from all f faulty processes are included. Thus, after the completion of the write, more than

$$\frac{N+2f}{2} - f$$

correct and *informed* processes store the pair (wts, v) in their variables ts and val, and the remaining up to (but less than)

$$N - \frac{N+2f}{2}$$

correct processes are *uninformed*.

During the read, process q receives more than $(N+2f)/2$ VALUE messages containing timestamp/value pairs, of which up to f may be from faulty processes and contain arbitrary data, and less than

$$N - \frac{N+2f}{2}$$

may be from uninformed processes. Subtracting the latter from the former, there are still more than f messages from informed processes and contain the pair (wts, v). Consequently, the function *byzhighestval*$(\cdot)$ does not filter out this pair and wts is the largest timestamp received from any correct process; larger timestamps received from faulty processes occur at most f times and are eliminated. Hence, *byzhighestval*$(\cdot)$ returns v.

Performance. The algorithm uses the same number of messages as Algorithm 4.2 for regular registers in the fail-silent model, from which it is derived. In total, it uses one communication roundtrip and $O(N)$ messages for every operation.

4.7 $(1, N)$ Byzantine Regular Register

When a write operation updates the stored data concurrently to a read operation in Algorithm 4.14, the read may return the default value v_0. Although permitted by safe semantics, this violates *regular* semantics. The problem is that the reader cannot distinguish old timestamp/value pairs and newly written ones from the ones that may have been forged by the faulty processes, and returns v_0 in case of doubt. For extending Algorithm 4.14 to implement a $(1, N)$ *regular* register abstraction, however, the algorithm would need to return either the last written value or the concurrently written one. We define the $(1, N)$ Byzantine regular register abstraction in this section and consider two algorithms to implement it. The first algorithm (Sect. 4.7.2) uses data authentication through digital signatures, where as the second one (Sect. 4.7.3) does not.

4.7.1 Specification

The $(1, N)$ *Byzantine regular register* abstraction is basically the same as a $(1, N)$ regular register, but with an explicit identification of the writer w and the restriction

Module 4.6: Interface and properties of a $(1, N)$ Byzantine regular register

Module:

Name: $(1, N)$-ByzantineRegularRegister, **instance** *bonrr*, with writer w.

Events:

Request: $\langle$ *bonrr*, *Read* $\rangle$: Invokes a read operation on the register.

Request: $\langle$ *bonrr*, *Write* $\mid v \rangle$: Invokes a write operation with value v on the register. Executed only by process w.

Indication: $\langle$ *bonrr*, *ReadReturn* $\mid v \rangle$: Completes a read operation on the register with return value v.

Indication: $\langle$ *bonrr*, *WriteReturn* $\rangle$: Completes a write operation on the register. Occurs only at process w.

Properties:

BONRR1–BONRR2: Same as properties ONRR1–ONRR2 in a $(1, N)$ regular register (Module 4.1).

of readers and writers to crash faults, as for Byzantine safe registers. The details of the abstraction are given in Module 4.6.

4.7.2 Fail-Arbitrary Algorithm: Authenticated-Data Byzantine Quorum

With the help of digital signatures one can easily circumvent the problem in Algorithm 4.14 mentioned earlier and obtain an implementation of a $(1, N)$ Byzantine regular register. This solution has even better resilience (requiring only $N > 3f$) than Algorithm 4.14 (whose resilience is $N > 4f$).

The idea behind the following "Authenticated-Data Byzantine Quorum" algorithm, shown in Algorithm 4.15, is for the writer to sign the timestamp/value pair and to store it together with the signature at the processes. The writer *authenticates* the data with its signature. The reader verifies the signature on each timestamp/value pair received in a VALUE message and ignores those with invalid signatures. A Byzantine process is thus prevented from returning an arbitrary timestamp and value in the VALUE message, although it may include a signed value with an outdated timestamp. Algorithm 4.15 is now obtained from the "Majority Voting" algorithm in the fail-silent model by adding data authentication and by employing Byzantine (majority) quorums (Sect. 2.7.3) instead of ordinary (majority) quorums.

Note that only the clients, i.e., the reader and the writer, need to perform cryptographic digital signature operations; the server processes simply store the signatures and may ignore their meaning.

