ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

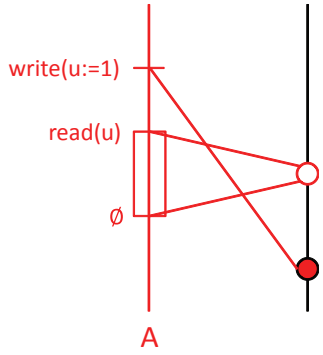
HS 2010 M. Kuhn / S. Welten / Prof. R. Wattenhofer

Distributed Systems Part II

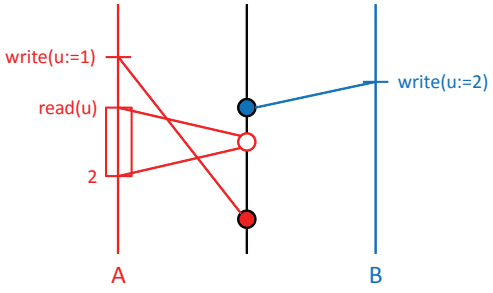
Solution to Exercise Sheet 4

1 Sequential Consistency: Warm Up

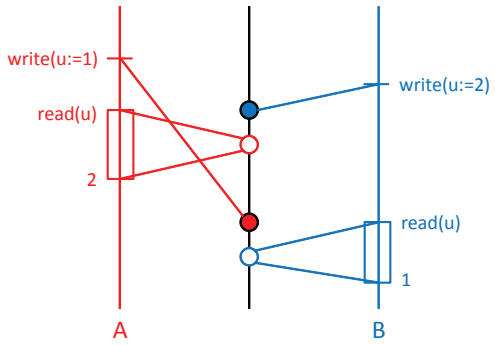
a) The following figure shows an execution that is not sequentially consistent. On the one hand, the read operation is required to happen before the write operation according to the return values ($r_A : r_A < w_A(u := 1)$, where o_X means that operation o was executed by client X). On the other hand, the client partial order requires the write operation to happen before the read operation ($<_c$: $w_A(u := 1) < r_A$). As these two conditions cannot be fulfilled at the same time, the execution is not sequentially consistent.



b) The following figure shows how the execution can be made sequentially consistent by adding one write operation. A valid sequence is: (1) $w_A(u := 1)$, (2) $w_B(u := 2)$, (3) r_A.



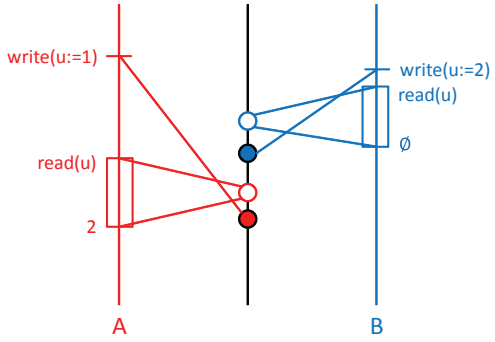
c) The following figure shows how the execution can again be made not sequentially consistent by adding a read operation at the end.



The execution is not sequentially consistent, as:

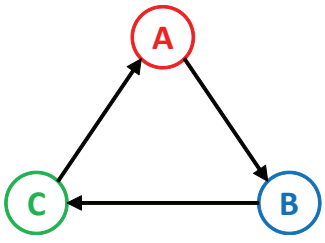
- r_A(together with $<_c$) : $w_A(u := 1) < w_B(u := 2)$
- $r_B : w_B(u := 2) < w_A(u := 1)$

Another possibility to construct an execution that is not sequentially consistent is shown below. Here the argument is analogous to subtask a).



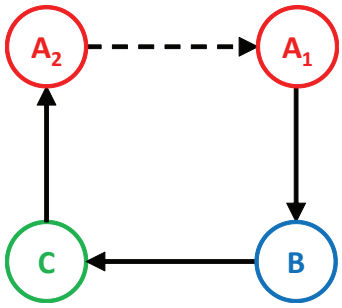
2 Sequential Consistency with 3 Clients

The solution of this task requires some basic ideas. We know that an execution is not sequentially consistent if it exhibits circular dependencies. Thus, a first idea is to make client A dependent from client B, client B dependent from client C, and client C again dependent from client A, as shown below:

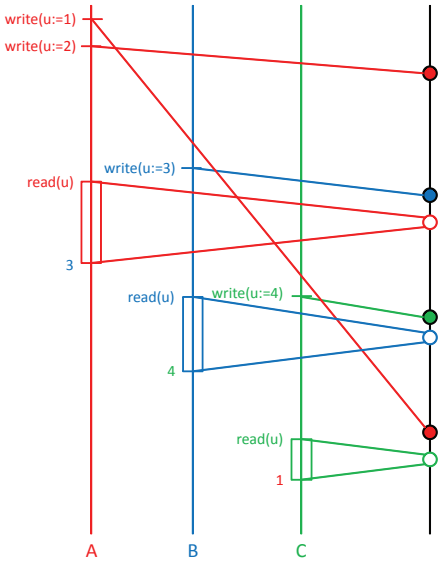


As soon as one node of the resulting triangle is removed, only simple dependencies remain (no matter which node is removed) and the execution becomes sequentially consistent. To construct the required dependencies, it seems reasonable that each client has to perform at least one write operation. Thus, we can first place these three write operations on our replica. Let's assume that the write of client A happens first, followed by the write of B and C (observe that the order does not matter, as so far everything is fully symmetric). We then introduce the first two of the required dependencies by making a read from A follow the write of B, a read from B follow the write of C. Ideally, we could now just introduce a read from C following the write of A. However, this would require to place C's read before C's write operation in C's client

order, which is not what we want. To nevertheless introduce the required dependency between C and A, we thus introduce a second write operation of client A (which is followed by C's read). To finally close the circle, we have to introduce another dependency between A's first and A's second write operation (order as seen on the replica). Basically, we want the second write to be before the first one (as indicated by the dashed arrow in the figure below). This can be achieved by placing the second write before the first one in A's client order.



