Timeouts

Preliminary

- The context object is created at the initialization of a stream. This ctx is immutable.
- Subsequent requests don’t include a new context object.
- I.e. contexts provided to multicasts and quorumcalls only has one job:
  - If the context is canceled, then it cancels the stream
    - Meaning a new stream has to be created to send new requests
    - Also meaning it is not possible to cancel a request without creating a new config

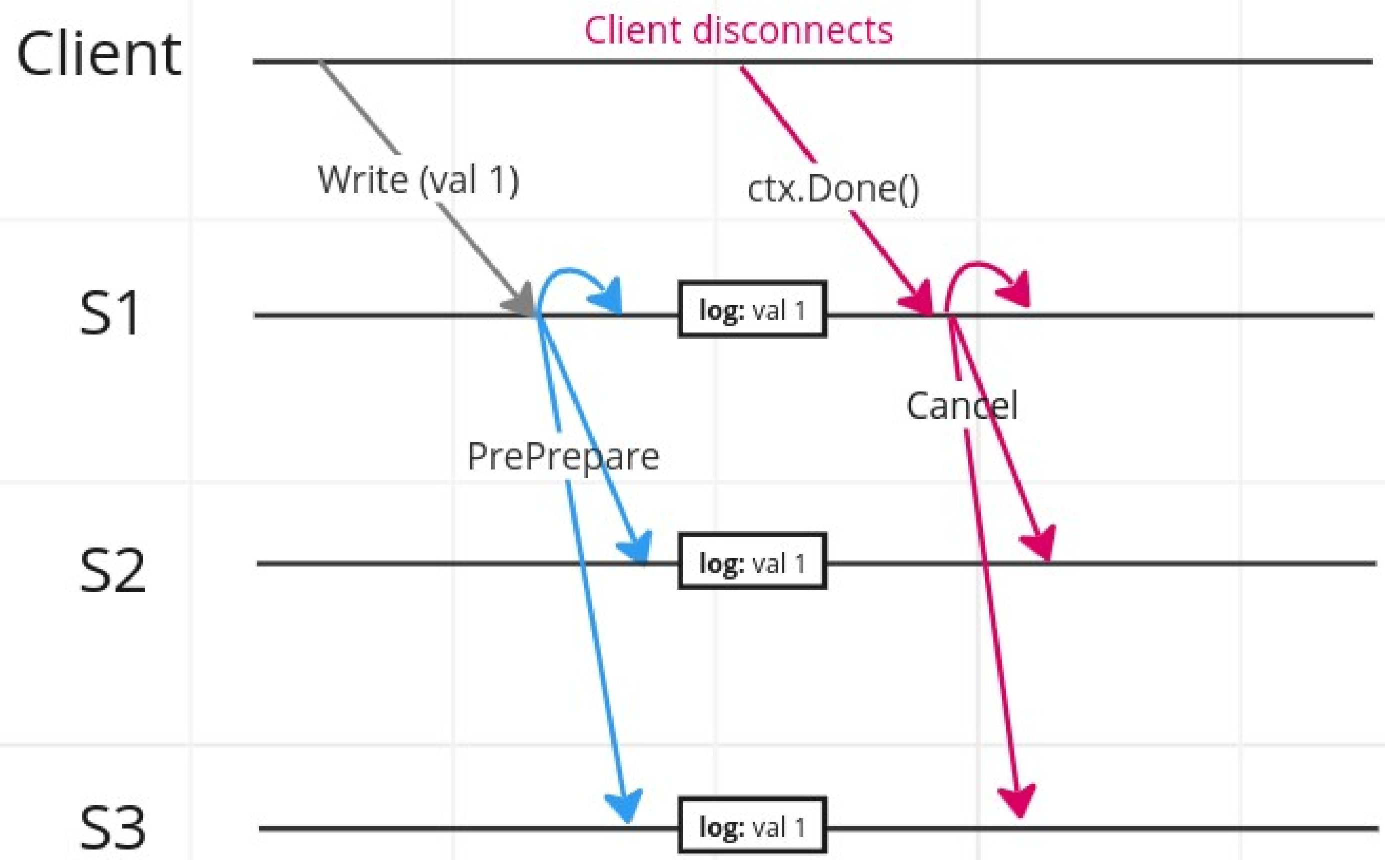
Desired outcome

- A client should be able to cancel a request. (give up on waiting)
- Servers should be able to timeout in order to stop “stuck” jobs

