Saltzer

Topics in the Engineering of Computer Systems

CHAPTER SEVEN: COORDINATION, ATOMICITY, AND RECOVERY

March, 1985

TABLE OF CONTENTS

Overview

This chapter explores two closely related system engineering problems: coordinating parallel activities, and recovering from failures of an activity in a computer system. A single conceptual framework, the *atomic operation*, provides a useful way of thinking about both of the problems. An atomic operation appears either to complete or to do nothing at all (in the case of a failure) and it appears to act either completely before or completely after every other atomic operation; in short there is no way to discover that an atomic operation may actually be composed of several suboperations. After defining atomicity and looking at some ad hoc examples of atomic operations we explore systematic ways of achieving both failure atomicity and coordination atomicity. This exploration has three major components: the conceptually elegant *version history*, the commonly encountered *log-lock* system, and the use of *messages* to accomplish multi-site coordination. Several case studies in the appendices provide real-world examples of atomicity as a tool for recovery and coordination.

Glossary

atomic operation—An operation that, when performed by an interpreter, produces no externally visible evidence that it might have been composed of intermediate steps. An operation may be atomic under parallelism, which means that its internal composition is not noticeable by other simultaneously operating interpreters. It may also be atomic under expected error, which means that if some specified error occurs during the operation, the operation will appear either never to have been invoked or else to have been accomplished completely.

atomic stable storage—Storage that is both atomic and stable.

atomic storage—Storage for which a multiword *put* operation can have only two possible outcomes: 1) all data was stored successfully, or 2) no data was stored; previous data unchanged. (Compare with *stable storage.*)

back-out-list protocol—An atomicity logging rule in which each log entry contains a data object identifier and its old value, so that if the system crashes before the atomic operation completes, a recovery algorithm can use the log entry to undo the partially completed atomic operation.

cell storage—Storage in which the *write* or *put* operation destroys old information. Most physical storage devices such as magnetic disk or silicon transistor memory implement cell storage. Cell storage has the problem that the act of writing commits the writer.

commit—For an atomic operation, to renounce the ability to abandon the operation unilaterally. For example, one usually would commit an operation before making the results of the

operation available to other operations. Before committing, the operation can be abandoned and one can pretend that it had never been attempted. After committing, the operation must be able to complete. A committed operation cannot be abandoned; if it can be determined precisely how far its results have propagated, it may be possible to undo the operation by *compensation.* (q.v.) Commitment also includes the expectation that the results will survive any later failure either of the atomic operation or of the system as a whole.

compensation—Performing a set of operations that reverses the effect and all of the consequences of some previously committed operation. Usually applied to situations in which it is necessary to track down and reverse other, a priori unknown operations that may have used the results of the operation being compensated. (Compare with *undo.*)

deadlock—A cycle of parallel activities in which every member of the cycle is waiting for other members of the cycle to complete before it can continue.

decay set—A set of storage blocks, words, tracks, etc., in which several or all members of the set may spontaneously fail together, but independently of any other decay set.

detectable error—An error that can be reliably detected. An error that is not detectable is automatically an *intolerable error.* (q.v.)

expected error—When the outcome of an operation is not the desired result, but is within some range of anticipated undesired behavior for which an automated recovery strategy can be devised. A prerequisite for an expected error is that it be detectable. In contrast, automated recovery cannot be expected to cope with unplanned-for or undetectable errors. (Compare with *intolerable error.*)

hint—Redundant information provided to improve performance, but whose integrity, currency, or validity is not guaranteed. Some mechanism must always be provided for establishing the currency or validity of a hint before it is relied upon.

idempotent sequence—A sequence of operations with the property that the sequence can be stopped at any point and restarted from the beginning, any number of times, and still lead to a correct result. (Note that "idempotent" is correctly pronounced with the accent on the second syllable, not on the first and third.)

intentions-list protocol—An atomicity logging rule in which each log entry contains a data identifier and its new value, so that if the system crashes after the atomic operation passes its commit point, a recovery algorithm can use the log entry to redo any cell storage updates that were lost in the crash.

interpretation level—The operation repertoire of a conceptual interpreter, usually composed of a real interpreter and one or more programs. Other operations (e.g., those of the real interpreter or

of lower levels of interpretation) are not considered part of the repertoire; this restriction may be accomplished either by fiat or by mechanical barriers.

interpreter—An active agent of a computer system; the entity that executes or evaluates functions, programs, requests, commands, instructions, or operations. An interpreter is characterized by a repertoire of operations that it is prepared to perform and a closure variable that identifies the current point of interpretation and the environment in which the operation is to be performed. An interpreter is a piece of hardware, typically supplemented by a program that makes it appear to have a different repertoire of operations. See also the definitions of *repertoire* and *interpretation level.*

intolerable error—An error that is either undetectable or not planned for, and therefore cannot be handled by an automatic recovery strategy. (Compare with *expected error* and *detectable error.*)

lock—A mark made by one operation to protect a data value from reading or writing by another, parallel operation. Used to achieve coordination atomicity.

lock point—In a locking coordination system, the first instant in an atomic operation when every lock that will ever be in the *lock set* has been seized.

lock set—The collection of locks held by an atomic operation when it reaches its *lock point.*

mutual exclusion—A coordination constraint among potentially parallel operations that they must not operate simultaneously, but either may go first. (Compare with *sequence coordination.*)

normal sequence—The usual operation of a interpreter, in which upon completing an operation the interpreter embarks upon the next operation designated by the program being interpreted.

process—The abstract unit of parallel activity. The term "process" has been used so often without careful definition or with varied definitions that it is now unsafe to use for precise communication. The terms *task, activity, locus of execution, execution point, virtual processor,* and *thread* have all been used in the literature for versions of this concept.

progress—A desirable property of a mechanism that coordinates parallel operations: that some useful work get done. Its usual definition is that of the set of parallel operations being coordinated, the coordination mechanism guarantees that it will not abort at least one of the set. In practice, lack of a theoretical progress guarantee can be compensated for by a try-again-later strategy.

raw storage—Storage that may fail, undetectably, at any time. (Compare with *volatile storage.*)

recoverable object—An object whose overall representation is designed so as to survive expected errors of individual storage components or expected errors of an interpreter that makes a change to the object.

recoverable operation—An operation that is atomic under expected errors.

recovery log—An append-only record of a system that implements atomic operations. Generally used to record old data values and completed or anticipated operation sequences to allow undo-redo processing if an error occurs.

redo—To perform some operation again, typically as part of an error recovery scenario. As a general rule, the data being operated on may not be the same as before, because other transactions may have taken place in the interim.

remote procedure call—A protocol for invoking a procedure in a parallel processor that may fail independently of the processor of the invoker. A too-simple remote procedure call protocol may lead to uncertainty or duplicate invocations in some failure scenarios. With careful design a remote procedure call protocol will reliably assure that the procedure is invoked exactly once, no matter when failures occur.

repertoire—The set of operations an interpreter is prepared to perform or evaluate. Since one can, by writing a program, construct operations not part of the original repertoire of an interpreter, it is often useful to talk about the operation repertoire at a given *interpretation level.* (q.v.)

sequence coordination—A coordination constraint among potentially parallel operations that for correctness one of the operations must precede the other. If they run in the wrong order, the result may be incorrect. (Compare with *mutual exclusion.*)

sequence delay—When an interpreter waits for a signal from another interpreter before embarking on the next operation designated by the program being interpreted.

sequence deviation—A sequence delay or a sequence disruption.

sequence disruption—When an interpreter does not proceed to the next operation designated by the program being interpreted.

simple locking—A locking protocol for coordination of atomic operations that requires that no data be read or written until after the *lock point* (q.v.) If the atomic operation is to be recoverable, a further requirement is that no locks be released until commit time. Compare with *two-phase locking.*

stable storage—Storage that never fails to return on a *get* the data that was stored by a previously successful *put.* In practice, storage can be considered stable when the probability of failure is

sufficiently low that it can be ignored for the application. (Compare with *volatile storage* and *atomic stable storage.*)

transaction—An operation composed of a set of smaller operations in a computer system. Often used as a shorthand for "recoverable transaction", which is another term for an operation that is atomic under expected errors.

two–phase commit—A protocol that creates atomic operations out of separate steps that are implemented on different systems that can crash independently. The protocol goes through a preparation phase, in which each site tentatively commits or aborts, and a completion phase, in which one site, acting as coordinator, makes a final decision; thus the name two–phase. Two phase commit has no connection with the similar–sounding term *two–phase locking.*

two–phase locking—A locking protocol for coordination of atomic operations that requires that no locks be released until all locks have been acquired (that is, there must be a *lock point.*) If the atomic operation is also to be recoverable, a further requirement is that no locks for objects to be written be released until commit time. Compare with *simple locking.* Two–phase locking has no connection with the similar–sounding term *two–phase commit.*

undo—To perform an operation that reverses the effect of some previously completed, but not yet committed, operation. (Compare with *compensation.*)

volatile storage—Storage that may return on a *get* either the data that was stored on a previously successful *put* or else a recognizable null value that indicates that the previously *put* data has been lost. (Compare with *stable storage* and *raw storage.*)

write–ahead log protocol—A rule that requires that a log entry be written to stable storage before the corresponding data in cell storage is updated.

A. Introduction

Most computer systems involve several activities that proceed somewhat independently of one another and in parallel, at least conceptually if not in reality. The independence is usually constrained by the need to coordinate the activities. For example, one would want to make sure that two airline agents do not sell the same seat, that a bank transfer involve both a credit to one account and a debit to another, that opening of a vent in an oil refinery be verified before application of heat to the contents of a tank, that batches of printed output lines from two computer users not end up interleaved on the paper. All these constraints require that the system designer have a well–developed technology of coordination of parallel activities.

Whether or not the activities of a system proceed in parallel, errors may occur during system operation. Errors may occur for many reasons: a failure in the electronic hardware, a mistake in a

program, an out–of–reason data value arriving as an input to the system, the breakdown of a piece of mechanical equipment such as a badge reader. Whatever the source of the error, it is usually a requirement of a system design that errors be handled in a systematic, graceful way. An important insight on error recovery comes from the realization that when an activity encounters an error, that line of activity is effectively blocked, at least until some recovery action occurs; the activity following the recovery action operates conceptually in parallel with the original activity, as suggested by figure 7–1. Thus the technology of coordinating parallel activities plays a part in error recovery. One of the key approaches in providing for recovery after an error is the same systematic design technique required to allow coordination.

Conversely, it is often the case that correct coordination of parallel activities implies that one activity should be delayed on the chance that another will interact with it; but it may turn out that no interaction will actually happen; one must actually perform one or both activities to find out. Delaying one until the other is complete is the safe course. If such delays are common, they may affect performance. Therefore, a common design strategy is to estimate the chance that an interaction will actually occur in practice, and if it seems small enough, allow both activities to proceed, with the intention of detecting an interaction if it does happen, and treating it exactly as though an error occurred, with recovery and restart of one or both activities. Thus again we find that the techniques of error recovery and the techniques of coordination have common elements.

The systematic design technique previously mentioned involves the concept of *atomicity*. Atomicity is the performing of several operations so that they appear to be a single step. Parallel activities that are not part of the atomic operation have no opportunity to influence the result once the operation starts and can observe no intermediate results. In the case of an error, atomicity means that there are no leftover partially completed effects that can be discovered above the level that performed the atomic operation. Thus in this chapter we explore first the concept of atomicity and techniques for achieving it, and then the methods by which these techniques are used to accomplish coordination of parallel activities and recovery from errors.

Ideally, these techniques should be able not just to do the required coordination and error recovery, but to do so in a way that is easy to design, easy to understand (for later maintenance) and for which it is easy to verify correctness. The importance of these desiderata should not be underestimated, since the difficulty of discovering and diagnosing mistakes in coordination of parallel activities is orders of magnitude greater than that of finding mistakes in a single, purely sequential, activity. The difficulty arises from the astronomical number of possible combinations in the actual time sequences of even a few parallel activities. When a coordination mistake is noticed, but the evidence is inadequate for diagnosis, it may be impossible to discover exactly which of many possible sequences of operations actually happened. It is therefore effectively impossible to reproduce the trouble under more carefully controlled circumstances. Thus techniques that tend naturally to lead to correct coordination are particularly valuable.