Correctness. Under the assumption that $N > 3f$, the *termination* property is straightforward to verify: as there are $N - f$ correct processes, the reader and the

Algorithm 4.15: Authenticated-Data Byzantine Quorum

Implements:
$(1, N)$-ByzantineRegularRegister, **instance** *bonrr*, with writer w.
Uses:
AuthPerfectPointToPointLinks, **instance** al.

upon event $\langle$ *bonrr*, *Init* $\rangle$ **do**
$(ts, val, \sigma) := (0, \bot, \bot)$;
$wts := 0$;
acklist $:= [\bot]^N$;
rid $:= 0$;
readlist $:= [\bot]^N$;

upon event $\langle$ *bonrr*, *Write* | v $\rangle$ **do** // only process w
$wts := wts + 1$;
acklist $:= [\bot]^N$;
$\sigma :=$ *sign*(*self*, *bonrr*||*self*||WRITE||wts||v);
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* | q, [WRITE, wts, v, σ] $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, [WRITE, ts', v', σ'] $\rangle$ **such that** $p = w$ **do**
if $ts' > ts$ **then**
$(ts, val, \sigma) := (ts', v', \sigma')$;
trigger $\langle$ *al*, *Send* | p, [ACK, ts'] $\rangle$;

upon event $\langle$ *al*, *Deliver* | q, [ACK, ts'] $\rangle$ **such that** $ts' = wts$ **do**
acklist[q] := ACK;
if $\#$(*acklist*) $> (N + f)/2$ **then**
acklist $:= [\bot]^N$;
trigger $\langle$ *bonrr*, *WriteReturn* $\rangle$;

upon event $\langle$ *bonrr*, *Read* $\rangle$ **do**
rid $:=$ *rid* $+ 1$;
readlist $:= [\bot]^N$;
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* | q, [READ, *rid*] $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, [READ, r] $\rangle$ **do**
trigger $\langle$ *al*, *Send* | p, [VALUE, r, ts, val, σ] $\rangle$;

upon event $\langle$ *al*, *Deliver* | q, [VALUE, r, ts', v', σ'] $\rangle$ **such that** $r =$ *rid* **do**
if *verifysig*(q, *bonrr*||w||WRITE||ts'||v', σ') **then**
readlist[q] := (ts', v');
if $\#$(*readlist*) $> \frac{N+f}{2}$ **then**
$v :=$ *highestval*(*readlist*);
readlist $:= [\bot]^N$;
trigger $\langle$ *bonrr*, *ReadReturn* | v $\rangle$;

writer receive

$$N - f > \frac{N + f}{2}$$

replies and complete their operations.

To see the *validity* property, consider a read operation by process q that is not concurrent with any write. Assume that some process p executed the last write, and that p has written value v with associated timestamp wts. When the read is invoked, more than $(N + f)/2$ processes have acknowledged to p that they would store wts and v in their local state. The writer has not signed any pair with a larger timestamp than wts. When the reader obtains VALUE messages from more than $(N + f)/2$ processes, at least one of these message originates from a correct process and contains wts, v, and a valid signature from p. This holds because every two sets of more than $(N + f)/2$ processes overlap in at least one correct process, as they form Byzantine quorums. The reader hence returns v, the last written value, because no pair with a timestamp larger than wts passes the signature verification step.

Consider now the case where the read is concurrent with some write of value v with associated timestamp wts, and the previous write was for value v' and timestamp $wts - 1$. If any process returns wts to the reader q then q returns v, which is a valid reply. Otherwise, the reader receives at least one message containing v' and associated timestamp $wts - 1$ from a correct process and returns v', which ensures regular semantics.

Performance. The algorithm uses the same communication pattern as Algorithm 4.2 for regular registers in the fail-silent model and incurs two communication round-trips and $O(N)$ messages for every operation. The algorithm adds the cryptographic operations for creating and verifying digital signatures by the clients.

The same approach can also be used to transform Algorithm 4.6–4.7 for a $(1, N)$ atomic register in the fail-silent model into a $(1, N)$ *Byzantine* atomic register, defined analogously in the fail-arbitrary model.

4.7.3 Fail-Arbitrary Algorithm: Double-Write Byzantine Quorum

In this section, we describe an algorithm with resilience $N > 3f$ that implements a $(1, N)$ Byzantine regular register abstraction and does not use digital signatures. As we have seen with the previous algorithm, digital signatures greatly simplify the design of fail-arbitrary algorithms.

Tolerating less than $N/3$ arbitrary-faulty processes is optimal in this context. For instance, no algorithm for implementing even a safe register on $N = 3$ processes without data authentication tolerates only one Byzantine process. Consider how such an algorithm would operate. Even without concurrency, the read operation should be wait-free, that is, not block after receiving a reply from only $N - f = 2$ processes. Hence, the reader should choose a return value from only two replies. But it may be that the third process is correct and only slow, and one of the replies was forged by the Byzantine process. As the two responses look equally plausible, the reader might return the forged value and violate the *validity* property of the register in this case.