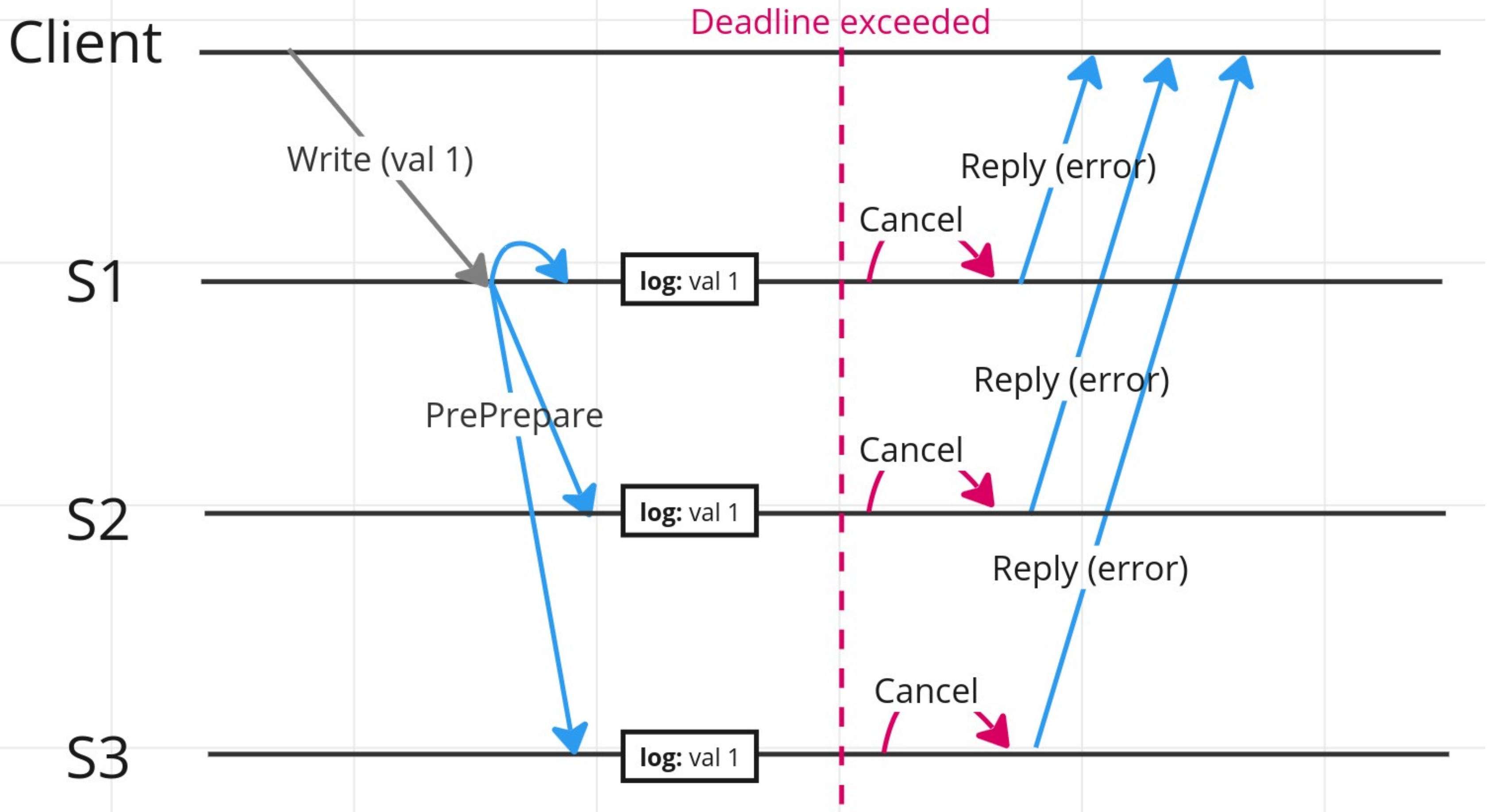
Problem

- A client can either timeout or disconnect. Should both be handled equally?
- How long should a timeout be?
- How to prevent servers from timing out when they are not supposed to?
- How to prevent byzantine servers to send fake cancellations?
- How should a server handle a timeout?
- When can a server safely assume that a request has finished?
- How to guarantee correct execution of algorithms?
  - What level of manual control over timeouts should the implementer have?
- The problem gets easier by constraining the user implementation, but this might be counter productive? E.g. by enforcing rules such as “Only the leader can cancel a request.”
- It could be necessary for the client to wait for replies when it cancels a request. Instead of canceling a request it instead sends a cancellation request. This can also fail and suffer from the problems which timeouts are trying to solve.
- A timeout can’t be considered as a server failure or as a request never arrived at the server. This is because the server can correctly complete all steps in an algorithm, but the execution might still be wrong due to external influence (slow servers/network) not considered by the implementer.

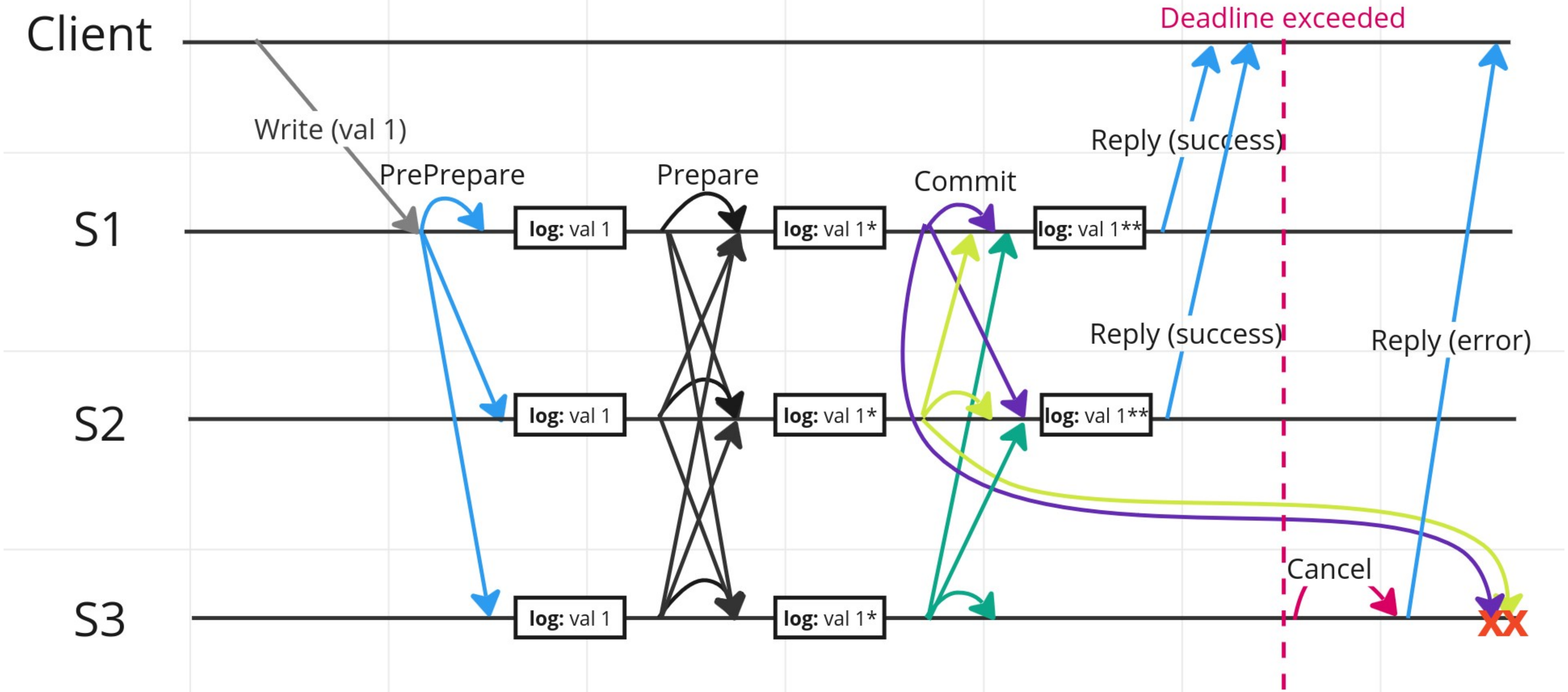
Client disconnects and thus the stream is broken. S1 sends a cancel request to the other servers.