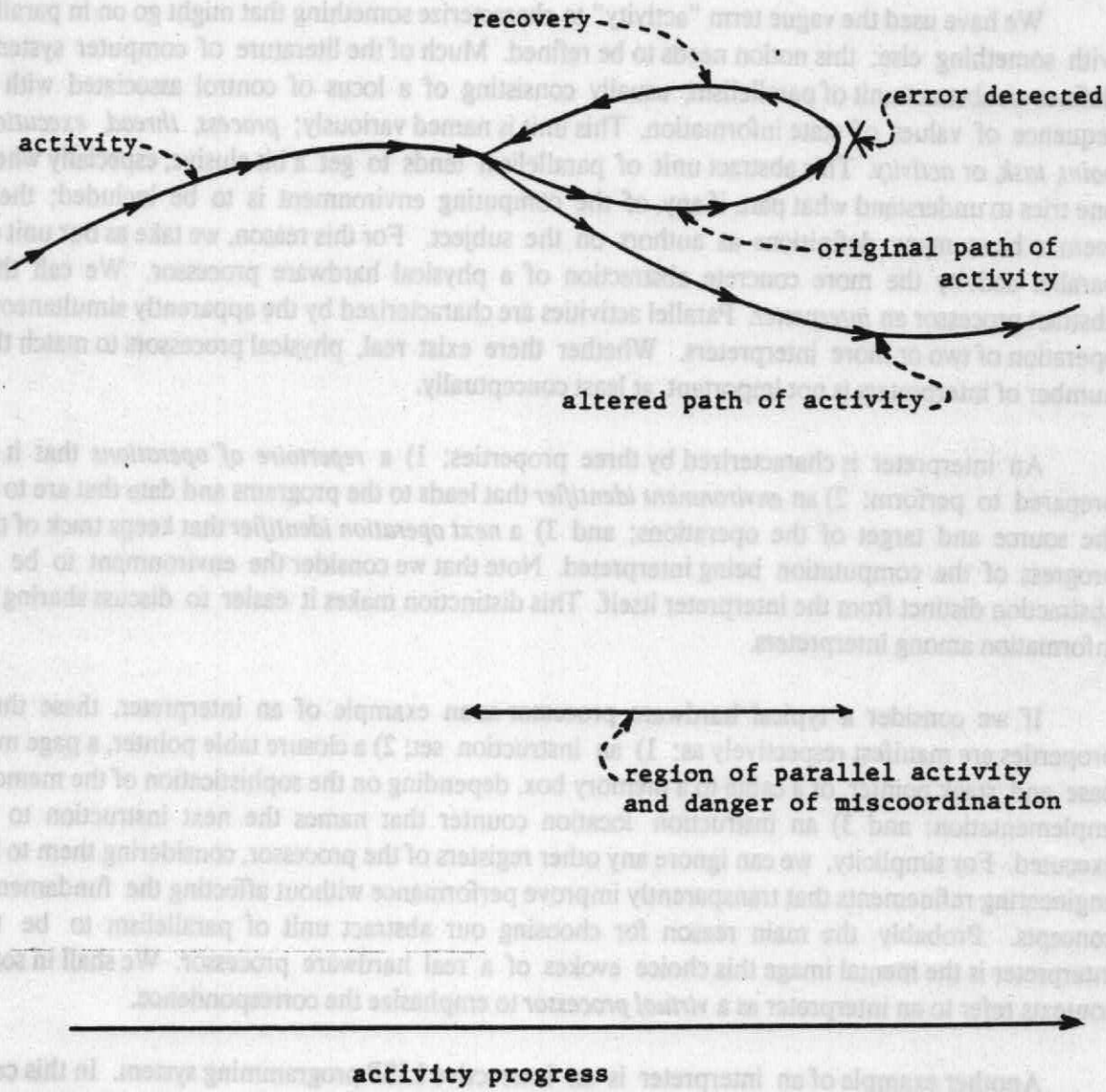


Figure 7-1. How error recovery can introduce parallelism into a purely sequential activity, and consequently introduce the need for coordination.

B. Atomicity

1. Interpreters

We have used the vague term "activity" to characterize something that might go on in parallel with something else; this notion needs to be refined. Much of the literature of computer systems defines an abstract unit of parallelism, usually consisting of a locus of control associated with a sequence of values of state information. This unit is named variously: *process, thread, execution point, task,* or *activity.* This abstract unit of parallelism tends to get a bit elusive, especially when one tries to understand what part, if any, of the computing environment is to be included; there seem to be as many definitions as authors on the subject. For this reason, we take as our unit of parallel activity the more concrete abstraction of a physical hardware processor. We call this abstract processor an *interpreter.* Parallel activities are characterized by the apparently simultaneous operation of two or more interpreters. Whether there exist real, physical processors to match the number of interpreters is not important, at least conceptually.

An interpreter is characterized by three properties: 1) a *repertoire of operations* that it is prepared to perform; 2) an *environment identifier* that leads to the programs and data that are to be the source and target of the operations; and 3) a *next operation identifier* that keeps track of the progress of the computation being interpreted. Note that we consider the environment to be an abstraction distinct from the interpreter itself. This distinction makes it easier to discuss sharing of information among interpreters.

If we consider a typical hardware processor as an example of an interpreter, these three properties are manifest respectively as: 1) an instruction set; 2) a closure table pointer, a page map base and stack pointer, or a cable to a memory box, depending on the sophistication of the memory implementation; and 3) an instruction location counter that names the next instruction to be executed. For simplicity, we can ignore any other registers of the processor, considering them to be engineering refinements that transparently improve performance without affecting the fundamental concepts. Probably the main reason for choosing our abstract unit of parallelism to be the interpreter is the mental image this choice evokes of a real hardware processor. We shall in some contexts refer to an interpreter as a *virtual processor* to emphasize the correspondence.

Another example of an interpreter is an interactive LISP programming system. In this case, the three properties are again manifest: 1) the operation repertoire is the set of LISP language operations; 2) the environment identifier is harder to find; it is the name of the active workspace that contains the programs and data of the user; and 3) the next operation identifier is a variable not usually of concern to the user that moves from one LISP statement to the next in the course of executing functions; it often "points" to the input terminal, since that is the source of the next operation to be interpreted. Note that we use the term "interpreter" to mean the combination of a

hardware engine and the LISP language interpretation program that runs on that engine.[1] We shall presently discover that this composite character of the interpeter turns out to be quite important.

2. *Interpreter sequencing*

The normal state of affairs for an interpreter is to follow some program, operation by operation. When one operation is complete, the interpreter proceeds to another, perhaps the next sequential instruction, an alternative one because of a successful if test, a subroutine, or a signalled procedure. In each of these cases, the next step is apparent by examining the program.[2] The term *normal sequencing* applies to this usual case. There is also a variety of situations in which the normal sequence cannot or should not be followed. We might call these situations *sequence deviations.* It is because of the possibility of sequence deviations that we become interested in atomic operations. There are two common kinds of sequence deviations: *sequence delays*, and *sequence disruptions.*

A sequence delay is a deviation only in timing of the operation of the interpreter. The next operation follows the current one with the normal inevitability, but only after waiting for arrival of a signal from some source outside the interpreter, usually another interpreter. Sequence delays may be imposed upon the interpreter from the outside (perhaps because the real interpreter is being multiplexed among several virtual interpreters,) in which case they are "invisible" to the running program, or they may be requested by the running program by executing an operation to wait for a signal or otherwise coordinate with some activity outside the interpreter.

Sequence disruptions are deviations in what operation is next to be performed by the interpreter. The next operation supplied by the running program for some reason cannot be honored, perhaps because of an error in that program or perhaps because of orders to that effect arriving from outside the interpreter. Sequence disruptions can range from simply stopping the interpreter dead in its tracks, through "backing it up" and undoing previously completed operations, to diverting it to a completely different sequence. All of these kinds of sequence disruptions are, in some semantic sense, "visible" to the running program, in contrast to the normal sequence or sequence delays imposed from outside.

---

1. Frequently the LISP language interpretation program alone is called an "interpreter". That usage of the term is almost always clear from context, though we shall avoid it.

2. We use the terminology of sequential programs here in order to be concrete. It should be understood that a completely equivalent set of observations could be made using the terminology of evaluating applicative functions.

What real world situations might seem to call for sequence deviations, either delays or disruptions? Here are some examples:

1. A hardware error for which there may be a way to recover:
   - parity failure in memory
   - dust on a disk
   - unreadable tape
2. Software or hardware encounters something that may be wrong:
   - attempt to divide by zero
   - negative argument to square root subroutine
3. Dynamic binding or resource allocation is required:
   - missing page
   - context initialization was postponed, now needed
4. There is more important work to do:
   - this job is running longer than expected
   - the president wants an answer to a different question, now
   - multiplexing/time-sharing
5. The user notices the first output values look wrong:
   - calculating *e*, the program starts to display 3.1415...
   - asked to print the wrong file
6. A deadlock has happened:
   - job A has seized a tape drive, and is waiting for the printer
   - job B has seized the printer, and is waiting for the tape drive
7. Several users want to update the same record:
   - airline seat record
   - bank account
   - two typists editing the same memorandum
   - two programmers making changes to a compiler
8. A user needs to change two records "at once":
   - debit one bank account, credit another
   - replace six pieces of the compiler simultaneously, so users get either all old or all new pieces

In the last two examples, it is not so obvious where the sequence deviations might occur. Consider that if one interpreter is to accomplish the job, other interpreters may have to put up with delays or maybe even disruptions.

Although it may not be apparent just yet, all these examples of requirements calling for sequence deviations can be framed in terms of wanting to see some sequence of operations performed either completely or not at all; that is, an atomic operation is involved. To give some insight into how this generalization might be a reasonable one, we first consider in detail how one might handle errors detected by an interpreter, for which consideration we first look more carefully into the structure of a typical interpreter.

3. *Layers of Interpreters*

Practical interpreters are usually built in layers, either systematically or otherwise, starting with a hardware engine that has a fairly primitive repertoire of operations, and adding successive layers of programs so as to provide an increasingly rich or specialized different repertoire of operations. A full-blown application system may involve four or five identifiable layers of interpretation.

Consider, for example, a word processing system for use in preparing correspondence and memoranda. The word processing system as a whole is an interpreter of the requests coming from the typist—requests to retrieve a file, to select a letter, to change part of the letter, to print it on a hard-copy output device. The set of acceptable requests constitutes the operation repertoire of the highest interpretation level.

The word processing system may be implemented in some programming language, such as LISP, PASCAL, PL/I, or BASIC. Each request of the word processing system is typically implemented not by one, but by a sequence of operations in the programming language. The application program thus transforms the operation repertoire of the programming language into the operation repertoire of the word processing system; if this transformation is done carefully and completely, the typist will never notice any evidence of the composite nature of the implementation or of the underlying operation repertoire of the programming language. The operation repertoire of the programming language represents the next-highest level of interpretation of this system.

The programing language operations, in turn, may be interpreted by a lower level program, which uses an operation repertoire of register loads and bit-test instructions, commonly called a "machine language". Each operation of the higher level language typically requires several steps at the lower level. Again, if this interpreter is carefully implemented, the composite nature of the implementation in terms of machine language will be completely hidden from the programmer who writes in the higher-level programming language.[1]

---

1. Usually, the interpreter for the higher-level language translates the program into a form allowing more rapid interpretation; often the translation is into the machine language itself, in which case the program has been *compiled.* We can ignore these translations as being refinements for efficiency

Finally, the machine language itself, though apparently implemented by a piece of hardware, may actually be created by a combination of a hardware microprocessor and a microprogram. The microprocessor has an instruction repertoire consisting of simple data movement and control branching operations; the microprogram interprets a single machine language instruction by executing a sequence of operations from the microprocessor repertoire. Again, with careful design the composite nature of individual machine instructions will be invisible to the machine language programmer.

Thus our hypothetical word processing interpreter might actually be implemented in four layers, as in figure 7-2. Each layer acts in conjunction with the layers below it to provide a complete interpreter at the next higher level. The interpreter at any level accomplishes single operations in the repertoire of the next higher level by a sequence of one or more operations in the repertoire of its own level. The choice of the number of levels that should be implemented for a particular application system and the exact nature of the operation repertoire of the intermediate levels is a vast area of engineering design involving many tradeoffs among speed, cost, availability of interpreter programs and hardware, and compatibility with other systems. Those considerations are very important, but not our current topic. We are interested in this layered multilevel organization for the insight it gives us into atomicity, coordination, and error recovery.

So consider now what happens if an error is detected by the lowest level interpreter, the hardware microprocessor—maybe a register overflow condition. The microprogram is probably "in the middle" of interpreting a machine language instruction, say an add instruction. The machine language add instruction is from a part of the LISP interpreter program that is "in the middle" of interpreting a LISP expression to scan an array. That LISP expression in turn is "in the middle" of interpreting a request from the typist to change the name "Smith" to "Jones". Clearly, the report "Overflow in register 4 while trying to execute microprogram instruction 4174" is not intelligible to the typist; that is a description that is intelligible only at the microprogram level. Unfortunately, the implication of being "in the middle" of higher-level operations is that the only accurate description of the current state of affairs is in terms of the progress of the microprogram.

The actual state of affairs in our word processing example as understood by an all-seeing observer might be the following: the overflow at the microprocessor level was caused by adding one to a register that contained a two's complement negative one at the machine language level. That machine language add instruction was part of an operation to scan an array of characters at the LISP level and a zero means that the end of the array has been reached. The array scan was embarked upon by the LISP level in response to the typist's request to change the name "Smith" to "Jones". The highest level interpretation of the error is thus "the name 'Smith' didn't appear anywhere in this letter". We want to make sure that this report goes to the typist, rather than the one about overflow in the microprocessor. Not only that but we want to be able to assure the typist that this mistake has

---

that, if done correctly, are invisible to all concerned and do not affect the conceptual layer structure.

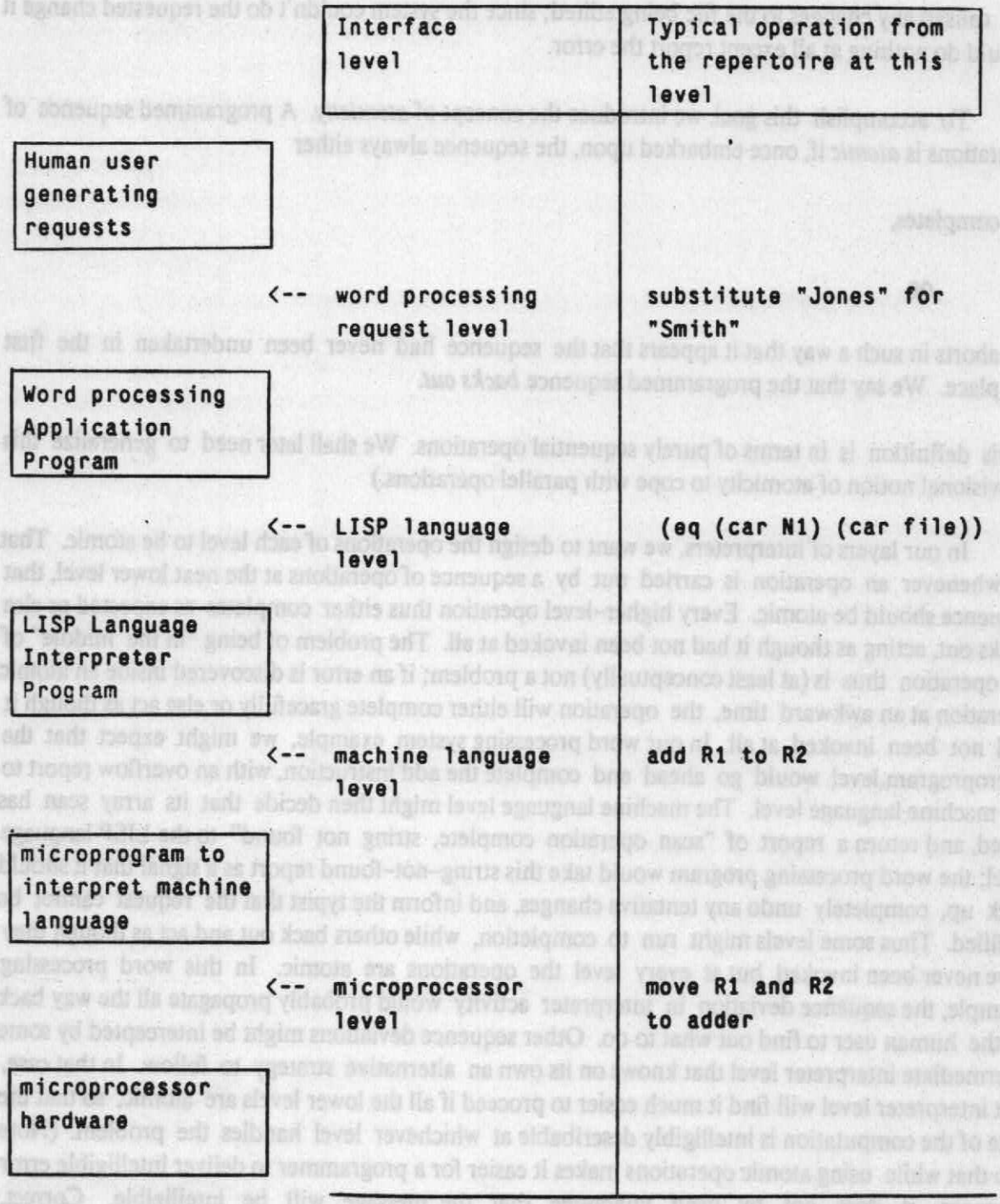


Figure 7-2. Example of an application system that exhibits four levels of interpretation, each with its own operation repertoire.

not caused any changes to the file being edited; since the system couldn't do the requested change it should do nothing at all except report the error.

