Disk Paxos

Eli Gafni
Computer Science Department
UCLA

Leslie Lamport
Microsoft Research

14 May 2002

Abstract

We present an algorithm, called Disk Paxos, for implementing a reliable distributed system with a network of processors and disks. Like the original Paxos algorithm, Disk Paxos maintains consistency in the presence of arbitrary non-Byzantine faults. Progress can be guaranteed as long as a majority of the disks are available, even if all processors but one have failed.

Contents

1 Introduction

Fault tolerance requires redundant components. Maintaining consistency in the event of a system partition makes it impossible for a two-component system to make progress if either component fails. There are innumerable fault-tolerant algorithms for implementing distributed systems, but all that we know of equate *component* with *processor*. But there are other types of components that one might replicate instead. In particular, modern networks can now include disk drives as independent components. Because commodity disks are cheaper than computers, it is attractive to use them as the replicated components for achieving fault tolerance. Commodity disks differ from processors in that they are not programmable, so we can't just substitute disks for processors in existing algorithms.

We present here an algorithm called *Disk Paxos* for implementing an arbitrary fault-tolerant system with a network of processors and disks. It maintains consistency in the event of any number of non-Byzantine failures. That is, a processor may pause for arbitrarily long periods, may fail completely, and may restart after failure, remembering only that it has failed; a disk may become inaccessible to some or all processors, but it may not be corrupted. Disk Paxos guarantees progress if the system is stable and there is at least one nonfaulty processor that can read and write a majority of the disks. Stability means that each processor is either nonfaulty or has failed completely, and nonfaulty processors can access nonfaulty disks. For example, it allows a system of two processors and three disks to make progress after the failure of any one processor and any one disk.

Disk Paxos is a variant of the classic Paxos algorithm [3, 12, 14], a simple, efficient algorithm that has been used in practical distributed systems [15, 18]. Classic Paxos can be viewed as an implementation of Disk Paxos in which there is one disk per processor, and a disk can be accessed directly only by its processor.

In the next section, we recall how to reduce the problem of implementing an arbitrary distributed system to the consensus problem. Section 3 informally describes Disk Synod, the consensus algorithm used by Disk Paxos. It includes a sketch of an incomplete correctness proof and explains the relation between Disk Synod and the Synod protocol of classic Paxos. Section 4 briefly discusses some implementation details and contains the conventional concluding remarks. An appendix gives formal specifications of the consensus problem and the Disk Synod algorithm, and sketches a rigorous correctness proof.

An earlier version of this work, with an abridged version of the appendix lacking any proof, appeared earlier [5].

2 The State-Machine Approach

The state-machine approach [7, 16] is a general method for implementing an arbitrary distributed system. The system is designed as a deterministic state machine that executes a sequence of commands, and a consensus algorithm ensures that, for each n, all processors agree on the n^{th} command. This reduces the problem of building an arbitrary system to solving the consensus problem. In the consensus problem, each processor p starts with an input value $input[p]$, and it may output a value. A solution should be:

Nontrivial Any value output should have been the value of $input[p]$ at some time, for some processor p. (The value of $input[p]$ may change if p fails and restarts.)

Consistent All values output are the same.

Nonblocking If the system is stable and a nonfaulty processor can communicate with a majority of disks, then the processor will eventually output a value.

It has long been known that a consistent, nonblocking consensus using asynchronous message passing always requires at least two message delays [6]. Nonblocking algorithms that use fewer message delays don't guarantee consistency. For example, the group communication algorithms of Isis [2] permit two processors belonging to the current group to disagree on whether a message was broadcast in a previous group to which they both belonged. This algorithm cannot, by itself, guarantee consistency because disagreement about whether a message had been broadcast can result in disagreement about the output value.

The classic Paxos algorithm [3, 12, 14] uses a three-phase consensus protocol, called the *Synod* algorithm, where each of the first two phases requires two message delays and the third phase just broadcasts the output value. However, the value to be output is not chosen until the second phase. When a new leader is elected, it executes the first phase just once for the entire sequence of consensus algorithms performed for all later system commands. Only the last two phases are performed separately for each individual command.

In the *Disk Synod* algorithm, the consensus algorithm used by Disk Paxos, each processor has an assigned block on each disk. The algorithm has two phases. In each phase, a processor writes to its own block and reads each other processor's block on a majority of the disks.[1] Only the last phase needs to be executed anew for each command. So, in the normal steady-state case, a leader chooses a state-machine command by executing a single write to each of its blocks and a single read of every other processor's blocks.

Disk Paxos, like classic Paxos, makes no timing assumptions; processes may be completely asynchronous. The classic result of Fischer, Lynch, and Paterson [4] implies that a purely asynchronous nonblocking consensus algorithm is impossible. So, clocks and real-time assumptions must be introduced. The typical industry approach is to use an *ad hoc* algorithm based on timeouts to elect a leader, and then have the leader choose the output [17, 19]. It is easy to devise a leader-election algorithm that works when the system is stable, which means that it works most of the time. It is very hard to make one that always works correctly even when the system is unstable. Both classic Paxos and Disk Paxos also assume a real-time algorithm for electing a leader. However, the leader is used only to ensure progress. Consistency is maintained even if there are multiple leaders. Thus, if the leader-election algorithm fails because the network is unstable, the system can fail to make progress; it cannot become inconsistent. The system will again make progress when it becomes stable and a single leader is elected.

3 An Informal Description of Disk Synod

We now informally describe the Disk Synod algorithm and explain why it works. We also discuss its relation to classic Paxos's Synod Protocol. Remember that, in normal operation, only a single leader will be executing the algorithm. The other processors do nothing; they simply wait for the leader to inform them of the outcome. However, the algorithm must preserve consistency even when it is executed by multiple processors, or when the leader fails before announcing the outcome and a new leader is chosen.

3.1 The Algorithm

We assume that each processor p starts with an input value $input[p]$.[2] As in Paxos's Synod algorithm, a processor executes a sequence of numbered ballots, with increasing ballot numbers. A ballot number is a positive integer, and different processors use different ballot numbers. For example, if

[1] There is also an extra phase that a processor executes when recovering from a failure.

[2] If processor p fails, it can restart with a new value of $input[p]$.

the processors are numbered from 1 through N, then processor i could use ballot numbers i, $i + N$, $i + 2N$, etc. A processor p executes a ballot in two phases, the first trying to choose a value and the second trying to commit that value:

Phase 1 Determine whether p can choose its input value $input[p]$ or must choose some other value.

Choose a value v.

Phase 2 Try to commit v.

The choice of the value v occurs in the transition from phase 1 to phase 2. The value is committed, and can be output, when p finishes phase 2.

In either phase, a processor aborts its ballot if it learns that another processor has begun a higher-numbered ballot. In that case, the processor may then choose a higher ballot number and start a new ballot. (It will do so if it still thinks it is the leader.) If the processor completes phase 2 without aborting—that is, without learning of a higher-numbered ballot—then value v is *committed* and the processor can output it. A processor p does not need to know the value of $input[p]$ until it enters phase 2, so phase 1 can be performed in advance for any number of separate instances of the algorithm.

To ensure consistency, we must guarantee that two different values cannot be successfully committed—either by different processors (because the leader-election algorithm has not yet succeeded) or by the same processor in two different ballots (because it failed and restarted). To ensure that the algorithm is nonblocking, we must guarantee that, if there is only a single processor p executing it, then p will eventually commit a value.

In practice, when a processor successfully commits a value, it will write on its disk block that the value was committed and also broadcast that fact to the other processors. If a processor learns that a value has been committed, it will abort its ballot and simply output the value. It is obvious that this optimization preserves correctness; we will not consider it further.

To execute the algorithm, a processor p maintains a record $dblock[p]$ containing the following three components:

mbal The current ballot number.

bal The largest ballot number for which p entered phase 2.

inp The value p tried to commit in ballot number bal.

Initially, *bal* equals 0, *inp* equals a special value *NotAnInput* that is not a possible input value, and *mbal* is any of its possible ballot numbers. We let $disk[d][p]$ be the block on disk d in which processor p writes $dblock[p]$. We assume that reading and writing a block are atomic operations.

Processor p executes phase 1 or 2 of a ballot as follows. For each disk d, it tries first to write $dblock[p]$ to $disk[d][p]$ and then to read $disk[d][q]$ for all other processors q. It aborts the ballot if, for any d and q, it finds $disk[d][q].mbal > dblock[p].mbal$. The phase completes when p has written and read a majority of the disks, without reading any block whose *mbal* component is greater than $dblock[p].mbal$. When it completes phase 1, p chooses a new value of $dblock[p].inp$, sets $dblock[p].bal$ to $dblock[p].mbal$ (its current ballot number), and begins phase 2. When it completes phase 2, p has committed $dblock[p].inp$.

To complete our description of the two phases, we now describe how processor p chooses the value of $dblock[p].inp$ that it tries to commit in phase 2. Let *blocksSeen* be the set consisting of $dblock[p]$ and all the records $disk[d][q]$ read by p in phase 1. Let *nonInitBlks* be the subset of *blocksSeen* consisting of those records whose *inp* field is not *NotAnInput*. If *nonInitBlks* is empty, then p sets $dblock[p].inp$ to its own input value $input[p]$. Otherwise, it sets $dblock[p].inp$ to $bk.inp$ for some record bk in *nonInitBlks* having the largest value of $bk.bal$.

Finally, we describe what processor p does when it recovers from a failure. In this case, p reads its own block $disk[d][p]$ from a majority of disks d. It then sets $dblock[p]$ to any block bk it read having the maximum value of $bk.mbal$, and it starts a new ballot by increasing $dblock[p].mbal$ and beginning phase 1.

The algorithm is summarized informally in Figure 1, which describes how a processor p executes a single ballot. The processor begins the ballot by executing the Start Ballot operation. It can begin a new ballot if a ballot aborts, or at any other time—except when it has failed, in which case it must execute the Restart After Failure operation. A precise specification of the algorithm appears in the appendix.

3.2 Why the Algorithm Works

Safety

We explain intuitively why the Disk Synod algorithm satisfies the two safety properties of nontriviality and consistency. Nontriviality is trivial, since the *val* field of any block is always set either to the *val* field of some other block

Start Ballot

Set $dblock[p].mbal$ to a value larger than its current value.

Set $blocksSeen$ to $\{dblock[p]\}$.

Begin Phase 1.

Phase 1 or 2

Concurrently for every disk d **do**

Write $dblock[p]$ to $disk[d][p]$.

for every processor $q \neq p$ **do**

Read $disk[d][q]$ and insert it in the set $blocksSeen$.

Abort the ballot if $disk[d][q].mbal > dblock[p].mbal$.

until this has been done for a majority of disks.

If phase 1

then Set $dblock[p].bal$ to $dblock[p].mbal$.

Let $nonInitBlks$ be the set of elements bk in $blocksSeen$ with $bk.inp \neq NotAnInput$.

If $nonInitBlks$ is empty

then Set $dblock[p].inp$ to $input[p]$.

else Set $dblock[p].inp$ to an element bk of $nonInitBlks$ with a maximal value of $bk.bal$.

Begin phase 2.

else Commit $dblock[p].inp$.

Restart After Failure

Set $tempSet$ to the empty set.

Concurrently for every disk d **do**

Read $disk[d][q]$ and insert it in the set $tempSet$.

until this has been done for a majority of disks.

Set $dblock[p]$ to an element bk of $tempSet$ with a maximal value of $mbal$.

Begin Start Ballot.

Figure 1: The algorithm by which a processor p executes a single ballot.

or to *input*$[p]$ for a processor p. Hence, a committed value must at one time have been an input value of some processor.

We now explain why the Disk Synod algorithm maintains consistency. First, we consider the following shared-memory version of the algorithm that uses single-writer, multiple-reader regular registers.[3] Instead of writing to disk, processor p writes *dblock*$[p]$ to a shared register; and it reads the values of *dblock*$[q]$ for other processors q from the registers. A processor chooses its *bal* and *inp* values for phase 2 the same way as before, except that it reads just one *dblock* value for each other processor, rather than one from each disk. We assume for now that processors do not fail.

To prove consistency, we must show that, for any processors p and q, if p finishes phase 2 and commits the value v_p and q finishes phase 2 and commits the value v_q, then $v_p = v_q$. Let b_p and b_q be the respective ballot numbers on which these values are committed. Without loss of generality, we can assume $b_p \leq b_q$. Moreover, using induction on b_q, we can assume that, if any processor r starts phase 2 for a ballot b_r with $b_p \leq b_r < b_q$, then it does so with $dblock[r].inp = v_p$.

When reading in phase 2, p cannot have seen the value of $dblock[q].mbal$ written by q in phase 1—otherwise, p would have aborted. Hence p's read of *dblock*$[q]$ in phase 2 did not follow q's phase 1 write. Because reading follows writing in each phase, this implies that q's phase 1 read of *dblock*$[p]$ must have followed p's phase 2 write. Hence, q read the current (final) value of *dblock*$[p]$ in phase 1—a record with *bal* field b_p and *inp* field v_p. Let *bk* be any other block that q read in its phase 1. Since q did not abort, $b_q > bk.mbal$. Since $bk.mbal \geq bk.bal$ for any block *bk*, this implies $b_q > bk.bal$. By the induction assumption, we obtain that, if $bk.bal \geq b_p$, then $bk.inp = v_p$. Since this is true for all blocks *bk* read by q in phase 1, and since q read the final value of *dblock*$[p]$, the algorithm implies that q must set $dblock[q].inp$ to v_p for phase 2, proving that $v_p = v_q$.

To obtain the Disk Synod algorithm from the shared-memory version, we use a technique due to Attiya, Bar-Noy, and Dolev [1] to implement a single-writer, multiple reader register with a network of disks. To write a value, a processor writes the value together with a version number to a majority of the disks. To read, a processor reads a majority of the disks and takes the value with the largest version number. Since two majorities of disks contain at least one disk in common, a read must obtain either the last

[3] A regular register is one in which a read that does not overlap a write returns the register's current value, and a read that overlaps one or more writes returns either the register's previous value or one of the values being written [8].

version for which the write was completed, or else a later version. Hence, this implements a regular register. With this technique, we transform the shared-memory version into a version for a network of processors and disks.

The actual Disk Synod algorithm simplifies the algorithm obtained by this transformation in two ways. First, the version number is not needed. The *mbal* and *bal* values play the role of a version number. Second, a processor p need not choose a single version of $dblock[q]$ from among the ones it reads from disk. Because *mbal* and *bal* values do not decrease, earlier versions have no effect.

So far, we have ignored processor failures. There is a trivial way to extend the shared-memory algorithm to allow processor failures. A processor recovers by simply reading its *dblock* value from its register and starting a new ballot. A failed process then acts like one in which a processor may start a new ballot at any time. We can show that this generalized version is also correct. However, in the actual disk algorithm, a processor can fail while it is writing. This can leave its disk blocks in a state in which no value has been written to a majority of the disks. Such a state has no counterpart in the shared-memory version. There seems to be no easy way to derive the recovery procedure from a shared-memory algorithm. The proof of the complete Disk Synod algorithm, with failures, is much more complicated than the one for the simple shared-memory version. Trying to write the kind of behavioral proof given above for the simple algorithm leads to the kind of complicated, error-prone reasoning that we have learned to avoid. Instead, we sketch a rigorous assertional proof in the appendix.

Liveness

Liveness (progress) of the Disk Synod algorithm requires liveness of a leader-election algorithm. A processor executes steps of the Disk Synod algorithm iff it believes itself to be the leader. We show that a value will be committed if, eventually, a single nonfaulty processor p that can read and write a majority of the disks is forever the unique leader.[4]

Suppose p is the unique leader and it can read and write a majority of the disks. Since p can access a majority of the disks, each phase it executes either completes or aborts. A phase aborts only if p reads an *mbal* value greater than its own, and p increases its own *mbal* value when it does abort. Since p is the unique leader, only it writes to the disks. So, if p does not complete phases 1 and 2, then its *mbal* value will eventually be greater than

[4] Actually, p needs to be the unique leader just long enough to commit the value.

that of every disk block that it reads. Hence, p must eventually complete phases 1 and 2 without aborting, thus committing a value.

3.3 Deriving Classic Paxos from Disk Paxos

In the usual view of a distributed fault-tolerant system, a processor performs actions and maintains its state in local memory, using stable storage to recover from failures. An alternative view is that a processor maintains the state of its stable storage, using local memory only to cache the contents of stable storage. Identifying disks with stable storage, a traditional distributed system is then a network of disks and processors in which each disk belongs to a separate processor; other processors can read a disk only by sending messages to its owner.

Let us now consider how to implement Disk Synod on a network of processors that each has its own disk. To perform phase 1 or 2, a processor p would access a disk d by sending a message containing $dblock[p]$ to disk d's owner q. Processor q could write $dblock[p]$ to $disk[d][p]$, read $disk[d][r]$ for all $r \neq p$, and send the values it read back to p. However, examining the Disk Synod algorithm reveals that there's no need to send back all that data. All p needs are (i) to know if its *mbal* field is larger than any other block's *mbal* field and, if it is, (ii) the *bal* and *inp* fields for the block having the maximum *bal* field. Hence, q need only store on disk three values: the *bal* and *inp* fields for the block with maximum *bal* field, and the maximum *mbal* field of all disk blocks. Of course, q would have those values cached in its memory, so it would actually write to disk only if any of those values are changed.

A processor must also read its own disk blocks to recover from a failure. Suppose we implement Disk Synod by letting p write to its own disk before sending messages to any other processor. This ensures that its own disk has the maximum value of $disk[d][p].mbal$ among all the disks d. Hence, to restart after a failure, p need only read its block from its own disk. In addition to the *mbal*, *bal*, and *inp* value mentioned above, p would also keep the value of $dblock[p]$ on its disk.

