Concurrent Programming

Bernard Sufrin

Hilary Term 2016-17

7a: A Ring-Structured Database

Supplementary Reading: A.W. Roscoe, *The Theory and Practice of Concurrency*, *pp* 423-430

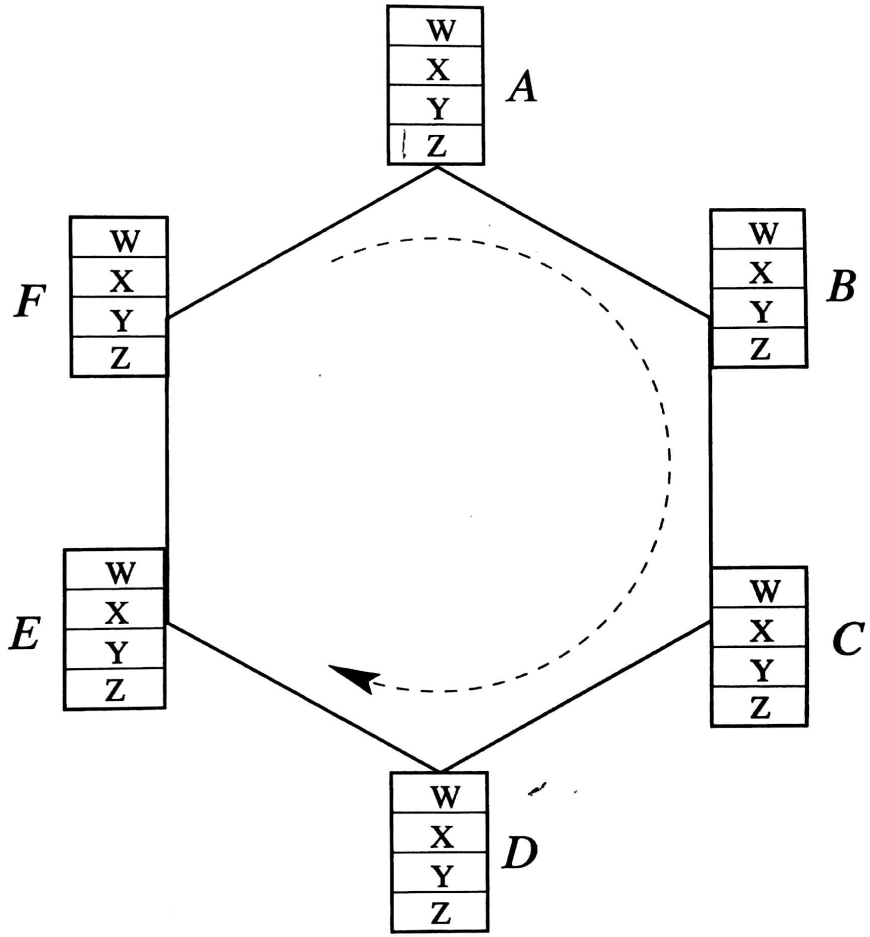
Supplementary Reading: A.W. Roscoe, *Maintaining Consistency in Distributed Databases* (course website)

Introduction

* N active nodes, with distinct identifiers $id : PID$ are organized in a ring
* Each holds a replica of a finite mapping of type `Map[K,V]`
* Each can update its own replica
* Each can send update messages to its successor
* The nodes can reside on different computer systems
* Updates to the mappings are executed locally (at individual nodes)
* Updates to the mappings are communicated round the ring
* The replicas are to be kept as nearly synchronized as is practical, *to wit:*

  if all the nodes stop updating for a while, the replicas should "soon" become identical
* Nodes are composed of an (application-oriented) client process and a ring interface process

Towards an algorithm: the problem



* Updates $\{k \mapsto v\}$ do one lap of the ring, without overtaking, then stop
* Problem: near-simultaneous updates can be seen in different orders, *e.g.*

  $\{X \mapsto V_A\}@A$
  $\{X \mapsto V_D\}@D$

  at the same time can result in different updates at different processes

  $B : \{X \mapsto V_D\}$
  $F : \{X \mapsto V_A\}$

Simple solution: a Token-Ring – first attempt

* A `Token` is either `Empty` or an `Update(pid: PID, k: K, v: V)`
* There is exactly one `Token` in circulation, implemented by the ring interface processes
* If an update is (soon) available when an empty token arrives it is injected into the ring

```
def ringInterface[K,V,PID]
    (pid: PID, fromClient: ?[(K,V)], map:  Map[K,V],
                pred:      ?[Token], succ: ![Token]) = proc(s"I$pid")
{ def inject =
   alt( fromClient         =?=> { case (k:K, v:V) => succ!Update(pid, k, v) }
      | after(100*microSec) ==> { succ!Empty }
      | orelse              ==> { succ!Empty }
      )
  repeat
  { pred?{ case Empty => inject
           case tok@Update(p:PID, k:K, v:V) =>
                if (p==pid) succ!Empty else { map+=((k,v)); succ!tok }
         }                  //////////
  }
  pred.closeIn; succ.closeOut; fromClient.closeIn
}
```

Observations

* Client-Interface architecture means mapping has to be thread-safe.

