Práctica 2.
Chat Distribuido

Andrés Gavín Murillo 716358

Borja Aguado 741440

8 de noviembre de 2018

Introducción

Se ha desarrollado un chat distribuido, implementando un protocolo de interacción de difusión amplia (*multicast*) con ordenación total. Para ello se ha implementado en Elixir el algoritmo de *Ricart-Agrawala* utilizando, además, los relojes lógicos de *Lamport*.

Sección Principal

Resumen

El chat distribuido consiste en la ejecución del programa, desarrollado en Elixir, en distintas máquinas, donde cada usuario podrá enviar mensajes de texto y todos verán los mensajes en el mismo orden de envío.

Para la implementación del algoritmo de *Ricart-Agrawala*, se ha optado por traducir de la forma más exacta posible el algoritmo original [1]. Sin embargo, para garantizar el acceso atómico a ciertos elementos del programa, se ha implementado una base de datos y un semáforo como se muestra a continuación.

Arquitectura del sistema

Cada máquina ejecuta 6 procesos principales distintos (los mensajes de *reply* y *request* se resuelven en nuevos procesos), cuyas descripciones son las siguientes:

database

Es el proceso que almacena las variables globales del chat. Los procesos que quieran acceder a dichas variables envían una petición de lectura o de escritura. Se compone de la siguiente función:

```
# osn = our_sequence_number
# hsn = highest_sequence_number
# orc = outstanding_reply_count
# rcs = requesting_critical_section
# rd = reply_deferred
def database(osn, hsn, orc, rcs, rd) do
        …
end
```

binary_semaphore

Se trata de un semáforo implementado como proceso, los demás procesos lo utilizan para acceder a las variables compartidas. Se compone de la siguiente función:

```
def binary_semaphore(shared_vars,
pendiente) do
  receive do
    {pid, :wait} ->
      if shared_vars == 1 do
        send(pid, {:ok})
        binary_semaphore(0, [])
      else
        binary_semaphore(shared_vars, [pid])
      end
    {pid, :signal} ->
      if length(pendiente) == 0 do
        binary_semaphore(1, pendiente)
      else
        send(hd(pendiente), {:ok})
        binary_semaphore(0, [])
      end
  end
end
```

request_receiver

Proceso que se encarga de recibir las peticiones de otros nodos. Contesta con un *ack* inmediatamente si se cumplen las condiciones específicas explicadas en el algoritmo,

[1] Algoritmo de Ricart-Agrawala: https://dl.acm.org/citation.cfm?id=358537

y si no, añade a la cola el identificador del nodo que ha pedido la petición, y el proceso *mutual_exclusion_invoker* se encargará de mandar el *ack* una vez se haya salido de la sección crítica. Se compone de la siguiente función:

```
  defp request_receiver(pidmutex, piddb, me) do
    receive do # Waiting for a request
      {:request, k, j, idnode} ->
        spawn(fn ->
          # k is the sequence number begin requested, j is the node number making the request
          defer_it = false
          send(pidmutex, {self(), :wait})
          receive do
            {:ok} ->
              send(piddb, {:read, :hsn, self()})
              receive do
                {:hsn, hsn} ->
                  if hsn < k do
                    send(piddb, {:write, :hsn, k})
                  end
              end
              #defer_it = ( RCS && k>OSN ) || (k=OSN && j>me)
              send(piddb, {:read, :rcs, self()})
              receive do
                {:rcs, rcs} ->
                  send(piddb, {:read, :osn, self()})
                  receive do
                    {:osn, osn} ->
                      defer_it = (rcs and k>osn) or (k==osn and j>me)
                  end
              end
              # Defer_it will be true if we have priority over node j's request
              #if defer_it then Reply_deferred[j] = true
              if defer_it do
                send(piddb, {:read, :rd, self()})
                receive do
                  {:rd, lista_rd} ->
                    # modificada[j] = true
                    modificada = List.replace_at(lista_rd, j-1, true)
                    send(piddb, {:write, :rd, modificada})
                end
              else
                send({:reply_process, idnode}, {:reply})
              end
              send(pidmutex, {self(), :signal})
          end
        end)
    end

    request_receiver(pidmutex, piddb, me)
  end
```

