Institute of Computer Science, University of Bern
Distributed Algorithms, Spring 2020

Prof. Christian Cachin
Jovana Mićić

Solution for exercise 5

5.1 Regular register executions (4pt)

Figures 1 and 2 show two examples of regular register executions. Process p does two *write* operations; processes q and r together execute three *read* operations, where the last two overlap with the *write* operation of p.

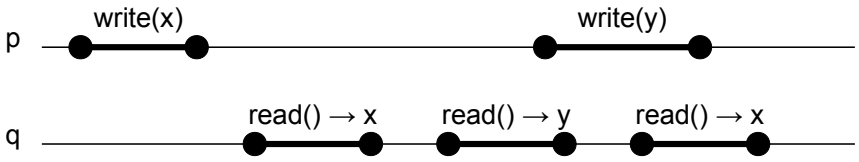

Figure 1. Execution A of a regular register.

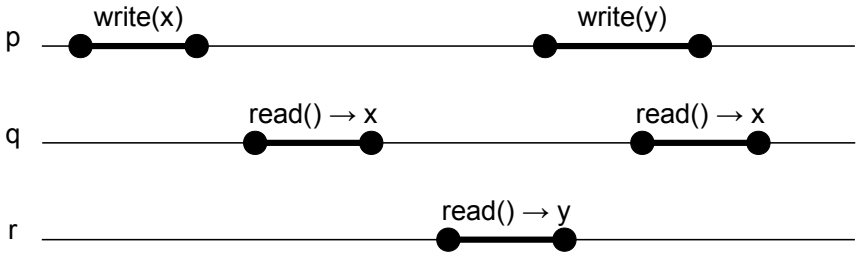

Figure 2. Execution B of a regular register.

Your task is to describe (or draw) execution steps of

a) Algorithm 4.1: Read-One Write-All

b) Algorithm 4.2: Majority Voting Regular Register

of [CGR11] that result in the scenarios described in the figures, if possible, or to conclude that this is not possible.

Solution. Execution A: Algorithm 4.1. Figure 3 shows execution steps of Algorithm 4.1 for execution A. The steps are:

1. Process p writes x and broadcasts a message [WRITE, x]. Upon receiving this message, each process stores x in the local variable *val* and sends a message [ACK].
2. Process q reads from the local variable *val* and returns x.
3. Process p writes y and broadcasts a message [WRITE, y]. Upon receiving this message, each process stores y in *val* and sends a message [ACK].

4. Process q reads concurrently to *write*(y). Because the register is regular, this read may return x or y. In this execution it returns y.
5. Process q reads from the local variable *val* and returns y.

Conclusion: Execution in Figure 1 cannot happen when Algorithm 4.1 is executed.

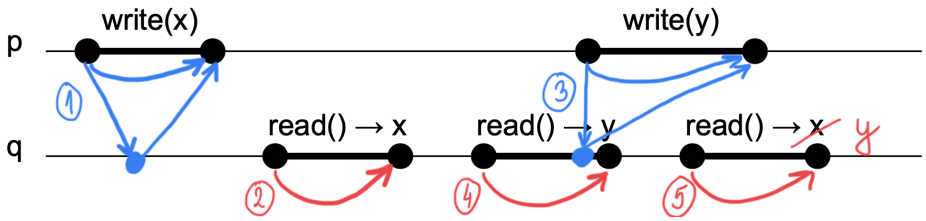

Figure 3. Execution steps of Algorithm 4.1 in Execution A.