* If an interface sends and receives `Empty` on successive turns there are no updates in transit (though some may have already have been buffered by clients). This gives us a handle on deciding whether the ring has become quiescent.
    - Special-purpose nodes can be used to monitor traffic, to store the mapping stably if it becomes quiescent, *etc*.
    - A (special-purpose) node can delay restarting updates by delaying passing the empty token on – simulating quiescence.

* Inter-node interference can be avoided by ensuring that only a single node has the right to update each key.

Fixing the Token Ring Bug

* Our implementation has at least one serious bug
* Consider the ring: A, B, C, D, E, F evolving as follows:
  - A starts the update $k \mapsto V_A$ circulating.
  - It updates C after C has locally mapped $k \mapsto V_C$ *and queued the update* $k \mapsto V_C$.
  - The update $k \mapsto V_A$ finishes circulating
  - The update $k \mapsto V_C$ starts circulating (from C)
  - The ring quiesces
  - All but C now map $k \mapsto V_C$, but C now maps $k \mapsto V_A$
* Solution: when an update for k passes a ring interface from which it didn't originate it should *both* update the local mapping, *and* eliminate any (other) value for k waiting for the token.
* There is also an unnecessary potential delay within `inject`

* We replace the independent `fromClient` and `map` structures for communicating with clients by a unitary thread-safe map-like structure shared between client and interface
* Clients use
  - `put(k,v)` – to change the mapping at k and enqueue that update for circulation
  - `get(k)` – to acquire the value of the mapping at k

```
class QMap[K,V]
{ /** The queue of keys of updates waiting to be distributed */
  protected val queue  = new collection.mutable.LinkedHashSet[K]

  /** The mapping */
  protected val map  = new collection.mutable.HashMap[K,V]
  /* Invariant: if k is in the queue, then map is defined at k */

  /** Client changes local mapping; key joins queue of outgoing updates */
  def put (k: K, v: V): Unit = synchronized { queue += k; map += ((k,v)) }

  /** Client interrogates local mapping */
  def get (k: K): Option[V] = synchronized { map.get(k) }
```

* The ring interfaces use:
  - `dequeue()` – to acquire the next update (if any) from the queue
  - `remap(k, v)` – to change the mapping at k to v, and *remove* any queued updates to k from the queue

```
/** Interface interrogates the queue of outgoing updates */
def dequeue: Option[(K,V)] = synchronized
{ if (queue.isEmpty) None else
  { val first = queue.head
    queue.remove(first)
    Some((first, map(first)))
  }
}

/** Interface forces change to local mapping  */
def remap(k: K, v: V) = synchronized
{ queue.remove(k); map += ((k,v)) }
```

}

* The correct(ed) ring interface:

```
def ringInterface[K,V,PID]
    (pid: PID, map: QMap[K,V], pred: ?[Token], succ: ![Token])=proc(s"I$pid")
{ def inject = map·dequeue match
      { case None          ⇒ succ!Empty
        case Some((k, v))  ⇒ succ!Update(pid, k, v)
      }

  repeat
  { pred?{ case Empty ⇒ inject
           case tok@Update(p: PID, k: K, v: V) ⇒
             if (p==pid) succ!Empty // update finished circulating
             else
             {  map·remap(k, v)     // update might supersede a waiting one
                succ!tok
             }
         }
  }
  pred·closeIn; succ·closeOut
}
```

Cursory Performance Analysis

* Once an update is accepted from a client it takes one token-lap to distribute it
* When all N clients are active *and have buffered updates with different keys*
  - each lap of a token delivers an update from a single client
  - the next lap delivers an update from the next client
  - maximum update delivery rate *per client* is one per N token-laps $= \frac{1}{N^2}$ message times
  - most of the channels spend most of the time doing nothing

Roscoe's Algorithm

* Strategy to enable more updates to be circulating simultaneously
  - Each node updates its own mapping locally (as before)
  - Circulating updates may not overtake each other (as before)
  - Each node retains a queue of outgoing updates (as before)
  - Each node has a queue, E of updates $(k : K, v : V)$ that it has injected into the ring and is expecting back
  - Nodes, and by extension the updates they originate, have *priorities*
  - Consistency is maintained in the presence of multiple clashing updates to the same key by *stopping* the circulation of lower priority clashing updates and *cancelling* their expected-back versions.

Consider the behaviour of the interface at node pid:

* If there is space available on the ring and an update has been queued then de-queue it, inject it into the ring, and enqueue it on E_{pid}

* If an `Update`(pid, k, v) arrives, then remove it from the front of E_{pid}.
  It has finished its lap of the ring.