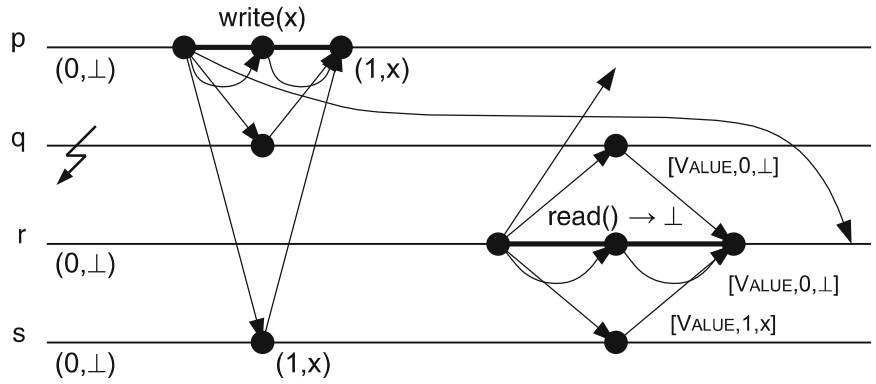

Figure 4.8: A nonregular register execution with one Byzantine process q and a one-phase write algorithm

In order to achieve optimal resilience $N > 3f$ and regular semantics, the writer process p uses two phases to write a new value in Algorithm 4.16–4.17, a *pre-write* phase and a *write* phase. This explains why the algorithm is called "Double-Write Byzantine Quorum."

A two-phase write operation is actually necessary to ensure regular semantics with resilience $N > 3f$, as the execution of a hypothetical algorithm with only one write phase in Fig. 4.8 illustrates. Suppose that the algorithm relies on a timestamp/value pair stored by all processes, like previous algorithms. Initially, every correct process stores $(0, \perp)$. If the algorithm would implement a Byzantine regular register abstraction then a write operation by process p to write the pair $(1, x)$ could have sent this to all processes, receive acknowledgments from processes p, q, and s, and complete. The message from p to process r is delayed. Consider a subsequently invoked read operation by r that obtains VALUE messages with replies from q, r, and s. Process q is Byzantine and replies with $[\text{VALUE}, 0, \perp]$, process r only knows the initial value and sends $[\text{VALUE}, 0, \perp]$, and process s replies with $[\text{VALUE}, 1, x]$. How should the reader select a return value?

- The reader cannot return x, because only one process replied with value x, so the value might have been forged by the faulty process and violate the *validity* property.
- The reader cannot return the initial value $\perp$, because the value x might indeed have been written before the read was invoked, as the execution shows. This would also violate the *validity* property.
- The reader must return *something* as required by the *termination* property. But the reader cannot afford to wait for a reply from process p. It may be that p has crashed and never sends a reply; indeed, if the writer had crashed in the middle of a write and only s had received its (single) message, the replies would represent a valid state.

Hence, the read will not satisfy the properties a regular register.

The two-phase write operation avoids this problem. In the first phase, the writer sends PREWRITE messages with the current timestamp/value pair. Then it waits until it receives PREACK messages from $N - f$ processes, which acknowledge

Algorithm 4.16: Double-Write Byzantine Quorum (part 1, write)

Implements:
$(1, N)$-ByzantineRegularRegister, **instance** *bonrr*, with writer w.
Uses:
AuthPerfectPointToPointLinks, **instance** al.

upon event $\langle$ *bonrr*, *Init* $\rangle$ **do**
$(pts, pval) := (0, \bot)$;
$(ts, val) := (0, \bot)$;
$(wts, wval) := (0, \bot)$;
preacklist $:= [\bot]^N$;
acklist $:= [\bot]^N$;
rid $:= 0$;
readlist $:= [\bot]^N$;

upon event $\langle$ *bonrr*, *Write* $\mid v \rangle$ **do** // only process w
$(wts, wval) := (wts + 1, v)$;
preacklist $:= [\bot]^N$;
acklist $:= [\bot]^N$;
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* $\mid q$, [PREWRITE, $wts, wval$] $\rangle$;

upon event $\langle$ *al*, *Deliver* $\mid p$, [PREWRITE, $pts', pval'$] $\rangle$
such that $p = w \wedge pts' = pts + 1$ **do**
$(pts, pval) := (pts', pval')$;
trigger $\langle$ *al*, *Send* $\mid p$, [PREACK, pts] $\rangle$;