To accomplish this goal, we introduce the concept of *atomicity*. A programmed sequence of operations is *atomic* if, once embarked upon, the sequence always either

1. completes,

or

2. aborts in such a way that it appears that the sequence had never been undertaken in the first place. We say that the programmed sequence *backs out*.

(This definition is in terms of purely sequential operations. We shall later need to generalize this provisional notion of atomicity to cope with parallel operations.)

In our layers of interpreters, we want to design the operations of each level to be atomic. That is, whenever an operation is carried out by a sequence of operations at the next lower level, that sequence should be atomic. Every higher-level operation thus either completes as expected or else backs out, acting as though it had not been invoked at all. The problem of being "in the middle" of an operation thus is (at least conceptually) not a problem; if an error is discovered inside an atomic operation at an awkward time, the operation will either complete gracefully or else act as though it had not been invoked at all. In our word processing system example, we might expect that the microprogram level would go ahead and complete the add instruction, with an overflow report to the machine language level. The machine language level might then decide that its array scan has failed, and return a report of "scan operation complete, string not found" to the LISP language level; the word processing program would take this string-not-found report as a signal that it should back up, completely undo any tentative changes, and inform the typist that the request cannot be fulfilled. Thus some levels might run to completion, while others back out and act as though they have never been invoked, but at every level the operations are atomic. In this word processing example, the sequence deviation in interpreter activity would probably propagate all the way back to the human user to find out what to do. Other sequence deviations might be intercepted by some intermediate interpreter level that knows on its own an alternative strategy to follow. In that case, that interpreter level will find it much easier to proceed if all the lower levels are atomic, so that the state of the computation is intelligibly describable at whichever level handles the problem. (Note also that while using atomic operations makes it easier for a programmer to deliver intelligible error messages, it does not by itself guarantee that the message will be intelligible. Correct, human-engineered design is still required.)

Atomicity is not usually achieved accidentally, but rather by careful design and planning. To get some insight into what is involved, let us examine some examples of both non-atomic and atomic operations.

4. Examples of atomic and non–atomic operations

One of the simplest and most common examples of an atomic operation is sometimes provided in the bus interface between a processor and a memory module. The bus may provide a "hold" line that, when set high by a processor, prevents all other active bus participants from using the bus. The purpose of this feature is to allow a processor to read a value from memory, change it, and write it back, all without any other processor or input/output device changing the memory value in the interim. The processor designer usually takes some special precaution to avoid sequence deviations while performing one of these atomic read–alter–rewrite operations. For example, if interrupts may arise from input/output devices, the processor may defer recognizing them until the end of the atomic operation. Thus the operation of reading, updating, and rewriting the memory value is guaranteed, once undertaken, to run to the end. (Or, if for example, the atomic operation attempts to read a non–existent address, the operation can abort before changing anything visible, and therefore appear never to have been tried at all.)

Non–atomic operations that are troublesome have often been discovered upon adding virtual memory features to a computer architecture. Unless the original machine architect arranged things so that every machine language instruction was an atomic operation, there will be cases in which a missing page can be discovered "in the middle" of an instruction, after some information has been irretrievably lost. When such a situation arises, the designer of the virtual memory feature is trapped. The instruction cannot run to the end, because one of the operands it needs is not in real memory. While the missing page is being retrieved from secondary storage, the designer would like to allow the processor to be used for something else (for example to run the program that fetches the missing page) but reusing the processor requires that the state of the currently executing program be saved, so that it can be restored later when the missing page is available.

If every instruction is an atomic operation, one can simply set the next–instruction pointer back to point once again to the current instruction (that is, the one that encountered the missing page) and then save the program's state. The resulting saved state description shows that the program is between two instructions, one of which has been completely executed, and the next one of which has not yet begun. Later, when the page is available, the program can be restarted simply by reloading all the program–visible registers and transferring control to the instruction that previously faulted; this time it may succeed. When the instruction set is non–atomic, this simple, clean interrupt scheme is not possible. Several techniques have been invented to "retrofit" atomicity at the machine language level. Appendix 7–A describes some examples of machine architectures and the techniques that were used to add virtual memory to them.

A second example of an interface that needs to be atomic is a call to a supervisor routine. For example, a common supervisor function is one to read the next typed character from an attached keyboard. There is a good chance that at the instant the application program calls the supervisor no character has been typed yet, and the question arises as to what the supervisor should do. Different systems choose one of three possibilities, the first one of which is non–atomic:

1. The supervisor program waits for the user to type a character. Although this possibility is conceptually simpler than the other two, note that the state description (for example, to an application debugging program) is "blocked in the middle of the supervisor". If the supervisor operates in a protection domain different from the user, this state description is not at all helpful.

2. Unwind the call, adjust the return point to the original call instruction, and transfer directly to the supervisor "wait" function. Now, if the user asks about the current state of his program, the answer is "blocked, poised to execute a read call". This description is not only intelligible to the programmer, the programmer can save it and then restore the program to that state later.

3. Return to the calling program with a zero–length result, expecting that the program itself can cope with the problem. (The program would probably test the length of the result and if zero, call the supervisor "wait" function.) This approach also leaves the programmer with a simple, clean description of the current state of affairs.

The second and third alternatives correspond to the two possibilities in the definition of an atomic operation. In the second alternative the supervisor program aborts in such a way that it appears that the call had never taken place, while in the third alternative the supervisor program completes every time it is called. Both alternatives make the supervisor programs atomic operations, and both lead to a user–intelligible state description if a sequence deviation should happen to occur while waiting.

In most systems, the collection of supervisor functions acts as a kind of programmed extension of the machine language level, interposed between the application program and the interpreter of the machine language. In terms of figure 7–2, one can talk of an additional level, just above the machine language level, the supervisor language level. When viewed this way, it becomes more apparent why it is of interest to design supervisor entries to act as atomic operations.

5. Coordinating parallel interpreters

In many computer systems it is a helpful abstraction to imagine that there are several program interpreters operating in parallel, simultaneously. The actual underlying hardware system may be implemented with only a single (or a few) hardware processors that are *multiplexed* to produce the effect of many virtual processors, or there may be direct hardware interpretation of some or all of the virtual processors. Let us assume that multiplexing is successfully hidden from our considerations, and that virtual processors, however created, are the units of interest for coordination. Such parallel program interpreters each individually follow programs that appear to be sequential in nature, so they do not represent any special problem until their paths cross. The way in which paths cross can always be described in terms of shared objects. A *shared object* is some piece of stored, changeable information that two different interpreters happen to take an interest in at the same time. From the point of view of the programmer of an application, there are two quite

different kinds of coordination requirements: *sequencing* and *mutual exclusion. Sequencing* is a constraint of the type "W must happen before X". For correctness the first one mentioned must complete before the second begins. For example, reading of input data from a typing keyboard must be complete before the program to present that data on a display can operate. As a general rule, sequencing constraints are anticipated at the time a program is written, and the identity of the parallel activities is known by the programmer. Sequencing constraints are thus usually explicitly programmed, using either special language constructs or, if such constructs are not available, operating system entry points and shared variables.

In contrast, *mutual exclusion* is a constraint that two operations should not run at the same time, but either one may operate first. One might put this "either W must happen before X, or else X must happen before W". Generally, the programmer of one such operation, such as W in our example, does not know the identity (e.g., X) of the other operations that might require exclusion. This lack of knowledge makes it difficult to accomplish mutual exclusion by explicit program steps, and thus the programmer would be aided by automatic, implicit mechanisms that assure that mutual exclusion happens when it is needed. Such implicit mutual exclusion can be achieved with the aid of atomic operations. Consider, for example, a banking application. We might define a procedure named "transfer" that debits one account and credits a second account, as follows:

```
transfer:  procedure(debitaccount, creditaccount, amount);
           debitaccount <-- debitaccount - amount;
           creditaccount <-- creditaccount + amount;
           return;
```

If this procedure is applied to accounts A (initially containing $300) and B (initially containing $100) as in

```
call transfer(A, B, $10);
```

we expect account A, the debit account, to end up with $290, and account B, the credit account, to end up with $110. Suppose, however, a second, parallel program interpreter is executing the statement

```
call transfer(B, C, $25);
```

where account C starts with $175. We expect B to end up with $85 and C with $200. Further, this expectation should be fulfilled no matter which of the two transfers happens first. The danger occurs if the two transfers happen at about the same time. To understand this danger, consider figure 7-3, in which several possible time sequences of read and write operations of the two interpreters are illustrated. With each sequence is shown the history of values of the cell containing the balance of account B. If both the operations 1-1 and 1-2 precede both the operations 2-1 and 2-2, (or vice-versa) the two transfers will work as expected, and B ends up with $85. If, however, operation 2-1 occurs after operation 1-1, but before operation 1-2, a mistake will occur; one of the

```
      interpreter #1                          interpreter #2
          .                                       .
          .                                       .
          .                                       .
1-1   read creditaccount(B)              2-1   read debitaccount(B)
          .                                       .
          .                                       .
          .                                       .
1-2   write creditaccount(B)             2-2   write debitaccount(B)
          .                                       .
          .                                       .
          .                                       .
```

Six possible histories of account B

	step	value	step	value	step	value	step	value	step	value	step	value
operation sequence	1-1	100	2-1	100	1-1	100	1-1	100	2-1	100	2-1	100
	1-2	110	2-2	75	2-1	100	2-1	100	1-1	100	1-1	100
	2-1	110	1-1	75	1-2	110	2-2	75	1-2	110	2-2	75
	2-2	85	1-2	85	2-2	75	1-2	110	2-2	75	1-2	110
	desired result				undesired result							

Figure 7–3. Possible histories if two interpreters with a shared variable do not coordinate their activities.

two transfers will not affect account B, even though it should have. The first two columns illustrate the history of shared variable B for the two cases in which the answers are the desired result; the remaining four columns show four different sequences leading to two undesirable values for B.

Thus we need to assure that one of the operation sequences of the first two columns is the one actually followed. The best way to describe this requirement is that the two steps 1–1 and 1–2 should be atomic, and the two steps 2–1 and 2–2 should similarly be atomic. In terms of the original procedure, the step

```
debitaccount <-- debitaccount - amount;
```

should be atomic. There should be no possibility that a parallel program interpreter that intends to change the value of the shared variable "debit-account" read its value between the read and write steps of this statement.

We can also find an example of a higher-level atomicity requirement in the same application. Suppose the following audit procedure is available; its purpose is to verify that the sum of the balances of all accounts is zero:

```
audit:  procedure;
        sum <-- 0;
        for A <-- each account do;
                sum <-- sum + balance of A;
                end;
        if sum ≠ 0 call for investigation;
        return;
```

Suppose that the audit procedure is running at the same time that someone else applies the transfer procedure between a pair of accounts. If audit examines one of the accounts before the transfer and the other account after the transfer, then the amount of money transferred will be picked up twice (or not at all—depending on which account it examined first) and the audit will fail. So the entire audit procedure should occur either before or after any transfer: we want the audit procedure to be an atomic operation. Similarly, if the audit program should run after the statement

```
debitaccount <-- debitaccount - amount;
```

and before the statement

```
creditaccount <-- creditaccount + amount;
```

audit will calculate a sum that does not include "amount"; we conclude that the transfer procedure itself ought to be an atomic operation. That is, the transfer operation should occur either completely before or completely after any audit operation.

6. *The definition of atomicity*

What exactly is the meaning of "atomic" with respect to coordination of parallel activities? Our earlier notion that the sequence of sub–steps either completes or backs out is not sufficient; there must be more to atomicity if the concept applies also to coordination. We replace our earlier, provisional description of atomicity with this final, more general one:

An operation is atomic if there is no way to discover that its implementation is composite.

This definition is the fundamental specification of the concept of atomicity. We immediately draw two important consequences of this specification:

1. From the point of view of an activity that invokes an atomic operation, the atomic operation always appears either to complete as expected, or to do nothing but report why it cannot. This consequence is the one that makes atomic operations useful in recovering from expected errors.

2. From the point of view of a parallel activity, an atomic operation acts as though it occurs either *completely before* or *completely after* every other parallel operation. This consequence is the one that makes atomic operations useful for coordinating parallel activities.

These two consequences are not really different. They are simply two perspectives, the first from inside and the second from outside the operation itself; both points of view follow from the single idea that the composite nature of the operation is not visible outside the operation. Such hiding of internal structure is the essence of modularity, and we have here defined a strong form of modularity. We are hiding not just the details of which operations compose the atomic operation, but the very fact that it is composite. Note that we have used the phrases "appears to complete" and "acts as though" rather than demanding that an atomic operation actually do those things. This apparently relaxed view is all that is required for correctness, and it leaves an opportunity for the implementation to operate in any way that provides the atomic effect.

