PDP - Exercício 2

Arthur Adolfo Gabriel Seibel

7 de Maio de 2019

1 Introdução

Este documento descreve uma solucão elaborada para a tarefa de reescrever o algoritmo de coleta em árvore usando o modelo assíncrono de sistemas distribuídos de Nancy Lynch (Lynch 1996). Este paradigma pressupõe processos *multithread*, cada um com uma *thread* que realiza *sends*, outra que realiza *receives*, e uma última que implementa a lógica específica da família de processos descrita em código.

O algoritmo em questão, também conhecido como Probe/Echo, é caracterizado por duas fases:

1. Difusão de pedido de informação (*probe*)
2. Coleta de informação (*echo*)

Na primeira fase, cada processo da árvore envia aos seus vizinhos (salvo o nodo "pai") uma mensagem de *probe*. Depois, na segunda fase, o processo espera seus filhos (nodos a quem ele enviou *probe anteriormente*) lhe retornarem um *echo* contendo alguma informação relevante (a topologia do grafo, por exemplo).

2 Pseudocódigo

O pseudocódigo descrito a seguir é feito à semelhança da linguagem de programação Kotlin. O funcionamento do mecanismo é explicado na sessão seguinte. No código é definida uma classe para representar a topologia do grafo, pois esta foi a escolha de informação a ser coletada pelo algoritmo. Em seguida, é descrita a classe Harvester, subclasse de uma suposta classe Process. É nesta classe "coletora" em que está modelado o algoritmo nos moldes do modelo em questão.

```
class Topology {
    ...
    combine(other: Topology) { ... }
}

class Harvester: Process {
    Signatures {
        Input {
            receive("probe", null) j,i
            receive("echo", topo : Topology) j,i
        }
        Output {
            start() i
        }
    }

    States {
```

```
    val starterId: Int = readStdin()

    val started: Bool = false

    val parentId: Int = 0

    var receivedEchos: Int = 0

    val topology = Topology(nbrs)

    // clear send buffers
    for (j in nbrs) {
        sendBuffer(j) = emptyQueue<Tag, Topology>()
    }
}

Transitions {
    start() i {
        Precondition:
            i == starterId
            started == false
        Effect:
            started = true

            // probe a todos os vizinhos
            for (nbr in nbrs)
                sendBuffer(nbr).push(["probe", null])
    }

    receive("probe", null) j,i {
        Precondition:
            started == true
        Effect:
            // que envia probe é pai de quem recebe
            parentId = j

            if (nbrs.length() >= 2) { // se há filhos, encaminha probe

                for (nbr in nbrs) {
                    if (nbr != parentId)
                        sendBuffer(nbr).push(["probe", null])
                }

            } else { // se não há filhos, envia echo ao pai

                sendBuffer(parentId).push(["echo", topology])

            }
    }

    receive("echo", topo: Topology) j,i {
        Precondition:
            started == true
        Effect:

            // combina topologia com a do filho que enviou echo
```

```
                    topology.combine(topo)

                    // verifica se se ja recebeu echo de todos os filhos
                    if (++receivedEchos == nbrs.length() - 1) {

                        if (i == starterId) { // se é processo iniciador, reporta e termina

                            report(topology)
                            started = false

                        } else { // se é processo normal, encaminha topologia coletada ao pai

                            sendBuffer(parent).push(["echo", topology])

                        }
                    }
                }
            }
        }
```

3 Funcionamento do Mecanismo

A classe Harvester define, como exige o modelo de programação de Lynch, *signature*, *states* e *transitions* de uma família de processos. Sua "assinatura", composta por declarações de transições, é dividida em *input* e *output*, de acordo com a natureza das declaracões. Os "Estados" do processo aglutinam variáveis e suas respectivas inicializações. Por último, são definidas de fato as transicões, invocando rotinas que são executadas se a precondição for verdadeira.

A transição "*start() i*" inicia o processo, executando quando o algoritmo ainda não tiver começado e se i (que identifica o processo a atual) for o starterId - que é lido da entrada padrão do programa na inicialização dos processos. A rotina dessa transição manda uma mensagem de "probe" para todos os vizinhos do nodo iniciador. Isso é feito por inserir um par [marcador, topologia] com marcador "probe" e topologia vazia na fila de envios para outro processo, fila esta tratada pela *thread sender*.

As duas transições classificadas como *input*, *"receive("probe", null) j,i"* e *"receive("echo", topo: Topology) j,i"*, podem executar quando a *thread receiver* do processo encontrar os pares [marcador, topologia] correspondentes às declaracões (desde o algoritmo já tenha iniciado). A primeira aceita apenas *receives* com marcadores "probe" e topologia null, enquanto que a segunda aceita *receives* com marcadores "echo" mas com qualquer topologia (variável "topo"). As variáveis j e i são, respectivamente, identicadores do processo que enviou e do que recebeu o *receive*.

O "efeito" da execução da transição *"receive("probe", null) j,i"* é de identificar o processo rementente como o pai do destinatário (que executa a transição), e de encaminhar para os vizinhos (salvo o pai), a mensagem de difusão; se o processo for um "nó folha", sem filhos, ele responde ao seu pai um echo com sua topologia. Já a *"receive("echo", topo: Topology) j,i"* é responsável por coletar e agregar echos e topologias dos filhos de um processo; se ele for o "nó raiz", iniciador do algoritmo, então a topologia dos filhos agregada é final e pode ser reportada, terminando a execução.

4 Referências

Lynch, Nancy A. 1996. *Distributed Algorithms*. Elsevier.