upon event $\langle$ *al*, *Deliver* $\mid q$, [PREACK, pts'] $\rangle$ **such that** $pts' = wts$ **do**
preacklist$[q]$:= PREACK;
if $\#$(*preacklist*) $\geq N - f$ **then**
preacklist $:= [\bot]^N$;
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* $\mid q$, [WRITE, $wts, wval$] $\rangle$;

upon event $\langle$ *al*, *Deliver* $\mid p$, [WRITE, ts', val'] $\rangle$
such that $p = w \wedge ts' = pts \wedge ts' > ts$ **do**
$(ts, val) := (ts', val')$;
trigger $\langle$ *al*, *Send* $\mid p$, [ACK, ts] $\rangle$;

upon event $\langle$ *al*, *Deliver* $\mid q$, [ACK, ts'] $\rangle$ **such that** $ts' = wts$ **do**
acklist$[q]$:= ACK;
if $\#$(*acklist*) $\geq N - f$ **then**
acklist $:= [\bot]^N$;
trigger $\langle$ *bonrr*, *WriteReturn* $\rangle$;

that they have stored the data from the PREWRITE message. In the second phase, the writer sends ordinary WRITE messages, again containing the current timestamp/value pair. It then waits until it receives ACK messages from $N - f$ processes, which acknowledge that they have stored the data from the WRITE message.

Every process stores two timestamp/value pairs, one from the pre-write phase and one from the write phase. Intuitively, the above problem now disappears when

Algorithm 4.17: Double-Write Byzantine Quorum (part 2, read)

upon event $\langle$ *bonrr*, *Read* $\rangle$ **do**
    $rid := rid + 1$;
    $readlist := [\perp]^N$;
    forall $q \in \Pi$ **do**
        trigger $\langle$ *al*, *Send* | q, [READ, rid] $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, [READ, r] $\rangle$ **do**
    trigger $\langle$ *al*, *Send* | p, [VALUE, $r, pts, pval, ts, val$] $\rangle$;

upon event $\langle$ *al*, *Deliver* | q, [VALUE, $r, pts', pval', ts', val'$] $\rangle$ **such that** $r = rid$ **do**
    if $pts' = ts' + 1 \vee (pts', pval') = (ts', val')$ **then**
        $readlist[q] := (pts', pval', ts', val')$;
    if exists (ts, v) in an entry of *readlist* **such that** *authentic*$(ts, v, readlist) =$ TRUE
        and exists $Q \subseteq readlist$ **such that**
            $\#(Q) > \frac{N+f}{2} \wedge$ *selectedmax*$(ts, v, Q) =$ TRUE **then**
        $readlist := [\perp]^N$;
        trigger $\langle$ *bonrr*, *ReadReturn* | v $\rangle$;
    else
        trigger $\langle$ *al*, *Send* | q, [READ, r] $\rangle$;

the reader is no longer forced to select a value immediately. For example, if q and r reply with the timestamp/value pair $(0, \perp)$ for their pre-written and written pair, and s replies with $(1, x)$ for its pre-written and written pair, then the reader r can infer that one of the three processes must be faulty because the writer received $N - f$ acknowledgments during the pre-write phase for $(1, x)$ (hence, either s sent the wrong written pair or q sent the wrong pre-written pair, as r itself is obviously correct). In this case, process r can safely wait for a reply from the fourth process, which will break the tie.

The algorithm cannot quite satisfy the $(1, N)$ Byzantine regular register abstraction of Module 4.6 in its *termination* property. More precisely, for Algorithm 4.16–4.17, we relax the *termination* property (BONRR1) as follows. Instead of requiring that *every* operation of a correct process eventually terminates, a *read* operation that is concurrent with *infinitely* many write operations may not terminate. In other words, every write eventually completes and either every read eventually completes or the writer invokes infinitely many writes. Hence, algorithms implementing such registers satisfy only this so-called *finite-write termination* property, but are clearly not wait-free. It has been shown that such a relaxation is necessary.

Algorithm 4.16–4.17 relies on Byzantine quorums (Sect. 2.7.3). The reader sends a READ message to all processes as usual and waits for replies containing *two* timestamp/value pairs. According to the algorithm for writing, two cases may occur: either the pre-written timestamp is one higher than the written timestamp or the two pairs are equal; the reader retains such pairs. In any other case, the sender must be faulty and its reply is ignored.