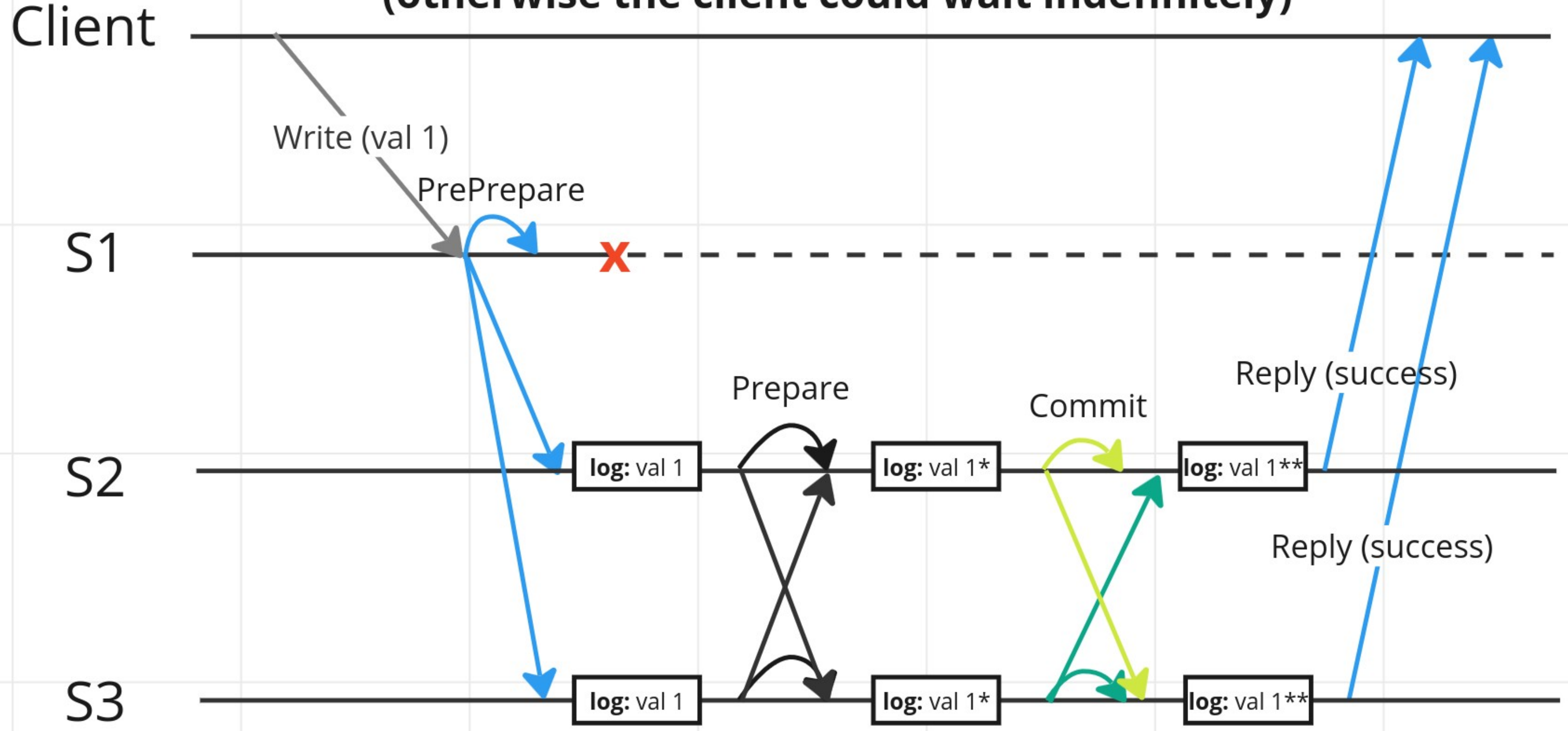
Context deadline is exceeded and all servers reply with an error.



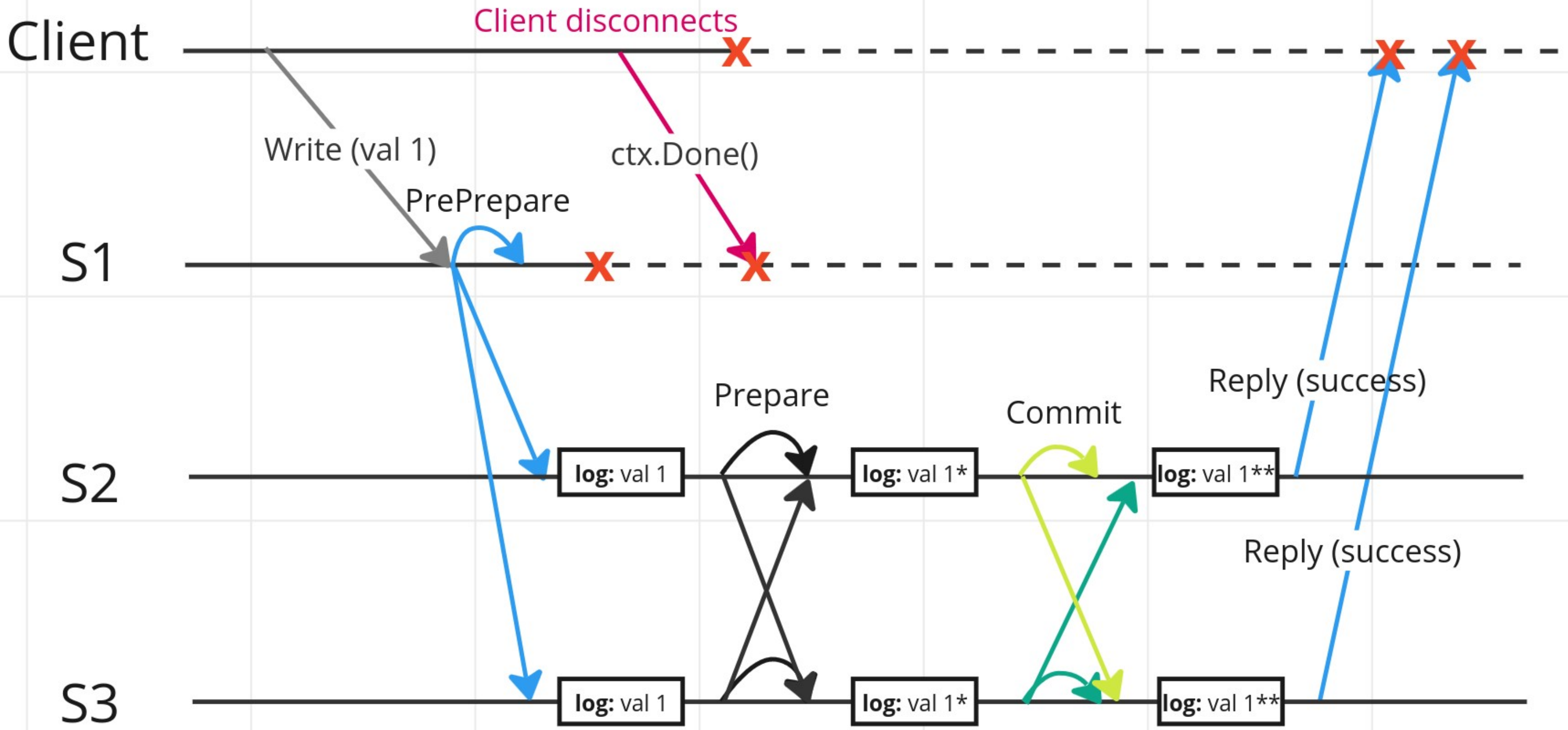
Context deadline exceeded. How should S1 and S2 know that S3 did not complete? Should the timeout be ignored or should the request be reverted?