reply_receiver

Proceso encargado de recibir los *ack* de los distintos nodos y actualizar la variable que controla los *acks* que quedan por recibir. Se compone de la siguiente función:

```
  defp reply_receiver(pidmutex, piddb, pidinvoker) do
    receive do # Waiting for a reply
      {:reply} ->
        spawn(fn ->
          send(pidmutex, {self(), :wait})
          receive do
            {:ok} ->
              send(piddb, {:read, :orc, self()})
              receive do
                {:orc, orc} ->
                  if orc == 1 do
                    send(pidinvoker, {:reply_end})
                  end
                  send(piddb, {:write, :orc, orc - 1})
              end

              send(pidmutex, {self(), :signal})
          end
        end)
    end

    reply_receiver(pidmutex, piddb, pidinvoker)
  end
```

chat_receiver

Proceso que recibe mensajes de chat y los imprime por pantalla con el formato “{nodo}: {mensaje}”. Se compone de la siguiente función:

```
  defp chat_receiver() do
    receive do
      {:message, idnodo, msj} ->
        IO.puts "#{idnodo}: #{msj}"
        chat_receiver()
    end
  end
```

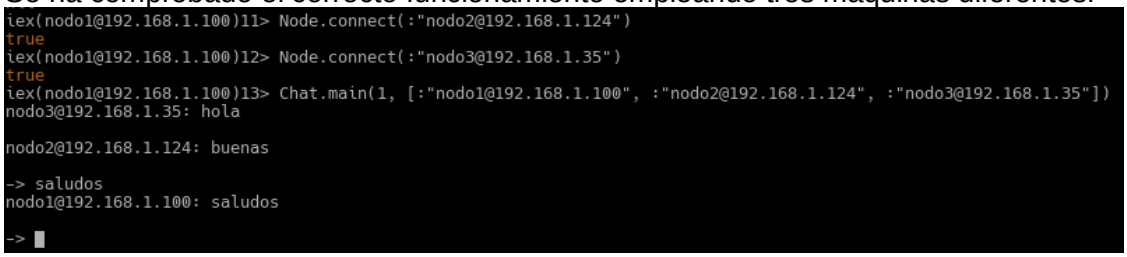
mutual_exclusion_invoker

Proceso principal. Se trata de la traducción del proceso principal del algoritmo. Cuando el usuario introduce un mensaje que desea enviar, accede a la base de datos modificando las variables globales, para indicar a los demás procesos que desea entrar en SC para enviar el mensaje. Invoca a la función *all_nodes_sender* para enviar peticiones a los demás nodos, y cuando recibe todos los *ack*, envía el mensaje. Se compone de las siguientes funciones:

```
  defp all_nodes_sender([node | nodes], msj_type, msj, sendMe)
  defp post_protocol([node | nodes], [rd | rds], result, piddb) do
  def mutual_exclusion_invoker(pidmutex, piddb, me, n, nodes) do
    message = IO.gets "-> "
    # Pre-protocol
    send(pidmutex, {self(), :wait})
    receive do
      {:ok} ->
        send(piddb, {:write, :rcs, true})
        send(piddb, {:read, :hsn, self()})
        receive do
          {:hsn, hsn} -> send(piddb, {:write, :osn, hsn + 1})
        end
        send(piddb, {:write, :orc, n - 1})
        send(piddb, {:read, :osn, self()})
        receive do
          {:osn, osn} ->
            all_nodes_sender(nodes, :request_process, {:request, osn, me, Node.self()}, false)
            # Sent a REQUEST message
        end
        send(pidmutex, {self(), :signal})
    end
    # Now wait for a REPLY
    receive do
      {:reply_end} ->
        # Critical Section
        all_nodes_sender(nodes, :chat_process, {:message, Node.self(), message}, true)
        # Release the Critical Section
    end
    # Post-protocol
    send(pidmutex, {self(), :wait})
    receive do
      {:ok} ->
        send(piddb, {:read, :rd, self()})
        receive do
          {:rd, rd} ->
            post_protocol(nodes, rd, [], piddb)
        end
        send(piddb, {:write, :rcs, false})
        send(pidmutex, {self(), :signal})
    end
    mutual_exclusion_invoker(pidmutex, piddb, me, n, nodes)
  end
```

Validación

Se ha comprobado el correcto funcionamiento empleando tres máquinas diferentes:



Conclusiones

Se ha observado cómo el algoritmo de *Ricart-Agrawala* es eficaz a la hora de garantizar el acceso a la sección crítica (envío de mensaje) de todos los nodos del sistema. El problema de este algoritmo es que, si cualquiera de los nodos falla por cualquier motivo, el sistema entero deja de funcionar, ya que para que los nodos accedan a la sección crítica es necesario que el resto envíe un mensaje de *ack*. Si uno solo de ellos no responde, el nodo se bloquea y no deja a los demás avanzar.