Disk Paxos

Eli Gafni[1] and Leslie Lamport[2]

[1] Computer Science Department, UCLA
[2] Compaq Systems Research Center

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.

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, the algorithm tolerates faulty processors that pause for arbitrarily long periods, fail completely, and possibly restart; and it tolerates lost and delayed messages. 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.

Disk Paxos is a variant of the classic Paxos algorithm [3, 10, 12], a simple, efficient algorithm that has been used in practical distributed systems [13, 16]. 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. Further discussion of the specifications and a sketch of a rigorous correctness proof appear in [5].

2 The State-Machine Approach

The state-machine approach [6, 14] 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 all processors output the same value, which equals $input[p]$ for some p. A solution should be:

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 algorithm requires a three-phase commit protocol [15], with *voting*, *prepare to commit*, and *commit* phases. Nonblocking algorithms that use fewer phases 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, 10, 12] achieves its efficiency by using a three-phase commit protocol, called the *Synod* algorithm, in which the value to be committed 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.

The classic result of Fischer, Lynch, and Patterson [4] implies that a purely asynchronous nonblocking consensus algorithm is impossible. So, real-time clocks 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. 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. (A formal specification appears in the appendix.) 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 the processors are numbered from 1 through N, then processor i could use ballot numbers i, $i+N$, $i+2N$, etc. A ballot has two phases:

Phase 1 Choose a value v.
Phase 2 Try to commit v.

[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 $input[p]$.

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. 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. Since a processor does not choose the value to be committed until phase 2, phase 1 can be performed once 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 or by the same processor in two different ballots. 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 reached phase 2.
inp The value p tried to commit in ballot number bal.

Initially, bal equal 0, inp equals a special value $NotAnInput$ that is not a possible input value, and $mbal$ is any ballot number. 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.

3.2 Why the Algorithm Works

Suppose processor p can read and write a majority of the disks, and all processors other than p stop executing the algorithm. In this case, p will eventually choose a ballot number greater than the *mbal* field of all blocks on the disks it can read, and its ballot will succeed. Hence, this algorithm is nonblocking, in the sense explained above.

We now explain, intuitively, 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 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.

[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 [7].

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. A sketch of a rigorous assertional proof is given in [5].

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 [10]. 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.[4] The Synod Protocol'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,

[4] In the Synod Protocol, a processor q does not bother sending a response if p sends 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 [10].

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 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 [10, 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 [8]. 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.

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 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. In Marios Mavronicolas and Philippas Tsigas, editors, *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG 97)*, volume 1320 of *Lecture Notes in Computer Science*, pages 111–125, Saarbruken, Germany, 1997. Springer-Verlag.
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. Technical Report ??, Compaq Systems Research Center, July 2000.
6. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
7. Leslie Lamport. On interprocess communication. *Distributed Computing*, 1:77–101, 1986.
8. Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.
9. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
10. Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.
11. Leslie Lamport. Specifying concurrent systems with TLA$^+$. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, pages 183–247, Amsterdam, 1999. IOS Press.
12. 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.
13. 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.
14. Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
15. Marion Dale Skeen. *Crash Recovery in a Distributed Database System*. PhD thesis, University of California, Berkeley, May 1982.
16. 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.
17. 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 precise specifications of the consensus problem solved by the Disk Synod algorithm and of the algorithm itself. The specifications are written in TLA$^+$, a formal language that combines the temporal logic of actions (TLA) [9], set theory, and first-order logic with notation for making definitions and encapsulating them in modules. These specifications have been debugged with the aid of the TLC model checker [17]. (However, errors may have been introduced by the manual process of translating from TLA$^+$ to LaTeX.) TLA$^+$ is described in [11]; annotated versions of the specifications, with fuller explanations of the TLA$^+$ constructs, appear in [5].

We feel that the algorithm's nonblocking property is sufficiently obvious not to need a rigorous specification and proof, so we consider only consistency. We therefore do not specify any liveness properties, so we make very little use of temporal logic.

The Specification of Consensus

We assume that there are N processors, numbered 1 through N. Each processor p has two registers: an input register $input[p]$ that initially equals some element of the 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
- if any processor r (including p itself) later sets $output[r]$ to some value w other than $NotAnInput$, then $w = v$.

We first define a specification $ISpec$ that has two additional variables: $allInput$, the set of all inputs chosen so far, and $chosen$, which is set to the first output value chosen. The actual specification $SynodSpec$ is obtained from $ISpec$ by hiding $allInput$ and $chosen$. Hiding in TLA is expressed by the temporal existential quantifier $\boldsymbol{\exists}$. To formally define $SynodSpec$ in TLA$^+$, we define $ISpec$ in a submodule that is then instantiated. However, the reader not familiar with TLA$^+$ can ignore these details and pretend that $SynodSpec$ is simply defined to equal $\boldsymbol{\exists}\, allInput, chosen : ISpec$.