* If an `Update`(pid', k, v) arrives from a different node,
  and there is no clashing update (k, v') in E_{pid}
  - the arriving update is executed locally and passed on round the ring
  - any clashing outgoing updates queued at pid are deleted

(note that this is almost the same as the token ring)

* (Stopping lower priority clashing updates)
  If lower priority `Update`(pid', k, v) arrives from pid'
  and there is a clashing update (k, v') in E_{pid}
  ◦ the arriving update is stopped: *i.e.* not passed on round the ring.

* (Cancelling lower priority expected-back updates)
  If a higher priority `Update`(pid', k, v) arrives from pid'
  and there are clashing expected updates (k, v') in E_{pid}
  ◦ the arriving update is executed locally and passed on round the ring
  ◦ any clashing outgoing updates queued at pid are deleted
  ◦ any clashing expected updates in E_{pid} are removed

Contents

Note 1: A test rig for the first token ring attempt 4 ☞

```
import io·threadcso·_
import scala·collection·concurrent·{Map, TrieMap}
object tokenring1
{ trait Token {}
  case object Empty extends Token
  case class  Update[K,V,PID](pid: PID, k: K, v: V) extends Token

  @inline def elapsedTime: Long = nanoTime-startTime
  type VAL = (String, Int, String)

  ··· Client Definition
  ··· Probe Definition
  ··· Ring Interface Definition
  ··· Main Program Definition
}
```

This simple client process associates a sequence of timestamped numbers with a single key, pausing from time to time, then stops.

```
def client(name: String, init: Int, toInterface: ![(String,VAL)], map: Map[String,VAL]) = proc(name)
{  val key=name·substring(0, 1)
   var n=init
   repeat (n<15)
   { val upd = (key, (name, n, elapsedTime·hms))
     n   += 1
     map += upd
     sleep(seconds((if (n%3==0) 4·0 else 0·9)))
     toInterface!upd
   }
   println(s"$name␣(started␣from␣$init)␣terminated␣@${elapsedTime·hms}")
   toInterface·closeOut
}
```

This probe process forwards tokens round the ring. It invokes `quiesce` and doubles em whenever em consecutive empty tokens have been forwarded. It closes the ring down if `stopNS` nanoseconds have elapsed since the last non-empty token went past.

```
def probe(quiesce: ⇒ Unit, stopNS: Long, pred: ?[Token], succ: ![Token]) = proc("probe")
{ var lastUpdate = nanoTime
  var ec         = 0l         // most recent number of adjacent Empty tokens
  var em         = 100l       // invariant:  ∃ n · em = 100 · 2^n
  var probing    = true
  while (probing)
  { val elapsed = nanoTime-lastUpdate
    if (elapsed > stopNS) probing=false else
    pred ? {  case Empty ⇒
                   ec += 1
                   if (ec==em) { em += em; print(s"$ec␣${(elapsed)·hms}:␣"); quiesce }
                   succ ! Empty
              case update ⇒
                   lastUpdate = nanoTime
                   ec = 0
                   em = 100
                   succ ! update
           }
  }
  pred·closeIn; succ·closeOut
}
```

The main program constructs a ring of nodes and starts it running by injecting a single empty token. If there's a probe in the ring, then that closes the ring down after it has been quiescent for 10 seconds.

```
def main(args: Array[String]): Unit =
{ val chans = for (arg←args) yield OneOne[Token](s"$arg->")
  val maps  = for (arg←args) yield new TrieMap[String, VAL]
  def showMaps {
      println(s"@${elapsedTime.hms}\n${args(0)}:␣${maps(0)}")
      for (i←1 until args.size if args(i)!="probe" && maps(i) != maps(0))
          println(s"${args(i)}:␣${maps(i)}")
  }
  val ring =
  || { for (i←0 until args.size) yield {
       val name = args(i)
       val map = maps(i)
       val toI = OneOneBuf[(String, VAL)](2, s"$name␣to␣I$name")
       if (name=="probe") probe({showMaps}, seconds(10), chans(i), chans((i+1)%chans.size)) else
       (  client(name, i, toI, map)
       || ringInterface(name, toI, map, chans(i), chans((i+1)%chans.size))
       )
     }}
  println(debugger)
  run(proc {chans(0)!Empty} || ring)
  println(s"Terminated␣@${elapsedTime.hms}")
  showMaps
  exit
}
```

Note 2: Aspects of the preliminary token-ring solution worth discussing 4 ☞

Fairness: what happens if a returning update is replaced by another update by invoking `inject` in the ring interface (see the listing at /////). Is there potential for starvation?

Update latency: how long does it take an update to reach every interface?

Buffering of the channel from the client: can this induce unfairness? WOuld it even be correct if it were fair?

Utilization of inter-interface channels forming the ring.

Termination: by consensus or in response to an error.

Testing.

Interference between client updates.

Correctness: Do all processes see the same mapping when the system is quiescent?

Note 3: 9 ☞
The test rig for the corrected token ring interface is similar to that for the earlier attempt. See `tokenring2.scala`.

Note 4: 12 ☞
Roscoe observes that it is by no means obvious that a circulating update that originated at pid will still be at the front of E_{pid} when it returns; but he goes on to prove this fact. We will take it for granted.