We can now compare this algorithm with classic Paxos's Synod protocol [12]. The *mbal*, *bal*, and *inp* components of $dblock[p]$ are just $lastTried[p]$, $nextBal[p]$, and $prevVote[p]$ of the Synod Protocol. Phase 1 of the Disk Synod algorithm corresponds to sending the *NextBallot* message and receiving the *LastVote* responses in the Synod Protocol. Phase 2 corresponds to sending the *BeginBallot* and receiving the *Voted* replies.[5] The Synod Pro-

[5]In the Synod Protocol, a processor q does not bother sending a response if p sends

tocol's *Success* message corresponds to the optimization mentioned above of recording on disk that a value has been committed.

This version of the Disk Synod algorithm differs from the Synod Protocol in two ways. First, the Synod Protocol's *NextBallot* message contains only the *mbal* value; it does not contain *bal* and *inp* values. To obtain the Synod Protocol, we would have to modify the Disk Synod algorithm so that, in phase 1, it writes only the *mbal* field of its disk block and leaves the *bal* and *inp* fields unchanged. The algorithm remains correct, with essentially the same proof, under this modification. However, the modification makes the algorithm harder to implement with real disks.

The second difference between this version of the Disk Synod algorithm and the Synod Protocol is in the restart procedure. A disk contains only the aforementioned *mbal*, *bal*, and *inp* values. It does not contain a separate copy of its owner's *dblock* value. The Synod Protocol can be obtained from the following variant of the Disk Synod algorithm. Let *bk* be the block $disk[d][p]$ with maximum *bal* field read by processor p in the restart procedure. Processor p can begin phase 1 with *bal* and *inp* values obtained from any disk block bk', written by any processor, such that $bk'.bal \geq bk.bal$. It can be shown that the Disk Synod algorithm remains correct under this modification too.

4 Conclusion

4.1 Implementation Considerations

Implicit in our description of the Disk Synod algorithm are certain assumptions about how reading and writing are implemented when disks are accessed over a network. If operations sent to the disks may be lost, a processor p must receive an acknowledgment from disk d that its write to $disk[d][p]$ succeeded. This may require p to explicitly read its disk block after writing it. If operations may arrive at the disk in a different order than they were sent, p will have to wait for the acknowledgment that its write to disk d succeeded before reading other processors' blocks from d. Moreover, some mechanism is needed to ensure that a write from an earlier ballot does not arrive after a write from a later one by the same processor, overwriting the later value with the earlier one. How this is achieved will be system dependent. (It is impossible to implement any fault-tolerant system if writes to

it a disk block with a value of *mbal* smaller than one already on disk. Sending back the maximum *mbal* value is an optimization mentioned in [12].

disk can linger arbitrarily long in the network and cause later values to be overwritten.)

Recall that, in Disk Paxos, a sequence of instances of the Disk Synod algorithm is used to commit a sequence of commands. In a straightforward implementation of Disk Paxos, processor p would write to its disk blocks the value of $dblock[p]$ for the current instance of Disk Synod, plus the sequence of all commands that have already been committed. The sequence of all commands that have ever been committed is probably too large to fit on a single disk block. However, the complete sequence can be stored on multiple disk blocks. All that must be kept in the same disk block as $dblock[p]$ is a pointer to the head of the queue. For most applications, it is not necessary to remember the entire sequence of commands [12, Section 3.3.2]. In many cases, all the data that must be kept will fit in a single disk block.

In the application for which Disk Paxos was devised (a future Compaq product), the set of processors is not known in advance. Each disk contains a directory listing the processors and the locations of their disk blocks. Before reading a disk, a processor reads the disk's directory. To write a disk's directory, a processor must acquire a lock for that disk by executing a real-time mutual exclusion algorithm based on Fischer's protocol [9]. A processor joins the system by adding itself to the directory on a majority of disks.

4.2 Concluding Remarks

We have presented Disk Paxos, an efficient implementation of the state machine approach in a system in which processors communicate by accessing ordinary (nonprogrammable) disks. In the normal case, the leader commits a command by writing its own block and reading every other processor's block on a majority of the shared disks. This is clearly the minimal number of disk accesses needed for a consensus algorithm that can make progress despite the failure of any minority of the disks and of any single processor.

Disk Paxos was motivated by the recent development of the Storage Area Network (SAN)—an architecture consisting of a network of computers and disks in which all disks can be accessed by each computer. Commodity disks are cheaper than computers, so using redundant disks for fault tolerance is more economical than using redundant computers. Moreover, since disks do not run application-level programs, they are less likely to crash than computers.

Because commodity disks are not programmable, we could not simply substitute disks for processors in the classic Paxos algorithm. Instead we took the ideas of classic Paxos and transplanted them to the SAN environment. What we obtained is almost, but not quite, a generalization of classic Paxos. Indeed, when Disk Paxos is instantiated to a single disk, we obtain what may be called Shared-Memory Paxos. Algorithms for shared memory are usually more succinct and clear than their message passing counterparts. Thus, Disk Paxos for a single disk can be considered yet another revisiting of classic Paxos that exposes its underlying ideas by removing the message-passing clutter. Perhaps other distributed algorithms can also be made more clear by recasting them in a shared-memory setting.

References

[1] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, January 1995.

[2] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[3] Roberto De Prisco, Butler Lampson, and Nancy Lynch. Revisiting the PAXOS algorithm. *Theoretical Computer Science*, 243:35–91, 2000.

[4] Michael J. Fischer, Nancy Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[5] Eli Gafni and Leslie Lamport. Disk paxos. In Maurice Herlihy, editor, *Distributed Computing: 14th International Conference, DISC 2000*, volume 1914 of *Lecture Notes in Computer Science*, pages 330–344. Springer-Verlag, 2000.

[6] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults—a tutorial. TechnicalReport MIT-LCS-TR-821, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, 02139, May 2001. also published in SIGACT News 32(2) (June 2001).

[7] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[8] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1:77–101, 1986.

[9] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[10] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[11] Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, August-September 1995.

[12] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[13] Leslie Lamport. Specifying concurrent systems with TLA$^+$. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, pages 183–247, Amsterdam, 1999. IOS Press.

[14] Butler W. Lampson. How to build a highly available system using consensus. In Ozalp Babaoglu and Keith Marzullo, editors, *Distributed Algorithms*, volume 1151 of *Lecture Notes in Computer Science*, pages 1–17, Berlin, 1996. Springer-Verlag.

[15] Edward K. Lee and Chandramohan Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 84–92, New York, October 1996. ACM Press.

[16] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[17] William E. Snaman, Jr. Application design in a VAXcluster system. *Digital Technical Journal*, 3(3):16–26, Summer 1991.

[18] Chandramohan Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, New York, October 1997. ACM Press.

[19] Werner Vogels et al. The design and architecture of the microsoft cluster service. In *Proceedings of FTCS98*, pages 422–431. IEEE, June 1998.

[20] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66, Berlin, Heidelberg, New York, September 1999. Springer-Verlag. 10th IFIP wg 10.5 Advanced Research Working Conference, CHARME '99.

Appendix

We now give a precise specification of the consensus problem solved by the Disk Synod algorithm and of the algorithm itself. The specification is written in TLA$^+$ [13], a formal language that combines the temporal logic of actions (TLA) [10], set theory, and first-order logic with notation for making definitions and encapsulating them in modules. In the course of writing the specifications, we try to explain any TLA$^+$ notation whose meaning is not self-evident. These specifications have been debugged with the aid of the TLC model checker [20].[6]

We prove only consistency of the algorithm. We feel that the nonblocking property is sufficiently obvious not to need a formal proof. We therefore do not specify or reason about liveness properties. This means that we make hardly any use of temporal logic.

A.1 The Specification of Consensus

We now formally specify the consensus problem. We assume N processors, numbered 1 through N. Each processor p has two registers: an input register $input[p]$ that initially equals some element of a set $Inputs$ of possible input values, and an output register $output[p]$ that initially equals a special value $NotAnInput$ that is not an element of $Inputs$. Processor p chooses an output value by setting $output[p]$. It can also fail, which it does by setting $input[p]$ to any value in $Inputs$ and resetting $output[p]$ to $NotAnInput$. The precise condition to be satisfied is that, if some processor p ever sets $output[p]$ to some value v, then

- v must be a value that is, or at one time was, the value of $input[q]$ for some processor q

[6] The typeset versions were generated manually from the actual TLA$^+$ specifications by a procedure that may have introduced errors.

- if any processor r (including p itself) later sets $output[r]$ to some value w other than $NotAnInput$, then $w = v$.

We specify only safety. There is no liveness requirement, so the specification is satisfied if no processor ever changes $output[p]$.

TLA$^+$ specifications are organized into modules. The specification of consensus is in a module named *Synod*, which begins:

MODULE *Synod*

EXTENDS *Naturals*

The EXTENDS statement imports the *Naturals* module, which defines the set *Nat* of natural numbers and the usual arithmetic operations. It also defines $i \mathrel{..} j$ to be the set of natural numbers from i through j. We next declare the specification's two constants: the number N of processors, and the set *Inputs* of inputs; and we assert the assumption that N is a positive natural number. (TLA$^+$, like ordinary mathematics, is untyped.)

CONSTANT N, *Inputs*
ASSUME $(N \in Nat) \land (N > 0)$

In TLA$^+$, every value is a set, so we don't have to assert that *Inputs* is a set. We next define two constants: the set *Proc* of processors, and the value *NotAnInput*. In TLA$^+$, $\triangleq$ means *is defined to equal*, and CHOOSE $x : F(x)$ equals an arbitrary value x such that $F(x)$ is true (if such an x exists).

$Proc \;\triangleq\; 1 \mathrel{..} N$
$NotAnInput \;\triangleq\; \text{CHOOSE } c : c \notin Inputs$

Note that the constants *Proc* and *NotAnInput* are defined, while the constants N and *Inputs* are simply declared.

We next declare the variables *input* and *output*.

VARIABLES *input*, *output*

To write the specification, we introduce two internal variables: *allInput*, which equals the set of all current and past values of $input[p]$, for all processors p; and *chosen*, which records the first input value output by some processor (and hence, the value that all processors must henceforth output). These variables are internal or "hidden" variables. In TLA, such variables are bound variables of the temporal existential quantifier $\boldsymbol{\exists}$. Since internal variables aren't part of the specification, they should not be declared in module *Synod*. One way to introduce such variables in TLA$^+$ is to declare them in a submodule. So, we introduce a submodule called *Inner*.

MODULE *Inner*

VARIABLES *allInput*, *chosen*

Before going further, we explain some TLA$^+$ notation. In programming languages, the variables *input* and *output* would be arrays indexed by the *Proc*. What programmers call an array indexed by S, mathematicians call a function with domain S. TLA$^+$ uses the notation $[x \in S \mapsto e(x)]$ for the function f with domain S such that $f[x] = e(x)$ for all x in S. It denotes by $[S \to T]$ the set of all functions f with domain S such that $f[x] \in T$ for all $x \in S$. TLA$^+$ allows a conjunction or disjunction to be written as a list of formulas bulleted by $\wedge$ or $\vee$. Indentation is used to eliminate parentheses.

We now define *IInit* to be the predicate describing the initial state.

$$
\begin{array}{lll}
\mathit{IInit} \triangleq & \wedge\ \mathit{input} & \in\ [\mathit{Proc} \to \mathit{Inputs}] \\
& \wedge\ \mathit{output} & =\ [p \in \mathit{Proc} \mapsto \mathit{NotAnInput}] \\
& \wedge\ \mathit{chosen} & =\ \mathit{NotAnInput} \\
& \wedge\ \mathit{allInput} & =\ \{\mathit{input}[p] : p \in \mathit{Proc}\}
\end{array}
$$

We next define the two actions, $\mathit{IChoose}(p)$ and $\mathit{IFail}(p)$, that describe the operations that a processor p can perform. In TLA, an action is a formula with primed and unprimed variables that describes the relation between the values of the variables in a new (primed) state and their values in an old (unprimed) state. For example, in a system with the two variables x and y, the action $(x' = x + 1) \wedge (y' = y)$ corresponds to the programming-language statement $x := x + 1$. A conjunct with no primed variables is an enabling condition.

In TLA$^+$, the expression $[f \text{ EXCEPT } ![x] = e]$ represents the function $\widehat{f}$ that is the same as f except that $\widehat{f}[x] = e$. Thus, $f' = [f \text{ EXCEPT } ![c] = e]$ corresponds to the programming-language statement $f[c] := e$, except that it says nothing about variables other than f (whereas $f[c] := e$ asserts that other variables are unchanged). An action must explicitly state what remains unchanged. We do this with the expression UNCHANGED v, which means $v' = v$. Leaving a tuple $\langle v_1, \ldots, v_n \rangle$ unchanged is equivalent to leaving all its components v_i unchanged.

The $\mathit{IChoose}(p)$ action represents the processor p choosing its output. It is enabled iff $\mathit{output}[p]$ equals *NotAnInput*. If *chosen* is *NotAnInput*, then *chosen* and $\mathit{output}[p]$ are set to any element of *allInput*. Otherwise, $\mathit{output}[p]$ is set to *chosen*.

$$
\begin{array}{l}
\mathit{IChoose}(p) \triangleq \\
\quad \wedge\ \mathit{output}[p] = \mathit{NotAnInput}
\end{array}
$$

$$
\begin{array}{l}
\land \ \text{IF} \ \ chosen = NotAnInput \\
\quad \ \ \text{THEN} \ \exists\, ip \in allInput : \land\ chosen' \ = \ ip \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \land\ output' \ = \ [output \ \text{EXCEPT} \ ![p] = ip] \\
\quad \ \ \text{ELSE} \ \ \land\ output' = [output \ \text{EXCEPT} \ ![p] = chosen] \\
\qquad\qquad\quad \land\ \text{UNCHANGED} \ chosen \\
\land \ \text{UNCHANGED} \ \langle input, allInput \rangle
\end{array}
$$

The $IFail(p)$ action represents processor p failing. It is always enabled. It sets $output[p]$ to $NotAnInput$, sets $input[p]$ to any element of $Inputs$, and adds that element to the set $allInput$.

$$
\begin{array}{l}
IFail(p) \ \triangleq \\
\quad \land\ output' = [output \ \text{EXCEPT} \ ![p] = NotAnInput] \\
\quad \land\ \exists\, ip \in Inputs : \land\ input' \ \ = \ [input \ \text{EXCEPT} \ ![p] = ip] \\
\qquad\qquad\qquad\qquad\quad\ \land\ allInput' \ = \ allInput \cup \{ip\} \\
\quad \land\ \text{UNCHANGED} \ chosen
\end{array}
$$

We next define the next-state action $INext$, which describes all possible steps. We then define $ISpec$, the specification with the internal variables $chosen$ and $allInput$ visible. It asserts that the initial state satisfies $IInit$, and every step either satisfies $INext$ or leaves all the variables unchanged. Formula $ISpec$ is defined to be a temporal formula, using the ordinary operator $\Box$ (always) of temporal logic, and the TLA notation that $[A]_v$ equals $A \lor (v' = v)$, for any action A and state function v, for any action A and state function v. These definitions end the submodule.

$$
\begin{array}{l}
INext \ \triangleq \ \exists\, p \in Proc : IChoose(p) \lor IFail(p) \\
ISpec \ \triangleq \ IInit \land \Box[INext]_{\langle input,\, output,\, chosen,\, allInput \rangle}
\end{array}
$$

Finally, we define $SynodSpec$, the complete specification, to be $ISpec$ with the variables $chosen$ and $allInput$ hidden—that is, quantified with the temporal existential quantifier $\boldsymbol{\exists}$ of TLA. The precise meaning of the TLA$^+$ constructs used here is unimportant.

$$
\begin{array}{l}
IS(chosen, allInput) \ \triangleq \ \text{INSTANCE} \ Inner \\
SynodSpec \ \triangleq \ \boldsymbol{\exists}\, chosen, allInput : IS(chosen, allInput)!ISpec
\end{array}
$$

This ends module $Synod$.

A.2 The Disk Synod Algorithm

The Disk Synod algorithm is specified by a module *DiskSynod* that imports all the declarations and definitions from the *Synod* module.

MODULE *DiskSynod*

EXTENDS *Synod*

The algorithm assumes that different processors use different ballot numbers. Instead of fixing some specific choice of ballot numbers, we let $Ballot(p)$ represent the set of ballot numbers that processor p can use, where *Ballot* is an unspecified constant operator.

We have described the algorithm in terms of a majority of disks. The property of majorities we need is that any two majorities has a disk in common. If there are an even number d of disks, we can maintain that property even if we consider certain sets containing $d/2$ disks to constitute a majority. We let *IsMajority* be an unspecified predicate so that if $IsMajority(S)$ and $IsMajority(T)$ is true for two sets S and T of disks, then S and T are not disjoint. (To rule out the trivial case when no set is a majority, we require that $IsMajority(Disk)$ be true.)

The module now declares *Ballot*, *IsMajority*, and the constant *Disk* that represents the set of disks. It also asserts the assumptions we make about them. In TLA$^+$, the expression SUBSET S denotes the set of all subsets of the set S.