S1 crashes. Should the client regard this as an interrupted request? Or should it wait until the timeout? This could mean that a timeout is required (otherwise the client could wait indefinitely)

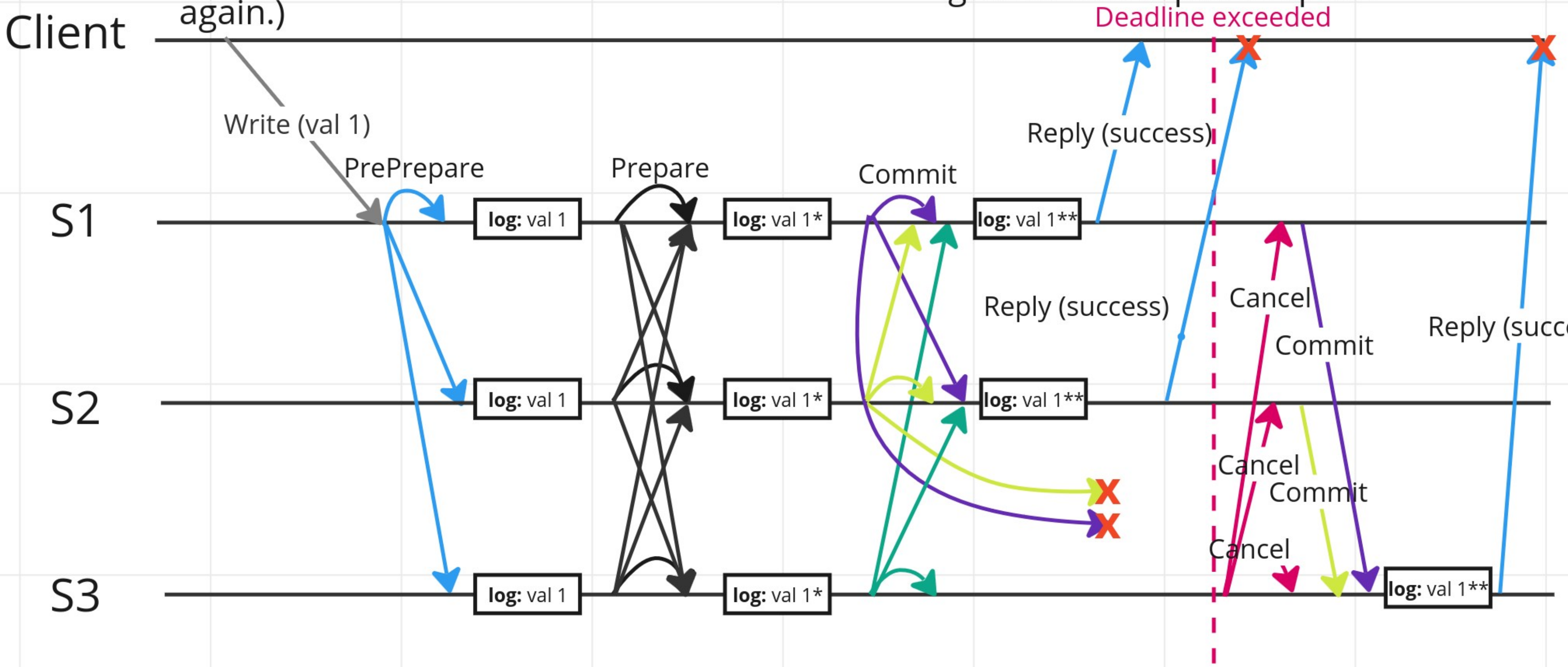


S1 and the client crashes. What should the other servers do?



Solution propsal 1: (case 1)
Broadcast cancel to all servers. Only reply to client with error if byz quorum of cancels. Servers don't broadcast cancel if already replied to client. (Instead invokes last method again.)

Issue:
The client will still have regarded the request as failed because it did not get enough responses within the deadline. Hence, it ignores subsequent replies.

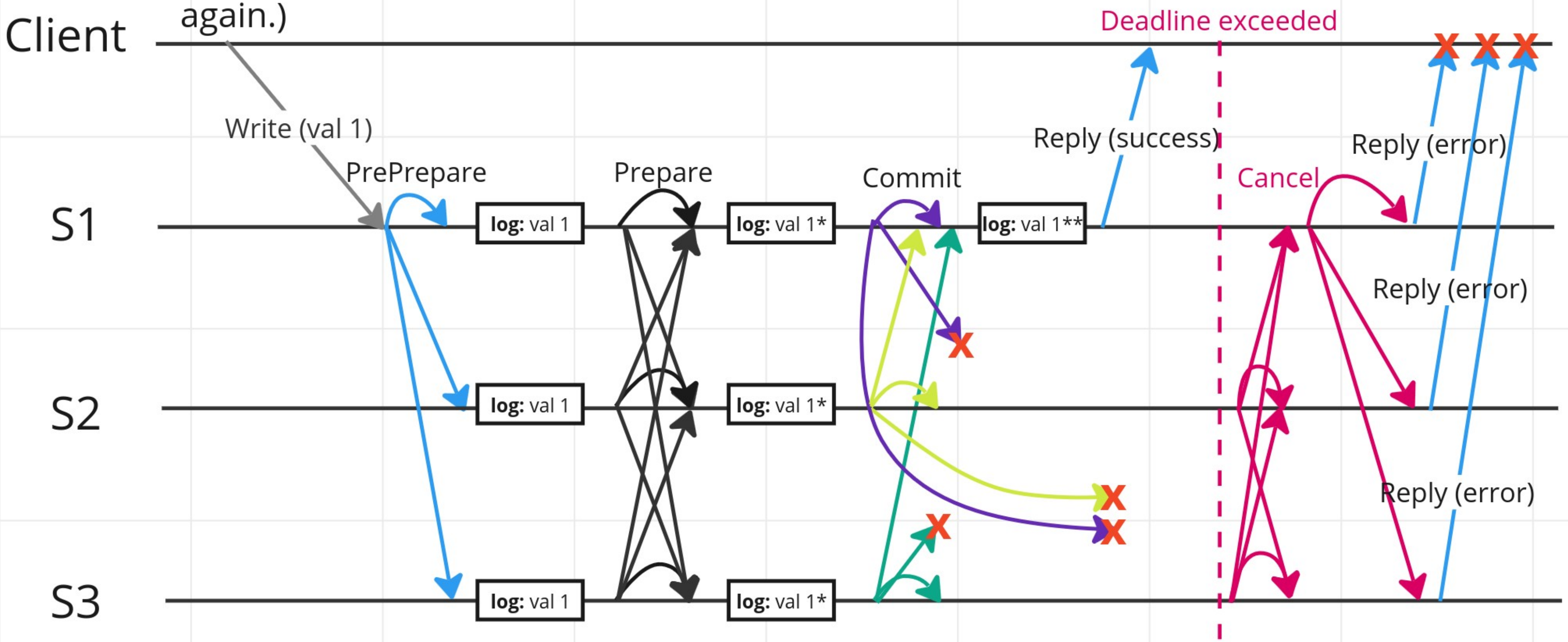


Solution propsal 1: (case 2)

Broadcast cancel to all servers. Only reply to client with error if byz quorum of cancels. Servers don't broadcast cancel if already replied to client. (Instead invokes last method again.)