The reader unfamiliar with TLA can consider the specification $ISpec$ to consist of two parts: the initial predicate $IInit$ and the next-state action $INext$, which is a predicate relating the new (primed) state with the old (unprimed) state.

Most of the TLA$^+$ notation used in the definitions should be self-evident, except for the following function constructs: $[x \in S \mapsto g(x)]$ is the function f

with domain S such that $f[x] = g(x)$ for all x in S; $[S \rightarrow T]$ is the set of all functions with domain S and range a subset of T; and $[f \text{ EXCEPT } ![x] = e]$ is the function $\widehat{f}$ that is the same as f except that $\widehat{f}[x] = e$. TLA$^+$ allows conjunctions and disjunctions to be written as bulleted lists, with indentation used to eliminate parentheses.

The specification is contained in the following module named *SynodSpec*. The module begins with an EXTENDS statement that imports the *Naturals* module, which defines the set *Nat* of natural numbers and the usual arithmetic operations. The *Naturals* module also defines $i \,..\, j$ to be the set of natural numbers from i through j.

———— MODULE *SynodSpec* ————

$$
\begin{array}{l}
\textsc{extends } \textit{Naturals} \\[4pt]
\textsc{constant } N,\ \textit{Inputs} \\
\textsc{assume } (N \in \textit{Nat}) \land (N > 0) \\[4pt]
\textit{Proc} \triangleq 1 \,..\, N \\
\textit{NotAnInput} \triangleq \textsc{choose } c : c \notin \textit{Inputs} \\[4pt]
\textsc{variables } \textit{input},\ \textit{output}
\end{array}
$$

———— MODULE *Inner* ————

$$
\begin{array}{l}
\textsc{variables } \textit{allInput},\ \textit{chosen} \\[4pt]
\textit{IInit} \triangleq \begin{array}[t]{l}
\land\ \textit{input} \in [\textit{Proc} \rightarrow \textit{Inputs}] \\
\land\ \textit{output} = [p \in \textit{Proc} \mapsto \textit{NotAnInput}] \\
\land\ \textit{chosen} = \textit{NotAnInput} \\
\land\ \textit{allInput} = \{\textit{input}[p] : p \in \textit{Proc}\}
\end{array} \\[4pt]
\textit{Choose}(p) \triangleq \\
\quad \land\ \textit{output}[p] = \textit{NotAnInput} \\
\quad \land\ \textsc{if } \textit{chosen} = \textit{NotAnInput} \\
\qquad\quad \textsc{then } \exists\, \textit{ip} \in \textit{allInput} : \begin{array}[t]{l} \land\ \textit{chosen}' = \textit{ip} \\ \land\ \textit{output}' = [\textit{output} \text{ EXCEPT } ![p] = \textit{ip}] \end{array} \\
\qquad\quad \textsc{else } \begin{array}[t]{l} \land\ \textit{output}' = [\textit{output} \text{ EXCEPT } ![p] = \textit{chosen}] \\ \land\ \textsc{unchanged } \textit{chosen} \end{array} \\
\quad \land\ \textsc{unchanged } \langle \textit{input}, \textit{allInput} \rangle \\[4pt]
\textit{Fail}(p) \triangleq \begin{array}[t]{l}
\land\ \textit{output}' = [\textit{output} \text{ EXCEPT } ![p] = \textit{NotAnInput}] \\
\land\ \exists\, \textit{ip} \in \textit{Inputs} : \begin{array}[t]{l} \land\ \textit{input}' = [\textit{input} \text{ EXCEPT } ![p] = \textit{ip}] \\ \land\ \textit{allInput}' = \textit{allInput} \cup \{\textit{ip}\} \end{array} \\
\land\ \textsc{unchanged } \textit{chosen}
\end{array} \\[4pt]
\textit{INext} \triangleq \exists\, p \in \textit{Proc} : \textit{Choose}(p) \lor \textit{Fail}(p) \\[4pt]
\textit{ISpec} \triangleq \textit{IInit} \land \Box[\textit{INext}]_{\langle \textit{input}, \textit{output}, \textit{chosen}, \textit{allInput} \rangle}
\end{array}
$$

————————————

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

————————————

The Disk Synod Algorithm

The Disk Synod algorithm's specification appears in module *DiskSynod*, which uses an EXTENDS statement to import all the declarations and definitions from the *SynodSpec* module. The specification introduces three new constant parameters: an operator *Ballot* such that $Ballot(p)$ is the set of ballot numbers that processor p can use; a set *Disk* of disks; and a predicate *IsMajority*, which generalizes the notion of a majority. The specification asserts the assumptions that different processors have disjoint sets of ballot numbers, and that, for any subsets S and T of *Disk*, if $IsMajority(S)$ and $IsMajority(T)$ are true, then S and T are not disjoint.