CONSTANTS $Ballot(\_),\ Disk,\ IsMajority(\_)$

$$
\begin{array}{ll}
\text{ASSUME} & \land\ \forall p \in Proc : \land\ Ballot(p) \subseteq \{n \in Nat : n > 0\} \\
& \qquad\qquad\qquad\quad \land\ \forall q \in Proc \setminus \{p\} : Ballot(p) \cap Ballot(q) = \{\} \\
& \land\ IsMajority(Disk) \\
& \land\ \forall S, T \in \text{SUBSET } Disk : \\
& \qquad IsMajority(S) \land IsMajority(T) \Rightarrow (S \cap T \neq \{\})
\end{array}
$$

We next define two constants: the set *DiskBlock* of all possible records that a processor can write to its disk blocks, and the record *InitDB* that is the initial value of all disk blocks. In TLA$^+$, $[f_1 \mapsto v_1, \ldots, f_n \mapsto v_n]$ is the record r with fields $f_1, \ldots, f_n$ such that $r.f_i = v_i$, for all i in $1 \,..\, n$, and $[f_1 : S_1, \ldots, f_n : S_n]$ is the set of all such records with v_i an element of the set S_i, for all i in $1 \,..\, n$. The set $\bigcup S$, the union of all the elements of S, is written UNION S. For example, UNION $\{A, B, C\}$ equals $A \cup B \cup C$.

$$
\begin{array}{lll}
DiskBlock \triangleq & [\, mbal & : (\text{UNION } \{Ballot(p) : p \in Proc\}) \cup \{0\}, \\
& \ \ bal & : (\text{UNION } \{Ballot(p) : p \in Proc\}) \cup \{0\}, \\
& \ \ inp & : Inputs \cup \{NotAnInput\}\,]
\end{array}
$$

$$InitDB \triangleq [mbal \mapsto 0,\ bal \mapsto 0,\ inp \mapsto NotAnInput]$$

We now declare all the specification's variables—except for *input* and *output*, whose declarations are imported from *Synod*. We have described the variables *disk* (the contents of the disks) and *dblock* in Section 3. We let $phase[p]$ be the current phase of processor p, which will be set to 0 when p fails and to 3 when p chooses its output. For convenience, we let each processor start in phase 0 and begin the algorithm as if it were recovering from a failure. The variables *disksWritten* and *blocksRead* record a processor's progress in the current phase; $disksWritten[p]$ is the set of disks that processor p has written, and $blocksRead[p][d]$ is the set of values p has read from disk d. More precisely, $blocksRead[p][d]$ is a set of records with *block* and *proc* fields, where $[block \mapsto bk,\ proc \mapsto q]$ is in $blocksRead[p][d]$ iff p has read the value bk from $disk[d][q]$ in the current phase. For convenience, we declare *vars* to be the tuple of all the specification's variables. We also define the predicate *Init* that defines the initial values of all variables.

$$\text{VARIABLES } disk,\ dblock,\ phase,\ disksWritten,\ blocksRead$$

$$vars \triangleq \langle input,\ output,\ disk,\ phase,\ dblock,\ disksWritten,\ blocksRead \rangle$$

$$\begin{aligned} Init \triangleq\ & \land\ input \in [Proc \rightarrow Inputs] \\ & \land\ output = [p \in Proc \mapsto NotAnInput] \\ & \land\ disk = [d \in Disk \mapsto [p \in Proc \mapsto InitDB]] \\ & \land\ phase = [p \in Proc \mapsto 0] \\ & \land\ dblock = [p \in Proc \mapsto InitDB] \\ & \land\ disksWritten = [p \in Proc \mapsto \{\}] \\ & \land\ blocksRead = [p \in Proc \mapsto [d \in Disk \mapsto \{\}]] \end{aligned}$$

We now define two operators that describe the state of a processor during the current phase: $hasRead(p, d, q)$ is true iff p has read $disk[d][q]$, and $allBlocksRead(p)$ equals the set of all $disk[d][q]$ values that p has read during the current phase. The TLA$^+$ expression LET *def* IN *exp* equals expression *exp* in the context of the local definitions in *def*.

$$hasRead(p, d, q) \triangleq \exists\, br \in blocksRead[p][d] : br.proc = q$$

$$\begin{aligned} & allBlocksRead(p) \triangleq \\ & \quad \text{LET } allRdBlks \triangleq \text{UNION } \{blocksRead[p][d] : d \in Disk\} \\ & \quad \text{IN } \{br.block : br \in allRdBlks\} \end{aligned}$$

We now define $InitializePhase(p)$ to be an action that sets $disksWritten[p]$ and $blocksRead[p]$ to their initial values, to indicate that p has done no reading or writing yet in the current phase. This action will be used to define other actions that make up the next-state relation; it itself is not part of the next-state relation.

$$
\begin{array}{l}
\mathit{InitializePhase}(p) \;\triangleq \\
\quad \land\ \mathit{disksWritten}' \;=\; [\mathit{disksWritten}\ \text{EXCEPT}\ ![p] = \{\}] \\
\quad \land\ \mathit{blocksRead}' \;=\; [\mathit{blocksRead}\ \text{EXCEPT}\ ![p] = [d \in \mathit{Disk} \mapsto \{\}]]
\end{array}
$$

We now define the actions that will form part of the next-state action. These actions describe all the atomic actions of the algorithm that a processor p can perform. The first is $\mathit{StartBallot}(p)$ in which p initiates a new ballot. We allow p to do this at any time during phase 1 or 2. The action sets $\mathit{phase}[p]$ to 1, increases $\mathit{dblock}[p].\mathit{mbal}$, and initializes the phase.

$$
\begin{array}{l}
\mathit{StartBallot}(p) \;\triangleq \\
\quad \land\ \mathit{phase}[p] \in \{1, 2\} \\
\quad \land\ \mathit{phase}' = [\mathit{phase}\ \text{EXCEPT}\ ![p] = 1] \\
\quad \land\ \exists\, b \in \mathit{Ballot}(p) : \begin{array}[t]{l} \land\ b > \mathit{dblock}[p].\mathit{mbal} \\ \land\ \mathit{dblock}' = [\mathit{dblock}\ \text{EXCEPT}\ ![p].\mathit{mbal} = b] \end{array} \\
\quad \land\ \mathit{InitializePhase}(p) \\
\quad \land\ \text{UNCHANGED}\ \langle \mathit{input}, \mathit{output}, \mathit{disk} \rangle
\end{array}
$$

In action $\mathit{Phase1or2Write}(p, d)$, processor p writes $\mathit{disk}[d][p]$ and adds d to the set $\mathit{disksWritten}[p]$ of disks written by p. The action is enabled iff p is in phase 1 or 2.[7] In the TLA$^+$ expression $[f\ \text{EXCEPT}\ ![c] = e]$, an @ appearing in e stands for $f[c]$. Thus, $x' = [x\ \text{EXCEPT}\ ![c] = @ + 1]$ corresponds to the programming-language statement $x[c] := x[c] + 1$. The EXCEPT construct also has a more general form for "arrays of arrays". For example, the formula $x' = [x\ \text{EXCEPT}\ ![a][b] = e]$ corresponds to the programming-language statement $x[a][b] := e$.

$$
\begin{array}{l}
\mathit{Phase1or2Write}(p, d) \;\triangleq \\
\quad \land\ \mathit{phase}[p] \in \{1, 2\} \\
\quad \land\ \mathit{disk}' = [\mathit{disk}\ \text{EXCEPT}\ ![d][p] = \mathit{dblock}[p]] \\
\quad \land\ \mathit{disksWritten}' = [\mathit{disksWritten}\ \text{EXCEPT}\ ![p] = @ \cup \{d\}] \\
\quad \land\ \text{UNCHANGED}\ \langle \mathit{input}, \mathit{output}, \mathit{phase}, \mathit{dblock}, \mathit{blocksRead} \rangle
\end{array}
$$

Action $\mathit{Phase1or2Read}(p, d, q)$ describes p reading $\mathit{disk}[d][q]$. It is enabled iff d is in $\mathit{disksWritten}[p]$, meaning that p has already written its block to disk d. (This implies that p is in phase 1 or 2.) We allow p to reread a disk

[7] We could add the enabling condition $d \notin \mathit{disksWritten}[p]$, but it's not necessary because the action is a no-op, leaving all variables unchanged, if p has already written its current value of $\mathit{dblock}[p]$ to disk d.

block it has already read. If $disk[d][q].mbal$ is less than p's current $mbal$ value, then $blocksRead[p][d]$ is updated and p continues executing its ballot. Otherwise, p aborts the ballot and begins a new one.

$$
\begin{array}{l}
Phase1or2Read(p, d, q) \;\triangleq \\
\quad \land\; d \in disksWritten[p] \\
\quad \land\; \textsc{if}\;\; disk[d][q].mbal < dblock[p].mbal \\
\qquad\quad \textsc{then}\;\; \land\; blocksRead' = \\
\qquad\qquad\qquad\quad [blocksRead\;\textsc{except} \\
\qquad\qquad\qquad\qquad !\,[p][d] = @ \cup \{[block \mapsto disk[d][q],\, proc \mapsto q]\}] \\
\qquad\qquad\quad\;\;\; \land\; \textsc{unchanged} \\
\qquad\qquad\qquad\qquad \langle input, output, disk, phase, dblock, disksWritten\rangle \\
\qquad\quad \textsc{else}\;\;\; StartBallot(p)
\end{array}
$$

The action $EndPhase1or2(p)$ describes processor p successfully finishing phase 1 or 2. It is enabled when p is in phase 1 or 2 and, on a majority of the disks, p has written its block and read every other processor's block. When p finishes phase 1, it sets $dblock[p].inp$ and $dblock[p].bal$ as described in Section 3.1 and starts phase 2. When p finishes phase 2, it sets $output[p]$, sets $phase[p]$ to 3, and terminates. (However, it could still fail and start again.) The TLA$^+$ EXCEPT construct applies to records as well as functions, and it can have multiple "replacements" separated by commas.

$$
\begin{array}{l}
EndPhase1or2(p) \;\triangleq \\
\quad \land\; IsMajority(\{d \in disksWritten[p] : \\
\qquad\qquad\qquad\qquad \forall\, q \in Proc \setminus \{p\} : hasRead(p, d, q)\}) \\
\quad \land\; \lor\; \land\; phase[p] = 1 \\
\qquad\quad\; \land\; dblock' = \\
\qquad\qquad\quad [\,dblock\;\textsc{except} \\
\qquad\qquad\qquad !\,[p].bal = dblock[p].mbal, \\
\qquad\qquad\qquad !\,[p].inp = \\
\qquad\qquad\qquad\quad \textsc{let}\;\; blocksSeen \;\triangleq\; allBlocksRead(p) \cup \{dblock[p]\} \\
\qquad\qquad\qquad\qquad\quad nonInitBlks \;\triangleq \\
\qquad\qquad\qquad\qquad\qquad \{bs \in blocksSeen : bs.inp \neq NotAnInput\} \\
\qquad\qquad\qquad\qquad\quad maxBlk \;\triangleq \\
\qquad\qquad\qquad\qquad\qquad \textsc{choose}\; b \in nonInitBlks : \\
\qquad\qquad\qquad\qquad\qquad\quad \forall\, c \in nonInitBlks : b.bal \geq c.bal \\
\qquad\qquad\qquad\quad \textsc{in}\quad \textsc{if}\;\; nonInitBlks = \{\}\;\; \textsc{then}\;\; input[p] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\;\; \textsc{else}\;\;\; maxBlk.inp\,] \\
\qquad\quad\; \land\; \textsc{unchanged}\;\; output
\end{array}
$$

$$
\begin{array}{l}
\quad\quad \lor \land\ phase[p] = 2 \\
\quad\quad\quad \land\ output' = [output \text{ EXCEPT } ![p] = dblock[p].inp] \\
\quad\quad\quad \land\ \text{UNCHANGED } dblock \\
\quad \land\ phase' = [phase \text{ EXCEPT } ![p] = @ + 1] \\
\quad \land\ InitializePhase(p) \\
\quad \land\ \text{UNCHANGED } \langle input, disk \rangle
\end{array}
$$

Action $Fail(p)$ represents a failure by processor p. The action is always enabled. It chooses a new value of $input[p]$, sets $phase[p]$ to 0 and initializes $dblock[p]$, $output[p]$, $disksWritten[p]$, and $blocksRead[p]$.

$$
\begin{array}{l}
Fail(p) \ \triangleq \\
\quad \land\ \exists\, ip \in Inputs : input' = [input \text{ EXCEPT } ![p] = ip] \\
\quad \land\ phase' \ = [phase \text{ EXCEPT } ![p] = 0] \\
\quad \land\ dblock' \ = [dblock \text{ EXCEPT } ![p] = InitDB] \\
\quad \land\ output' \ = [output \text{ EXCEPT } ![p] = NotAnInput] \\
\quad \land\ InitializePhase(p) \\
\quad \land\ \text{UNCHANGED } disk
\end{array}
$$

The next two actions describe failure recovery. In $Phase0Read(p, d)$, processor p reads $disk[d][p]$, recording the value read in $blocksRead[p]$. Again, we allow redundant reads of the same disk block. In $EndPhase0(p)$, processor p completes its recovery and enters phase 1, as described in Section 3.1.

$$
\begin{array}{l}
Phase0Read(p, d) \ \triangleq \\
\quad \land\ phase[p] = 0 \\
\quad \land\ blocksRead' = [blocksRead \text{ EXCEPT} \\
\quad\quad\quad\quad\quad\quad\quad ![p][d] = @ \cup \{[block \mapsto disk[d][p],\ proc \mapsto p]\}] \\
\quad \land\ \text{UNCHANGED } \langle input, output, disk, phase, dblock, disksWritten \rangle
\end{array}
$$

$$
\begin{array}{l}
EndPhase0(p) \ \triangleq \\
\quad \land\ phase[p] = 0 \\
\quad \land\ IsMajority(\{d \in Disk : hasRead(p, d, p)\}) \\
\quad \land\ \exists\, b \in Ballot(p) : \\
\quad\quad\quad \land\ \forall\, r \in allBlocksRead(p) : b > r.mbal \\
\quad\quad\quad \land\ dblock' = [dblock \text{ EXCEPT} \\
\quad\quad\quad\quad\quad\quad ![p] = [\,(\text{CHOOSE } r \in allBlocksRead(p) : \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \forall\, s \in allBlocksRead(p) : r.bal \geq s.bal) \\
\quad\quad\quad\quad\quad\quad\quad\quad \text{EXCEPT } !.mbal = b]\,] \\
\quad \land\ InitializePhase(p) \\
\quad \land\ phase' = [phase \text{ EXCEPT } ![p] = 1] \\
\quad \land\ \text{UNCHANGED } \langle input, output, disk \rangle
\end{array}
$$

As in most TLA specifications, we define the next-state action *Next* that describes all possible steps of all processors. We then define the formula *DiskSynodSpec*, our specification of the algorithm, to assert that the initial state satisfies *Init* and every step either satisfies *Next* or leaves all the variables unchanged.

$$
\begin{array}{lll}
Next & \triangleq & \exists\, p \in Proc : \\
& & \quad \lor\ StartBallot(p) \\
& & \quad \lor\ \exists\, d \in Disk : \lor\ Phase0Read(p, d) \\
& & \quad \ \lor\ Phase1or2Write(p, d) \\
& & \quad \ \lor\ \exists\, q \in Proc \setminus \{p\} : Phase1or2Read(p, d, q) \\
& & \quad \lor\ EndPhase1or2(p) \\
& & \quad \lor\ Fail(p) \\
& & \quad \lor\ EndPhase0(p)
\end{array}
$$

$$DiskSynodSpec \triangleq Init \land \Box[Next]_{vars}$$

The module ends by asserting the correctness of the algorithm, which means that the algorithm's specification implies the formula *SynodSpec* that is its correctness condition.

THEOREM $DiskSynodSpec \Rightarrow SynodSpec$

A.3 An Assertional Proof

To prove correctness of the Disk Synod algorithm, we must prove that *DiskSynodSpec* implies *SynodSpec*, which is the theorem asserted at the end of module *DiskSynod*. In general, a theorem and its proof must appear in a context that defines the meaning of the identifiers they mention. When proving a theorem that appears in a module, we assume the context (the definitions and declarations) provided by the module.

In our proof of the theorem that *DiskSynodSpec* implies *SynodSpec*, we will be informal in our use of identifier names. We will use identifiers like *ISpec* that are defined in submodule *Inner* of the *Synod* module and assume that they have their expected meaning. Readers who understand the fine points of TLA$^+$ will realize that those identifiers are not defined in the context of module *DiskSynod*, and they should be prefaced with $IS(chosen, allInput)!$, as in $IS(chosen, allInput)!ISpec$. However, we will ignore this formal detail. (We chose our identifier names so that dropping the $IS(chosen, allInput)!$ causes no name clashes.)

We now sketch the proof that *DiskSynodSpec* implies *SynodSpec*. Formula *SynodSpec* equals $\exists\, chosen, allInput : ISpec$. To prove such a formula, we must find Skolem functions with which to instantiate the bound variables *chosen* and *allInput*, and then prove that *DiskSynodSpec* implies *ISpec*, when *chosen* and *allInput* are defined to equal those Skolem functions. The choice of Skolem functions is called a *refinement mapping*. However, we cannot define such a refinement mapping because *chosen* and *allInput* record history that is not present in the actual state of the algorithm. Instead, we add *chosen* and *allInput* to the algorithm specification as *history variables*. Formally, we define a specification *HDiskSynodSpec* such that

$$DiskSynodSpec \;\equiv\; \exists\, chosen, allInput \;:\; HDiskSynodSpec$$

We then prove that *HDiskSynodSpec* implies *ISpec*, from which we infer by simple logic that *DiskSynodSpec* implies *SynodSpec*.