The reader collects such replies in a variable *readlist*, where now every entry consists of two timestamp/value pairs. It stops collecting replies when some entry

of *readlist* contains a pair (ts, v) that (1) is found in the entries of more than f processes, and such that (2) there is a Byzantine quorum (Q) of entries in *readlist* whose *highest* timestamp/value pair, selected among the pre-written or written pair of the entries, is (ts, v).

More precisely, a timestamp/value pair (ts, v) is called *authentic* in *readlist*, and the predicate *authentic*$(ts, v, readlist)$ returns TRUE, whenever

$$\#\Big(\{p \mid readlist[p] = (pts', pv', ts', v') \\ \wedge \big((pts', pv') = (ts, v) \vee (ts', v') = (ts, v)\big)\}\Big) \; > \; f.$$

Hence, an authentic timestamp/value pair is found somewhere in the replies of more than f processes and cannot have been forged by a faulty process.

Furthermore, a pair (ts, v) is called a *selected maximum* in a list S with two timestamp/value pairs in every entry, and the predicate *selectedmax*(ts, v, S) is TRUE, whenever it holds for all $(pts', pv', ts', v') \in S$ that

$$\big(pts' < ts \wedge ts' < ts\big) \vee (pts', pv') = (ts, v) \vee (ts', v') = (ts, v).$$

Thus, a selected maximum pair satisfies, for every entry in S, that it is either equal to one of the timestamp/value pairs in the entry or its timestamp is larger than the timestamps in both pairs in the entry.

Given these notions, the read returns now only if there exists an authentic pair (ts, v) in some entry of *readlist* and a sublist $Q \subseteq readlist$ exists that satisfies $\#(Q) > (N + f)/2$ and *selectedmax*$(ts, v, Q) =$ TRUE.

Note that, the algorithm enforces FIFO-order delivery of all PREWRITE and WRITE messages from the writer to every process, through the timestamp included in these messages. This contrasts with previous algorithms for implementing registers, but here it ensures that a tuple (pts', pv', ts', v') containing the pre-written and written timestamp/value pairs of a correct process always satisfies $pts' = ts'$ or $pts' = ts' + 1$.

When the reader receives a VALUE message with a reply but the condition is not satisfied, the reader sends another READ message to the process and waits for more replies. This is needed to ensure liveness. As a process may not yet have received some messages from a complete write operation, waiting like this ensures that all correct processes eventually reply with timestamps and values that are at least as recent as the last complete write.

Note that, the reader cannot simply wait for the pair with the highest timestamp found anywhere in *readlist* to become authentic, because this might have been sent by a Byzantine process and contain an arbitrarily large timestamp. But, the correct processes send a reply with correct data, and therefore, enough replies are eventually received.

To illustrate the algorithm, consider the execution in Fig. 4.9, where all correct processes initially store $(2, x; 2, x)$ as their two timestamp/value pairs, and process q is Byzantine. Process p now starts to write y, and before it receives enough PREACK messages, process r invokes a read operation. Due to scheduling, the messages

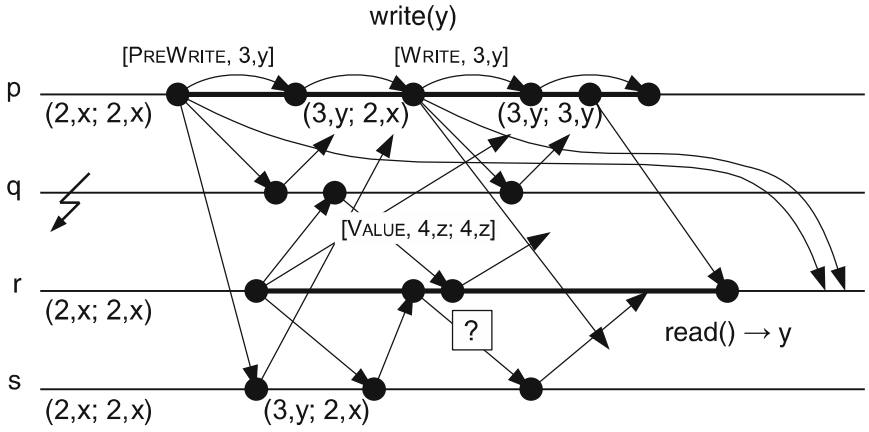

Figure 4.9: Sample execution of the "Double-Write Byzantine Quorum" algorithm implementing a regular register