This slightly relaxed view is often stated in the following way: parallel operations are considered to be correctly coordinated if their result is one that could have been obtained by *some* purely sequential application of those same operations. So long as the only coordination requirement is mutual exclusion, any order will do. Underlying this perspective is a straightforward modularity goal: being able to make an argument for correctness of a coordination mechanism without getting tangled up in questions of whether or not the application using the mechanism is correct. Figure 7–4 shows, abstractly, the effect of applying some atomic operation to a system: the

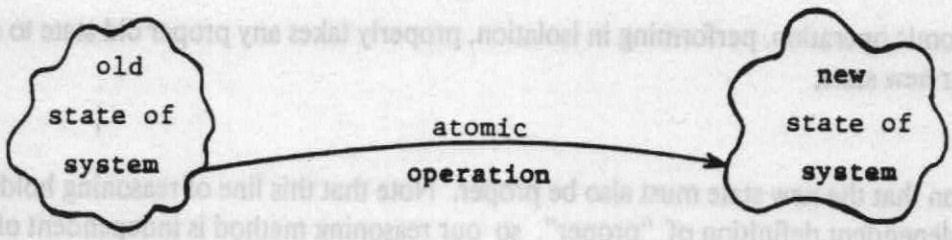


Figure 7-4. A single atomic operation takes a system from one state to another.

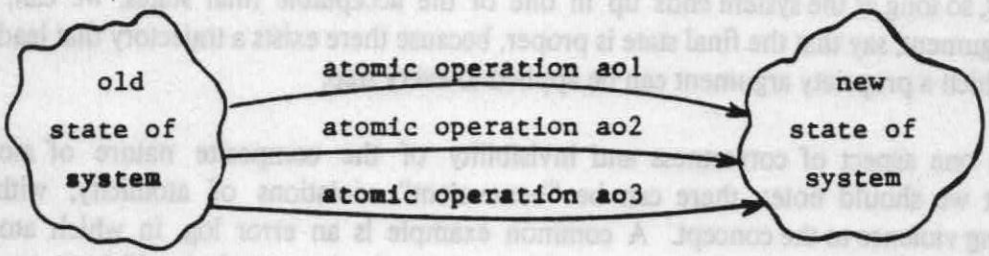


Figure 7-5. When several atomic operations act in parallel, they together produce a new state.

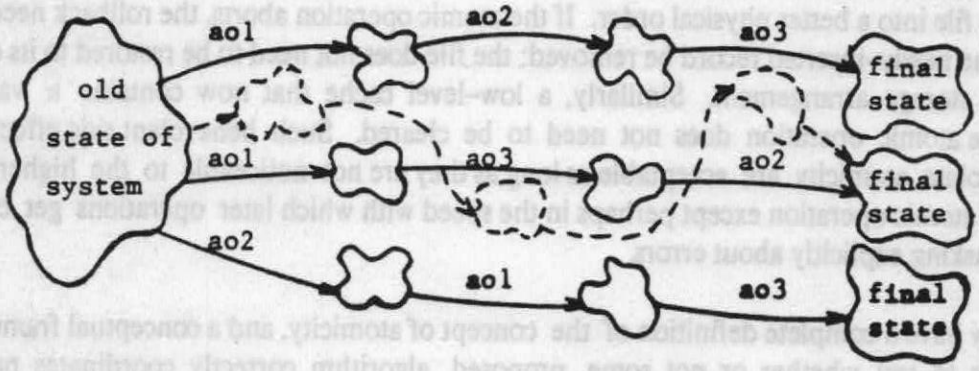


Figure 7-6. We insist that the final state be one that could have been reached by some sequential ordering of the atomic operations, but we don't care which ordering. In addition, we do not need to insist that the intermediate states ever actually exist. The actual state trajectory could be that shown by the dotted lines, *but only if there is no way of observing the intermediate state from outside.*

state of the system changes. If we assume

1. the old state of the system was proper, and

2. the atomic operation, performing in isolation, properly takes any proper old state to a proper new state,

then we can reason that the new state must also be proper. Note that this line of reasoning holds for any application–dependent definition of "proper", so our reasoning method is independent of that definition and of the application. Now, our coordination requirement is that when several atomic operations act in parallel, as in figure 7–5, the resulting new state ought to be one of those that would result from some sequential ordering, as in figure 7–6. We do *not* need to insist that the system actually traverse the intermediate states along any particular path of figure 7–6—it may actually follow the dotted trajectory through intermediate states that are not by themselves proper ones. However, so long as the system ends up in one of the acceptable final states, we can, for purposes of argument, say that the final state is proper, because there exists a trajectory that leads to that state for which a propriety argument can be applied to every step.

There is one aspect of correctness and invisibility of the composite nature of atomic operations that we should note: there can be "benevolent" violations of atomicity, without necessarily doing violence to the concept. A common example is an error log, in which atomic operations that run into trouble record the nature of the error for later analysis. If the operation were perfectly atomic, then when an error leads to rollback, the audit log would be rolled back, too; but rolling it back would defeat its purpose—we prefer that the error log record the fact that the atomic operation tried and failed. A further example of a benevolent atomicity violation occurs often in data management systems: an atomic operation asks to insert a new record into a file, and the data management system decides as a performance optimization that now is the time to rearrange the file into a better physical order. If the atomic operation aborts, the rollback need only insure that the newly–inserted record be removed; the file does not need to be restored to its older, less efficient, storage arrangement. Similarly, a low–level cache that now contains a variable touched by the atomic operation does not need to be cleared. Such benevolent side effects that apparently violate atomicity are acceptable as long as they are not noticeable to the higher–level client of the atomic operation except perhaps in the speed with which later operations get carried out, or when asking explicitly about errors.

We now have a complete definition of the concept of atomicity, and a conceptual framework that allows us to test whether or not some proposed algorithm correctly coordinates parallel activities. We have not yet identified a corresponding framework for discussing recovery in a systematic way; that is the topic of the next section.

C. Recovery

1. Recovery models

In order to talk systematically about atomicity and recovery from errors there are some important distinctions regarding errors that help in the design of recovery algorithms. As a general rule, one can design recovery algorithms to cope only with specific, *expected* errors. Further, a recovery algorithm can cope only with errors that are actually *detected*. Thus when making arguments about whether or not a system design has satisfactory recovery procedures, it is helpful to distinguish among three kinds of error events:

1. A *detectable error* can be detected reliably. If it occurs, we will discover it with certainty.

2. An *expected error* is one for which there is some procedure available for recovery.

3. An *intolerable error* is any error that is either undetectable or unexpected.

The term "intolerable" does not mean that the error cannot happen, but rather that if it does the system should not be expected to meet its specifications—it cannot tolerate such errors. Similarly, the term "expected" implies that even if such errors occur, the system should still operate correctly—it is designed to tolerate such errors, provided they are detected. Figure 7–7 illustrates how these kinds of error events overlap.

The usual effect of an error is that some part of the state of the system is wrong. A subtle consequence of the concept of an expected error is that there must be a well-defined boundary around that part of the state that might be wrong. The recovery procedure must restore all of the state within that boundary, using only information that is safely outside the boundary. The real meaning of detectable, then, is that the error is discovered before its consequences have propagated beyond this well-defined boundary.

The distinctions among detectable, expected, and intolerable errors are the basis for a systematic recovery design procedure that goes as follows:

1. Analyze the system for all possible failure events. Categorize them into those that can be reliably detected and those that cannot. At this stage, detectable or not, all errors are intolerable.

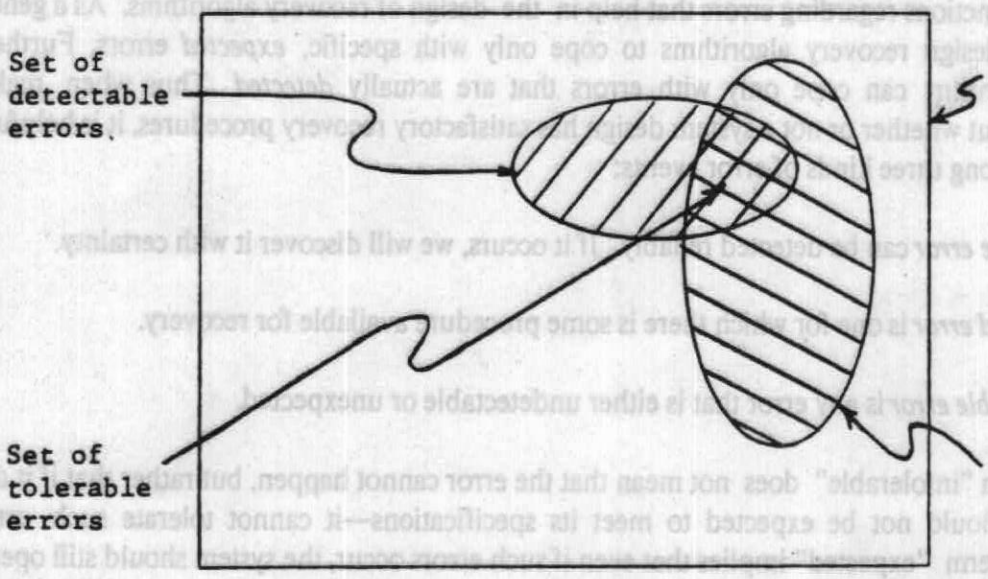


Figure 7–7. Detectable, expected, and tolerable errors. All errors that are not both expected and detectable are intolerable.

2. For each undetectable error, evaluate the probability of its occurrence. If not negligible, modify the system design in whatever way necessary to make the error reliably detectable. Otherwise the undetectable error will be the system's downfall.

3. For each detectable error, evaluate the probability of its occurrence. If not negligible, devise a recovery algorithm for it and reclassify it as a tolerable error.

When finished with such a design procedure, the designer should have a useful recovery specification for the system. Some errors, which have negligible probability of occurrence, will be intolerable, while others will have specified recovery algorithms. A review of the system recovery strategy can separately focus on two distinct (but sometimes confused) questions:

1. Is the designer's list of intolerable failures complete, and is the assessment of negligible probability for them realistic?

2. Is the designer's set of algorithms to cope with the expected errors complete and correct?

These two questions are very different in nature. The first is a question of models of the real world. It addresses an issue of experience and judgement about real-world probabilities and whether or not all real-world modes of failure have been discovered. Two different engineers, with different real-world experiences, may reasonably disagree on such judgements—they may have different models of the real world. The second question is more abstract and also more absolutely answerable, in that an argument for correctness—unless it is a hopelessly complicated argument—or a counter-example showing an error that the recovery algorithm does not handle, should be something that everyone can agree on. In system design, it seems very helpful to follow design procedures that distinctly separate these classes of questions. When someone questions a reliability feature, the designer can first ask "are you questioning the correctness of my recovery algorithm or are you questioning my model of what needs to be recovered?," and thereby properly focus the discussion or argument.

Let us apply this method of reasoning to a simple error-correcting code for memory words, just to see how it works. Suppose that with each 32-bit block of memory are stored enough extra bits (in this case five) to allow any single-bit error to be corrected and any double-bit error to be detected, using a standard encoding.[1] When one reads an encoded word from memory, before

---

1. A code that can correct all one-bit errors and detect all two-bit errors is constructed by cleverly choosing as legitimate data values only some of the total number of possible bit patterns. The ones that are chosen all have the property that to transform any one of them to any other, at least four bits must be changed. Then, if one bit gets changed accidentally, not only will the result be detectably wrong (since no legitimate code value has this pattern) but it is obvious which bit got changed, since the result differs from only one legitimate bit pattern in only one bit value. If two bits get changed, one can still detect that the coded word is wrong, but not decide which was the original correct value, so two-bit errors are not correctable. If three or more bits get changed, the resulting word may look like a one-bit error in some other value, and all bets are off. The systematic construction of such codes is a field of study by itself. Chapter 9, section 7, suggests some reading on the subject.

applying the error-decoding algorithm, the following event analysis applies:

```
                                                          probability
desired event:      all 37 raw bits are correct             modest
undesired events:
    expected:       exactly one bit is wrong                O(p)
    expected:       exactly two bits are wrong              O(p squared)
    intolerable:    three or more bits are wrong            O(p cubed)
```

After applying the error decoding algorithm, the event analysis changes to:

```
                                                          probability
desired event:      all 32 data bits are correct            very high
undesired events:
    intolerable:    reported error (because two bits were wrong)  O(p squared)
    intolerable:    unreported error (because more than           O(p cubed)
                    two bits were wrong)
```

The recovery algorithm has eliminated all the failures with probability of order p. It has not eliminated the reported two-bit errors, so that one must be considered intolerable. (If this error correcting code were embedded in a system that provided a recovery procedure for values reported to be in error, we could reclassify the two-bit error as expected.) We can now question this design on two independent points. First, we ask whether or not the estimate of bit failure probability is realistic, and whether or not it is realistic to suppose that multiple bit failures are statistically independent of one another (failure independence appeared in the analysis in the claim that the probability of an n-bit failure has the order of the nth power of the probability of a one-bit failure.) Those questions concern the real world, and the accuracy of our model of it. Second, we can ask whether or not the coding algorithm actually corrects all one-bit errors and detects all two-bit errors. That question is explored by examining the mathematical technique used to construct the code values and is quite independent of anybody's estimate or measurement of real-world failure rates.