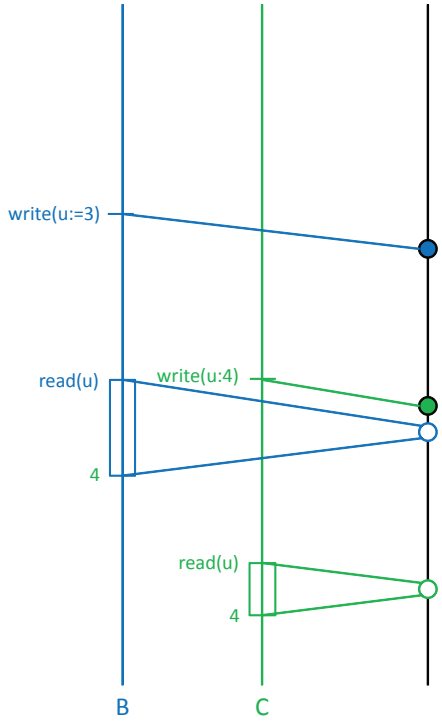
The resulting execution looks as illustrated below:



The following relationships show that we have indeed constructed the intended circular dependency which makes the execution not sequentially consistent:

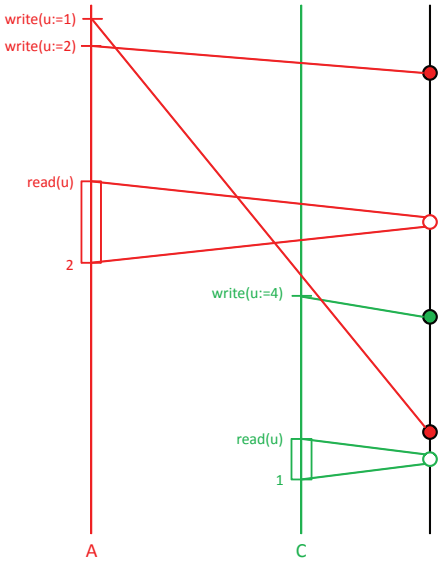
- $<_C$ on client A: $write(u := 1) < write(u := 2)$
- r_A (together with $<_C$): $write(u := 2) < write(u := 3)$
- r_B (together with $<_C$): $write(u := 3) < write(u := 4)$
- r_C (together with $<_C$): $write(u := 4) < write(u := 1)$

The execution without client A is sequentially consistent and looks as follows:



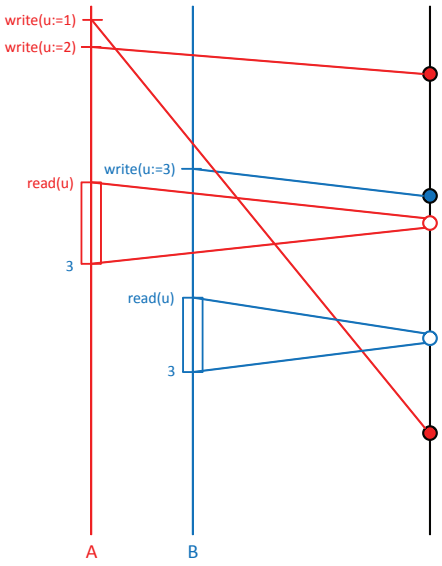
A valid sequence is: (1) $write(u := 3)$ (2) $write(u := 4)$ (3) r_B (4) r_C.

The execution without client B is sequentially consistent and looks as follows:



A valid sequence is: (1) $write(u := 4)$ (2) $write(u := 1)$ (3)r_C (4) $write(u := 2)$ (5) r_A.

The execution without client C is sequentially consistent and looks as follows:



A valid sequence is: (1) $write(u := 1)$ (2) $write(u := 2)$ (3) $write(u := 3)$ (4) r_A (5) r_B.

3 Three Phase Commit

a) The five steps are:

i) Step 2: Participants wait for VOTE request
ii) Step 3: Coordinator waits for votes
iii) Step 4: Participants wait for coordinators decision
iv) Step 5: Coordinator waits for acknowledgments
v) Step 6: Participants wait for COMMIT

b) The reactions are:

i) no one has yet decided on commit, so abort
ii) no one has yet decided on commit, so abort
iii) some processes already know the coordinators decision, some do not. Some might already aborted, or have sent ACK. The processes have to elect a new coordinator, the new coordinator has to find out what the decision was and resend the decision to all the processes which do not already know it.
iv) a crashed processes did not send an ACK. But the transaction can continue because the correct processes are prepared to commit.
v) the next message must be COMMIT, but committing would violate the non-blocking property as some processes may not yet have received PREPARE. So the processes first have to elect an new coordinator, which has to find out about the decision and inform uninformed processes about it.

c) Assume there are three processes p_1, p_2 and p_3. Process p_1 is the coordinator. All the processes vote `YES` on the transaction. p_1 receives and processes the votes, but p_2 and p_3 detach from p_1 before receiving the `PREPARE` message sent by p_1.

p_2 is elected as new coordinator. It sees both p_2 and p_3 are undecided. Not knowing their state p_2 decides to abort, sending the `ABORT` message. p_3 receives the message, then detaches from p_2. If now p_1 and p_3 become connected they cannot progress: p_1 already decided to commit, p_3 already decided to abort.