windows, puede que deban instalar antes pip: sudo apt-get install python3-pip Si el comando para instalar pip3 falla pueden intentar con: sudo apt-get update Para las pruebas unitarias es necesario instalar pytest con: pip3 install pytest

5. Pruebas automáticas con test.py

El archivo test.py incluye pruebas unitarias que validan la implementación de cada algoritmo. Se comprueban las variables identifiers (vecinos de vecinos), padre e hijos (árbol generador), y mensaje (broadcast). Para el algoritmo extra no hay prueba automatizada.

6. Nombres de variables esperadas

- identifiers (n.identifiers): lista donde cada nodo almacena los otros nodos que conoce. - padre (n.padre): referencia al nodo padre en el árbol generador. - hijos (n.hijos): lista de nodos hijos en el árbol generador. - mensaje (n.mensaje): mensaje difundido por el nodo distinguido en el algoritmo de broadcast.

7. Lineamientos de entrega

Se debe entregar un archivo Practica2.zip que contenga: - Código fuente completo. - Un ReadMe con número de práctica, integrantes y explicación de la implementación. - En caso de resolver el extra, incluir NodoConvergecast.py y, si es necesario, CanalConvergecast.py. Solo un integrante sube la práctica y el otro la marca como entregada.

8. Conclusiones

En esta práctica los alumnos aprenden a modelar la comunicación entre procesos con canales, implementar algoritmos básicos de propagación y recolección de información en redes distribuidas, y validar sus implementaciones mediante pruebas automatizadas. Estos conceptos son la base de protocolos reales en sistemas distribuidos.