Execution A: Algorithm 4.2. Figure 4 shows execution steps of Algorithm 4.2 for execution A. The steps are:

1. Process p writes x and broadcasts a message [WRITE, 1, x]. Upon receiving this message, every process stores locally a tuple $(1, x)$ and sends a message [ACK]. Writing is over when more than $\frac{N}{2}$ [ACK] messages are received.
2. Process q broadcasts a message [READ, 1]. Upon receiving this message, every process sends a message [VALUE, 1, 1, x]. Reading terminates when the process receives more than half of the VALUE-messages. The value with highest writing timestamp is read, which is x.
3. Process p writes y and broadcasts a message [WRITE, 2, y]. Upon receiving this message, every process stores locally a tuple $(2, y)$ and sends a message [ACK].
4. Process q broadcasts a message [READ, 2]. Upon receiving this message, every process sends a message [VALUE, 2, 2, y] and q waits for receiving this message from both. The value with highest write-timestamp is read, which is y.
5. Process q broadcasts a message [READ, 3]. Upon receiving this message, every process sends a message [VALUE, 3, 2, y] and q waits for receiving this message from both. The value with highest write-timestamp is read, which is y.

Conclusion: Execution in Figure 1 cannot happen when Algorithm 4.2 is executed.

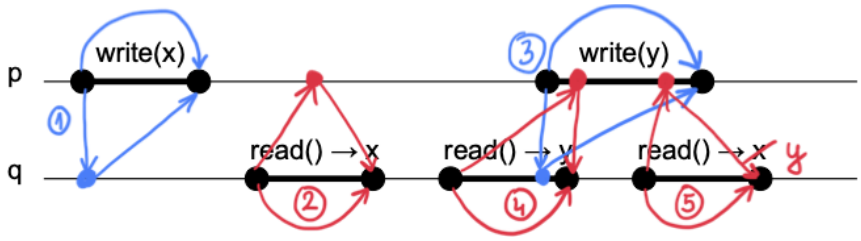

Figure 4. Execution steps of Algorithm 4.2 in Execution A.

Execution B: Algorithm 4.1. Figure 5 shows execution steps of Algorithm 4.1 for execution B. The steps are:

1. Process p writes x and broadcasts a message [WRITE, x]. Upon receiving this message, each process stores x in the local variable *val* and sends a message [ACK].
2. Process q reads from the local variable *val* and returns x.
3. Process p writes y and broadcasts a message [WRITE, y]. Upon receiving this message, processes p and r store y in *val* and send a message [ACK].
4. Process r reads concurrently to *write*(y). Because the register is regular, this read may return x or y. In this execution it returns y.
5. Process q reads from the local variable *val* and returns x.
6. Process q receives the message [WRITE, y] and stores y in *val* and sends a message [ACK].

Conclusion: Because the message [WRITE, y] is delivered with a delay to process q, q returns x in step 5, so execution B can happen.

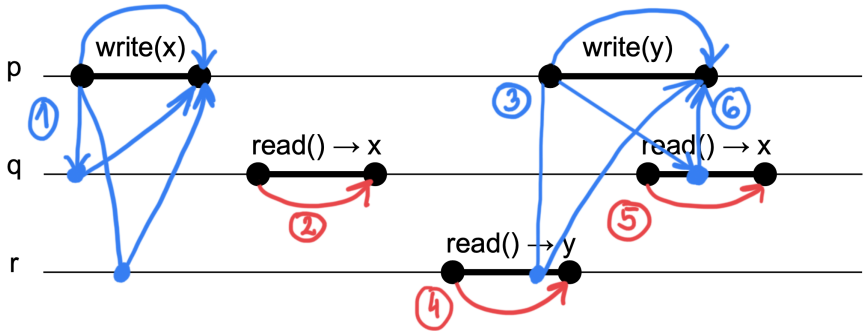

Figure 5. Execution steps of Algorithm 4.1 in Execution B.

Comment: *A closer look at Algorithm 4.1 reveals that the* read *operation consists of a single event handler, which is executed upon the invocation of the* read *operation. Therefore, the second* read$() \rightarrow x$ *by* q *should correspond to a single event that occurs either before or after* q *receives the [WRITE,* y*] message, and* q *might also return* y*. The reason is that according to the convention handling events, they are not executed concurrently.*

Execution B: Algorithm 4.2. Figure 6 shows execution steps of Algorithm 4.2 for execution B. The steps are:

