4.1 Emulating a (1,N) register from (1,1) registers

(a) Emulation is SAFE if $br.q$ are SAFE *binary* (1,1)-registers

> **SAFE:** A read() not concurrent with a write, returns the value written by the most recent write() operation.

Upon the event $\langle br.q\text{-}WriteReturn \rangle$ the event $\langle onr\text{-}WriteReturn \rangle$ is only triggered if all $q \in \Pi$ have triggered $\langle br.q\text{-}WriteReturn \rangle$. Therefore when onr will acknowledge that it has written, every process q must have finished the writing event and hence every q stores the same value.
Upon the event $\langle onr\text{-}Read \rangle$ which is not concurrent to any write operation the return value will be the right one because it does not matter which $br.q$ is read because all of them will store the same most recent written value.
Therefore the emulation is SAFE.

(b) Is emulation REGULAR if REGULAR *binary* (1,1)-registers are used?

> **REGULAR:** A read() not concurrent with a write returns the most recently written value. Otherwise read() returns the most recently written value or the concurrently written value.

***First Case* - NOT CONCURRENT:**
If the *write* and *read* events are not concurrent the same argumentation holds as in (a).

***Second Case* - CONCURRENT:**
We assume that the new and old value differ (otherwise it would be trivial):
Because we only use binary registers and w.l.o.g. the new written value is 1 and the already stored value is 0 a concurrent read event will either return 0 or 1. Therefore the REGULAR assumption holds and the emulation is indeed a REGULAR *binary* (1,N)-register.

(c) Is emulation REGULAR *multi-valued* if REGULAR *multi-valued* (1,1)-registers are used?

> **REGULAR:** A read() not concurrent with a write returns the most recently written value. Otherwise read() returns the most recently written value or the concurrently written value.

***First Case* - NOT CONCURRENT:**
If the *write* and *read* events are not concurrent the same argumentation holds as in (a).

***Second Case* - CONCURRENT:**
We assume that the new and old value differ (otherwise it would be trivial):
Because we use q which are REGULAR *multi-valued* (1,1)-registers upon the trigger $\langle br.q-Read\rangle$ they will return either the most recent or the concurrent written value. Therefore the $\langle onr - Read\rangle$ will either return the most recent or the concurrent written value and hence is a REGULAR *mult-valued* (1,N)-register.

4.2 Multivalued register from binary registers

MVR:

Reg[0, 1, ..., k] init. to [1, 0, ..., 0]

upon Read()
  for $j = 0$ to k do
    if Reg[j].Read == 1 do
    return j

upon Write(v)
  Reg[v].Write(1)
  for $j = v - 1$ to 0 do
    Reg[j].Write(0)

The Read()-operation will search for the first 1 in the array, and returns its index.
The Write(v)-operation will write 1 in the register with index v. Then it will start "cleaning" the array (set all to zero) in reverse order beginning from the newly written register - because the Read-operaton will only return the first index of the register with a one, the registers coming after this certain register does not matter. Therefore it is ensured that either the register contains the old value (if the new value is greater than the old value and the algorithm has not finished cleaning) or the ne value (if the new value is smaller than the old value or the algorithm has finished cleaning).

4.3 Register emulations without correct majority?

An eventually perfect failure detector can suspect processes of failing even though they did not fail (as it will only be perfect at some point in the future). We already know that the Majority Voting Regular Register will be wrong if the Correct Majority assumption is violated. Initially the system may behave as in the Faile Silent Mode, because the failure Detector can initially be wrong. (Processes can Fail, but the Failure detector does not notice it).
Therefore the same assumptions as in the Fail Silent mode are needed during the initial period. If we don't have these assumptions, the same things can go wrong as in the fail-silent mode.