from p to r are delayed. The reader r obtains the following VALUE messages with responses, each containing a pre-written and a written timestamp/value pair: the forged values $(4, z; 4, z)$ from q, the initial state $(2, x; 2, x)$ from r itself, and the half-way written state $(3, y; 2, x)$ from s. At this point, denoted by $\boxed{?}$ in the figure, the read cannot yet terminate because the only selected maximum in *readlist* is the pair $(4, z)$, but $(4, z)$ is not authentic. Hence, the reader continues by sending an additional READ message. Later on a response from p arrives, containing $(3, y; 3, y)$ from the write phase for value y. The variable *readlist* of the reader now contains four responses, and the sublist of *readlist* containing the responses from processes p, r, and s has $(3, y)$ as its selected maximum. As the pair $(3, y)$ is contained in the responses from p and s, it is also authentic and the read returns y.

Correctness. Assuming that $N > 3f$, the algorithm implements a $(1, N)$ Byzantine regular register abstraction with the relaxed *finite-write termination* property. It is easy to see that every write operation terminates because all $N - f$ correct processes properly acknowledge every PREWRITE and every WRITE message.

Read operations are only required to terminate when no more write operations are invoked. Assume that the last complete write operation o_w used a timestamp/value pair (ts, v) and a subsequent write operation o'_w failed because the writer crashed; if the writer crashed immediately after starting to execute o'_w, the situation looks for all other processes as if o'_w had never been invoked. Suppose o'_w used timestamp/value pair (ts', v') with $ts' = ts + 1$.

We distinguish two cases. If the writer crashed before sending any WRITE message during o'_w then since o_w completed, all $N - f$ correct processes eventually store (ts, v) as their written pair and do not change it afterward. Once they all reply with this value to the reader, the pair (ts, v) is authentic and represents a selected maximum of the Byzantine quorum corresponding to the replies of the

$$N - f > \frac{N + f}{2}$$

correct processes. Thus, the read returns v.

Otherwise, if the writer crashed after sending some WRITE message of o'_w then it has previously sent a PREWRITE message containing (ts', v') to all processes. All $N - f$ correct processes eventually store (ts', v') as their pre-written pair and do not change it afterward. Once they all reply with this value to the reader, the pair (ts', v') is authentic and represents a selected maximum of the Byzantine quorum corresponding to the replies of the correct processes. Thus, the read returns v'.

The arguments for these two cases demonstrate that the algorithm satisfies the *finite-write termination* property.

For the *validity* property, consider a read operation o_r and assume the last write operation o_w that completed before the invocation of o_r used a timestamp/value pair (ts, v). The writer may have invoked a subsequent operation to write some value $\overline{v}$. We need to ensure that the value returned by o_r is either v or $\overline{v}$. According to the algorithm, this means that the return value must be associated to timestamp ts or $ts + 1$. From the definition of authentic timestamp/value pairs used by the reader, and because at most f processes might reply with bad values to the reader, only a value written by the writer can be returned. Specifically, because o_w completed, at least $N-2f > f$ correct processes store a value with timestamp ts or higher in their pre-written and written pair. When their VALUE messages reach the reader during o_r, then (ts, v) or $(ts + 1, \overline{v})$ is authentic. Because the reader uses a Byzantine quorum Q of more than $(N + f)/2$ replies to determine the return value and because Q intersects in at least one correct process with the set of processes storing (ts, v) or $(ts + 1, \overline{v})$, the set Q contains at least one entry in which both timestamps are ts or higher. Thus, the selected maximum cannot contain a smaller timestamp than ts. Hence, o_r returns v or $\overline{v}$, and the *validity* property holds.

Performance. A write operation takes two communication roundtrips between the writer and all processes. A read operation that is not concurrent with a write involves only one communication roundtrip; otherwise, a read operation may continue to send messages as long as writes are concurrently invoked. In the absence of concurrency, both operations require $O(N)$ messages to be exchanged.

4.8 $(1, N)$ Byzantine Atomic Register

Recall the atomic register abstraction introduced in the model with crash-stop processes. Toward its clients, the register appears to execute one common sequence of atomic operations. In the fail-arbitrary model, with our restriction of readers and writers to crash faults, an atomic register obeys the same rules.

Implementing such an atomic register abstraction with Byzantine processes is more difficult than with crash-stop processes. As the presentation of the two $(1, N)$ Byzantine regular register algorithms in the previous section has shown, especially when no data authentication with digital signatures is available, algorithms may involve multiple rounds of communication between client and server processes. In this section, we will present an algorithm where the server processes may send response messages to a reading client spontaneously, without receiving an explicit request before triggering a response. This algorithm achieves the same resilience