The first step in our proof that *DiskSynodSpec* implies *SynodSpec* is to define the required formula *HDiskSynodSpec* and to state formally and prove the theorem that it implies *ISpec*. To define *HDiskSynodSpec*, we must define its initial predicate and next-state action. The initial predicate *HInit* is the conjunction of the initial predicate *Init* of *DiskSynodSpec* with formulas that specify the initial values of *chosen* and *allInput*. Its next-state action *HNext* is the conjunction of the next-state action *Next* of *DiskSynodSpec* with formulas that specify the values of $chosen'$ and $allInput'$ as functions of the (unprimed and primed) values of the other variables. A general theorem of TLA asserts that, if no variable among the tuple $\mathbf{x}$ of variables occurs in I, N, or the tuple $\mathbf{y}$ of variables, then

$$I \land \Box[N]_{\mathbf{y}} \;\equiv\; \exists\,\mathbf{x} : (I \land (\mathbf{x} = f(\mathbf{y}))) \land \Box[N \land (\mathbf{x}' = g(\mathbf{x}, \mathbf{y}, \mathbf{y}'))]_{\langle \mathbf{x}, \mathbf{y}\rangle}$$

for any f and g. Substituting *Init* for I, *Next* for N, and the formulas implied by the definitions of *HInit* and *HNext* below for f and g, this result implies that the specification obtained from *HDiskSynodSpec* by hiding (existentially quantifying) *chosen* and *allInput* is equivalent to *DiskSynodSpec*. Hence, as explained above, proving that *HDiskSynodSpec* implies *ISpec* will show that *DiskSynodSpec* implies *SynodSpec*, proving the correctness of the Disk Synod algorithm.

We define *HDiskSynodSpec* in a module *HDiskSynod* that extends the *DiskSynod* module and declares *chosen* and *allInput* as variables.

MODULE *HDiskSynod*

EXTENDS *DiskSynod*

VARIABLES *allInput*, *chosen*

The initial values of *chosen* and *allInput* are the same as in the initial predicate of *Ispec*.

$$
\begin{array}{lll}
HInit & \triangleq & \land\ Init \\
 & & \land\ chosen\ \ = NotAnInput \\
 & & \land\ allInput\ = \{input[p] : p \in Proc\}
\end{array}
$$

The action *HNext* ensures that *chosen* equals the first *output* value that is different from *NotAnInput*, and that *allInput* always equals the set of all *input* values that have appeared thus far.

$$
\begin{array}{l}
HNext \ \triangleq \\
\quad \land\ Next \\
\quad \land\ chosen' = \text{LET } hasOutput(p) \ \triangleq\ output'[p] \neq NotAnInput \\
\qquad\qquad\qquad \text{IN } \ \text{IF } \lor\ chosen \neq NotAnInput \\
\qquad\qquad\qquad\qquad\quad\ \ \lor\ \forall\, p \in Proc : \neg hasOutput(p) \\
\qquad\qquad\qquad\qquad\ \ \text{THEN } chosen \\
\qquad\qquad\qquad\qquad\ \ \text{ELSE } output'[\text{CHOOSE } p \in Proc : hasOutput(p)] \\
\quad \land\ allInput' = allInput \cup \{input'[p] : p \in Proc\}
\end{array}
$$

The module then defines *HDiskSynodSpec* in the usual way, and asserts that it implies *ISpec*, with *chosen* and *allInput* replaced by the variables of the same name declared in the current module. (Again, the details of how this is expressed in TLA$^+$ are not important.)

$$HDiskSynodSpec \ \triangleq\ HInit \land \Box[HNext]_{\langle vars,\, chosen,\, allInput \rangle}$$

$$\text{THEOREM } HDiskSynodSpec \Rightarrow IS(chosen, allInput)!ISpec$$

To prove the correctness of the Disk Synod algorithm, it suffices to prove the theorem above, that *HDiskSynodSpec* implies *ISpec*. (Remember that we are dropping the $IS(chosen, allInput)!$ from identifiers defined in submodule *Inner*.) We now outline the proof of this theorem. Let *ivars* be the tuple of all variables of *ISpec*:

$$ivars \ \triangleq\ \langle input, output, chosen, allInput \rangle$$

To prove that *HDiskSynodSpec* implies *ISpec* we must prove

$$\text{THEOREM } R1 \quad HInit \Rightarrow IInit$$

$$\text{THEOREM } R2 \quad HInit \land \Box[HNext]_{\langle vars,\, chosen,\, allInput \rangle} \ \Rightarrow\ \Box[INext]_{ivars}$$

The proof of $R1$ is trivial. To prove $R2$, standard TLA reasoning shows that it suffices to find a state predicate *HInv* for which we can prove:

$$\text{THEOREM } R2a \quad HInit \land \Box[HNext]_{\langle vars,\, chosen,\, allInput \rangle} \Rightarrow \Box HInv$$

$$\text{THEOREM } R2b \quad HInv \land HInv' \land HNext \Rightarrow INext \lor (\text{UNCHANGED } ivars)$$

A predicate *HInv* satisfying $R2a$ is said to be an invariant of the specification $HInit \land \Box[HNext]_{\langle vars,\, chosen,\, allInput \rangle}$. To prove $R2a$, we make *HInv* strong enough to satisfy:

$$\text{THEOREM } I1 \quad HInit \Rightarrow HInv$$

$$\text{THEOREM } I2 \quad HInv \land HNext \Rightarrow HInv'$$

A predicate *HInv* satisfying $I2$ is said to be an invariant of the action *HNext*. A standard TLA theorem asserts that $I1$ and $I2$ imply $R2a$. Hence, $R2b$, $I1$, and $I2$ together imply $HDiskSynodSpec \Rightarrow ISpec$, which implies the correctness of the algorithm. So, we must now just define *HInv* and prove $R2b$, $I1$, and $I2$.

There are two general approaches to defining *HInv*. In both, we write *HInv* as a conjunction $HI_1 \land \ldots \land HI_k$. In the bottom-up method, we define the HI_i in increasing order of i, so that each conjunction $HI_1 \land \ldots \land HI_k$ is an invariant of *HNext*. We stop when we obtain an invariant strong enough to prove $R2b$. In the top-down method, we start by defining HI_k so that $R2b$ is satisfied with HI_k substituted for *HInv*. We then define the HI_i in decreasing order of i so that $HI_i \land \ldots \land HI_k \land HNext \Rightarrow HI'_{i+1}$, stopping when we obtain an invariant of *HNext*. In practice, one uses a combination of the two methods—with a lot of backtracking. Here, we present the invariant in a bottom-up fashion.

If the set of disks is empty, then $IsMajority(D)$ is false for all subsets D of *Disk*. (This follows from the assumption about *IsMajority* by substituting D for both S and T.) Hence, *HDiskSynodSpec* implies that the system remains forever in its initial state, trivially satisfying *ISpec*. It therefore suffices to consider only the case when *Disk* is nonempty:

$$\text{ASSUME } Disk \neq \{\}$$

The standard starting point for a TLA proof is a simple "type invariant", which we call $HInv1$, asserting that all variables have the correct type:

$$\begin{aligned}
&HInv1 \;\triangleq \\
&\quad \land\ input \in [Proc \rightarrow Inputs] \\
&\quad \land\ output \in [Proc \rightarrow Inputs \cup \{NotAnInput\}]
\end{aligned}$$

$$
\begin{array}{l}
\quad \land\ disk \in [Disk \rightarrow [Proc \rightarrow DiskBlock]] \\
\quad \land\ phase \in [Proc \rightarrow 0\ ..\ 3] \\
\quad \land\ dblock \in [Proc \rightarrow DiskBlock] \\
\quad \land\ output \in [Proc \rightarrow Inputs \cup \{NotAnInput\}] \\
\quad \land\ disksWritten \in [Proc \rightarrow \text{SUBSET}\ Disk] \\
\quad \land\ blocksRead \in [Proc \rightarrow [Disk \rightarrow \\
\qquad\qquad\qquad\qquad\qquad \text{SUBSET}\ [block : DiskBlock, proc : Proc]]] \\
\quad \land\ allInput \in \text{SUBSET}\ Inputs \\
\quad \land\ chosen \in Inputs \cup \{NotAnInput\}
\end{array}
$$

Our first lemma asserts that $HInv1$ is an invariant of $HNext$:

LEMMA $I2a$ $\quad HInv1 \land HNext \Rightarrow HInv1'$.

The proofs of Theorem $R2b$ and of most lemmas appear in Section A.4 below.

Before going any further, we define some useful state functions. First, we let $MajoritySet$ be the set of all subsets of the set of disks containing a majority of them; we let $blocksOf(p)$ be the set of all copies of p's disk blocks in the system—that is, $dblock[p]$, p's blocks on disk, and all blocks of p read by some processor; and we let $allBlocks$ be the set of all copies of all disk blocks of all processors.

$$
\begin{array}{l}
MajoritySet \ \triangleq\ \{D \in \text{SUBSET}\ Disk : IsMajority(D)\} \\
blocksOf(p) \ \triangleq \\
\quad \text{LET}\ rdBy(q, d) \ \triangleq\ \{br \in blocksRead[q][d] : br.proc = p\} \\
\quad \text{IN}\ \ \{dblock[p]\} \cup \{disk[d][p] : d \in Disk\} \\
\qquad\qquad \cup\ \{br.block : br \in \text{UNION}\ \{rdBy(q, d) : q \in Proc, d \in Disk\}\} \\
allBlocks \ \triangleq\ \text{UNION}\ \{blocksOf(p) : p \in Proc\}
\end{array}
$$

The next conjunct of $HInv$ describes some simple relations between the values of the different variables.

$$
\begin{array}{l}
HInv2 \ \triangleq \\
\quad \land\ \forall p \in Proc : \\
\qquad \forall bk \in blocksOf(p) : \land\ bk.mbal \in Ballot(p) \cup \{0\} \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ bk.bal \in Ballot(p) \cup \{0\} \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ (bk.bal = 0) \equiv (bk.inp = NotAnInput) \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ bk.mbal \geq bk.bal \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ bk.inp \in allInput \cup \{NotAnInput\}
\end{array}
$$

$$
\begin{array}{l}
\wedge\ \forall\, p \in \mathit{Proc},\, d \in \mathit{Disk}\ : \\
\quad \wedge\ (d \in \mathit{disksWritten}[p]) \Rightarrow \wedge\ \mathit{phase}[p] \in \{1,2\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathit{disk}[d][p] = \mathit{dblock}[p] \\
\quad \wedge\ (\mathit{phase}[p] \in 1,2) \Rightarrow \wedge\ (\mathit{blocksRead}[p][d] \neq \{\}) \Rightarrow \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (d \in \mathit{disksWritten}[p]) \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \neg \mathit{hasRead}(p, d, p) \\
\wedge\ \forall\, p \in \mathit{Proc}\ : \\
\quad \wedge\ (\mathit{phase}[p] = 0) \Rightarrow \wedge\ \mathit{dblock}[p] = \mathit{InitDB} \\
\qquad\qquad\qquad\qquad\qquad \wedge\ \mathit{disksWritten}[p] = \{\} \\
\qquad\qquad\qquad\qquad\qquad \wedge\ \forall\, d \in \mathit{Disk}\ : \forall\, \mathit{br} \in \mathit{blocksRead}[p][d]\ : \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathit{br.proc} = p \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathit{br.block} = \mathit{disk}[d][p] \\
\quad \wedge\ (\mathit{phase}[p] \neq 0) \Rightarrow \wedge\ \mathit{dblock}[p].\mathit{mbal} \in \mathit{Ballot}(p) \\
\qquad\qquad\qquad\qquad\qquad \wedge\ \mathit{dblock}[p].\mathit{bal} \in \mathit{Ballot}(p) \cup \{0\} \\
\qquad\qquad\qquad\qquad\qquad \wedge\ \forall\, d \in \mathit{Disk}\ : \forall\, \mathit{br} \in \mathit{blocksRead}[p][d]\ : \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{br.block.mbal} < \mathit{dblock}[p].\mathit{mbal} \\
\quad \wedge\ (\mathit{phase}[p] \in \{2,3\}) \Rightarrow (\mathit{dblock}[p].\mathit{bal} = \mathit{dblock}[p].\mathit{mbal}) \\
\quad \wedge\ \mathit{output}[p]\ =\ \textsc{if}\ \mathit{phase}[p] = 3\ \textsc{then}\ \mathit{dblock}[p].\mathit{inp}\ \textsc{else}\ \mathit{NotAnInput} \\
\wedge\ \mathit{chosen} \in \mathit{allInput} \cup \{\mathit{NotAnInput}\} \\
\wedge\ \forall\, p \in \mathit{Proc}\ : \wedge\ \mathit{input}[p] \in \mathit{allInput} \\
\qquad\qquad\qquad\quad \wedge\ (\mathit{chosen} = \mathit{NotAnInput}) \Rightarrow (\mathit{output}[p] = \mathit{NotAnInput})
\end{array}
$$

The invariance of $\mathit{HInv}1 \wedge \mathit{HInv}2$ follows from Lemma $I2a$ and:

LEMMA $I2b$ $\quad \mathit{HInv}1 \wedge \mathit{HInv}2 \wedge \mathit{HNext} \Rightarrow \mathit{HInv}2'$

The next conjunct of *HInv* expresses the observation that if processors p and q have each read the other's block from disk d during their current phases, then at least one of them has read the other's current block.

$$
\begin{array}{l}
\mathit{HInv}3\ \triangleq\ \forall\, p, q \in \mathit{Proc},\ d \in \mathit{Disk}\ : \\
\qquad\qquad \wedge\ \mathit{phase}[p] \in \{1,2\} \\
\qquad\qquad \wedge\ \mathit{phase}[q] \in \{1,2\} \\
\qquad\qquad \wedge\ \mathit{hasRead}(p, d, q) \\
\qquad\qquad \wedge\ \mathit{hasRead}(q, d, p) \\
\qquad\qquad \Rightarrow \vee\ [\mathit{block} \mapsto \mathit{dblock}[q],\ \mathit{proc} \mapsto q] \in \mathit{blocksRead}[p][d] \\
\qquad\qquad\quad\ \ \vee\ [\mathit{block} \mapsto \mathit{dblock}[p],\ \mathit{proc} \mapsto p] \in \mathit{blocksRead}[q][d]
\end{array}
$$

LEMMA $I2c$ $\quad \mathit{HInv}1 \wedge \mathit{HInv}2 \wedge \mathit{HInv}3 \wedge \mathit{HNext} \Rightarrow \mathit{HInv}3'$

The next conjunct of the invariant expresses relations among the *mbal* and *bal* values of a processor and of its disk blocks. Its first conjunct asserts that, when p is not recovering from a failure, its *mbal* value is at least as large as the *bal* field of any of its blocks, and at least as large as the *mbal* field of its block on some disk in any majority set. Its second conjunct asserts that, in phase 1, its *mbal* value is actually greater than the *bal* field of any of its blocks. Its third conjunct asserts that, in phase 2, its *bal* value is the *mbal* field of all its blocks on some majority set of disks. The fourth conjunct asserts that the *bal* field of any of its blocks is at most as large as the *mbal* field of all its disk blocks on some majority set of disks.

$$
\begin{array}{l}
\mathit{HInv4} \;\triangleq \\
\quad \forall\, p \in \mathit{Proc} : \\
\qquad \land\ (\mathit{phase}[p] \neq 0) \;\Rightarrow \\
\qquad\qquad \land\ \forall\, \mathit{bk} \in \mathit{blocksOf}(p) : \mathit{dblock}[p].\mathit{mbal} \geq \mathit{bk}.\mathit{bal} \\
\qquad\qquad \land\ \forall\, D \in \mathit{MajoritySet} : \\
\qquad\qquad\qquad \exists\, d \in D : \land\ \mathit{dblock}[p].\mathit{mbal} \;\geq\; \mathit{disk}[d][p].\mathit{mbal} \\
\qquad\qquad\qquad\qquad\quad\;\; \land\ \mathit{dblock}[p].\mathit{bal} \;\;\;\geq\; \mathit{disk}[d][p].\mathit{bal} \\
\qquad \land\ (\mathit{phase}[p] = 1) \Rightarrow (\forall\, \mathit{bk} \in \mathit{blocksOf}(p) : \mathit{dblock}[p].\mathit{mbal} > \mathit{bk}.\mathit{bal}) \\
\qquad \land\ (\mathit{phase}[p] \in \{2, 3\}) \;\Rightarrow \\
\qquad\qquad (\exists\, D \in \mathit{MajoritySet} : \forall\, d \in D : \mathit{disk}[d][p].\mathit{mbal} = \mathit{dblock}[p].\mathit{bal}) \\
\qquad \land\ \forall\, \mathit{bk} \in \mathit{blocksOf}(p) : \\
\qquad\qquad \exists\, D \in \mathit{MajoritySet} : \forall\, d \in D : \mathit{disk}[d][p].\mathit{mbal} \geq \mathit{bk}.\mathit{bal}
\end{array}
$$

LEMMA *I2d* $\mathit{HInv1} \land \mathit{HInv2} \land \mathit{HInv2}' \land \mathit{HInv4} \land \mathit{HNext} \Rightarrow \mathit{HInv4}'$

Before going further, we define $\mathit{maxBalInp}(b, v)$ to assert that every block in *allBlocks* with *bal* field at least b has *inp* field v.

$$\mathit{maxBalInp}(b, v) \;\triangleq\; \forall\, \mathit{bk} \in \mathit{allBlocks} : (\mathit{bk}.\mathit{bal} \geq b) \Rightarrow (\mathit{bk}.\mathit{inp} = v)$$

We now come to a conjunct of *HInv* that provides some high-level insight into why the algorithm is correct. It asserts that, if a processor p is in phase 2, then either its *bal* and *inp* values satisfy *maxBalInp*, or else p must eventually abort its current ballot. Processor p will eventually abort its ballot if there is some processor q and majority set D such that p has not read q's block on any disk in D, and all of those blocks have *mbal* values greater than $\mathit{dblock}[p].\mathit{bal}$. (Since p must read at least one of those disks, it must eventually read one of those blocks and abort.)