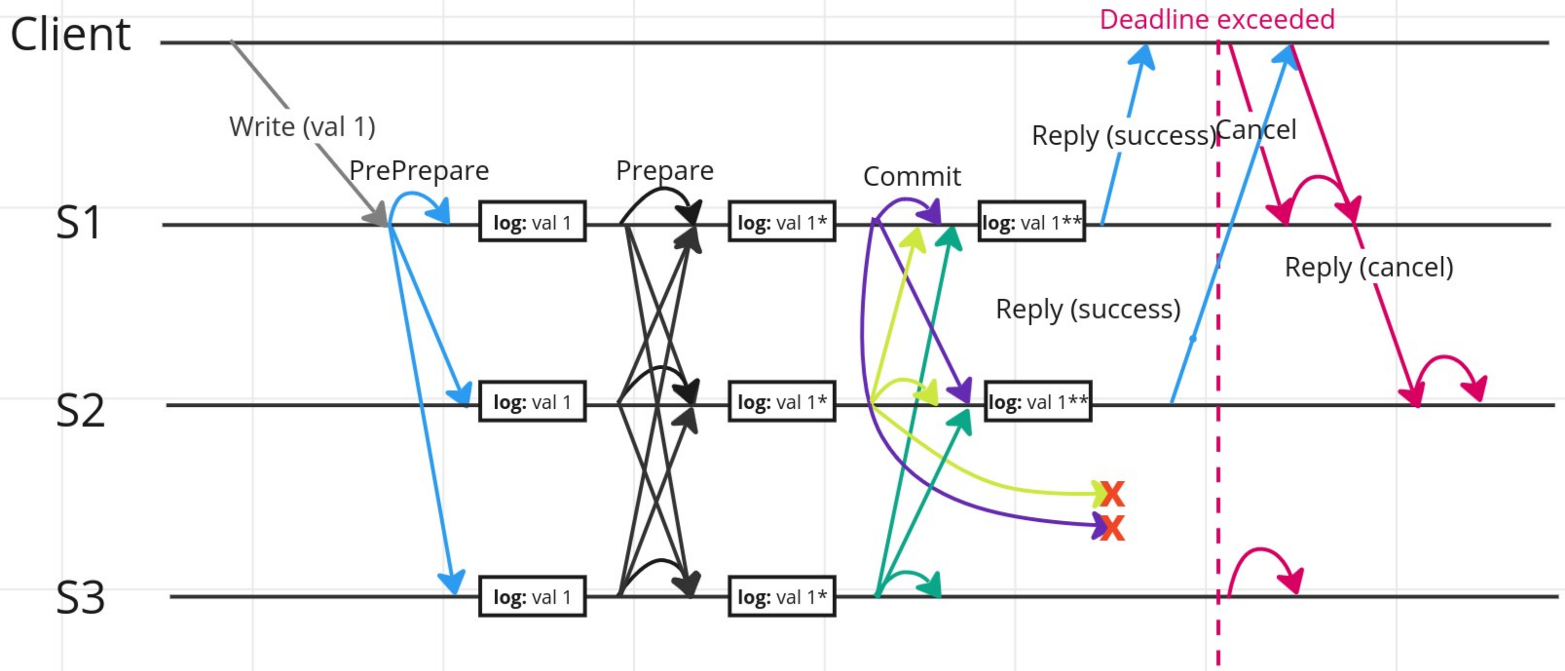
Issue:

Should the client still wait for error replies from all servers? Should the client just simply drop the first reply from S1?



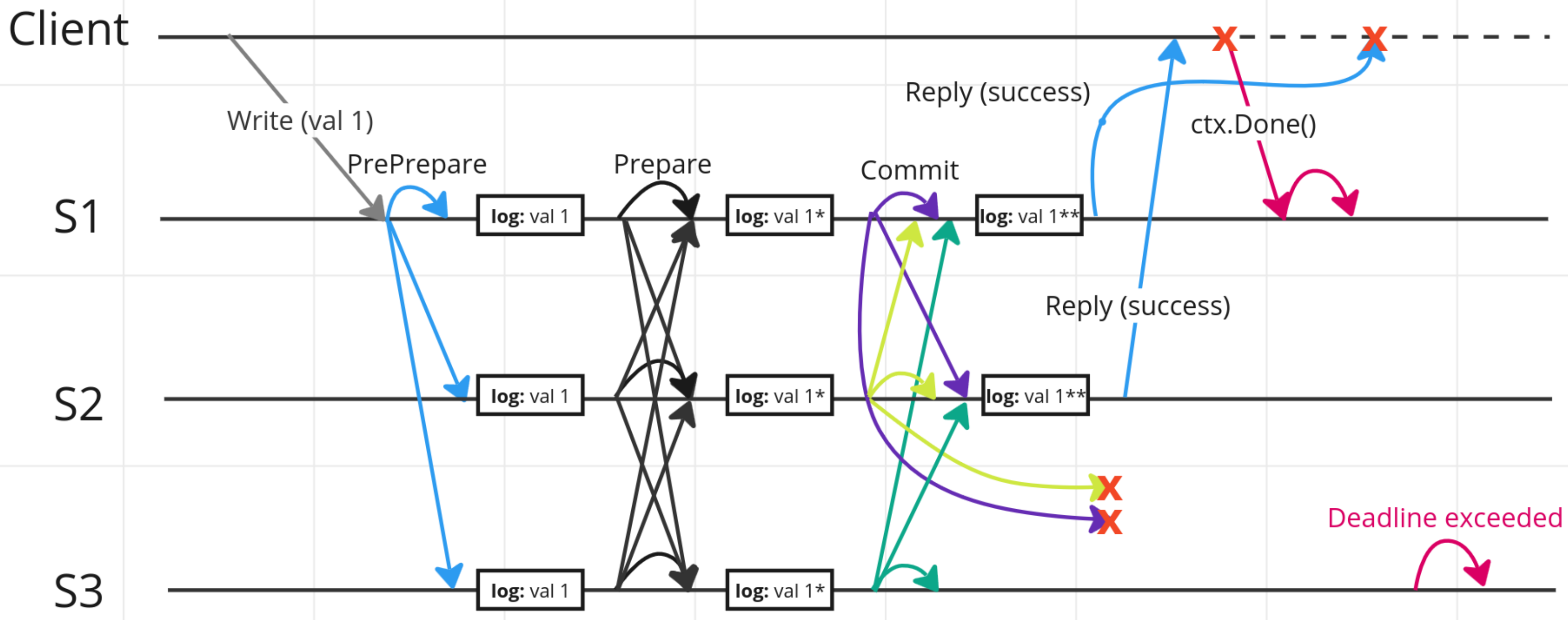
Solution propsal 2: (case 1)
Client broadcast cancel to all servers in config + all servers replied to client. Servers only cancels on itself if deadline is exceeded. Client also reply with cancels for late responses. Servers can assume a successful request if it has replied to client and not received a cancellation request within x minutes.

Issue:
This will not work if the client disconnects: Imagine S2 replied to the client within the deadline. There is no way for S2 to know that the client disconnected (only S1 has a connection (NodeStream) to the Client and knows when it is broken).