Error-detecting codes are the usual method for reducing the probability of intolerable errors in storage devices. All storage devices seem to be subject to physical failures of various kinds: an attempt to read or write a bit may produce the wrong bit value, or a bit may spontaneously change value. Storage that can fail without warning we label *raw* storage, and unless its probability of failure is extremely low, one must plan to apply some reliability-enhancing technique to it. In the case of a primary memory technology where one-bit errors are the most likely problem, such as electronic random access memory, one might store one extra bit with each byte, or a few extra bits

---

bits get changed, one can still detect that the bit pattern is wrong, but not decide which was the original correct value, so two-bit errors are not correctable. If three or more bits get changed, the resulting bit pattern may look like a one-bit error in some other value, and all bets are off. The systematic construction of such codes is a field of study by itself. Chapter 9, section 7, suggests some reading on the subject.

with a larger block, coding the extra bits to allow detection of one-bit errors. In the case of a disk memory, where bad spots in the magnetic medium may take out a group of adjacent bits, one might record a "longitudinal cyclic redundancy check" code of 32 or more bits at the end of each track. The details of the coding method are unimportant: what matters is that both of these techniques transform the raw storage into what may be termed *volatile* storage: storage that may fail, but in a *detectable* way. One must always keep in mind that there is a residual—one hopes very small—probability that an error will alter enough different bits in such a way that the result slips through without error detection. The definition of volatile storage is that the probability of intolerable (that is, undetected) errors is small enough to be ignored, but the probability of detectable errors is large enough that one must plan for them.

For long-term storage of files, volatile storage is not very satisfactory, so one usually resorts to multiple-copy techniques, placing the copies on different physical devices so that the probability of a detectable error in one copy is independent of the probability of error in the next one. A carefully designed multicopy strategy can produce what is called *stable* storage, whose definition is that the probability of any error, detectable or not, over the lifetime of the stored data, is so low as to be negligible. Note in this regard that some physical storage media, such as electronic random access memory, tends naturally to be best organized as volatile storage, because they depend on a continuous supply of electric power and will reset if the power fails or the system is accidentally unplugged. Because data stored in magnetic media, such as disk or tape, can survive power outages, magnetic media are usually considered better candidates for the extra effort of a stable storage implementation.

2. *Example: Atomic put using version numbers*

A realistic example of our recovery design procedure can become quite complicated. Let us start with a quite simple model that allows us to implement an atomic operation for disk update. The algorithm used in this model was first described in connection with the American Airlines *SABRE* seat reservation system, in 1961.

We start with a processor and operating system that use a volatile primary memory and a stable secondary memory (disk storage). If the processor and operating system components are designed without much thought given to system error recovery, it may actually be very difficult to build a reliable system out of them. However, only a minor effort is required to do it right. As a general rule, these two components do *not* have to be exceptionally reliable, but they *do* have to be able to detect their own mistakes. This rule will be seen repeatedly in the following detailed set of assumptions about these components.

Processor/Primary memory/Operating system

```
desired events:     Processor, memory, system follow specifications
undesired events:   something goes wrong in hardware or operating system
    expected:       failure detected, processor goes to restart point
                    before any disk writes
    intolerable:    failure not noticed, processor muddles along
                    and writes bad data on the disk
```

In other words, the system must be designed to "crash" in a predictable, systematic manner whenever anything goes wrong in either hardware or operating system software. This design requirement does not forbid including in the system design local error recovery procedures that, for example, retry failed instructions in the hope that they will work a second time. The essence of the requirement is that any time the result is going to come out wrong, the system crash immediately rather than continue without doing something about the error. Such a system is sometimes called *fail-fast*. The part of the system state that might be in error is all processor registers and all of primary memory, and the first step of recovery is to reset those registers and memory to zero. Restart will thus depend entirely on the disk storage system to provide application state storage, and any failure must be detected before anything incorrect gets written to the disk storage system.

Disk storage system

We assume the usual semantics of secondary storage. That is, there are two operations, *Put* and *Get*. Most important, we assume that the secondary storage is stable. That assumption, unfortunately, doesn't eliminate all undesired events, because a failure in the operating system can still disturb secondary storage, as seen in this event breakdown:

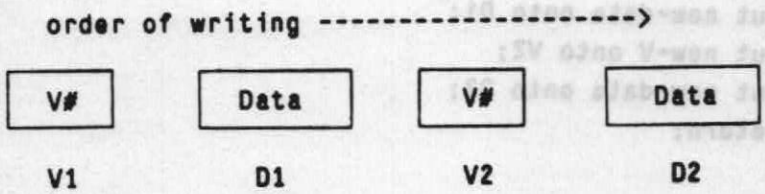
```
Put(address, data)

   desired event:       all words of data are written at address
   undesired events:
       expected:        system crashes during write; some new data has
                        overwritten some old data from the beginning.
       intolerable:     wrong data is written at address or the new data
                        is written at the wrong address.

Get(address, data)

   desired event:       previously written data is returned
   undesired events:
       expected:        none planned
       intolerable:     previously written data is read with errors
```

This model oversimplifies by assuming that the raw hardware provides stable storage directly; in a real magnetic disk one would expect that the intolerable error category contains occurrences of non-negligible probability. To meet this criticism, we later examine a better, but much more complex model. But for the moment let us proceed to develop an atomic update algorithm, just to see how it works. Note that all expected errors lead to the same situation: the system detects the error, resets all processor registers and primary memory, and it restarts. If a disk write was in progress at the instant of the crash, a single disk record may be partially overwritten with new data.

We also assume that when a crash occurs, the system will send a message to all attached terminals alerting their operators of the crash; an agent in the middle of a transaction now wants the atomic property to hold: either the transaction in progress is complete, or else it has not occurred at all. The agent will inquire as to which, and then if necessary reenter the transaction from the beginning. Finally, we assume that a transaction involves writing exactly one disk block.

The problem, of course, is that the *Put* operation does not provide the atomicity we need. It may overwrite only part of a disk record before the system crashes, and the next person to read that record may find an unrecognizable mish-mash of old and new data there.

What we want to do is create "atomic disk storage," in which we guarantee either to change the data completely and correctly or else not to touch it at all. The overall strategy is to write the data twice, preceded each time by a version number, taking care to do everything in a predictable *oraer*, as in the following diagram:



Suppose for a moment that some record has been correctly stored in this double format, and someone updates it with a new version. If the update follows the ordered double-write strategy, even if a crash occurs in the middle of the update a later reader can always be assured of getting a complete, consistent version of the data by reading all four records and examining the version numbers. If V2 is identical to V1, then data D1 must be OK since V2 is written after D1 is written and no crash could have taken place while writing D1. Similarly, if V2 is different from V1, then data D2 must be the old un-updated data, since the crash occurred before V2 was written. This correctness argument depends on the supposition that the two copies were intact when the update started, so the update algorithm should check to make sure they are intact before beginning the update.

Figure 7-8 shows a complete version of this algorithm, with programs named *atomic-get* and *atomic-put*. In that figure the individual steps are labelled *a* through *i*, for reference in the following discussion. We can make several observations about the algorithm of figure 7-8:

```
atomic-get:     get(V1, D1, V2, D2);
                return {if V1 = V2 then D1 else D2}

atomic-put      get(V1, D1, V2, D2);                                  a
                if V1 = V2 then {if D1 ≠ D2, put D1 onto D2}          b
                           else {put D2 onto D1;                      c
                                 put V2 onto V1};                     d
                new-V <-- V1 + 1;                                     e
                put new-V onto V1;                                    f
                put new-data onto D1;                                 g
                put new-V onto V2;                                    h
                put new-data onto D2;                                 i
                return;
                                                                      ↑
                                                  Step identification for
                                                  the notes in the text
```

Figure 7–8. The *SABRE* double-write atomicity algorithm.

1. This atomicity algorithm assumes that only one processor is working on the data. That processor can crash and restart any number of times; it can even crash during *atomic-put* and then do *atomic-get*. But we don't have to consider the possibility that both *atomic-get* and *atomic-put* are in progress simultaneously.

2. Steps *b*, *c*, and *d* of figure 7-8 are preparing for recovery in case a crash happens before completing all steps of *atomic-put*. Step *b* takes care of the case that the previous use of *atomic-put* on this data made it through step *h* but crashed in step *i*. Steps *c* and *d* fix things up in case the previous use of *atomic-put* made it through step *f* but crashed before getting through step *h*.

3. Steps *a* through *g* of the algorithm are *idempotent*; that means that one can crash out of them and repeat them from the beginning any number of times with the same ultimate result as if no crash had occurred.

4. Step *h* exposes the new data to *atomic-get* operations.

5. If we never get to step *i* it doesn't matter, since the next invocation of *atomic-put* will complete the job in step *b*.

6. In the failure-free case, *atomic-get* does four reads, and *atomic-put* does four reads and four writes, assuming each version number and data copy is in a separate disk record. These numbers can be halved by pairing version numbers with data in the same records, with no effect on expected failures.

It is interesting to analyze the visibility of the data, as seen by a caller to *atomic-get*, during the *atomic-put* operation. Figure 7-9 illustrates that at all times during the operation of *atomic-put*, an *atomic-get* operation will return only completely old or completely new data. Whenever *atomic-put* writes a data record, it writes on the copy that is hidden from *atomic-get* because of the values of the version numbers. And whenever *atomic-put* writes a version number, the two data copies are both known to be good copies. This visibility argument provides further confidence in the correctness of the algorithm.

As noted earlier, a real magnetic disk provides raw storage that one could consider stable only if the application had unusually high tolerance for lost data. A somewhat more elaborate model of disk failure modes would take into account the relatively high probability that a disk record may be written or read incorrectly, or that it may spontaneously decay. Such a model, due to Lampson and Sturgis, is found in Appendix 7-B, together with the more elaborate recovery algorithms required to deal with the additional expected error events, and thus produce real stable storage.

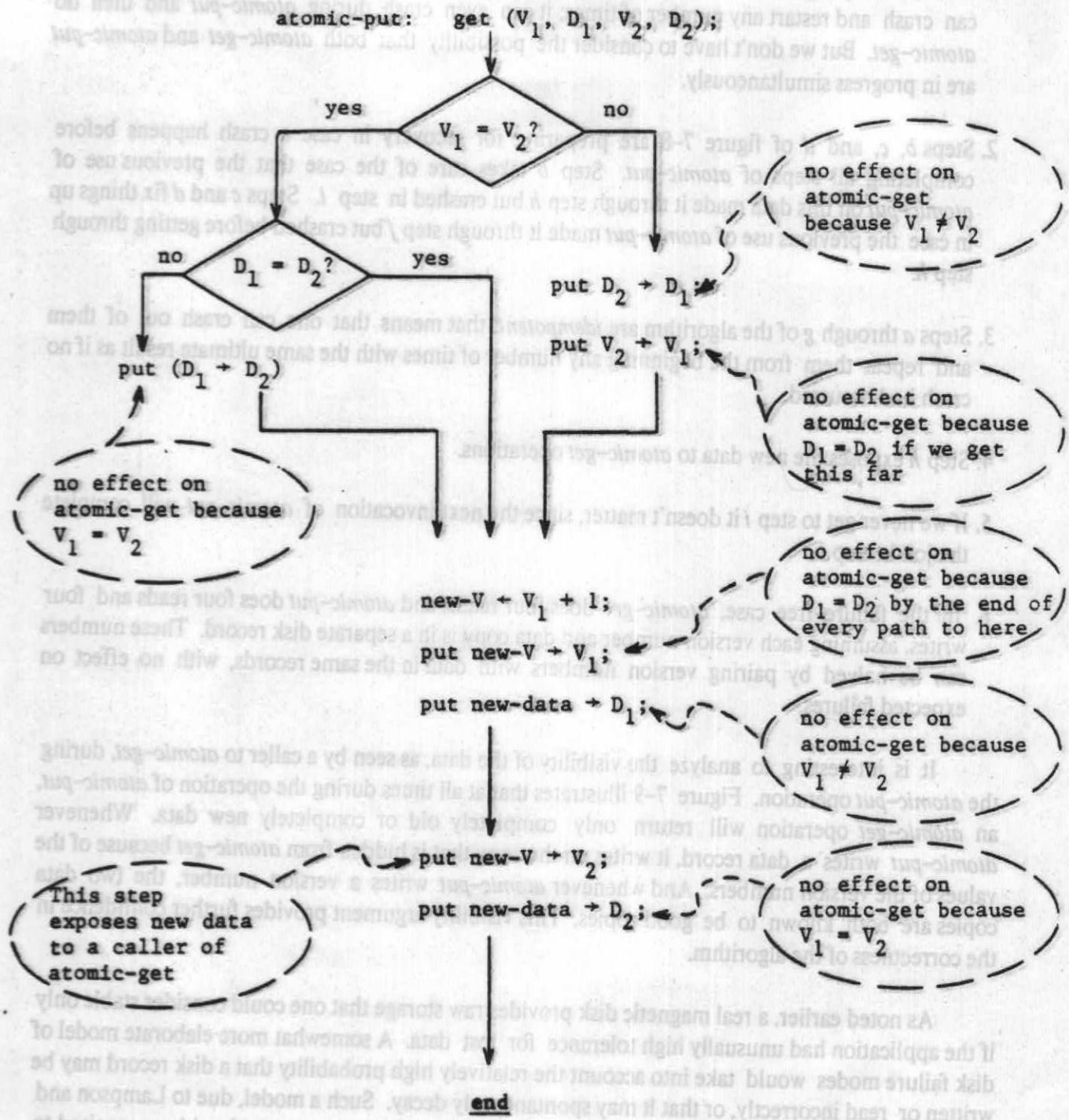


Figure 7-9. Visibility of data to a caller of *atomic-get* during operation of *atomic-put* when using the *SABRE* double-write-with-version-numbers algorithm.

D. Systematically achieving atomicity

Our definition of atomicity may lead us to believe that atomic operations would be nice to have, but it offers no clue as to how to achieve atomicity. Our example of the *SABRE* disk update algorithm demonstrates that atomicity can be attained, at least in some special cases, but again that example offers little guidance as to how to create an atomic operation. We approach this subject by first sketching a high–level, very general overview of how atomic operations are assembled, introducing the idea of a *commit* point. Then we look at one systematic way to achieve atomicity for both expected errors and coordination, using the idea of a version history. Finally, we look at logs and locks, a pair of techniques commonly employed to achieve atomicity.

1. Achieving atomicity: commit points

Ideally, one might like to be able to take any arbitrary sequence of operations in a program, surround that sequence with some sort of *begin* and *end* statements as in figure 7–10, and expect that the language compilers and lower level operating system will perform some magic that makes the surrounded sequence into an atomic operation. Although one can try hard to provide such a friendly programming environment, it appears that the programmer always needs to make some concession to the requirements of atomicity. This concession is expressed in the form of a discipline on the lower–level steps of the atomic operation.