$$
\begin{array}{l}
HInv5 \;\triangleq\\
\quad \forall\, p \in Proc \;:\\
\qquad (phase[p] = 2) \Rightarrow \lor\; maxBalInp(dblock[p].bal,\; dblock[p].inp)\\
\qquad\qquad\qquad\qquad\quad \lor\; \exists\, D \in MajoritySet,\; q \in Proc \;:\\
\qquad\qquad\qquad\qquad\qquad\quad \forall\, d \in D \;:\; \land\; disk[d][q].mbal > dblock[p].bal\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\; \land\; \neg hasRead(p,\; d,\; q)
\end{array}
$$

LEMMA $I2e$

$HInv1 \land HInv2 \land HInv2' \land HInv3 \land HInv4 \land HInv5 \land HNext \Rightarrow HInv5'$

Before defining our final conjunct, we define a predicate $valueChosen(v)$ that is true if v is the only possible value that can be chosen as an output. It asserts that there is some ballot number b such that $maxBalInp(b,\, v)$ is true. This condition is satisfied if there is no block bk in $allBlocks$ with $bk.bal \geq b$. So, $valueChosen(v)$ must require that some processor p has written blocks with bal field at least b to a majority set D of the disks. (By $maxBalInp(b, v)$, those blocks must have inp field v). We also ensure that, once $valueChosen(v)$ becomes true, it can never be made false. This requires the additional condition that no processor q that is currently executing phase 1 with $mbal$ value at least b can fail to see those blocks that p has written. So, $valueChosen(v)$ also asserts that, for every disk d in D, if q has already read $disk[d][p]$, then it has read a block with bal field at least b.

$$
\begin{array}{l}
valueChosen(v) \;\triangleq\\
\quad \exists\, b \in \textsc{union}\ \{Ballot(p) \;:\; p \in Proc\} \;:\\
\qquad \land\; maxBalInp(b, v)\\
\qquad \land\; \exists\, p \in Proc,\; D \in MajoritySet \;:\\
\qquad\qquad \forall\, d \in D \;:\; \land\; disk[d][p].bal \geq b\\
\qquad\qquad\qquad\qquad\; \land\; \forall\, q \in Proc \;:\\
\qquad\qquad\qquad\qquad\qquad \land\; phase[q] = 1\\
\qquad\qquad\qquad\qquad\qquad \land\; dblock[q].mbal \geq b\\
\qquad\qquad\qquad\qquad\qquad \land\; hasRead(q, d, p)\\
\qquad\qquad\qquad\qquad\quad \Rightarrow (\exists\, br \in blocksRead[q][d] \;:\; br.bal \geq b)
\end{array}
$$

It's obvious that, if $valueChosen(v) = valueChosen(w)$, then $v = w$.

The final conjunct of $HInv$ asserts that, once an output has been chosen, $valueChosen(chosen)$ holds, and each processor's output equals either $chosen$ or $NotAnInput$.

$$
\begin{array}{l}
HInv6 \;\triangleq\; \land\; (chosen \neq NotAnInput) \Rightarrow valueChosen(chosen)\\
\qquad\qquad\;\; \land\; \forall\, p \in Proc \;:\; output[p] \in \{chosen, NotAnInput\}
\end{array}
$$

LEMMA $I2f$ $\;HInv1 \land HInv2 \land HInv2' \land HInv3 \land HInv6 \land HNext \Rightarrow HInv6'$

We define $HInv$ to be the conjunction of $HInv1$–$HInv6$.

$$HInv \triangleq HInv1 \land HInv2 \land HInv3 \land HInv4 \land HInv5 \land HInv6$$

Theorem $I2$ then follows easily from Lemmas $I2a$–$I2f$.

A.4 Proofs

We now sketch the proofs of most of the lemmas from Section A.3 and of Theorem $R2b$. We give hierarchically structured proofs [11]. A structured proof consists of a sequence of statements and their proofs; each of those proofs is either a structured proof or an ordinary paragraph-style proof. The j^{th} step in the current level-i proof is numbered $\langle i \rangle j$. Within a paragraph-style proof, $\langle i \rangle j$ denotes the most recent statement with that number. The proof statement "$\langle i \rangle j$. Q.E.D." denotes the current goal—that is, the level $i-1$ statement being proved by this step. A proof statement

ASSUME: A
PROVE: P

asserts that the assumption A implies P. If P is the current goal, then this is abbreviated as

CASE: A

An assumption CONSTANT $c \in S$ asserts that c is a new constant parameter that we assume is in S. We prove $\forall\, c \in S : P(c)$ by proving

ASSUME: CONSTANT $c \in S$
PROVE: $P(c)$

The assumption CONSTANT $c \in S$ S.T. $A(c)$ also assumes that c satisfies $A(c)$. A proof statement

$\langle i \rangle j$ CHOOSE $c \in S$ S.T. $P(c)$

asserts the existence of a value c in S satisfying $P(c)$, and defines c to be such a value. To prove this statement, we must demonstrate the existence of c.

We recommend that proofs be read hierarchically, from the top level down. To read the proof of a long level-i step, you should first read the level-$(i+1)$ statements that form its proof, together with the proof of the final "Q.E.D." step (which is usually a short paragraph), and then read the proofs of the level-$(i+1)$ steps in any desired order.

We also use a hierarchical scheme for naming subformulas of a formula. If F is the name of a formula that is a conjunction, then $F.i$ is the name of its i^{th} conjunct. A similar scheme is used for a disjunction, except using letters instead of numbers, so $F.c$ is the name of the third disjunct of F. If F is the name of the formula $P \Rightarrow Q$, then $F.L$ is the name of P and $F.R$ is the name of Q. If F is the name of the formula $\exists\, x : P(x)$ or $\forall\, x : P(x)$, then $F(e)$ is the name of the formula $P(e)$, for any expression e. This is generalized in the obvious way for abbreviated quantifications like $\exists\, x, y : P(x, y)$. For example, $\mathit{HInv}5(n).R.b(E, m)(dd).2$ is the formula $\neg \mathit{hasRead}(n,\ dd,\ m)$.

We now give the proofs. We omit the proofs of Lemmas $I2a$ and $I2b$, which require a simple but tedious case analysis for the different disjuncts of *Next*. In the informal paragraph-style proofs, we use $\mathit{HInv}1$ implicitly in many places by tacitly assuming that variables have values of the right type. For example, we deduce $\mathit{phase}'[p] = 2$ from

$$\mathit{phase}' = [\mathit{phase}\ \text{EXCEPT}\ ![p] = 2]$$

without mentioning that this follows only if *phase* is a function whose domain contains p, which is implied by $\mathit{HInv}1.4$.

A.4.1 Lemma I2c

We prove Lemma $I2c$ by proving:

ASSUME: 1. $\mathit{HInv}1 \wedge \mathit{HInv}2 \wedge \mathit{HInv}3 \wedge \mathit{HNext}$
2. CONSTANTS $p, q \in \mathit{Proc},\ d \in \mathit{Disk}$
3. $\mathit{HInv}3(p, q, d).L'$

PROVE: $\mathit{HInv}3(p, q, d).R'$

$\langle 1 \rangle 1$. CASE: $\neg \mathit{HInv}3(p, q, d).L$

$\langle 2 \rangle 1$. CASE: $(p \neq q) \wedge \mathit{Phase}1or2\mathit{Read}(p, d, q)$

$\langle 3 \rangle 1$. $(\mathit{phase}[q] \in \{1, 2\}) \wedge \mathit{hasRead}(q, d, p)$

PROOF: Assumption 3 implies

$$(\mathit{phase}'[q] \in \{1, 2\}) \wedge \mathit{hasRead}(q, d, p)'$$

and the level $\langle 2 \rangle$ case assumption implies that $\mathit{hasRead}(q, d, p)$ and $\mathit{phase}[q]$ are left unchanged.

$\langle 3 \rangle 2$. $\mathit{disk}[d][q] = \mathit{dblock}[q]$

PROOF: $\langle 3 \rangle 1$ and $\mathit{HInv}2.2(q, d).2$ imply $d \in \mathit{disksWritten}[q]$, which by $\mathit{HInv}2.2(q, d).1$ implies $\mathit{disk}[d][q] = \mathit{dblock}[q]$.

$\langle 3 \rangle 3$. Q.E.D.

PROOF: $\mathit{Phase}1or2\mathit{Read}(p, d, q)$ (the level $\langle 2 \rangle$ case assumption) implies:

$$[\mathit{block} \mapsto \mathit{disk}[d][q],\ \mathit{proc} \mapsto q] \in \mathit{blocksRead}'[p][d]$$

and we then obtain $HInv3(p,q).R.a'$ from $\langle 3\rangle 2$, since $(p \neq q) \wedge Phase1or2Read(p,d,q)$ implies $dblock'[q] = dblock[q]$.

$\langle 2\rangle 2$. CASE: $(p \neq q) \wedge Phase1or2Read(q,d,p)$

PROOF: The proof is the same as that of $\langle 2\rangle 1$ with p and q interchanged and $HInv3(p,q).R.a'$ replaced by $HInv3(p,q).R.b'$.

$\langle 2\rangle 3$. CASE: $EndPhase0(p)$

PROOF: This implies $\neg hasRead(p,d,q)'$, so $HInv3(p,q,d).L'$ is false, making $HInv3(p,q,d)'$ true.

$\langle 2\rangle 4$. CASE: $EndPhase0(q)$

PROOF: The proof is the same as that of $\langle 2\rangle 3$ with p and q interchanged.

$\langle 2\rangle 5$. Q.E.D.

PROOF: By assumption 3 and the level $\langle 1\rangle$ case assumption, one of the four conjuncts of $HInv3(p,q,d).L$ is changed from false to true. Steps $\langle 2\rangle 1$–$\langle 2\rangle 4$ cover the four subactions of $Next$ that can make one of those conjuncts true.

$\langle 1\rangle 2$. CASE: $HInv3(p,q,d).L$

PROOF: $HInv3(p,q,d).L$ and $HInv3$ (which holds by assumption 1) imply $HInv3(p,q,d).R$. The only subactions of $HNext$ that can change $HInv3(p,q,d).R$ from true to false are ones that remove elements from $blocksRead[p][d]$ or $blocksRead[q][d]$ or that change $dblock[p]$ or $dblock[q]$. All such subactions have an $InitializePhase(p)$ or $InitializePhase(q)$ conjunct that makes $HInv3(p,q,d).R'$ false, contrary to assumption 3.

$\langle 1\rangle 3$. Q.E.D.

PROOF: By $\langle 1\rangle 1$ and $\langle 1\rangle 2$.

A.4.2 Lemma BksOf

The following simple result will be used below.

LEMMA $BksOf$

$$HNext \wedge HInv1 \;\Rightarrow\; \forall\, p \in Proc : blocksOf(p)' \subseteq blocksOf(p) \cup \{dblock'[p]\}$$

The lemma follows from the observation that the only way an $HNext$ step creates a new block for a processor p (rather than copying an existing one, which leaves $blocksOf(p)$ unchanged) is by changing $dblock[p]$.

A.4.3 Lemma I2d

ASSUME: 1. $HInv1 \wedge HInv2 \wedge HInv2' \wedge HInv4 \wedge HNext$

2. CONSTANT $p \in Proc$

PROVE: $HInv4(p)'$

$\langle 1\rangle 1.$ $HInv4(p).1'$

$\langle 2\rangle 1.$ CASE: $(phase[p] = 0) \wedge (phase'[p] \neq 0)$

$\langle 3\rangle 1.$ $EndPhase0(p)$

PROOF: By the level $\langle 2\rangle$ case assumption, since $EndPhase0(p)$ is the only subaction of $HNext$ that changes $phase[p]$ from zero to a nonzero value.

$\langle 3\rangle 2.$ ASSUME: CONSTANT $bk \in blocksOf(p)'$ S.T. $bk \neq dblock'[p]$
PROVE: $dblock'[p].mbal \geq bk.bal$

$\langle 4\rangle 1.$ $bk \in blocksOf(p)$

PROOF: Lemma $BksOf$ and the level $\langle 3\rangle$ assumption.

$\langle 4\rangle 2.$ CHOOSE $D1 \in MajoritySet$ S.T.
$\forall\, d \in D1 : disk[d][p].mbal \geq bk.bal$

PROOF: $HInv4.4$ and $\langle 4\rangle 1$ imply the existence of $D1$.

$\langle 4\rangle 3.$ $\forall\, D \in MajoritySet : \exists\, d \in D : disk[d][p].mbal \geq bk.bal$

PROOF: By $\langle 4\rangle 2$, since for any majority set D, we can choose d to be a disk in $D1 \cap D$, which is nonempty because any two majority sets have an element in common.

$\langle 4\rangle 4.$ $\exists\, d \in Disk : \exists\, rb \in blocksRead[p][d] : rb.block.mbal \geq bk.bal$

$\langle 5\rangle 1.$ $\forall\, d \in Disk : \forall\, rb \in blocksRead[p][d] : rb.block = disk[d][p]$

PROOF: By $HInv2.3(p).1.R.3$, which holds by assumption 1 and case assumption $\langle 2\rangle$.

$\langle 5\rangle 2.$ $\forall\, d \in Disk : hasRead(p, d, p) \Rightarrow \exists\, rb \in blocksRead[p][d] :$
$rb.block = disk[d][p]$

PROOF: By $\langle 5\rangle 1$ and the definition of $hasRead(p, d, p)$, which implies that $blocksRead[p][d]$ is nonempty.

$\langle 5\rangle 3.$ $\exists\, D \in MajoritySet :$
$\forall\, d \in D : \exists\, rb \in blocksRead[p][d] : rb.block = disk[d][p]$

PROOF: By $\langle 5\rangle 2$ and step $\langle 3\rangle 1$, from which we deduce that $hasRead(p, d, p)$ holds for all d in some majority set.

$\langle 5\rangle 4.$ Q.E.D.

PROOF: Steps $\langle 4\rangle 3$ and $\langle 5\rangle 3$ imply that there is a disk d and an rb in $blocksRead[p][d]$ such that $rb.block.mbal = disk[d][p].mbal \geq bk.bal$.

$\langle 4\rangle 5.$ Q.E.D.

PROOF: $\langle 4\rangle 4$ and $\langle 3\rangle 1$ imply $dblock'[p].mbal > bk.bal$.

$\langle 3\rangle 3.$ $HInv4(p).1.R.2'$

$\langle 4 \rangle 1.$ $\exists\, D \in \mathit{MajoritySet} :$

$$\forall\, d \in D : \land\ \mathit{dblock}'[p].\mathit{mbal} > \mathit{disk}[d][p].\mathit{mbal}$$
$$\land\ \mathit{dblock}'[p].\mathit{bal} \geq \mathit{disk}[d][p].\mathit{bal}$$

PROOF: $\langle 3 \rangle 1$ implies $\mathit{dblock}'[p].\mathit{mbal} > \mathit{br}.\mathit{mbal}$ and $\mathit{dblock}'[p].\mathit{bal} \geq \mathit{br}.\mathit{bal}$, for all $\mathit{br} \in \mathit{allBlocksRead}(p)$. Step $\langle 3 \rangle 1$, the level $\langle 2 \rangle$ case assumption, and $\mathit{HInv}2.3(p).3$ imply that $\mathit{allBlocksRead}(p)$ contains all blocks $\mathit{disk}[d][p]$ for d in some majority set D of disks.

$\langle 4 \rangle 2.$ $\forall\, D \in \mathit{MajoritySet} :$

$$\exists\, d \in D : \land\ \mathit{dblock}'[p].\mathit{mbal} > \mathit{disk}[d][p].\mathit{mbal}$$
$$\land\ \mathit{dblock}'[p].\mathit{bal} \geq \mathit{disk}[d][p].\mathit{bal}$$

PROOF: By $\langle 4 \rangle 1$, since any two majority sets have a disk in common.

$\langle 4 \rangle 3.$ Q.E.D.

PROOF: $\mathit{HInv}4(p).1.R.2'$ follows from $\langle 4 \rangle 2$ and $\langle 3 \rangle 1$, which implies that disk is unchanged.

$\langle 3 \rangle 4.$ Q.E.D.

PROOF: By $\langle 3 \rangle 2$ and $\langle 3 \rangle 3$, since $\langle 3 \rangle 2$ implies $\mathit{HInv}4(p).1.R.1(\mathit{bk})'$ except for the case $\mathit{bk} = \mathit{dblock}'[p]$; and $\mathit{HInv}4(p).1.R.1(\mathit{bk})'$ follows from $\mathit{HInv}2.1(p)(\mathit{dblock}[p]).4'$ in that case.

$\langle 2 \rangle 2.$ CASE: $(\mathit{phase}[p] \neq 0) \land (\mathit{phase}'[p] \neq 0)$

$\langle 3 \rangle 1.$ $\land\ \mathit{dblock}'[p].\mathit{mbal} \geq \mathit{dblock}[p].\mathit{mbal}$
$\land\ \mathit{dblock}'[p].\mathit{bal} \geq \mathit{dblock}[p].\mathit{bal}$

PROOF: Only the following four subactions of Next change $\mathit{dblock}[p]$:

$\mathit{StartBallot}(p)$ $\mathit{EndPhase}1\mathit{or}2(p)$ $\mathit{EndPhase}0(p)$ $\mathit{Fail}(p)$

These four cases are checked as follows.