Solution propsal 2: (case 1.2)
Client broadcast cancel to all servers in config + all servers replied to client. Servers only cancels on itself if deadline is exceeded. Client also reply with cancels for late responses. Servers can assume a successful request if it has replied to client and not received a cancellation request within x minutes.

Issue:
This will not work if the client disconnects: Imagine S2 replied to the client within the deadline. There is no way for S2 to know that the client disconnected (only S1 has a connection (NodeStream) to the Client and knows when it is broken).

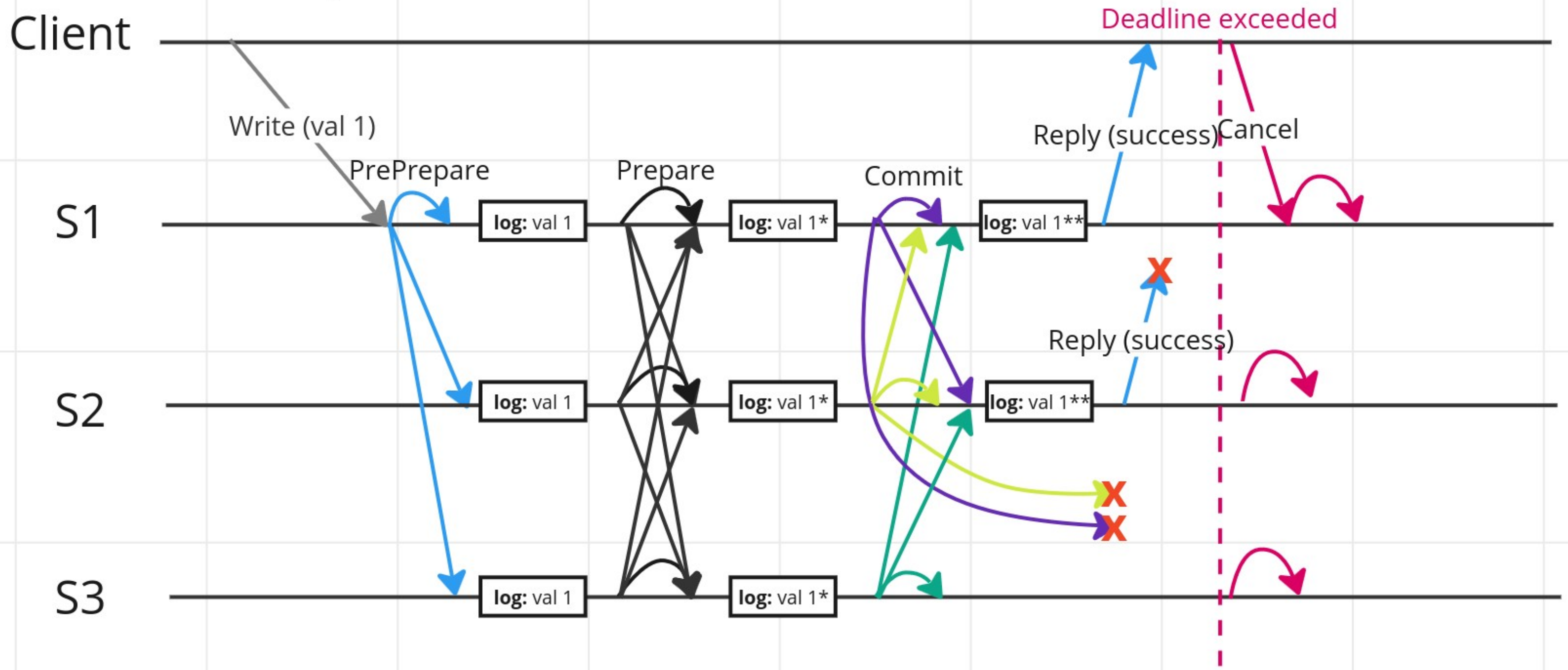


Solution propsal 2: (case 2)

Client broadcast cancel to all servers in config + all servers replied to client. Servers only cancels on itself if deadline is exceeded. Client also reply with cancels for late responses. Servers can assume a successful request if it has replied to client and not received a cancellation request within x minutes.

Issue:

What happens if a server not included in the config of the client finishes, but the reply is dropped in transit? Only accept a successful request if the client has replied/acknowledged the server reply.

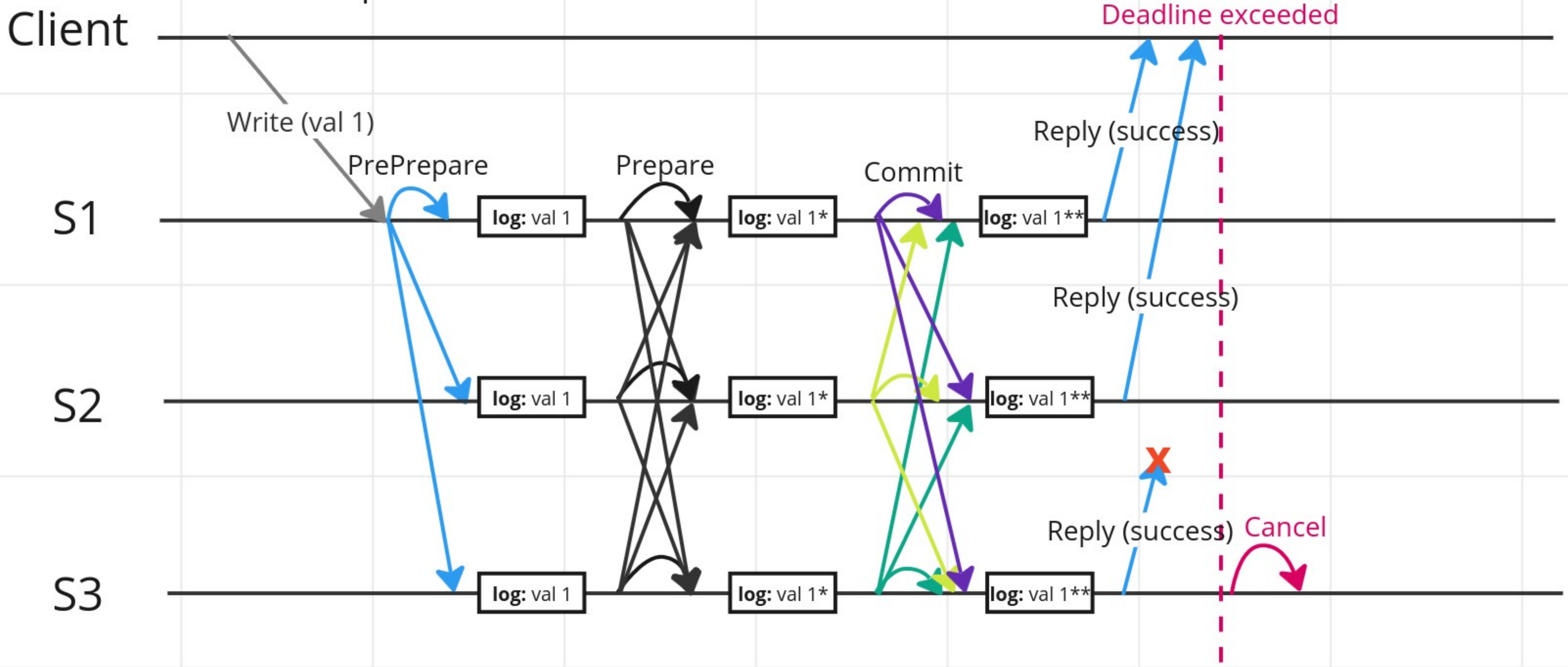


Solution propsal 2: (case 3)

Client broadcast cancel to all servers in config + all servers replied to client. Servers only cancels on itself if deadline is exceeded. Client also reply with cancels for late responses. Servers can assume a successful request if it has replied to client and not received a cancellation request within x minutes.