In its most general form, the discipline consists of identifying some step of the sequence of lower–level operations as the *commit point*. The atomic operation is thus divided into two phases, a *pre–commit phase* and a *post–commit phase*, as suggested by figure 7–11. During the pre–commit phase, the disciplining rule of design is that no matter what happens, it must be possible to back out of this atomic operation in a way that leaves no trace. During the post–commit phase the disciplining rule of design is that no matter what happens, the operation must run to the end successfully. Thus an atomic operation can have two outcomes. If the atomic operation backs out, we say that it *aborts*. If the atomic operation passes the commit point, we say that it *commits*.

We can make several observations about the nature of the pre–commit phase. The pre–commit phase must identify all the resources needed to complete the operation, and establish their availability. The names of data should be bound, access control should be checked, the pages to be read or written should be in memory, tapes should be mounted, space must be allocated, etc. In other words, all the steps needed to anticipate the severe run–to–the–end–without–faltering requirement of the post–commit phase should be completed during the pre–commit phase. In addition, the pre–commit phase must maintain the ability to abort at any instant. Any changes that

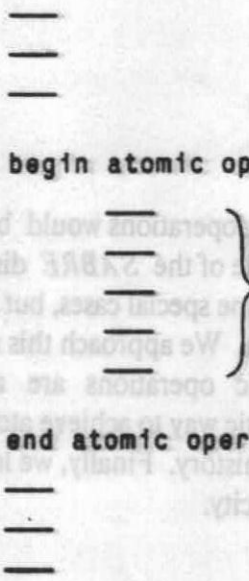


Figure 7–10. Imaginary semantics for painless programming of atomic operations.

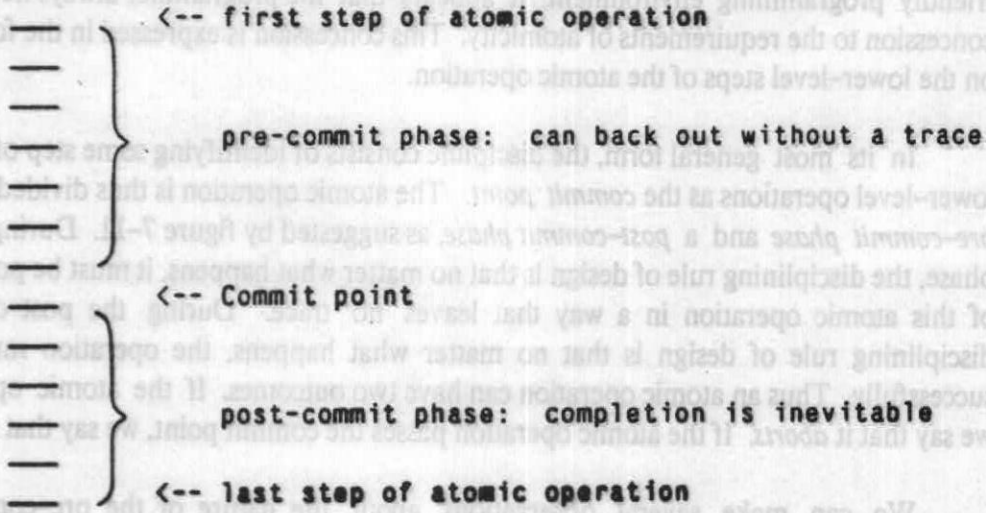


Figure 7–11. The commit point of an atomic operation.

the pre–commit phase makes to the state of the system must be both invisible to parallel activities and undoable in case this atomic action aborts. Usually, this requirement means that shared resources, once reserved, cannot be released until the commit point is passed. (Once an atomic operation releases a shared resource, some other, parallel operation may seize that resource indefinitely. If the resource is needed in order to undo some effect of the first atomic operation, releasing the resource is tantamount to abandoning the ability to abort.) Finally, the reversibility requirement means that the atomic operation should not do anything externally visible (e.g., print a check or fire a missile) prior to the commit point.[1]

In contrast, the post–commit phase can expose results to parallel activities, it can release reserved resources that are no longer needed, and it can perform externally visible actions (e.g., print a check). But it cannot try to reserve anything, because an attempt to reserve might fail, and the post–commit phase is not permitted the luxury of failure. The post–commit phase must confine itself to finishing just the activities that were planned during the pre–commit phase.

Two concrete examples illustrate several of these ideas:

1. *The dry run.* This technique was used, for example, in some IBM System/360 processors to ensure that the instruction set (which includes operations that touch several operands) is atomic despite the possibility of missing pages in the virtual memory:

   Pre–commit: Do a practice execution of the operation in order to generate its address trace, and try to touch all the pages that will be required to complete it. If a missing page is encountered, abort this instruction, fetch the missing page, then restart this instruction from the beginning. The practice execution is carried out delicately, being careful not to modify any of the programmer–visible processor registers.

   Post–commit: Now do the operation for real, fetching the operands, all of which are, for the moment, guaranteed to be in primary memory.

2. *Work on a copy.* This technique is used in some data management systems such as text file editors to ensure that following a system crash the user does not end up with a file containing some, but not all, of the intended changes:

---

1. We are intentionally avoiding a more complex and hard–to–verify possibility. Some systems allow other, parallel operations to see uncommitted results, and they allow externally visible actions before commit. Those systems must also be prepared to track down and abort those parallel operations (this tracking down is called *cascaded abort*) and perform compensating external actions (e.g., a send letter requesting return of the check or apologizing for the missile firing.)

Pre–commit: Create a complete duplicate copy of the data file that is to be modified. Then, make as many changes as desired to the copy.

Post–commit: Carefully exchange the copy with the original. Release the space that was occupied by the original.

It should be apparent that the *SABRE* atomic put is a specialized version of the work–on–a–copy strategy. These examples may increase our confidence that atomicity can be achieved in at least some special cases, but we have not yet identified a systematic way to create atomic operations.

A fundamental difficulty confronts a designer trying to create atomic operations: memory, when organized as a set of named storage cells, has semantics that are hard to cope with. The act of storing data destroys old data; storing commits a new data value. If the atomic operation later aborts, the old value is irretrievably gone; at best it can only be reconstructed from information kept elsewhere. In addition, storing data in a cell reveals it to the view of parallel activities, whether or not the atomic operation that stored it is ready for that exposure. If the atomic operation happens to have exactly one output value, then writing that value into a memory cell can be the mechanism of committing the operation, and there is no problem. But if the result is supposed to consist of several output values, all of which should be committed simultaneously, it is harder to see how to construct the atomic operation. Once the first output value is stored, the computation of the remaining outputs has to be successful; there is no going back. Examining the special technique of the text file editor above ("work on a copy") provides a useful clue, however. The essence of the mechanism that allows a text editor to make several changes to a file, yet not reveal any of the changes until commit time, is the following: the only way another prospective reader of a file can get to read it is by going through some catalog to find the file's physical address. Until commit time the editor is working on an uncatalogued, and therefore effectively invisible, copy of the file. The operation of cataloguing the new version (which might be accomplished by storing a pointer to it in a catalog memory cell) is the step that makes the entire set of updates simultaneously visible to other participants.

This observation suggests that atomic operations would be better served by a model of storage that behaves differently: instead of a model in which a store operation overwrites old data, we would instead create a new, tentative version of the data, in a way that the tentative version remains invisible to all readers outside this atomic operation until the operation commits. We can provide such semantics even though we start with a traditional cell–type memory by interposing a layer between it and the user who requires tools to implement atomic operations. This layer can implement *version storage*. The basic idea of version storage is quite straightforward: we associate with every named object not a single cell, but a list of cells; the values in the list represent the history of the variable. Figure 7–12 illustrates. Whenever anyone proposes to write into the object, the version storage manager appends the prospective new value to the end of the list. Clearly this approach, being history–preserving, offers some hope of being helpful, because if an atomic operation aborts, one can imagine a systematic way to locate and discard all of the new versions it wrote. Moreover, we can explicitly equip the version manager to expect to receive tentative values,

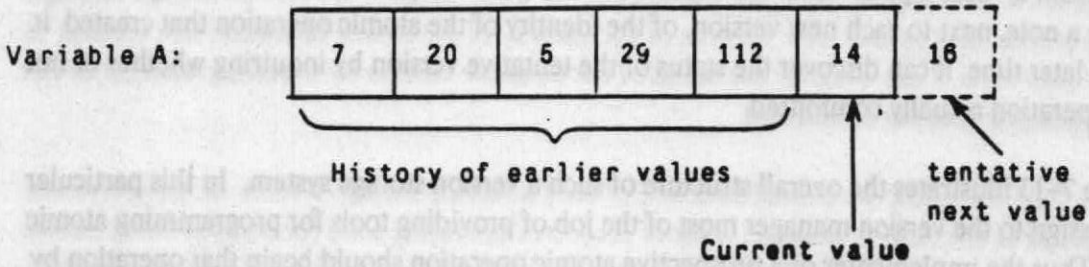


Figure 7–12. Version history of a variable.

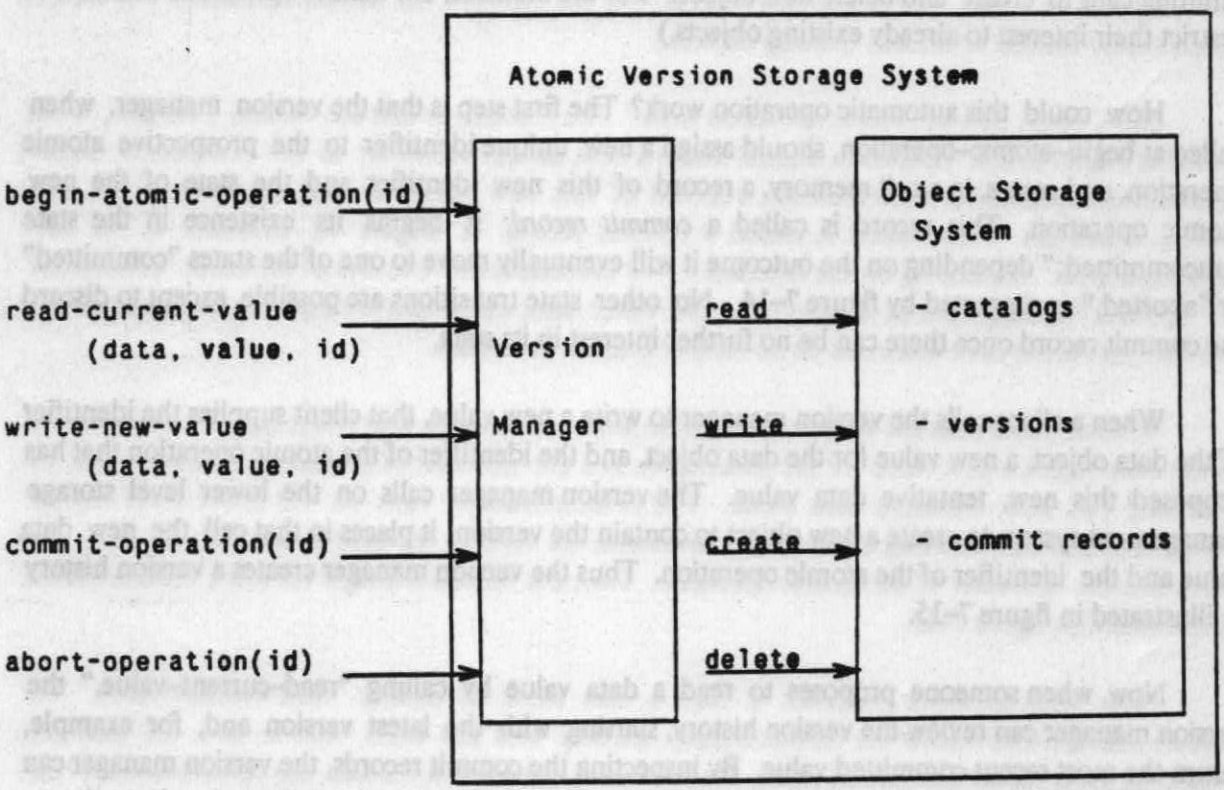


Figure 7–13. Interface to and internal organization of an atomic storage system based on version histories.

and to be prepared to ignore them if the atomic operation that created them fails to commit. The basic mechanism to accomplish such an expectation is quite simple; the version storage manager should make a note, next to each new version, of the identity of the atomic operation that created it. Then, at any later time, it can discover the status of the tentative version by inquiring whether or not the atomic operation actually committed.

Figure 7–13 illustrates the overall structure of such a version storage system. In this particular model, we assign to the version manager most of the job of providing tools for programming atomic operations. Thus the implementer of a prospective atomic operation should begin that operation by invoking the version manager entry "begin–atomic–operation", and later complete the operation by invoking either "commit–operation" or "abort–operation". If in addition the atomic operation performs all reads and writes of data by invoking the version manager's "read–current–value" and "write–new–value" entries, our hope is that the result will automatically be atomic with no further concern of the implementer. (We have somewhat simplified the version manager interface by omitting calls to create and delete new objects. For the moment, our atomic operations will have to restrict their interest to already existing objects.)

How could this automatic operation work? The first step is that the version manager, when called at begin–atomic–operation, should assign a new, unique identifier to the prospective atomic operation, and create, in a cell memory, a record of this new identifier and the state of the new atomic operation. This record is called a *commit record;* it begins its existence in the state "uncommitted;" depending on the outcome it will eventually move to one of the states "committed" or "aborted," as suggested by figure 7–14. No other state transitions are possible, except to discard the commit record once there can be no further interest in its state.

When a client calls the version manager to write a new value, that client supplies the identifier of the data object, a new value for the data object, and the identifier of the atomic operation that has proposed this new, tentative data value. The version manager calls on the lower level storage management system to create a new object to contain the version; it places in that cell the new data value and the identifier of the atomic operation. Thus the version manager creates a version history as illustrated in figure 7–15.

Now, when someone proposes to read a data value by calling "read–current–value," the version manager can review the version history, starting with the latest version and, for example, return the most recent committed value. By inspecting the commit records, the version manager can ignore those versions that were written by atomic operations that aborted or that haven't yet committed. The operations "read–current–value" and "write–new–value" thus follow the algorithms of figure 7–16.