- A $\mathit{StartBallot}(p)$ step increases $\mathit{dblock}[p].\mathit{mbal}$, and it does not change $\mathit{dblock}[p].\mathit{bal}$.
- An $\mathit{EndPhase}1\mathit{or}2(p)$ step leaves $\mathit{dblock}[p].\mathit{mbal}$ unchanged and changes $\mathit{dblock}[p].\mathit{bal}$ only by setting it to $\mathit{dblock}[p].\mathit{mbal}$ when $\mathit{phase}[p] = 1$, in which case $\mathit{HInv}2.1(p)(\mathit{dblock}[p]).4$ implies that its value is not decreased.
- $\mathit{EndPhase}0(p)$ and $\mathit{Fail}(p)$ are ruled out by the level $\langle 2 \rangle$ case assumption.

$\langle 3 \rangle 2.$ $\mathit{HInv}4(p).1.R.1'$

PROOF: If $\mathit{bk} \in \mathit{blocksOf}(p)$, then $\mathit{HInv}4(p).1.R.1(\mathit{bk})'$ follows from $\langle 3 \rangle 1$ and $\mathit{HInv}4(p).1.R.1$ (which holds by assumption 1 and the level $\langle 2 \rangle$ case assumption). If $\mathit{bk} = \mathit{dblock}'[p]$, then $\mathit{HInv}4(p).1.R.1(\mathit{bk})'$ follows from $\mathit{HInv}2.1(p)(\mathit{bk}).4'$. We then obtain $\mathit{HInv}4(p).1.R.1'$ from Lemma BksOf.

$\langle 3 \rangle 3.$ $\mathit{HInv}4(p).1.R.2'$

PROOF: *HNext* implies that $disk'[p][d]$ equals $disk[p][d]$ or $dblock[p]$, so $HInv4(p).1.R.2'$ follows from $\langle 3 \rangle 1$ and $HInv4(p).1.R.2$, which holds by assumption 1 and the level $\langle 2 \rangle$ case assumption.

$\langle 3 \rangle 4$. Q.E.D.

PROOF: By $\langle 3 \rangle 2$ and $\langle 3 \rangle 3$.

$\langle 2 \rangle 3$. Q.E.D.

PROOF: By $\langle 2 \rangle 1$ and $\langle 2 \rangle 2$, since $HInv4(p).1'$ is trivially true if $phase'[p]$ equals 0.

$\langle 1 \rangle 2$. $HInv4(p).2'$

$\langle 2 \rangle 1$. CASE: $(phase[p] \neq 1) \wedge (phase'[p] = 1)$

$\langle 3 \rangle 1$. CASE: $phase[p] = 0$

$\langle 4 \rangle 1$. $EndPhase0(p)$

PROOF: By *HNext* and the levels $\langle 2 \rangle$ and $\langle 3 \rangle$ case assumptions.

$\langle 4 \rangle 2$. $\forall\, bk \in blocksOf(p) :$
$$\exists\, D \in MajoritySet : \forall\, d \in D : disk[d][p].mbal \geq bk.bal$$

PROOF: By $HInv4(p).4$.

$\langle 4 \rangle 3$. $\forall\, bk \in blocksOf(p) :$
$$\forall\, D \in MajoritySet : \exists\, d \in D : disk[d][p].mbal \geq bk.bal$$

PROOF: By $\langle 4 \rangle 2$, since any two majority sets have a disk in common.

$\langle 4 \rangle 4$. $\forall\, bk \in blocksOf(p) :$
$$\exists\, br \in allBlocksRead(p) : br.mbal \geq bk.bal$$

PROOF: Step $\langle 4 \rangle 1$ implies that $blocksRead[p][d]$ is nonempty for all disks d in some majority set D, and $HInv2.3(p).1.R.3$ (which holds by assumption 1 and the level $\langle 3 \rangle$ case assumption) implies $rb.block = disk[d][p]$ for every $d \in D$ and $rb \in blocksRead[p][d]$. The result then follows from $\langle 4 \rangle 3$, since $rb \in blocksRead[p][d]$ implies $rb.block \in allBlocksRead(p)$.

$\langle 4 \rangle 5$. Q.E.D.

$\langle 5 \rangle 1$. $\forall\, br \in allBlocksRead(p) : dblock'[p].mbal > br.mbal$

PROOF: By $\langle 4 \rangle 1$.

$\langle 5 \rangle 2$. $\forall\, bk \in blocksOf(p) : HInv4(p).2.R(bk)'$

PROOF: By $\langle 5 \rangle 1$ and $\langle 4 \rangle 4$.

$\langle 5 \rangle 3$. $\exists\, br \in allBlocksRead(p) : dblock'[p].bal = br.bal$

PROOF: By $\langle 4 \rangle 1$.

$\langle 5 \rangle 4$. $HInv4(p).2.R(dblock[p])'$

PROOF: By $\langle 5 \rangle 1$, $\langle 5 \rangle 3$, and $HInv2.1(p)$.

$\langle 5 \rangle 5$. Q.E.D.

PROOF: $\langle 5 \rangle 2$, $\langle 5 \rangle 4$, and Lemma *BksOf* imply $HInv4(p).2.R'$.

$\langle 3 \rangle 2$. CASE: $phase[p] \in \{2, 3\}$

$\langle 4 \rangle 1.$ $\forall\, bk \in blocksOf(p) \,:\, dblock[p].mbal \geq bk.bal$

PROOF: $HInv4(p).1$ and the level $\langle 3 \rangle$ case assumption (which imply $HInv4(p).1.R.1$).

$\langle 4 \rangle 2.$ $\land\; dblock'[p].mbal > dblock[p].mbal$
$\quad\;\land\; dblock'[p].bal = dblock[p].bal$

PROOF: By $HNext$ and the level $\langle 2 \rangle$ and $\langle 3 \rangle$ case assumptions, which imply $StartBallot(p)$.

$\langle 4 \rangle 3.$ Q.E.D.

PROOF: By Lemma $BksOf$, it suffices to prove $HInv4(p).2.R(bk)'$ for $bk \in blocksOf(p)$ and $bk = dblock'[p]$. For $bk \in blocksOf(p)$, it follows from $\langle 4 \rangle 1$ and $\langle 4 \rangle 2$. For $bk = dblock'[p]$, it follows from $\langle 4 \rangle 2$ and $HInv2.1(p)(dblock[p]).4$.

$\langle 3 \rangle 3.$ Q.E.D.

PROOF: The level $\langle 2 \rangle$ case assumption implies that $\langle 3 \rangle 1$ and $\langle 3 \rangle 2$ cover all possibilities.

$\langle 2 \rangle 2.$ CASE: $(phase[p] = 1) \land (phase'[p] = 1)$

PROOF: By $HNext$, this implies $dblock'[p] = dblock[p]$, so Lemma $BksOf$ implies that $HInv4(p).2'$ follows from $HInv4(p).2$.

$\langle 2 \rangle 3.$ Q.E.D.

PROOF: Since $HInv4(p).2'$ is trivially true if $phase'[p] \neq 1$, the cases of $\langle 2 \rangle 1$ and $\langle 2 \rangle 2$ are exhaustive.

$\langle 1 \rangle 3.$ $HInv4(p).3'$

$\langle 2 \rangle 1.$ CASE: $(phase[p] \neq 2) \land (phase'[p] = 2)$

$\langle 3 \rangle 1.$ $EndPhase1or2(p) \land (phase[p] = 1)$

PROOF: By $HNext$ and the level $\langle 2 \rangle$ case assumption.

$\langle 3 \rangle 2.$ $\exists\, D \in MajoritySet \,:\, \forall\, d \in D \,:\, disk[d][p].mbal = dblock[p].mbal$

PROOF: By $HInv2.2(p).1$, since $\langle 3 \rangle 1$ implies that $disksWritten[p]$ contains a majority set of disks.

$\langle 3 \rangle 3.$ Q.E.D.

PROOF: $\langle 3 \rangle 1$ implies $dblock'[p].bal = dblock[p].mbal$ and $disk' = disk$, which by $\langle 3 \rangle 2$ implies $HInv4(p).3'$

$\langle 2 \rangle 2.$ CASE: $(phase[p] \in \{2, 3\}) \land (phase'[p] \in \{2, 3\})$

$\langle 3 \rangle 1.$ $dblock'[p].bal = dblock[p].bal$

PROOF: By $HNext$ and the level $\langle 2 \rangle$ case assumption.

$\langle 3 \rangle 2.$ $\forall\, d \in Disk \,:$
$$Phase1or2Write(p, d) \Rightarrow (disk'[d][p].mbal = dblock[p].bal)$$

PROOF: By the level $\langle 2 \rangle$ case assumption and $HInv2.3(p).3$.

$\langle 3 \rangle 3.$ Q.E.D.

PROOF: $HInv4(p).3'$ follows from $HInv4(p).3$, $\langle 3 \rangle 1$, and $\langle 3 \rangle 2$, since $HNext \land \neg Phase1or2Write(p, d)$ implies $disk'[d][p] = disk[d][p]$, for any disk d.

$\langle 2\rangle 3.$ Q.E.D.

PROOF: $HInv4(p).3'$ follows from $\langle 2\rangle 1$ and $\langle 2\rangle 2$ because it is trivially true if $phase'[p] \notin \{2, 3\}$, and $HNext \wedge (phase'[p] = 3)$ implies $phase[p] \in \{2, 3\}$,

$\langle 1\rangle 4.$ $HInv4(p).4'$

$\langle 2\rangle 1.$ CASE: $EndPhase1or2(p) \wedge (phase[p] = 1)$

$\langle 3\rangle 1.$ $\exists\, D \in MajoritySet : \forall\, d \in D : disk'[d][p].mbal = dblock'[p].bal$

PROOF: By $\langle 1\rangle 3$ and the level $\langle 2\rangle$ case assumption, which implies $phase'[p] = 2$.

$\langle 3\rangle 2.$ $disk' = disk$

PROOF: By the level $\langle 2\rangle$ case assumption.

$\langle 3\rangle 3.$ Q.E.D.

PROOF: If $bk \in blocksOf(p)$, then $HInv4(p).4(bk)'$ follows from $\langle 3\rangle 2$ and $HInv4(p).4(bk)$. If $bk = dblock'[p]$, then $HInv4(p).4(bk)'$ follows from $\langle 3\rangle 1$. By Lemma $BksOf$, this proves $HInv4(p).4'$.

$\langle 2\rangle 2.$ CASE: $Fail(p)$

PROOF: If $bk \in blocksOf(p)$, then $HInv4(p).4(bk)'$ follows easily from $HInv4(p).4(bk)$, since $Fail(p)$ implies $disk' = disk$. If $bk = dblock'[p]$, then $HInv4(p).4(bk)'$ holds because $Fail(p)$ implies $dblock'[p].bal = 0$. By Lemma $BksOf$, this proves $HInv4(p).4'$.

$\langle 2\rangle 3.$ CASE: $\exists\, d \in Disk : Phase1or2Write(p, d)$

$\langle 3\rangle 1.$ ASSUME: 1. $(d \in Disk) \wedge Phase1or2Write(p, d)$
2. $(bk \in blocksOf(p)) \wedge (D \in MajoritySet)$
3. $\forall\, dd \in D : disk[dd][p].mbal \geq bk.bal$

PROVE: $\forall\, dd \in D : disk'[dd][p].mbal \geq bk.bal$

$\langle 4\rangle 1.$ $disk'[d][p].mbal \geq bk.bal$

PROOF: Assumption 1 of $\langle 3\rangle 1$ implies $disk'[d][p] = dblock[p]$ and $phase[p] \neq 0$, so this follows from $HInv4(p).1.R.1$.

$\langle 4\rangle 2.$ Q.E.D.

PROOF: The conclusion of $\langle 3\rangle 1$ follows from its assumption 3 and $\langle 4\rangle 1$, since assumption 1 of $\langle 3\rangle 1$ implies that $disk'[dd] = disk[dd]$ if $dd \neq d$.

$\langle 3\rangle 2.$ Q.E.D.

PROOF: $HInv4(p).4'$ follows from $HInv4(p).4$, $\langle 3\rangle 1$, and the level $\langle 2\rangle$ case assumption, which implies $blocksOf(p)' \subseteq blocksOf(p)$.

$\langle 2\rangle 4.$ Q.E.D.

PROOF: The only way to change $HInv4(p).4$ from true to false is to add a new element to $\{bk.bal : bk \in blocksOf(p)\}$ or to change $disk[d][p]$, for some disk d. The cases covered by $\langle 2\rangle 1$, $\langle 2\rangle 2$, $\langle 2\rangle 3$ include all the subactions of *HNext* that can do this. (That *EndPhase*0 does not add any element to $\{bk.bal : bk \in blocksOf(p)\}$ follows from $Inv2.3(p).1.R.3$, which implies $allBlocksRead(p) \subseteq blocksOf(p)$.)

$\langle 1 \rangle$5. Q.E.D.

PROOF: By steps $\langle 1 \rangle 1$–$\langle 1 \rangle 4$.

A.4.4 Lemma I2e

Simple logic shows that, to prove Lemma $I2e$, it suffices to prove:

ASSUME: 1. $HInv1 \land HInv2 \land HInv2' \land HInv3 \land HInv4 \land HInv5 \land HNext$

2. CONSTANT $p \in Proc$

3. $phase'[p] = 2$

4. $\neg HInv5(p).R.a'$

PROVE: $HInv5(p).R.b'$

$\langle 1 \rangle$1. CASE: $(phase[p] \neq 2)$

$\langle 2 \rangle$1. $EndPhase1or2(p) \land (phase[p] = 1)$

PROOF: By *HNext*, assumption 3, and the level $\langle 1 \rangle$ case assumption.

$\langle 2 \rangle$2. CHOOSE $bk \in allBlocks$ S.T.

$(bk.bal \geq dblock'[p].bal) \land (bk \neq dblock'[p])$

$\langle 3 \rangle$1. CHOOSE $bk \in allBlocks'$ S.T.

$(bk.bal \geq dblock'[p].bal) \land (bk \neq dblock'[p])$

PROOF: Assumption 4 and the definition of *maxBalInp* imply the existence of *bk*.

$\langle 3 \rangle$2. CHOOSE $q \in Proc : bk \in blocksOf(q)'$

PROOF: $\langle 3 \rangle 1$ asserts $bk \in allBlocks$, so the existence of q follows from the definition of *allBlocks*.

$\langle 3 \rangle$3. $bk \in blocksOf(q)$

PROOF: We consider the two cases $q = p$ and $q \neq p$. In both cases, the result follows from $\langle 3 \rangle 2$ and Lemma *BksOf*. If $q = p$, it follows because $bk \neq dblock'[p]$ (by $\langle 3 \rangle 1$). If $q \neq p$, it follows because $\langle 2 \rangle 1$ implies $dblock'[q] = dblock[q]$, so the lemma implies $blocksOf(q)' \subseteq blocksOf(q)$.

$\langle 3 \rangle$4. Q.E.D.

PROOF: By $\langle 3 \rangle 1$, $\langle 3 \rangle 3$, and the definition of *allBlocks*.

$\langle 2 \rangle$3. CHOOSE $q \in Proc \setminus \{p\}$ S.T. $bk \in blocksOf(q)$

PROOF: By $\langle 2 \rangle 2$ and the definition of *allBlocks*, there is some processor q such that $bk \in blocksOf(q)$. Steps $\langle 2 \rangle 1$ and $\langle 2 \rangle 2$ imply $bk.bal \geq dblock[p].mbal$, so $phase[p] = 1$ (by $\langle 2 \rangle 1$) and $HInv4(p).2$ imply $q \neq p$.

$\langle 2 \rangle$4. $\exists\, D \in MajoritySet : \forall\, d \in D : disk[d][q].mbal \geq dblock'[p].bal$

PROOF: By $\langle 2\rangle 3$, $HInv4(q).4$, and $\langle 2\rangle 2$.

$\langle 2\rangle 5.$ $\exists\, D \in MajoritySet : \forall\, d \in D : disk[d][q].mbal > dblock'[p].bal$

PROOF: By $\langle 2\rangle 3$ (which implies $p \neq q$) and $\langle 2\rangle 4$, since $\langle 2\rangle 1$ (which by $HInv2.3(p).2$ implies $dblock'[p].bal > 0$), $HInv2.1$, and the assumption that different processors have distinct ballot numbers imply that $disk[d][q].mbal \neq dblock'[p].bal$.

$\langle 2\rangle 6.$ Q.E.D.

PROOF: $\langle 2\rangle 1$ implies $\neg hasRead(p, d, q)'$, for all disks d. Hence, $\langle 2\rangle 5$ implies $HInv5(p).R.b'$.

$\langle 1\rangle 2.$ CASE: $(phase[p] = 2) \wedge HInv5(p).R.a$

$\langle 2\rangle 1.$ CHOOSE $q \in Proc \setminus \{p\}$ S.T. $\wedge\, EndPhase1or2(q) \wedge (phase[q] = 1)$
$\wedge\, dblock'[q].bal > dblock[p].bal$
$\wedge\, dblock'[q].inp \neq dblock[p].inp$

$\langle 3\rangle 1.$ $dblock'[p] = dblock[p]$

PROOF: By $phase[p] = 2$ (the level $\langle 1\rangle$ case assumption), $phase'[p] = 2$ (assumption 3), and $HNext$.

$\langle 3\rangle 2.$ CHOOSE $q \in Proc$ S.T. $\wedge\, dblock'[q].bal \geq dblock[p].bal$
$\wedge\, dblock'[q].inp \neq dblock[p].inp$
$\wedge\, dblock'[q].bal \notin$
$\{bk.bal : bk \in blocksOf(q)\}$

PROOF: By $\langle 3\rangle 1$, $HInv5.R.a$ (from the level $\langle 1\rangle$ case assumption) and $\neg HInv5.R.a'$ (assumption 3), there exist a processor q and a bk in $allBlocks(q)' \setminus allBlocks(q)$ such that $bk.bal \geq dblock[p].bal$, $bk.inp \neq dblock[p].inp$, and $bk.bal \notin \{bb.bal : bb \in blocksOf(q)\}$. Lemma $BlksOf$ implies $bk = dblock'[q]$.