The specification uses the following variables: *input* and *output* are imported from the *SynodSpec* module; *dblock* and *disk* were explained in the informal description of the algorithm; $phase[p]$ is the current phase of processor p, which is set to 0 when p fails and to 3 when p chooses its output; $disksWritten[p]$ is the set of disks that processor p has written during its current phase; and $blocksRead[p][d]$ is the set of values p has read from disk d during its current phase.

Some additional TLA$^+$ notation is introduced in the specification. TLA$^+$ has the following record constructs: $[f_1 \mapsto v_1, \ldots, f_n \mapsto v_n]$ is the record r with fields $f_1, \ldots, f_n$ such that $r.f_i = v_i$, for each i; 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 each i. The EXCEPT construct has the following extensions: in $[f \text{ EXCEPT } ![x] = e]$, an @ in expression e denotes $f[x]$; the EXCEPT part can have multiple "replacements" separated by commas; and the construct generalizes to functions of functions in the obvious way—for example, $[f \text{ EXCEPT } ![x][y] = e]$. In TLA$^+$, SUBSET S is the set of all subsets of S, and UNION S is the union of all the elements of S.

The algorithm's specification is formula *DiskSynodSpec*, but the reader unfamiliar with TLA can consider the specification to be the initial predicate *Init* and the next-state action *Next*. The module ends by asserting the correctness of the algorithm, expressed in TLA by the statement that the algorithm's specification implies its correctness condition. On first reading, we recommend jumping from the definition of *Init* to the definition of *Next*, and then reading backwards to see what is defined in terms of what.

MODULE *DiskSynod*

EXTENDS *SynodSpec*

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

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

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

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

VARIABLES $disk,\ dblock,\ phase,\ disksWritten,\ blocksRead$

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

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

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

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

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

$$\begin{array}{l}
StartBallot(p) \triangleq \\
\quad \land\ phase[p] \in \{1, 2\} \\
\quad \land\ phase' = [phase \text{ EXCEPT } ![p] = 1] \\
\quad \land\ \exists\, b \in Ballot(p) : \land\ b > dblock[p].mbal \\
\qquad\qquad\qquad\qquad\quad \land\ dblock' = [dblock \text{ EXCEPT } ![p].mbal = b] \\
\quad \land\ InitializePhase(p) \\
\quad \land\ \text{UNCHANGED } \langle input, output, disk \rangle
\end{array}$$

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

$$\begin{array}{l}
Phase1or2Read(p, d, q) \triangleq \\
\quad \land\ d \in disksWritten[p] \\
\quad \land\ \text{IF } disk[d][q].mbal < dblock[p].mbal \\
\qquad\quad \text{THEN } \land\ blocksRead' = \\
\qquad\qquad\qquad\quad [blocksRead \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad ![p][d] = @ \cup \{[block \mapsto disk[d][q],\ proc \mapsto q]\}] \\
\qquad\qquad\quad\ \ \land\ \text{UNCHANGED} \\
\qquad\qquad\qquad\qquad \langle input, output, disk, phase, dblock, disksWritten \rangle \\
\qquad\quad \text{ELSE } StartBallot(p)
\end{array}$$

$$
\begin{array}{l}
EndPhase1or2(p) \triangleq \\
\quad \land\ IsMajority(\{d \in disksWritten[p] : \forall q \in Proc \setminus \{p\} : hasRead(p, d, q)\}) \\
\quad \land\ \lor\ \land\ phase[p] = 1 \\
\qquad\qquad \land\ dblock' = \\
\qquad\qquad\quad [\, dblock \text{ EXCEPT} \\
\qquad\qquad\qquad ![p].bal = dblock[p].mbal, \\
\qquad\qquad\qquad ![p].inp = \text{LET } blocksSeen \triangleq allBlocksRead(p) \cup \{dblock[p]\} \\
\qquad\qquad\qquad\qquad\qquad\quad nonInitBlks \triangleq \\
\qquad\qquad\qquad\qquad\qquad\qquad \{bs \in blocksSeen : bs.inp \neq NotAnInput\} \\
\qquad\qquad\qquad\qquad\qquad\quad maxBlk \triangleq \text{CHOOSE } b \in nonInitBlks : \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \forall c \in nonInitBlks : b.bal \geq c.bal \\
\qquad\qquad\qquad\qquad\qquad \text{IN}\ \ \text{IF } nonInitBlks = \{\} \text{ THEN } input[p] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } maxBlk.inp\,] \\
\qquad\qquad \land\ \text{UNCHANGED } output \\
\qquad \lor\ \land\ phase[p] = 2 \\
\qquad\qquad \land\ output' = [output \text{ EXCEPT } ![p] = dblock[p].inp] \\
\qquad\qquad \land\ \text{UNCHANGED } dblock \\
\quad \land\ phase' = [phase \text{ EXCEPT } ![p] = @ + 1] \\
\quad \land\ InitializePhase(p) \\
\quad \land\ \text{UNCHANGED } \langle input, disk \rangle
\end{array}
$$

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

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

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

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

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

THEOREM $DiskSynodSpec \Rightarrow SynodSpec$