The important property of this pair of algorithms is that they make tentative changes invisible outside the atomic operation that is proposing the changes. If atomic operation number 99 proposes to change the values of nineteen different data objects, all nineteen changes will be hidden from other readers until atomic operation 99 commits. Operation 99 reveals the entire set of changes

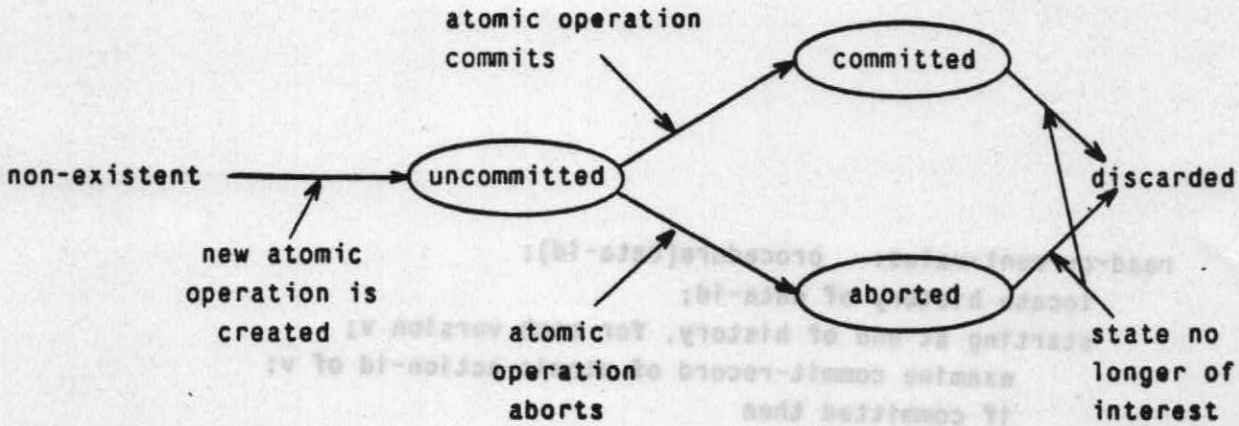


Figure 7-14. The allowed state transitions of a commit record.

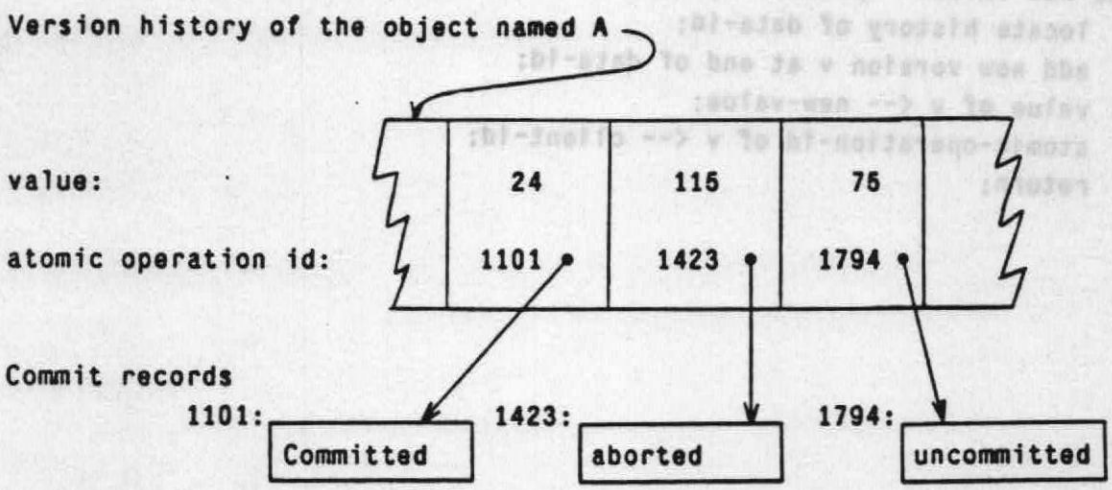
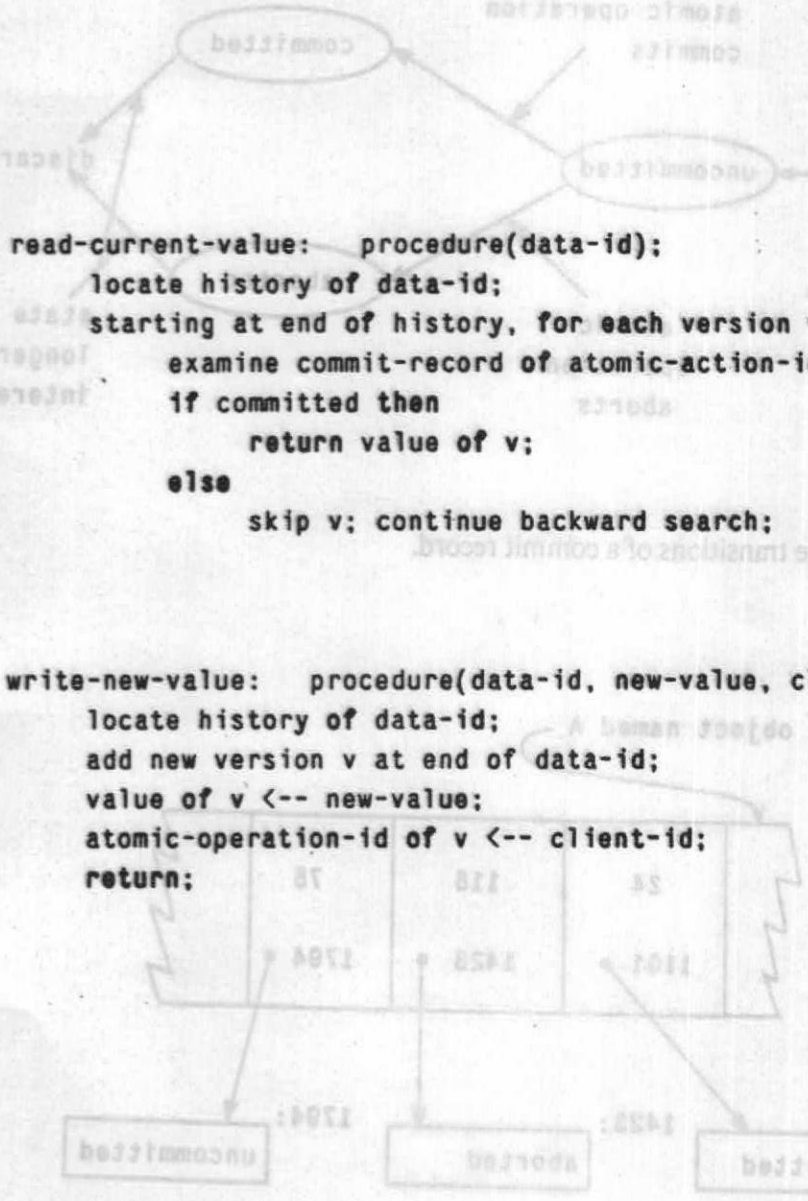


Figure 7-15. Portion of a version history, with commit records.



```
read-current-value:   procedure(data-id);
     locate history of data-id;
     starting at end of history, for each version v;
          examine commit-record of atomic-action-id of v;
          if committed then
               return value of v;
          else
               skip v; continue backward search;


write-new-value:   procedure(data-id, new-value, client-id);
     locate history of data-id;
     add new version v at end of data-id;
     value of v <-- new-value;
     atomic-operation-id of v <-- client-id;
     return;
```

Figure 7–16. Algorithms followed by read–current–value and write–new–value.

simultaneously and atomically, only at the instant that it changes its commit record from "uncommitted" to "committed". Should atomic operation 99 run into any kind of trouble, or should the system crash and forget completely that it was doing atomic operation 99, the uncommitted versions will never be observed by any other operation that reads data using our version manager.

The essence of the version history mechanism—the part that assures correctness—is what we have just examined. A practical system that implemented version histories would probably add several refinements that increase efficiency and reduce storage bulk. The most interesting of these refinements is to add to each commit record a list containing pointers to all the tentative versions that are dependent on this commit record. This list allows the operations *commit* or *abort*, after setting the commit record state, to also copy the new commit record state directly into the tentative versions (in effect replacing the commit record pointer with the commit record value.) That way, a reader of the version history can more quickly discover that the version belongs to a committed or aborted atomic operation—it can avoid an extra reference to the commit record. Even better, once all the tentative versions have been so updated, the commit record itself is no longer of any interest; it can be discarded and its storage space reclaimed. (If we do not have this refinement, the commit record may have to be kept forever, a kind of "tombstone" for the atomic operation.) In addition, if a crash interrupts the copying, the version manager continues to operate correctly; any future reader of the data may volunteer to continue the copying. In other words, updating versions with the final value of commit records and discarding the records themselves is a transparent performance improvement that has no effect on the atomicity algorithm itself.

A second refinement is to require that each prospective atomic operation predict its completion time (by an extra argument to *begin-atomic-operation*, for example). This prediction can be stored in the commit record for the atomic operation. Then, whenever anyone examines a commit record (for example, while reading a version history) the examiner can also compare the completion time prediction with the current time. If a tentative version is found linked to a commit record for an uncommitted atomic operation, and the current time has passed the predicted completion time, the examiner can assume that the original atomic operation got waylaid somewhere, and abort it. Prediction of completion time has one major flaw. Actual execution times for a given atomic operation can be extremely variable depending on system load, user responsiveness, and the state of the data. Worse, a reasonable prediction can be rendered obsolete by installation of a faster or slower processor, replacement of disk storage with a newer model or growth of a data base. When such a change of environment occurs, atomic operations that once worked fine may suddenly begin timing out; the prospect of pawing through a large system to root out and identify every timeout prediction is not a pleasant one.

A third refinement, and one that may be essential in practice, is to provide some kind of storage reclamation strategy that discards old versions. As we have defined read-current-value, versions older than the most recent committed version are inaccessible anyway and they might as well be discarded. Discarding could be accomplished either as an additional step in the program that implements the first refinement above, or as part of a separate garbage collection activity.

Again, discarding inaccessible older versions is a transparent performance improvement. (Note that for certain applications it may be very useful to keep older versions around longer and provide entry points that allow those older versions to be read by programs that need them. The banking industry abounds in requirements that make use of history information, such as reporting a consistent sum of balances in all bank accounts, paying interest on the fifteenth on balances as of the first of the month, or calculating the average balance last month.)

So far, the mechanism we have developed assures that a single operation composed of several steps will either get a chance to execute all of those steps or else have the effect of never having started the first step. That is, the operation is atomic with respect to expected errors, whether that error is detected by the operation itself (e.g., noticing an inappropriate input value) or the program interpreter decides on the basis of external events (e.g., an impending loss of electric power) to abandon the operation. Whatever the source of the error, if the error occurs before the commit point all data objects that are read or written with the read–current–value and write–new–value discipline will have values as if the operation had never been started. Conversely, any error that happens after the commit point will have no effect on the new values; at worst some performance–enhancing tune–ups will have to be finished off by later operations that notice the opportunity.

However, as we have seen earlier, the composite nature of a multistep operation can be discovered in two ways: either by a failure part way through the operation, or else by a parallel operation that happens to look in on the value of a variable in the midst of execution of the first operation. We have so far provided a systematic defense against only the first kind of discovery. Although the version history mechanism does prevent parallel operations from seeing tentative changes, that prevention by itself is sufficient only for failure atomicity; it doesn't prevent other parallel operations from reading other variables that aren't yet changed, but should be for consistency with the tentative changes that are now hidden. Making an operation atomic with respect to a parallel observer is the subject of the next section.

2. *Achieving Coordination Atomicity*

To understand how to design a mechanism to achieve coordination atomicity, we should first recall our criterion for correctness of coordination. We consider the coordination among several parallel operations to be correct *if the result is one that could have been obtained by some purely sequential application* of those same operations. Since our atomicity goal is that every atomic operation should act as though it ran either completely before or completely after each other atomic operation, correctness of coordination follows from achievement of coordination atomicity.

Our mechanism of version histories and commit records is a starting point from which we can develop several different, correct coordination strategies, each with subtly different properties. We have already discovered that we need to assign a unique identifier to each atomic operation. Suppose that we use sequential integers for these unique identifiers, and suppose further that we

impose on the program interpreter the following simple rule: atomic operation *n* may not begin its first step until atomic operation *(n-1)* has either committed or aborted. In other words, we force the operations to be purely sequential, in the order that atomic operation identifiers happen to be assigned. Since that order is one of the possible sequential orders of the various operations, by definition this approach will produce correct coordination. But it does so by being too conservative: it forbids all parallelism, and thus any speedup that could be obtained by using parallel interpreters. Nevertheless, this approach has two important values: first, in some practical applications it may actually be the best, on the basis of simplicity. Second, by looking a little more carefully at *why* it works, we can discover a whole range of less conservative approaches that allow more parallelism while at the same time preserving correctness with confidence.

The version history is the starting point for this consideration. Figure 7-17 illustrates in a single figure the version histories of four objects named A, B, C, and D, during the execution of six atomic operations, numbered with unique identifiers 1 through 6. The first atomic operation initializes all the objects to contain the value 0 (think of a banking system; the objects are bank accounts) and the following atomic operations transfer various amounts back and forth between pairs of account objects. This figure provides a straightforward interpretation of why the purely sequential discipline works correctly, and it provides clues as to how to increase parallelism. Consider atomic operation three, which must both read and write objects B and C in order to transfer funds from one to the other. One way for operation three to appear to run after operation two is that all of three's input objects have values that include the effect of operation two—if operation two committed, then any objects it changed should have new values; if operation two aborted, then any objects it tentatively changed should contain the values that they had when operation two started. Since operation three reads object B and operation two creates a new version of B, it is clear that a constraint that operation three not begin (or at least not read the value of B) until operation two either commits or aborts will produce a correct result. Looking at operation four (suppose we are just starting operation four and don't know yet that it will eventually abort) it becomes apparent just where the strictly sequential discipline is too strict. Operation four reads values only from objects A and D, yet operation three has no interest in either object. Thus the values of A and D will be the same whether or not operation three commits, and a discipline that requires that four wait for three to complete delays four unnecessarily. On the other hand, operation four does use an object that operation two modifies, so in any case four ought to wait for two to complete. (And the purely sequential discipline guarantees that it will, since four can't begin till three completes and operation three couldn't have started till operation two completed.)