Issue:

The algorithm successfully commits a value, but S3 does not receive a response from the client before the deadline. Thus it runs the cancellation process.

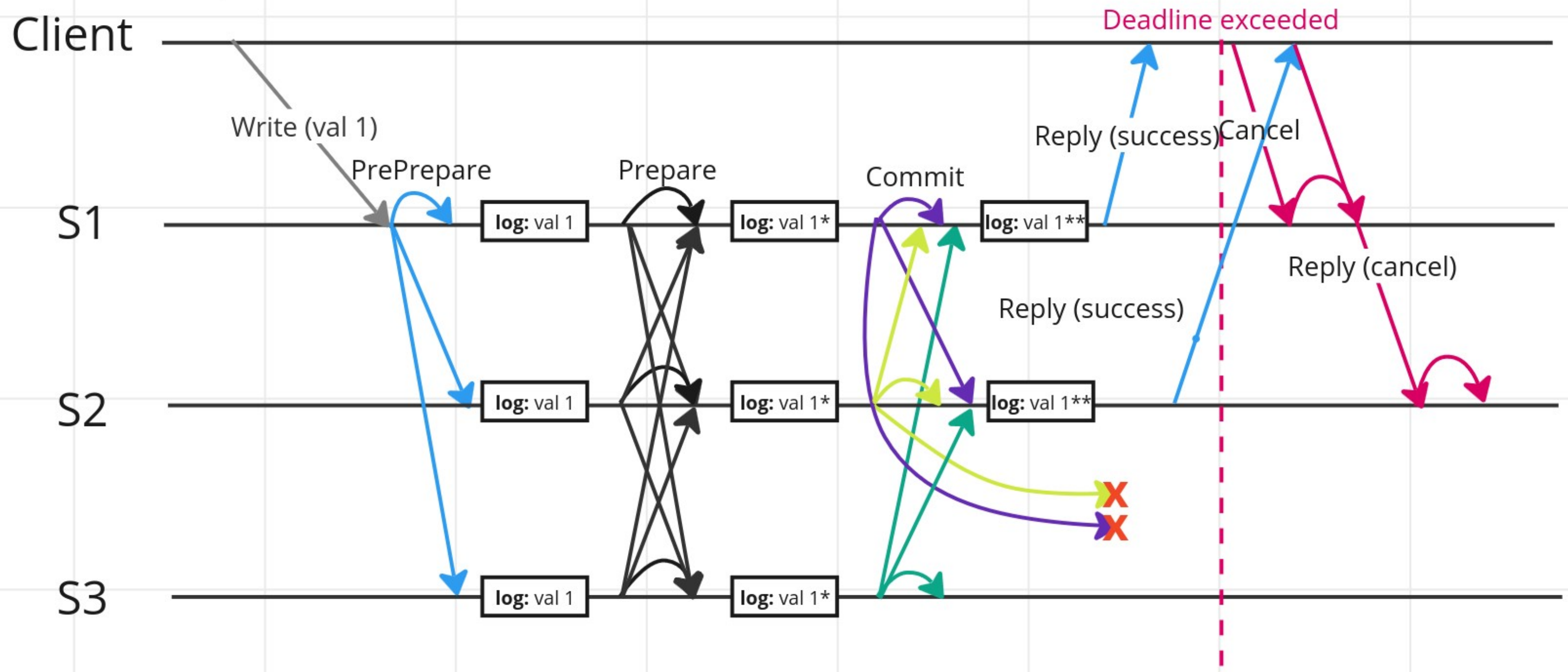


Solution propsal 3: (case 1)

Only the client can cancel a request and none else (i.e. servers and intermediate servers). The client broadcast cancel to all servers in config + all servers replied to client. Servers never cancels itself, but drops the client request after x minutes (e.g. 30 minutes). Client also reply with cancels for late responses. Servers can assume a successful request as longs as it has not received a cancellation request from the client.

Issue:

The cancellation needs to be propagated to all servers. Servers unkown to the client (e.g. S3) may have stale data if the implementer does not have a way of cleaning up. How to enforce that only the client can cancel and not a byz server?