$\langle 3\rangle 3.$ $dblock[p].bal > 0$

PROOF: By $HInv2.3(p).2.R.1$, and $HInv2.3(p).3$, since $phase[p] = 2$ by the level $\langle 1\rangle$ case assumption.

$\langle 3\rangle 4.$ $dblock'[q].bal > 0$

PROOF: By conjunct 1 of $\langle 3\rangle 2$ and $\langle 3\rangle 3$.

$\langle 3\rangle 5.$ $\neg EndPhase0(q)$

PROOF: By conjunct 3 of $\langle 3\rangle 2$, since $HInv2.3(q).1.R.3$ implies:
$\forall\, d \in Disk : blocksRead[q][d] \subseteq blocksOf(q)$

$\langle 3\rangle 6.$ $EndPhase1or2(q) \wedge (phase[q] = 1)$

PROOF: Conjunct 3 of $\langle 3\rangle 2$ implies $dblock'[q].bal \neq dblock[q].bal$. By $HNext$, this implies either $EndPhase1or2(q) \wedge (phase[q] = 1)$, $Fail(q)$, or $EndPhase0(q)$. The second possibility is ruled out by $\langle 3\rangle 4$ and the third is ruled out by $\langle 3\rangle 5$.

$\langle 3\rangle 7.$ $(q \neq p) \wedge (dblock'[q].bal \neq dblock[p].bal)$

PROOF: $\langle 3\rangle 6$ and $phase[p] = 2$ (by the level $\langle 1\rangle$ case assumption) imply $p \neq q$. We then obtain $dblock'[q].bal \neq dblock[p].bal$ from $HInv2.1$, $\langle 3\rangle 3$, and the assumption that different processors have distinct ballot numbers.

$\langle 3\rangle 8$. Q.E.D.

PROOF: By $\langle 3\rangle 2$, $\langle 3\rangle 6$, and $\langle 3\rangle 7$.

$\langle 2\rangle 2$. CHOOSE $D \in MajoritySet$ S.T.

$$\forall\, d \in D : \wedge\ disk[d][q].mbal > dblock[p].bal \\ \wedge\ hasRead(q, d, p)$$

PROOF: By $HInv2.2(q, d).1$ and conjunct 1 of $\langle 2\rangle 1$, there is a majority set D such that $hasRead(q, d, p)$ and $disk[d][q].mbal = dblock'[q].bal$, for all $d \in D$. The result then follows from conjunct 2 of $\langle 2\rangle 1$.

$\langle 2\rangle 3$. $\forall\, d \in D : [block \mapsto dblock[p],\ proc \mapsto p] \notin blocksRead[q][d]$

PROOF: By $HInv5(p).R.a$ (the level $\langle 1\rangle$ case assumption), conjunct 1 of $\langle 2\rangle 1$, and the definitions of $maxBalInp$ and $EndPhase1or2$, if $dblock[p]$ were in $allBlocksRead(q)$, then $dblock'[q].inp$ would equal $dblock[p].inp$, contradicting conjunct 3 of $\langle 2\rangle 1$.

$\langle 2\rangle 4$. $\forall\, d \in D : \neg\exists\, br \in blocksRead[p][d] : br.block.mbal \geq dblock[p].bal$

PROOF: By $HInv2.3(p).2.R.3$ and $HInv2.3(p).3$, since the level $\langle 1\rangle$ case assumption asserts $phase[p] = 2$.

$\langle 2\rangle 5$. $\forall\, d \in D : \neg hasRead(p, d, q)$

PROOF: We assume $d \in D$ and $hasRead(p, d, q)$, and we obtain a contradiction.

$\langle 3\rangle 1$. $[block \mapsto dblock[q],\ proc \mapsto q] \in blocksRead[p][d]$

PROOF: We have $phase[p] = 2$ (by the level $\langle 1\rangle$ case assumption), $phase[q] = 1$ (by conjunct 1 of $\langle 2\rangle 1$) and $hasRead(q, d, p)$ (by $\langle 2\rangle 2$), so this follows from $hasRead(p, d, q)$ by $HInv3(p, q, d)$ and $\langle 2\rangle 3$.

$\langle 3\rangle 2$. $dblock[q].mbal > dblock[p].bal$

PROOF: Conjunct 1 of $\langle 2\rangle 1$ implies $dblock'[q].bal = dblock[q].mbal$, so this follows from conjunct 2 of $\langle 2\rangle 1$.

$\langle 3\rangle 3$. Q.E.D.

PROOF: $\langle 3\rangle 1$ and $\langle 3\rangle 2$ contradict $\langle 2\rangle 4$.

$\langle 2\rangle 6$. Q.E.D.

PROOF: $\langle 2\rangle 2$ and $\langle 2\rangle 5$ imply $HInv5(p).R.b$. Conjunct 1 of $\langle 2\rangle 1$ implies that $disk$, $dblock[p].bal$ and $hasRead(p, d, q)$ are unchanged, for all $d \in Disk$, so $HInv5(p).R.b$ implies $HInv5(p).R.b'$.

$\langle 1\rangle 3$. CASE: $(phase[p] = 2) \wedge HInv5(p).R.b$

$\langle 2\rangle 1$. CHOOSE $D \in MajoritySet$, $q \in Proc$ S.T.

$$(q \neq p) \wedge HInv5(p).R.b(D, q)$$

PROOF: The level $\langle 1 \rangle$ case assumption implies the existence of D and q satisfying $HInv5(p).R.b(D, q)$. Since any two majority sets have a disk in common, $HInv4(p).3$ then implies $q \neq p$.

$\langle 2 \rangle 2$. CASE: $\exists\, d \in D \,:\, Phase1or2Write(q, d)$

$\langle 3 \rangle 1$. $dblock[q].mbal > dblock[p].bal$.

PROOF: Since D is a majority set (by $\langle 2 \rangle 1$), $HInv4(q).1.R.2$ implies $dblock[q].mbal \geq disk[d][p].mbal$ for some $d \in D$, so the result follows from $HInv5(p).R.b(D, q)$ (which holds by $\langle 2 \rangle 1$).

$\langle 3 \rangle 2$. Q.E.D.

PROOF: The level $\langle 2 \rangle$ case assumption implies that $dblock[p]$ and $hasRead(p, d, q)$ are left unchanged, for all d, and that $disk$ is unchanged except that $disk'[d][q] = dblock[q]$ for some disk d. It follows from this and $\langle 3 \rangle 1$ that $HInv5(p).R.b(D, q)$ (which holds by $\langle 2 \rangle 1$) implies $HInv5(p).R.b(D, q)'$.

$\langle 2 \rangle 3$. CASE: $\exists\, d \in D \,:\, Phase1or2Read(p, d, q)$

PROOF: By $HInv5(p).R.b(D, q)$ (from $\langle 2 \rangle 1$), we have $disk[d][q].mbal > dblock[p].bal$, for all $d \in D$. Since $phase[p] = 2$ (by the level $\langle 1 \rangle$ case assumption), $HInv2.3(p).3$ implies $dblock[p].bal = dblock[p].mbal$, so $disk[d][q].mbal > dblock[p].mbal$ for all $d \in D$. Thus, the case assumption implies $phase'[p] = 1$ (because the ballot must abort), contradicting assumption 3.

$\langle 2 \rangle 4$. Q.E.D.

PROOF: Since $phase'[p] = phase[p] = 2$ (by assumption 3 and the level $\langle 1 \rangle$ case assumption), $HNext$ implies that $dblock[p]$ is unchanged and that, for any $d \in D$:

$\wedge\ (disk'[d][q] \neq disk[d][q]) \Rightarrow Phase1or2Write(q, d)$

$\wedge\ hasRead(p, d, q)' \wedge \neg hasRead(p, d, q) \Rightarrow Phase1or2Read(p, d, q)$

Hence, $\langle 2 \rangle 2$ and $\langle 2 \rangle 3$ cover the only cases in which $HInv5(p).R.b(D, q)$ can be made false. In all other cases, $HInv5(p).R.b'$ follows from $HInv5(p).R.b(D, q)$ (which holds by $\langle 2 \rangle 1$).

$\langle 1 \rangle 4$. Q.E.D.

PROOF: Since $HInv5(p)$ holds by assumption 1, the cases in steps $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, and $\langle 1 \rangle 3$ are exhaustive.

A.4.5 Lemma I2f

The proof of Lemma $I2f$ uses:

LEMMA VC $\quad \forall\, v \in Inputs \,:\, HInv1 \wedge HInv4 \wedge HNext \wedge valueChosen(v)$
$\qquad \Rightarrow\ valueChosen(v)'$

We prove Lemma VC by proving:

ASSUME: 1. CONSTANT $b \in$ UNION $\{Ballot(p) : p \in Proc\}$
2. CONSTANTS $v \in Inputs,\ p \in Proc,\ D \in MajoritySet$
3. $maxBalInp(b, v)$
4. $valueChosen(v)(b).2(p, D)$

PROVE: $maxBalInp(b, v)' \land valueChosen(v)(b).2(p, D)'$

$\langle 1 \rangle 1.$ $maxBalInp(b, v)'$

$\langle 2 \rangle 1.$ CASE: $\exists\, q \in Proc : EndPhase1or2(q) \land (phase[q] = 1)$

$\langle 3 \rangle 1.$ CHOOSE $q \in Proc$ S.T. $EndPhase1or2(q) \land (phase[q] = 1)$

PROOF: q exists by the level $\langle 2 \rangle$ case assumption.

$\langle 3 \rangle 2.$ $allBlocks' \subseteq allBlocks \cup \{dblock'[q]\}$.

PROOF: Lemma $BlksOf$, $\langle 3 \rangle 1$, and the definition of $EndPhase1or2$.

$\langle 3 \rangle 3.$ CASE: $(p \neq q) \land (dblock[q].mbal \geq b)$

$\langle 4 \rangle 1.$ CHOOSE $d \in D$ S.T. $hasRead(q, d, p)$

PROOF: The existence of d follows from $\langle 3 \rangle 1$ and $p \neq q$ (from the level $\langle 3 \rangle$ case assumption), which imply that $hasRead(q, d, p)$ holds for all d in some majority set, since any two majority sets have a disk in common.

$\langle 4 \rangle 2.$ $\exists\, br \in blocksRead[q][d] : br.block.bal \geq b$

PROOF: This is the conclusion of $valueChosen(v)(b).2(p, D)(d).2$, which holds by assumption 4 since $\langle 4 \rangle 1$ implies $d \in D$. Its hypotheses are proved as follows:

- $phase[q] = 1$ holds by $\langle 3 \rangle 1$.
- $dblock[q].mbal \geq b$ holds by the level $\langle 3 \rangle$ case assumption.
- $hasRead(q, d, p)$ holds by $\langle 4 \rangle 1$.

$\langle 4 \rangle 3.$ $dblock'[q].inp = v$

PROOF: By $\langle 4 \rangle 2$, $maxBalInp(b, v)$ (assumption 3), $\langle 3 \rangle 1$, and the definition of $EndPhase1or2$.

$\langle 4 \rangle 4.$ Q.E.D.

PROOF: $maxBalInp(b, v)'$ holds by $\langle 4 \rangle 3$, $\langle 3 \rangle 2$, and $maxBalInp(b, v)$ (assumption 3).

$\langle 3 \rangle 4.$ CASE: $(p = q) \land (dblock[q].mbal \geq b)$

$\langle 4 \rangle 1.$ $\forall\, d \in D : disk[d][p].bal \geq b$

PROOF: By assumption 4.

$\langle 4 \rangle 2.$ $\exists\, d \in D : disk[d][p] = dblock[p]$

PROOF: The level $\langle 2 \rangle$ case assumption and $p = q$ (from the level $\langle 3 \rangle$ case assumption) imply that $disksWritten[p]$ contains a majority set, and hence an element d of D. The result then follows from $HInv2.2(p, d).1$.

$\langle 4 \rangle 3.$ $dblock'[p].inp = v$

PROOF: $\langle 4 \rangle 1$ and $\langle 4 \rangle 2$ imply $dblock[p].bal \geq b$, so $maxBalInp(b, v)$ (assumption 3), $\langle 3 \rangle 1$, $q = p$ (from the level $\langle 3 \rangle$ case assumption), and the definition of $EndPhase1or2$ imply $dblock'[p].inp = v$.

$\langle 4 \rangle 4.$ Q.E.D.

PROOF: Assumption 3, $\langle 3 \rangle 2$, $\langle 4 \rangle 3$, and $p = q$ (the level $\langle 3 \rangle$ case assumption) imply $maxBalInp(b, v)'$.

$\langle 3 \rangle 5.$ CASE: $dblock[q].mbal < b$

PROOF: By $\langle 3 \rangle 1$, this implies $dblock'[q].bal < b$, so $maxBalInp(b, v)$ (assumption 3) and $\langle 3 \rangle 2$ imply $maxBalInp(b, v)'$.

$\langle 3 \rangle 6.$ Q.E.D.

PROOF: By $\langle 3 \rangle 3$, $\langle 3 \rangle 4$, and $\langle 3 \rangle 5$.

$\langle 2 \rangle 2.$ CASE: $\exists\, q \in Proc : Fail(q)$

PROOF: By $maxBalInp(b, v)$ (assumption 3), since $b > 0$ (by assumption 1) and the definition of $Fail(q)$ imply:

$$\{bk \in allBlocks' : bk.bal \geq b\} \subseteq \{bk \in allBlocks : bk.bal \geq b\}$$

$\langle 2 \rangle 3.$ Q.E.D.

PROOF: By $\langle 2 \rangle 1$ and $\langle 2 \rangle 2$, since $HNext$ implies that the only kind of step that can add a new element to $\{\langle bk.bal, bk.inp \rangle : bk \in allBlocks\}$ is an $EndPhase1or2(q) \wedge (phase[q] = 1)$ step or a $Fail(q)$ step, for some processor q.

$\langle 1 \rangle 2.$ $valueChosen(v)(b).2(p, D)'$

$\langle 2 \rangle 1.$ ASSUME: CONSTANT $d \in D$

PROVE: $disk'[d][p].bal \geq b$

$\langle 3 \rangle 1.$ CASE: $Phase1or2Write(p, d)$

$\langle 4 \rangle 1.$ $\exists\, dd \in D : dblock[p].bal \geq disk[dd][p].bal$

PROOF: By $HInv4(p).1.R.2(D)$, since $D \in MajoritySet$ by assumption 2, and $phase[p] \neq 0$ by the level $\langle 3 \rangle$ case assumption.

$\langle 4 \rangle 2.$ $dblock[p].bal \geq b$

PROOF: By $\langle 4 \rangle 1$ and assumption 4, which implies $disk[dd][p].bal \geq b$ for all $dd \in D$.

$\langle 4 \rangle 3.$ Q.E.D.

PROOF: By the level $\langle 3 \rangle$ case assumption, $disk'[d][p] = dblock[p]$, so $\langle 4 \rangle 2$ implies $disk'[d][p].bal \geq b$.

$\langle 3 \rangle 2.$ CASE: $disk'[d][p] = disk[d][p]$

PROOF: In this case, assumption 4 and $d \in D$ (by the level $\langle 2 \rangle$ assumption) imply $disk'[d][p].bal \geq b$.

$\langle 3 \rangle 3.$ Q.E.D.

PROOF: By $\langle 3 \rangle 1$ and $\langle 3 \rangle 2$, since:

$$HNext \wedge (disk'[d][p] \neq disk[d][p]) \Rightarrow Phase1or2Write(p, d)$$

$\langle 2\rangle 2$. ASSUME: 1. CONSTANTS $q \in Proc,\ d \in D$
2. $phase'[q] = 1$
3. $dblock'[q].mbal \geq b$
4. $hasRead(q, d, p)'$

PROVE: $\exists\, br \in blocksRead'[q][d] : br.block.bal \geq b$

$\langle 3\rangle 1$. $phase[q] = 1$

PROOF: By the level $\langle 2\rangle$ assumptions 2 and 4, since:

$HNext \wedge (phase'[q] \neq phase[q]) \Rightarrow InitalizePhase(q)$

and $InitalizePhase(q)$ implies $\neg hasRead(q, d, p)'$.

$\langle 3\rangle 2$. $dblock'[q].mbal = dblock[q].mbal$

PROOF: By the level $\langle 2\rangle$ assumption 4, since:

$HNext \wedge (dblock'[q] \neq dblock[q]) \Rightarrow InitalizePhase(q)$

and $InitalizePhase(q)$ implies $\neg hasRead(q, d, p)'$.

$\langle 3\rangle 3$. CASE: $Phase1or2Read(q, d, p)$

PROOF: Assumption 4 and $d \in D$ (by the level $\langle 2\rangle$ assumption 1) imply $disk[d][p].bal \geq b$. By $Phase1or2Read(q, d, p)$ and the level $\langle 2\rangle$ assumption 4 (which implies that the action does not abort the ballot), this implies:

$[block \mapsto disk[d][p],\ proc \mapsto p] \in blocksRead'[q][d]$

proving the level $\langle 2\rangle$ goal.

