## 5.1 (new) Regular Register Executions

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

#### Execution A

The execution A is not possible for the read-one write-all algorithm because a *read* operation will always return the locally stored value. Hence if it is returning the new value a *read* operation sequentially after the first process on the same process will also return the new value and can't return the old one.

#### Execution B

The non-concurrent *write* and *read* operation working as expected and will follow the validity property. Therefore any healthy process will have written and stored the new value.
Now we consider the two concurrent read operations. Furthermore we assume that $p.Write(y)$ to happen immediately after it starts. The *read* operation of $r$ can therefore return the value $y$. As process $q$ might be a bit slower, as the *read* operation is caled it still can return the value $x$ because it has not received the or terminated the *write* operation.

### (b) Algorithm 4.2: Majority Voting Regular Register

#### Execution A

Again the non-concurrent *read/write* operations work according to the validity property.
For the two *read* operation which are concurrent with the *write* operation we assume the following: As $q$ is first reading it receives a message from $q$ and $p$. Process $q$ still has the value of $x$ stored but $p$ has already written $y$. Because $y$ will have the higher timestamp the *read* operation will return $y$. For the second *read* operation $q$ will receive messages from $q$ and $r$ which is also a moajority but both have not finished writing the new value and therefore returning $x$ hence the *read* operation is returning it as well.

#### Execution B

This execution is similar to the Execution A. Process $r$ can receive a message from a process which has already written the new value $y$ and process $q$ receives only messages of processes that don't.

## 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 $\langle beb, Broadcast \mid [\textsc{Read}, rid] \rangle$

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

upon event $\langle beb, Deliver \mid q, [\textsc{Read}, rid] \rangle$
  trigger $\langle pl, Send \mid q, [\textsc{Value}, rid, wid, val] \rangle$

upon event $\langle pl, Deliver \mid q, [\textsc{Value}, r, id', v'] \rangle$ 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

We can modify a reader process which is implemented as the Algorithm 4.2 such that it will store the latest value. The *read* operation is executed as described in Algorithm 4.2 but before the *ReadReturn* it can compare the value with the highest timestamp it received with the $(wts, val)$ pair it has stored. If the timestamp it has received is higher than the one it has stored it can update its pair. In the end it will return the value with the highest timestamp and hence we implemented an algorithm for a (1,1) ATOMIC register.