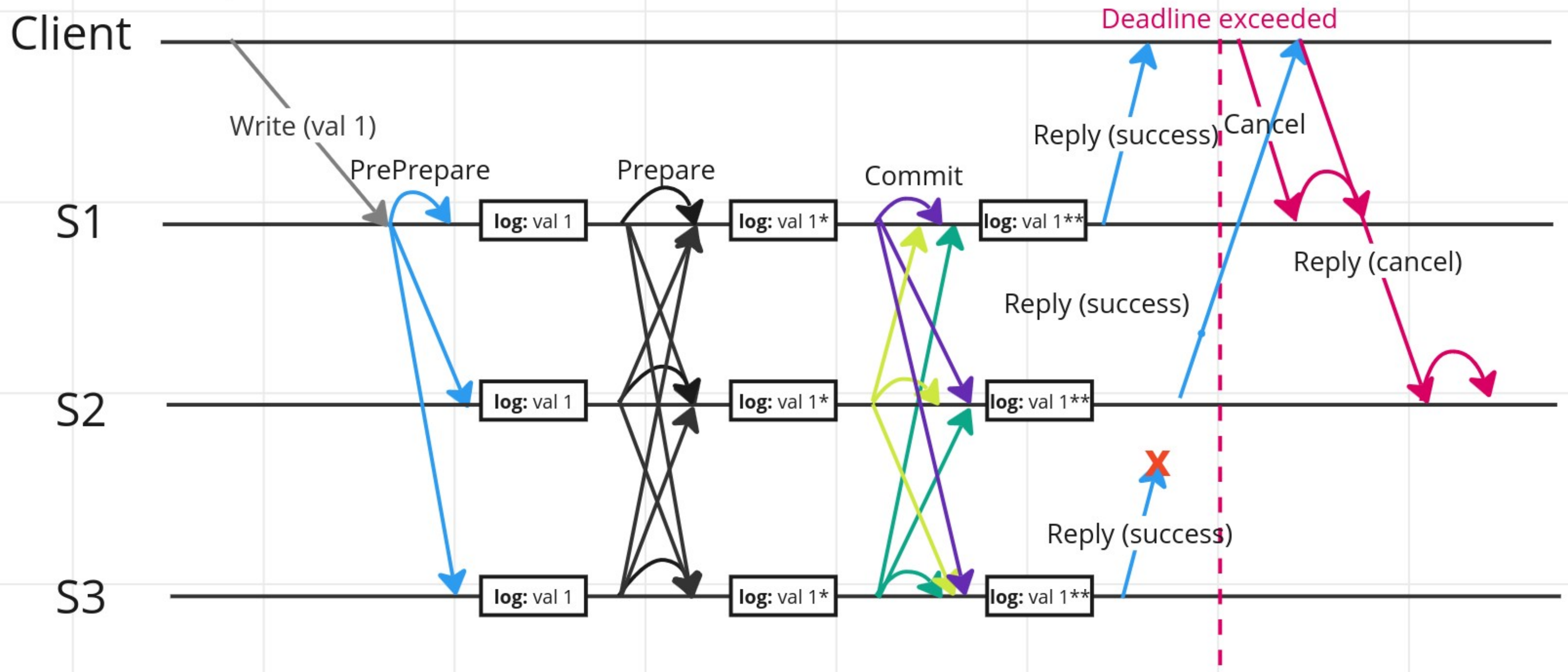


Solution propsal 3: (case 2)

Only the client can cancel a request and none else (i.e. servers and intermediate servers). The client broadcast cancel to all servers in config + all servers replied to client. Servers never cancels itself, but drops the client request after x minutes (e.g. 30 minutes). Client also reply with cancels for late responses. Servers can assume a successful request as longs as it has not received a cancellation request from the client.

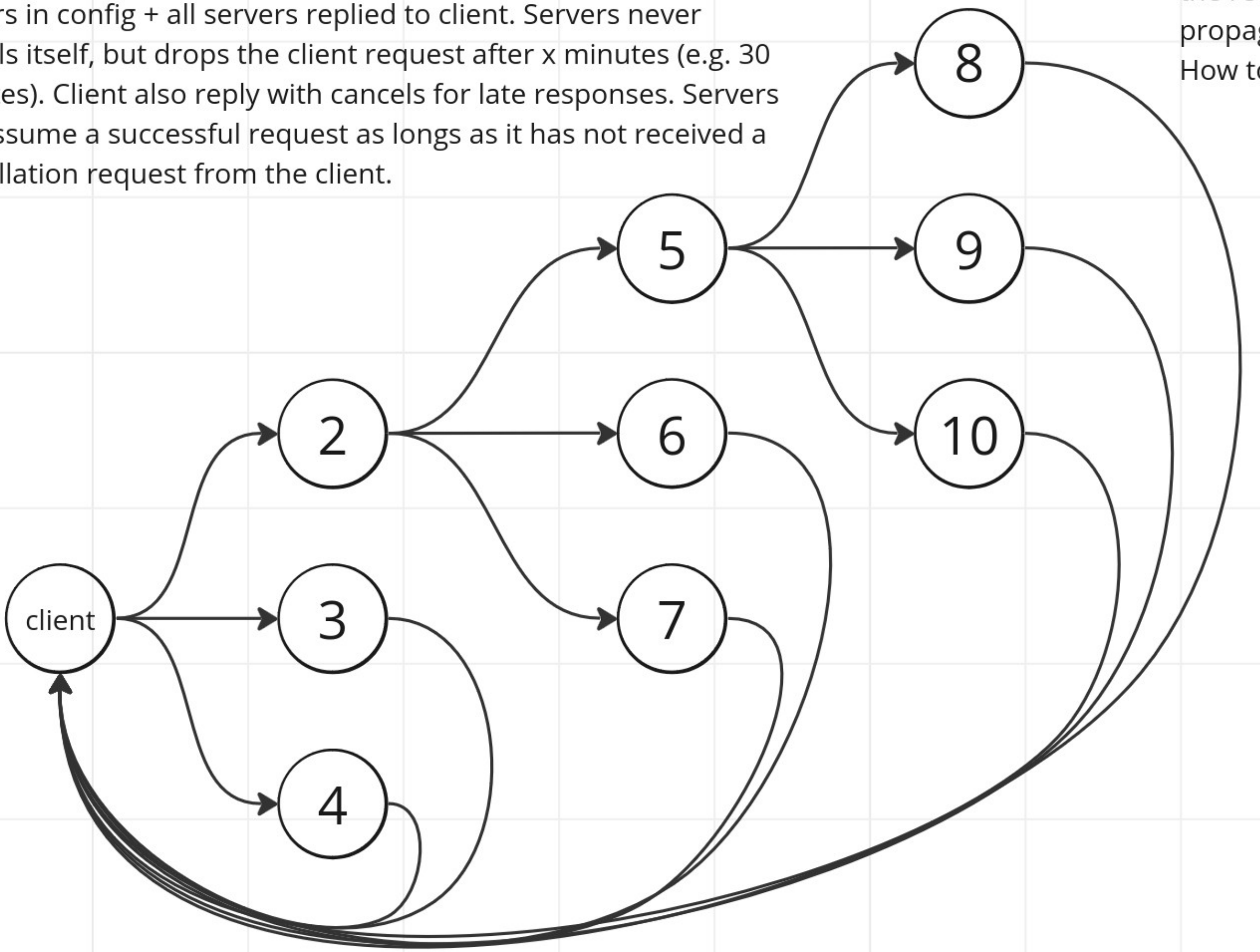
Issue:

Unkown servers may commit a value and assume that the request finished when it in fact was cancelled.



Solution propsal 3: (case 3)
Only the client can cancel a request and none else (i.e. servers and intermediate servers). The client broadcast cancel to all servers in config + all servers replied to client. Servers never cancels itself, but drops the client request after x minutes (e.g. 30 minutes). Client also reply with cancels for late responses. Servers can assume a successful request as longs as it has not received a cancellation request from the client.

Issue:
In gossiping, the intermediate servers will not know if the request was cancelled unless the cancellation is propagated. What to do about "fake" cancellation? How to get trust?



PBFT

- A server timeout can interfere with the correctness of the algorithm because it relies on an “all or nothing” execution (atomicity requirement)
- All servers must revert the state if a request is canceled. Hence, a cleanup function needs to be implemented.

A possible solution

- Employ a consensus algorithm to determine when a request has been canceled by a client or finished (either by replying to the client or by timing out)
- Can also regard requests as failed if nothing happens for 30 minutes (to prevent stale data). This can be overridden by the implementer.

Timeout options

- None:
  - A request cannot be canceled
  - The handling of timeouts is entirely left to the implementer
- Simple:
  - A server drops a request after the deadline/timeout or if the client disconnects
  - Implementer can provide a cleanup/timeout function
  - No consensus mechanisms
- Full fledged:
  - Employ consensus mechanisms to ensure all servers are in sync

Conclusion

- A client is too unreliable and therefore can never be the source of truth. Hence, a client can only try to cancel a request, but it cannot be given any gurantees. Furthermore, it should ask for the results for a given task/execution.
- There might not exist a “one size fits all” solution for timeouts. It is highly dependent on the implementation and can thus be a hindrance.