$\langle 3\rangle 4$. CASE: $\neg Phase1or2Read(q, d, p)$

$\langle 4\rangle 1$. $hasRead(q, d, p)$

PROOF: By the level $\langle 3\rangle$ case assumption and the level $\langle 2\rangle$ assumption 4, since:

$HNext \wedge \neg hasRead(q, d, p) \wedge hasRead(q, d, p)'$
$\quad \Rightarrow Phase1or2Read(q, d, p)$

$\langle 4\rangle 2$. $\exists\, br \in blocksRead[q][d] : br.block.bal \geq b$

PROOF: By assumption 4, since $d \in D$ by the level $\langle 2\rangle$ assumption 1, $phase[q] = 1$ by $\langle 3\rangle 1$, $dblock[q].mbal \geq b$ by $\langle 3\rangle 2$ and the level $\langle 2\rangle$ assumption 3, and $hasRead(q, d, p)$ by $\langle 4\rangle 1$.

$\langle 4\rangle 3$. Q.E.D.

PROOF: By $\langle 4\rangle 2$ and the level $\langle 2\rangle$ assumption 4, since:

$HNext \wedge hasRead(q, d, p)' \Rightarrow$
$\quad (blocksRead[q][d] \subseteq blocksRead[q][d]')$

$\langle 3\rangle 5$. Q.E.D.

PROOF: By $\langle 3\rangle 3$ and $\langle 3\rangle 4$.

$\langle 2\rangle 3$. Q.E.D.

PROOF: $\langle 2\rangle 1$ and $\langle 2\rangle 2$ imply $valueChosen(v)(b).2(p, D)'$.

$\langle 1\rangle 3$. Q.E.D.

PROOF: By $\langle 1\rangle 1$ and $\langle 1\rangle 2$.

We now prove Lemma $I2f$ by proving:

ASSUME: $HInv1 \wedge HInv2 \wedge HInv2' \wedge HInv3 \wedge HInv5 \wedge HInv6 \wedge HNext$
PROVE: $HInv6'$

$\langle 1 \rangle 1$. ASSUME: $chosen' \neq NotAnInput$
PROVE: $valueChosen(chosen)'$

$\langle 2 \rangle 1$. CASE: $chosen = NotAnInput$

$\langle 3 \rangle 1$. CHOOSE $p \in Proc$ S.T. $EndPhase1or2(p) \wedge (phase[p] = 2)$

PROOF: $HInv2.5$ and the level $\langle 2 \rangle$ case assumption imply $output[p] = NotAnInput$ for all processors p. From $HNext.2$ and the levels $\langle 1 \rangle$ and $\langle 2 \rangle$ assumptions, we deduce that $output'[p] \neq NotAnOutput$ for some $p \in Proc$. By $HNext$, this implies $EndPhase1or2(p) \wedge (phase[p] = 2)$.

$\langle 3 \rangle 2$. $maxBalInp(dblock[p].bal,\ dblock[p].inp)$

PROOF: $\langle 3 \rangle 1$ implies

$$\exists\, D \in MajoritySet : \forall\, d \in D,\ q \in Proc : hasRead(p, d, q)$$

Since any two majority sets have a disk in common, this implies $\neg HInv5(p).R.b$. Hence, $HInv5$ and $\langle 3 \rangle 1$ (which implies $phase[p] = 2$) imply $HInv5(p).R.a$.

$\langle 3 \rangle 3$. $maxBalInp(dblock[p].bal,\ chosen)'$

PROOF: $\langle 3 \rangle 1$, $HNext.2$, and the level $\langle 2 \rangle$ case assumption imply

$$(chosen' = dblock[p].inp) \wedge (dblock'[p].bal = dblock[p].bal)$$

which by $\langle 3 \rangle 2$ implies $maxBalInp(dblock'[p].bal,\ chosen')$. Lemma $BksOf$ and $\langle 3 \rangle 1$ imply that no new element is added to

$$\{\langle\, bk.bal,\ bk.inp \,\rangle : bk \in allBlocks\}$$

so $maxBalInp(b, v)' = maxBalInp(b, v)$ for any constants b and v. If b and v are constants, then $b = dblock'[p].bal$ and $v = chosen'$ imply $maxBalInp(b, v)' = maxBalInp(dblock[p].bal,\ chosen)'$.

$\langle 3 \rangle 4$. CHOOSE $D \in MajoritySet$ S.T.

$$\forall\, d \in D : \wedge\ disk[d][p] = dblock[p]$$
$$\wedge\ \forall\, q \in Proc \setminus \{p\} : hasRead(p, d, q)$$

PROOF: D exists by $\langle 3 \rangle 1$ and $HInv2.2(p, d).1$.

$\langle 3 \rangle 5$. ASSUME: CONSTANTS $q \in Proc,\ d \in D$ S.T.
$\wedge\ phase[q] = 1$
$\wedge\ dblock[q].mbal \geq dblock[p].bal$
$\wedge\ hasRead(q, d, p)$

PROVE: $[block \mapsto dblock[p],\ proc \mapsto p] \in blocksRead[q][d]$

PROOF: $\langle 3 \rangle 1$ and $HInv2.3(p).3$ imply $dblock[p].bal = dblock[p].mbal$; $HInv2.3(p).2.R.3$ and the assumption $dblock[q].mbal \geq dblock[p].bal$ then imply

$$[block \mapsto dblock[q],\ proc \mapsto q] \notin blocksRead[p][d]$$

The result now follows from the conclusion of $HInv3(p, q, d)$, whose hypotheses are proved as follows: $phase[p] = 2$ follows from $\langle 3 \rangle 1$); $phase[q] = 1$ is an assumption; $hasRead(p, d, q)$ follows from $\langle 3 \rangle 4$ (since $phase[p] \neq phase[q]$ implies $p \neq q$); and $hasRead(q, d, p)$ is an assumption.

$\langle 3 \rangle 6$. $\forall\, q \in Proc, d \in D$:

$$\begin{aligned} &\wedge\ phase'[q] = 1 \\ &\wedge\ dblock'[q].mbal \geq dblock[p].bal \\ &\wedge\ hasRead(q, d, p)' \\ &\Rightarrow (\exists\, br \in blocksRead'[q][d] \,:\, br.block.bal = dblock[p].bal) \end{aligned}$$

PROOF: $\langle 3 \rangle 1$ and the assumption $phase'[q] = 1$ imply $q \neq p$, so $\langle 3 \rangle 1$ implies $phase[q]$, $dblock[q]$, $hasRead(q, d, p)$, and $blocksRead[q][d]$ are unchanged, for all disks d. The result now follows from $\langle 3 \rangle 5$.

$\langle 3 \rangle 7$. Q.E.D.

PROOF: We deduce $valueChosen(chosen)'$ as follows:

- $valueChosen(chosen)'(dblock[p].bal).1$ follows from $\langle 3 \rangle 3$ because $\langle 3 \rangle 1$ implies $dblock[p].bal' = dblock[p].bal$.
- $valueChosen(chosen)'(dblock[p].bal).2(p, D).1$ follows from $\langle 3 \rangle 4$, since $\langle 3 \rangle 1$ implies $disk' = disk$.
- $valueChosen(chosen)'(dblock[p].bal).2(p, D).2$ follows from $\langle 3 \rangle 6$.

$\langle 2 \rangle 2$. CASE: $chosen \neq NotAnInput$

$\langle 3 \rangle 1$. $chosen' = chosen$

PROOF: By $HNext.2$ and the level $\langle 2 \rangle$ case assumption.

$\langle 3 \rangle 2$. Q.E.D.

PROOF: We deduce $valueChosen(chosen)$ from the level $\langle 2 \rangle$ case assumption and $HInv6.1$. By Lemma VC and $\langle 3 \rangle 1$, this implies the level $\langle 1 \rangle$ goal, $valueChosen(chosen)'$.

$\langle 2 \rangle 3$. Q.E.D.

PROOF: Immediate from $\langle 2 \rangle 1$ and $\langle 2 \rangle 2$.

$\langle 1 \rangle 2$. ASSUME: CONSTANT $p \in Proc$ S.T. $output'[p] \neq NotAnInput$
PROVE: $output'[p] = chosen'$

$\langle 2 \rangle 1$. CASE: $chosen = NotAnInput$

$\langle 3 \rangle 1$. $\forall\, q \in Proc \,:\, output[q] = NotAnInput$

PROOF: By $HInv2.5$ and the level $\langle 2 \rangle$ case assumption.

$\langle 3 \rangle 2$. Q.E.D.

PROOF: $\langle 3 \rangle 1$, the level $\langle 2 \rangle$ case assumption, and $HNext.2$ imply that if $output'[p] \neq NotAnInput$, then $chosen' = output'[p]$.

$\langle 2 \rangle 2$. CASE: $chosen \neq NotAnInput$

$\langle 3 \rangle 1$. $valueChosen(chosen)$

PROOF: By the level $\langle 2 \rangle$ case assumption and $HInv6.1$.

$\langle 3 \rangle 2.$ $valueChosen(chosen)'$

PROOF: By $\langle 1 \rangle 1$, since the level $\langle 2 \rangle$ case assumption and $HNext.2$ imply $chosen' \neq NotAnInput$.

$\langle 3 \rangle 3.$ $chosen' = chosen$

PROOF: By $\langle 3 \rangle 1$, $\langle 3 \rangle 2$, and Lemma VC, since $valueChosen(v)$ and $valueChosen(w)$ imply $v = w$.

$\langle 3 \rangle 4.$ CASE: $output[p] = NotAnInput$

$\langle 4 \rangle 1.$ $EndPhase1or2(p) \wedge (phase[p] = 2)$

PROOF: By the level $\langle 1 \rangle$ assumption, the level $\langle 3 \rangle$ case assumption, and $HNext$.

$\langle 4 \rangle 2.$ $\exists\, D \in MajoritySet : \forall\, q \in Proc \setminus \{p\} : hasRead(p, d, q)$

PROOF: By $\langle 4 \rangle 1$.

$\langle 4 \rangle 3.$ $\neg HInv5(p).R.b$

PROOF: Since any two majority sets have a disk in common, $\langle 4 \rangle 2$ implies $\neg HInv5(p).R.b(D, q)$ for any majority set D and any $q \neq p$. We then have only to prove $\neg HInv5(p).R.b(D, p)$ for any majority set D. Step $\langle 4 \rangle 1$ implies that $disksWritten[p]$ contains a disk d in D, and $HInv2.2(p, d).1.R.2$ and $HInv2.3(p).3$ then imply $disk[d][p].mbal = dblock[p].bal$, proving $\neg HInv5(p).R.b(D, p)$.

$\langle 4 \rangle 4.$ $maxBalInp(dblock[p].bal,\ dblock[p].inp)$

PROOF: $HInv5(p)$ and $phase[p] = 2$ (from $\langle 4 \rangle 1$) imply $HInv5(p).R$, so $\langle 4 \rangle 3$ implies $HInv5(p).R.a$.

$\langle 4 \rangle 5.$ CHOOSE $bk \in allBlocks,\ b \in$ UNION $\{Ballot(p) : p \in Proc\}$
S.T. $maxBalInp(b, chosen) \wedge (bk.bal \geq b)$

PROOF: The existence of bk and b follows from $\langle 3 \rangle 1$ and the definition of $valueChosen$.

$\langle 4 \rangle 6.$ $dblock[p].inp = chosen$

PROOF: If $dblock[p].bal \geq b$, then this follows from $\langle 4 \rangle 5$ and the definition of $maxBalInp(b, chosen)$. If $dblock[p].bal < b$, then $\langle 4 \rangle 4$ implies $bk.inp = dblock[p].inp$, while $\langle 4 \rangle 5$ implies $bk.inp = chosen$.

$\langle 4 \rangle 7.$ Q.E.D.

PROOF: $\langle 3 \rangle 3$, $\langle 4 \rangle 1$ (which implies $output'[p] = dblock[p].inp$), and $\langle 4 \rangle 6$ imply $output'[p] = chosen'$.

$\langle 3 \rangle 5.$ CASE: $output[p] \neq NotAnInput$

PROOF: In this case, $HInv2.3(p).4$, the level $\langle 1 \rangle$ assumption, and $HNext$ imply $output'[p] = output[p]$; and $HInv6.2$ and $\langle 3 \rangle 3$ imply $output'[p] = chosen'$.

$\langle 3 \rangle 6.$ Q.E.D.

PROOF: By $\langle 3 \rangle 4$ and $\langle 3 \rangle 5$

$\langle 2 \rangle 3.$ Q.E.D.

PROOF: By $\langle 2 \rangle 1$ and $\langle 2 \rangle 2$

$\langle 1 \rangle 3$. Q.E.D.

PROOF: $HInv6'$ follows immediately from $\langle 1 \rangle 1$ and $\langle 1 \rangle 2$.

A.4.6 Theorem R2b

We now prove Theorem $R2b$ by proving:

ASSUME: $HInv \land HInv' \land HNext$

PROVE: $(\exists\, p \in Proc : IFail(p) \lor IChoose(p)) \lor (\text{UNCHANGED } ivars)$

$\langle 1 \rangle 1$. CASE: $\exists\, p \in Proc : Fail(p)$

PROOF: We assume $p \in Proc$ and $Fail(p)$ and prove $IFail(p)$, which implies the goal. We obtain $IFail(p).1$ from $Fail(p).4$. From $Fail(p).1$ we infer the existence of $ip \in Inputs$ satisfying $IFail(p).2(ip).1$; it also satisfies $IFail(p).2(ip).2$ by $HNext.3$ and $HInv2.5$. We deduce $IFail(p).3$ from $Fail(p).4$, $HNext.2$ and $HInv2.5$.

$\langle 1 \rangle 2$. CASE: $\exists\, p \in Proc : (phase[p] = 2) \land EndPhase1or2$

$\langle 2 \rangle 1$. CHOOSE $p \in Proc$ S.T. $(phase[p] = 2) \land EndPhase1or2$

PROOF: p exists by the level $\langle 1 \rangle$ case assumption.

$\langle 2 \rangle 2$. $dblock[p].inp \in allInput$

PROOF: By $\langle 2 \rangle 1$ (which asserts $phase[p] = 2$), $HInv2.3(p).2.R.1$ and $HInv2.3(p).3.R$, we deduce $dblock[p].bal \neq 0$. By conjuncts 3 and 5 of $HInv2.1(p)(dblock[p])$, this implies $dblock[p].inp \in allInput$.

$\langle 2 \rangle 3$. CASE: $chosen = NotAnInput$

$\langle 3 \rangle 1$. $\forall\, q \in Proc : output[q] = NotAnInput$

PROOF: By the level $\langle 2 \rangle$ case assumption and $HInv2.5$.

$\langle 3 \rangle 2$. $\forall\, q \in Proc \setminus \{p\} : output'[q] = NotAnInput$

PROOF: $\langle 3 \rangle 1$ and $\langle 2 \rangle 1$.

$\langle 3 \rangle 3$. $chosen' = output'[p]$

PROOF: By $\langle 3 \rangle 2$, $\langle 2 \rangle 1$ (which implies $output'[p] \neq NotAnInput$), the level $\langle 2 \rangle$ case assumption, and $HNext.2$.

$\langle 3 \rangle 4$. Q.E.D.

$\langle 4 \rangle 1$. $IChoose(p).1$

PROOF: By $\langle 3 \rangle 1$.

$\langle 4 \rangle 2$. $IChoose(p).2$

PROOF: $\langle 2 \rangle 1$ and $\langle 3 \rangle 3$ imply

$\land\ chosen' = dblock[p].inp$

$\land\ output' = [output \text{ EXCEPT } ![p] = dblock[p].inp]$

$IChoose(p).2$ then follows from $\langle 2 \rangle 2$ and the level $\langle 2 \rangle$ case assumption.

$\langle 4 \rangle 3$. $IChoose(p).3$

PROOF: By $\langle 2 \rangle 1$ (which implies $input' = input$), $HInv2.5$, and $HNext.3$.

$\langle 4 \rangle 4$. Q.E.D.

PROOF: $\langle 4 \rangle 1$, $\langle 4 \rangle 2$, and $\langle 4 \rangle 3$ imply $IChoose(p)$, which implies our goal.

$\langle 2 \rangle 4$. CASE: $chosen \neq NotAnInput$

$\langle 3 \rangle 1$. $chosen' = chosen$

PROOF: By $HNext.2$ and the level $\langle 2 \rangle$ case assumption.

$\langle 3 \rangle 2$. $output'[p] = chosen$

PROOF: $HInv6.2'$ and $\langle 3 \rangle 1$ imply $output'[p]$ equals either $chosen$ or $NotAnInput$. Step $\langle 2 \rangle 1$ implies $output'[p] = dblock[p].inp$, which by $\langle 2 \rangle 2$ and $HInv1.9$ implies $output'[p] \neq NotAnInput$.

$\langle 3 \rangle 3$. Q.E.D.

PROOF: $\langle 2 \rangle 1$ and $HInv2.3(p).4$ imply $IChoose(p).1$; $\langle 2 \rangle 1$, $\langle 3 \rangle 1$, $\langle 3 \rangle 2$ and the level $\langle 2 \rangle$ case assumption imply $IChoose(p).2$; and $\langle 2 \rangle 1$, $HNext.3$, and $HInv2.5$ imply $IChoose(p).3$. This proves $IChoose(p)$, which implies the goal.

$\langle 2 \rangle 5$. Q.E.D.

PROOF: By $\langle 2 \rangle 3$ and $\langle 2 \rangle 4$.

$\langle 1 \rangle 3$. Q.E.D.

PROOF: By $\langle 1 \rangle 1$ and $\langle 1 \rangle 2$, since

$$HInv2.5 \land HNext \land (ivars' \neq ivars) \Rightarrow (input' \neq input) \lor (output' \neq output)$$

and

$$HNext \land ((input' \neq input) \lor (output' \neq output)) \Rightarrow \exists\, p \in Proc : Fail(p) \lor ((phase[p] = 2) \land EndPhase1or2)$$