1. Process p writes x and broadcasts a message [WRITE, 1, x]. Upon receiving this message, every process stores locally a tuple $(1, x)$ and sends a message [ACK]. Writing is over when more than $\frac{N}{2}$ [ACK] messages are received.
2. Process q broadcasts a message [READ, 1]. Upon receiving this message, every process sends a message [VALUE, 1, 1, x]. Reading terminates when the process receives more than half of the VALUE-messages. The value with the highest write-timestamp is read, which is x.
3. Process p writes y and broadcasts a message [WRITE, 2, y]. Upon receiving this message, every process stores locally a tuple $(2, y)$ and sends a message [ACK]. In this execution WRITE messages for processes p and q are delayed.
4. Process r broadcasts a message [READ, 2]. Upon receiving this message, process r sends a message [VALUE, 2, 2, y] and process q sends a message [VALUE, 2, 1, x]. As soon

as r receives two VALUE-messages, the value with the highest write-timestamp is read, which is y. It is because only messages from 2 out of 3 processes are needed to complete *read()* operation.

5. Process q broadcasts a message [READ, 3]. Upon receiving this message, processes p and q send a message [VALUE, 3, 1, x]. Note that the messages [WRITE, 2, y] from p to p and q have not yet arrived at p and q.
6. Processes p and q receive the [WRITE, 2, y] message from p and send [ACK] to p. After receiving two [ACK] messages, the write of p terminates.
7. Subsequently, q receives the messages [VALUE, 3, 1, x] from p and q. Since 2 out of 3 messages are enough for finishing operation, *read*() returns x.

Conclusion: Execution B can happen because of a message delays.

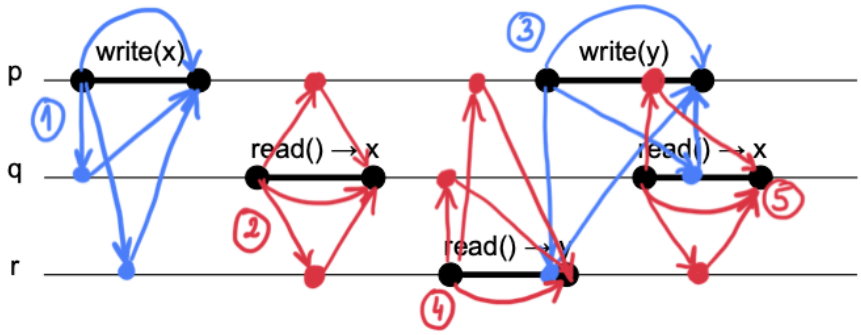

Figure 6. Execution steps of Algorithm 4.2 in Execution B.

5.2 Read-all write-one regular register (3pt)

Implement an algorithm providing a $(1, N)$ regular register. Your protocol should be similar to Algorithm 4.1 [CGR11, Sec. 4.2.2]), but instead follow a *read-all write-one* approach. It should use the *fail-stop model*, where a perfect failure detector is available. When a process crashes, the failure detector ensures that eventually all correct processes detect the crash (*strong completeness*), and no process is detected to have crashed until it has really crashed (*strong accuracy*).

Solution. Algorithm 1 shown in the pseudocode implements this approach. To ensure correctness, it has to satisfy *safety* and *liveness*. When no *write* operation executes, the algorithm has to implement safe register semantics. According to the protocol, every correct process then stores the most recently written value in its variable *val*. Hence, any subsequent *read* operation obtains this in at least one VALUE-message and returns it. The subtle point is that this works even if the writer has crashed. (Recall that implementations registers would actually be used between clients as processes that read and write and server replicas, which store values; in our formulation the clients are contained in the set of processes.)

Furthermore, liveness holds for the reader because it is a local operation. Based on the properties of the perfect failure detector, every crashed process is eventually detected and removed from *correct*. Therefore, also every *write* operation terminates.

Algorithm 1 Read-All Write-One.

Implements:

(1, N)-RegularRegister, **instance** *onrr*.

Uses:

BestEffortBroadcast, **instance** *beb*.

PerfectPointToPointLinks, **instance** *pl*.

PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event $\langle$ *onrr-Init* $\rangle$ **do**

val $:= \perp$;

ts $:= 0$;

readset $:= \emptyset$;

correct $:= \pi$;

upon event $\langle$ $\mathcal{P}$-*Crash* | p $\rangle$ **do**

correct $:=$ *correct* $\setminus \{p\}$;

upon event $\langle$ *onrr-Write* | v $\rangle$ **do**

ts $:=$ *ts* $+ 1$;

val $:= v$;

trigger $\langle$ *onrr-WriteReturn* $\rangle$;

upon event $\langle$ *onrr-Read* $\rangle$ **do**

trigger $\langle$ *beb-Broadcast* | [READ] $\rangle$;

upon event $\langle$ *beb-Deliver* | p, [READ] $\rangle$ **do**

trigger $\langle$ *pl-Send* | p, [VALUE, *ts*, *val*] $\rangle$;

upon event $\langle$ *pl-Deliver* | q, [VALUE, *ts*$'$, v] $\rangle$ **do**

if *ts*$'$ $>$ *ts* **then**

val $:= v$;

readset $:=$ *readset* $\cup \{q\}$;

upon *correct* $\subseteq$ *readset* **do**

readset $:= \emptyset$;

trigger $\langle$ *onrr-ReadReturn* | *val* $\rangle$;

5.3 $(1, 1)$ Atomic register (3pt)

Study Algorithm 4.3 [CGR11, Sec. 4.3.2], which is an abstract transformation from one $(1, N)$ regular register to an $(1, 1)$ atomic register. Use the underlying idea to describe modifications for Algorithm 4.2 ("Majority voting regular register") [CGR11, Sec. 4.2.3] such that the modified protocol implements a $(1, 1)$ atomic register in the fail-silent model (where less than $N/2$ processes may fail).

You might be thinking that a $(1, 1)$ register is not very useful because the two clients (reader and writer) could simply communicate with each other. However, these clients may not be online simultaneously and the approach gives insight into more complex protocols.

Solution. According to the idea that underlies Algorithm 4.3 [CGR11, Sec. 4.3.2], the reader simply stores the timestamp/value pair with the largest timestamp that it has observed so far, and returns this value if the received VALUE messages do not contain a larger timestamp. This can be achieved if the reader always includes its own timestamp/value pair in the variable *readlist*, from which it selects the return value.

Algorithm 2 shows pseudo code for those parts of Algorithm 4.2 [CGR11, Sec. 4.2.3] that must be modified. It uses the function *highest*$(\cdot)$, which returns the timestamp/value pair with the largest timestamp from its input. As in the original "Majority Voting" algorithm, every write operation requires one communication roundtrip between the writer and a majority of the processes, and every read operation requires one communication roundtrip between the reader and a majority of the processes. In both cases, $O(N)$ messages are exchanged.

Algorithm 2 Modification of Majority Voting to Implement a (1, 1) Atomic Register.

upon event $\langle$ *onrr-Read* $\rangle$ **do**
    rid $:=$ *rid* $+ 1$;
    readlist $:= [\perp]^N$;
    readlist[*self*] $:= (ts, val)$;
    trigger $\langle$ *beb-Broadcast* | [READ, *rid*] $\rangle$;

upon event $\langle$ *beb-Deliver* | p, [READ, r] $\rangle$ **do**
    if $p \neq$ *self* **then**
        trigger $\langle$ *pl-Send* | p, [VALUE, r, *ts*, *val*] $\rangle$;

upon event $\langle$ *pl-Deliver* | q, [VALUE, r, ts', v'] $\rangle$ **such that** $r =$ *rid* **do**
    readlist[q] $:= (ts', v')$;
    if $\#$(*readlist*) $> \frac{N}{2}$ **then**
        $(ts, val) :=$ *highest*(*readlist*);
        readlist $:= [\perp^N]$;
        trigger $\langle$ *onrr-ReadReturn* | *val* $\rangle$;

References

[CGR11] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming (Second Edition)*, Springer, 2011.