Reporte de Practica

320079629 Gonzazlez Murga Alan Alexzander
320016431 Morales Aguilar Marcos
Materia Computación Distribuida

Descripción general del desarrollo

El desarrollo de la práctica consistió en la implementación de algoritmos distribuidos clásicos sobre una red simulada usando Python y la librería SimPy. Cada nodo de la red se modeló como una clase independiente, con canales de comunicación para el envío y recepción de mensajes. Se implementaron los siguientes algoritmos: construcción de árbol generador (flooding), convergecast, búsqueda distribuida y ordenamiento distribuido. El diseño modular permitió reutilizar componentes como los canales y las funciones auxiliares (`cuadricula` y `k_merge`), facilitando la extensión y pruebas de los algoritmos en diferentes topologías y tamaños de red.

Análisis detallado de la implementación

- **Árbol Generador (Flooding):**
  - El algoritmo de flooding, implementado en `NodoGenerador.py`, permite construir un árbol de difusión en la red. El nodo distinguido (id 0) inicia el proceso enviando mensajes `GO` a sus vecinos. Cada nodo que recibe un `GO` y no tiene padre, asigna como padre al remitente y reenvía `GO` a sus otros vecinos. Cuando un nodo ha recibido todos los mensajes esperados, envía un mensaje `BACK` a su padre. Así, cada nodo conoce su padre e hijos, formando el árbol generador.
  - El pseudocódigo del algoritmo se muestra a continuación:

Árbol Generador

```
1:  Initially do
2:  begin:
3:      if p_s = p_i then                                  ▷ Si soy el nodo distinguido
4:          parent_i = i; expected_msg_i = |neighbors_i|
5:          for each j ∈ neighbors_i do send GO() to p_j
6:          end for
7:      else parent_i = ∅                                  ▷ Si no, solo inicializo mis variables
8:      end if
9:      children_i = ∅
10: end

11: when GO() is received from p_j 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 p_j
16:         else
17:             for each k ∈ neighbors_i \ {j} do send GO() to p_k
18:             end for
19:         end if
20:     else send BACK(∅) to p_j
21:     end if
22: end

23: when BACK(val_set) is received from p_j 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
```

- **Convergecast:**
  - El algoritmo convergecast, implementado en `NodoConvergecast.py`, permite recolectar información desde las hojas del árbol hacia la raíz. Cada nodo mantiene un conjunto de valores (`val_set`) que inicialmente contiene su propio valor. Las hojas envían su valor a su padre mediante un mensaje `BACK`. Los nodos internos esperan recibir mensajes `BACK` de todos sus hijos, agregan los valores recibidos a su

conjunto y luego envían el conjunto actualizado a su padre. El nodo raíz, tras recibir todos los valores, aplica una función agregadora (`f`) para calcular el resultado final.

- El pseudocódigo del algoritmo se muestra a continuación:

Convergecast

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

8: when BACK(data) is received from each p_j 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 p_s can compute f(val_set_i)
15:    end if
16: end
```

- **Búsqueda distribuida:**
  - El arreglo a buscar se divide en segmentos usando la función `cuadricula` (`Auxiliares.py`), asignando a cada nodo una parte del arreglo. Cada nodo verifica si el elemento buscado está presente en su segmento y envía el resultado al nodo coordinador (id 0). El nodo 0 recolecta los resultados y determina si el elemento está presente en algún segmento usando la función `any`. Este método permite realizar búsquedas paralelas y reduce el tiempo total de búsqueda.
- **Ordenamiento distribuido:**
  - El arreglo se divide en segmentos usando la función `cuadricula`. Cada nodo ordena localmente su segmento y envía el segmento ordenado al nodo 0. El nodo 0 recolecta todos los segmentos ordenados y utiliza la función `k_merge` para realizar un *k-way merge*, obteniendo el arreglo final completamente ordenado. Este enfoque aprovecha el paralelismo y es escalable para grandes volúmenes de datos.
- **Consideraciones generales:**
  - Todos los algoritmos se implementaron usando SimPy para modelar la concurrencia y los canales de comunicación entre nodos.
  - El diseño modular facilita la extensión y pruebas de los algoritmos.
  - Las funciones auxiliares permiten dividir el trabajo y combinar resultados de manera eficiente.
  - La simulación facilita la validación y el análisis del comportamiento de los algoritmos en diferentes topologías y tamaños de red.