16-124-836
Marcel ZAUDER

5.1 Regular Register Executions

(a) Algorithm 4.1: Read-One Write-All

We have one write process p, a reader process q which is divided into two processes q_1 and q_2 initialized with the storage of $\perp$. First p will call the *write* operation which because it is not concurrent will be acknowledged and q_1 and q_2 store the yalue of x. Therefore the *read* operation which is sequential after the *write* operation and will therefore return x because both q_1 and q_2 (so it does not matter which of them is read) have stored this value. We asume that the second *read* operation is called by q_1 but before it has finished process p is calling a *write* operation to store the value of y. The process q_1 has already stored this value before the *read* operation terminates and will therefore return y. The third *read* operation is called by q_2 which is a little bit slower to write the new value and is therefore returning the value of x before it will overwrite its value and acknowledge it.

(b) Algorithm 4.2: Majority Voting Regular Register

We have one write process p, a reader process q and five processes s_1, s_2, s_3, s_4 and s_5 initialized with the storage of $\perp$. The sequential operations *write* and *read* will terminated as expected so now every process will have stored the value x with the timestamp 1. Before the next *read* operation terminates only s_1 have stored the value of y with the timestamp 2 from the concurrent *write* operation. Process q will receive the messages from s_1, s_2, s_3 and s_4 and because the timestamp of s_1 is the highest it will return the value of y. For the third *read* operation the process q will receive messages from s_2, s_3, s_4 and s_5. Because all of them still have stored the value of x because the *write* operation has not terminated the *read* operation will return x as well.

16-124-836
Marcel ZAUDER

5.2 Read-All Write-One Regular Register

Implements:
(1,N)-RegularRegister, **instance** *onrr*

Uses:
BestEffortBroadcast, **instance** *beb*
PerfectPointToPointLinks, **instance** *pl*
PerfectFailureDetector, **instance** $\mathbb{P}$

upon **event** $\langle onrr, Init \rangle$ do
  $val \leftarrow \bot$
  $correct \leftarrow \Pi$
  $wid \leftarrow 0$
  $rid \leftarrow 0$
  $readList \leftarrow []$

upon **event** $\langle \mathbb{P}, Crash \mid p \rangle$ do
  $correct = correct \setminus \{p\}$

upon **event** $\langle onrr, Read \rangle$ do
  $rid = rid + 1$
  trigger $\langle beb, Broadcast \mid [\text{READ}, rid] \rangle$

upon **event** $\langle onrr, Write \mid v \rangle$ do
  $val \leftarrow v$
  $wid := wid + 1$
  trigger $\langle onrr, WriteReturn \rangle$

upon **event** $\langle beb, Deliver \mid q, [\text{READ}, rid] \rangle$
  trigger $\langle pl, Send \mid q, [\text{VALUE}, rid, wid, val] \rangle$

upon **event** $\langle pl, Deliver \mid q, [\text{VALUE}, r, id', v'] \rangle$ s.t. $r = rid$
  $readList[q] \leftarrow (id', v')$
  if $\#(readList) == \#(correct)$ then
    $v = highestval(readList)$
    $readList = []$
    trigger $\langle onrr, ReadReturn \mid v \rangle$

5.3 (1,1) Atomic Register

A reader process can update its (wts, val) pair in the last step of the *ReadReturn*, after the if condition is fulfilled. After the value with the highest timestamp is chosen the process can update its pair with these values so it is up to date again.