These observations suggest that other, more relaxed, disciplines conceivably could be used with correct results; they also suggest that any such discipline will probably involve detailed examination of exactly which objects each atomic operation reads and writes.

By drawing the version histories as in figure 7-17 we have created one representation of what may be thought of as a system state sequence history, but it is a little hard to interpret it that way. Figure 7-18 is a redrawn version of figure 7-17 in which the system state history is more explicit. In figure 7-18, it appears that each atomic operation has perversely created a new version of every

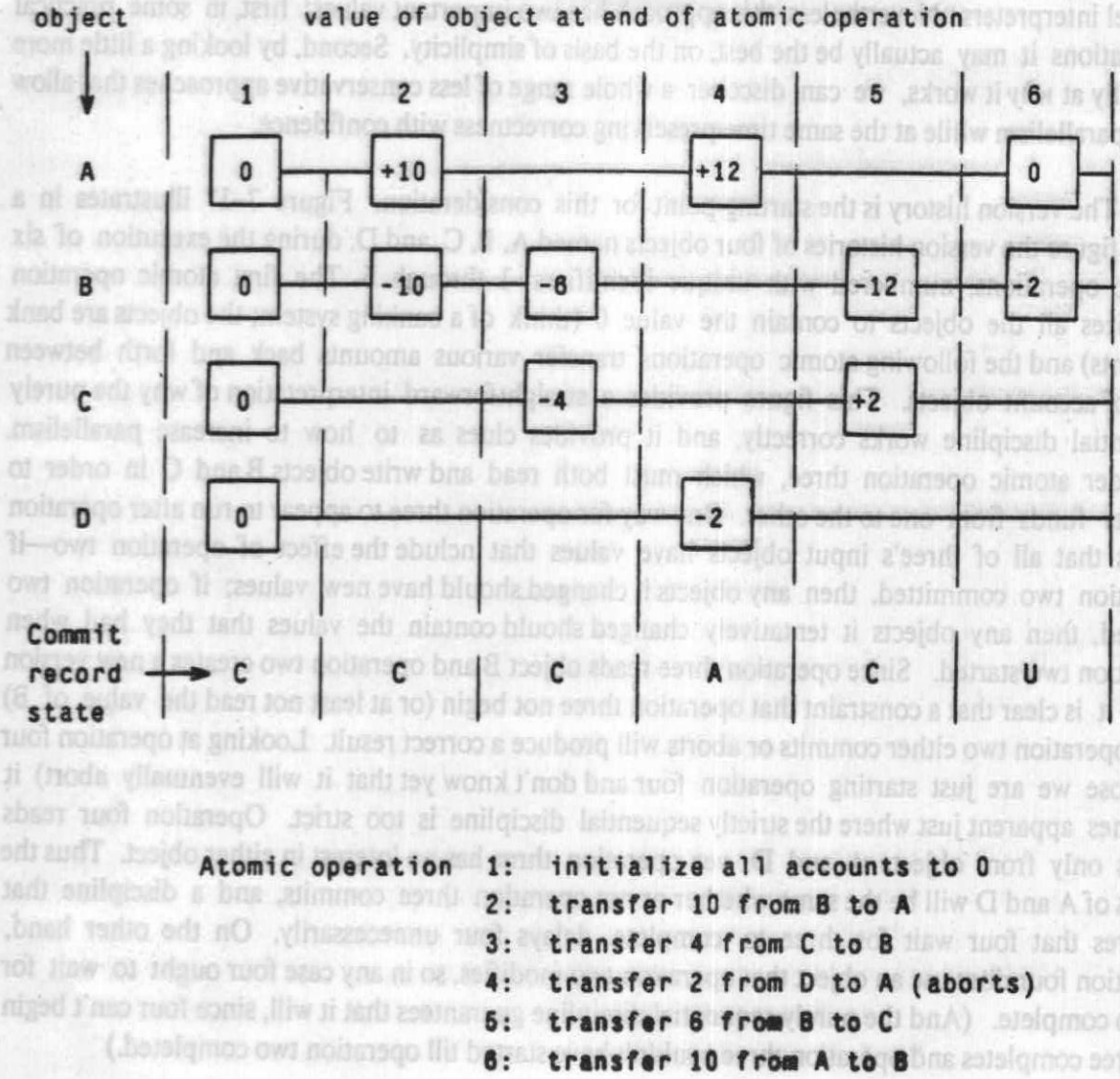


Figure 7–17. Version history of a banking system.

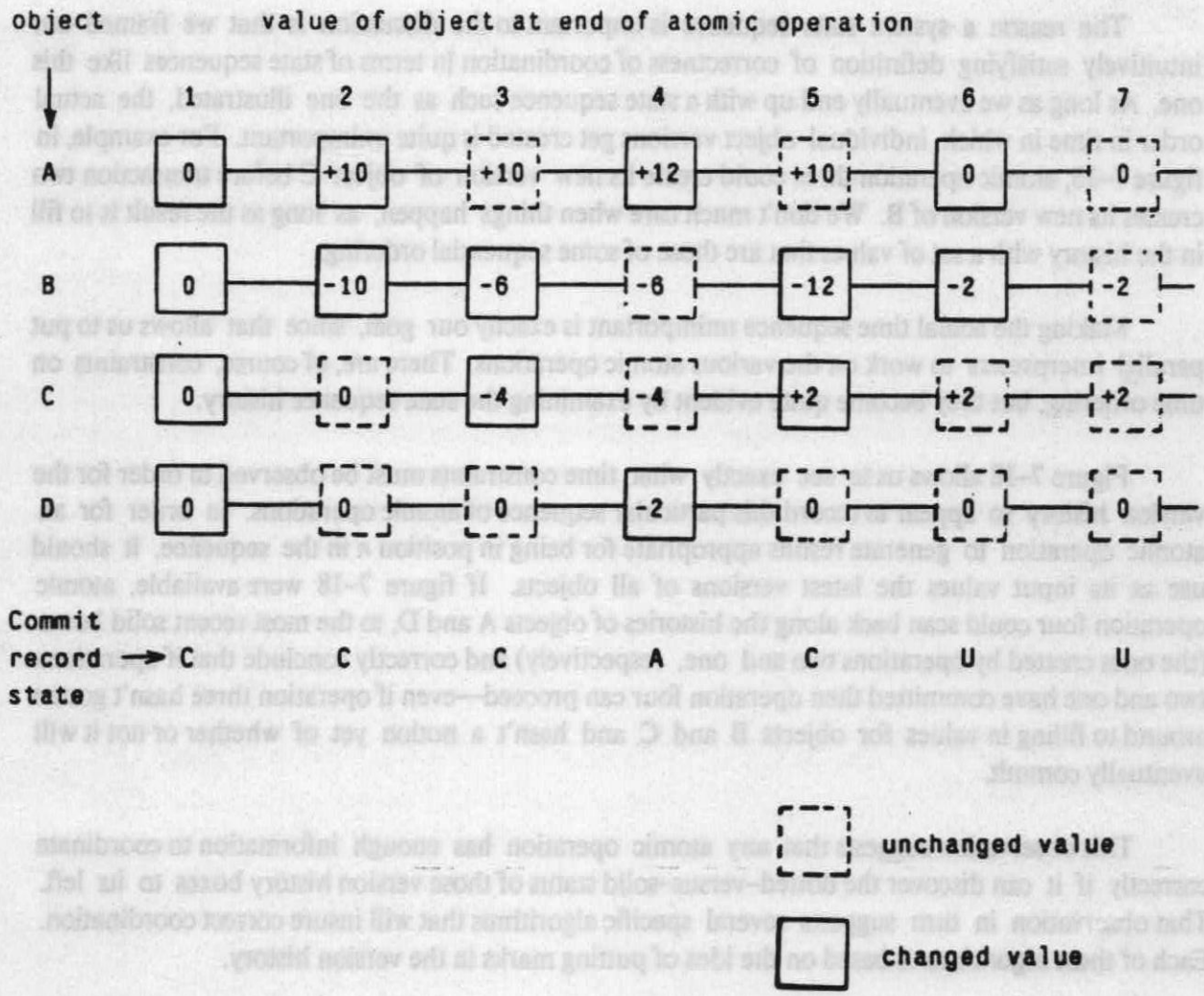


Figure 7–18. Version history with unchanged values filled in.

object in sight, with unchanged values in dotted boxes for those objects it wasn't really interested in. Although it would be silly to implement a system this way, the resulting conceptual picture of sequential system state is quite useful. For example, it emphasizes that the vertical slot for, say, atomic operation three is in effect a reservation in the state sequence for every object in the system; atomic operation three has an opportunity to propose a new value for every object if it wishes, but only at this point in the state sequence.

The reason a system state sequence is important to the discussion is that we framed our intuitively satisfying definition of correctness of coordination in terms of state sequences like this one. As long as we eventually end up with a state sequence such as the one illustrated, the actual order in time in which individual object versions get created is quite unimportant. For example, in figure 7–18, atomic operation three could create its new version of object C before transaction two creates its new version of B. We don't much care when things happen, as long as the result is to fill in the history with a set of values that are those of some sequential ordering.

Making the actual time sequence unimportant is exactly our goal, since that allows us to put parallel interpreters to work on the various atomic operations. There are, of course, constraints on time ordering, but they become quite evident by examining the state sequence history.

Figure 7–18 allows us to see exactly what time constraints must be observed in order for the version history to appear to record this particular sequence of atomic operations. In order for an atomic operation to generate results appropriate for being in position *n* in the sequence, it should use as its input values the latest versions of all objects. If figure 7–18 were available, atomic operation four could scan back along the histories of objects A and D, to the most recent solid boxes (the ones created by operations two and one, respectively) and correctly conclude that if operations two and one have committed then operation four can proceed—even if operation three hasn't gotten around to filling in values for objects B and C and hasn't a notion yet of whether or not it will eventually commit.

This observation suggests that any atomic operation has enough information to coordinate correctly if it can discover the dotted–versus–solid status of those version history boxes to its left. That observation in turn suggests several specific algorithms that will insure correct coordination. Each of these algorithms is based on the idea of putting marks in the version history.

3. The mark–point–sequential algorithm

The first algorithm we may call the mark–point–sequential algorithm. In this algorithm, the first step of every atomic operation is to *mark* every version it intends to write. (This algorithm will be useful only if the atomic operation can predict which data will be updated. If it is necessary to read one data object to discover the identity of another that requires update, a more complex algorithm may be required.) In terms of figure 7–18, the boxes under newly arrived atomic operation seven are all dotted; its first step is to mark the ones that it plans to make solid. This

marking could be done in practice quite simply by actually creating a new physical version now with the plan of supplying a data value later. For this purpose, we might split the write algorithm of figure 7–16 into two parts, create–new–version and write–new–value, as in figure 7–19. When finished marking, the operation is expected to announce that it has reached its *mark point.* It may then go about its business, reading and writing values as appropriate to its purpose. We must also modify read–current–value to wait for uncommitted values to become committed, rather than to skip them. Figure 7–19 shows this modification, too. There are two subtleties in the modification to read–current–value. First, if the atomic action marks some object history by creating a tentative version, and later tries to read the current value of that same object, we don't want a simple–minded read algorithm to wait for this tentative version to commit. Therefore, we must ask the atomic operation to provide its own identifier as an argument to read–current–value, and ask read–current–value to skip over any tentative versions created by its current client.

Second, because a later–arriving atomic operation can now run on ahead, create new versions, and commit, we want to be sure that this atomic operation doesn't accidentally read or wait for one of those later values. Thus read–current–value should also skip over any versions created by later atomic operations. Both of these subtleties are handled by one test in read–current–value in figure 7–19:

```
if atomic-operation-id of v ≥ client-id then skip v;
```

This test has one side–effect that could surprise some application programs. If an atomic operation makes an update and then, before committing, tries to read the value of the object it updated, it will get the old, previously committed value rather than its own, newly proposed value. If one wanted an atomic operation to be able to read its own updates, a slightly more complex algorithm would be required.

Finally, we make a rule, perhaps enforced by putting a test and a possible wait in begin–atomic–operation, that no atomic operation may begin until the preceding atomic operation reports that it has reached mark point.

It is this last rule that guarantees coordination correctness, and that also gives this algorithm its name. Because no atomic operation can start until the previous atomic operation has reached its mark point, all atomic operations earlier in the sequential ordering must also have passed their mark points, so every atomic operation earlier in the sequential ordering has already created all of the versions that it ever will. Since read–current–value now insists on waiting for preceding, uncommitted values to become committed or aborted, it will always return to its client a value that represents the final outcome of all preceding atomic operations. All input values to an atomic operation thus contain the committed result of all atomic operations that appear earlier in the sequential ordering, so the result of that atomic operation is now guaranteed to be the result that it would have produced if it had followed the purely sequential discipline. The result is identical to that produced by a sequential ordering, no matter what real order the various atomic operations end up writing actual data values into their version slots.

```
read-current-value:  procedure(data-id, value, client-id);
    locate history of data-id;
    starting at end of history, for each version v;
        if atomic-operation-id of v ≥ client-id then skip v;
        examine commit-record of atomic-operation-id of v;
        if uncommitted then
            wait for atomic-operation-id of v
                    to commit or abort;
        if committed then
            return value of v;
        else
            skip v; continue backward search;

create-new-version:  procedure(data-id, client-id);
    locate history of data-id;
    add new version v at end of data-id;
    value of v <-- null;
    atomic-operation-id of v <-- client-id;
    return;

write-new-value:  procedure(data-id, new-value, client-id);
    locate history of data-id;
    locate this client's version v of history of data-id
        (if not found, report coordination error)
    value of v <-- new-value;
    return;
```

Figure 7-19. Mark-point-sequential versions of read-current-value, create-new-version, and write-new-value.

We note in passing that the delays of the purely sequential discipline (which were all concentrated in begin-atomic-operation) are now distributed. Some delays may still occur in begin-atomic-operation, waiting for the preceding atomic operation to complete its mark phase; one expects that some atomic operations will complete their mark phase early, and thus that delay should be not as great as waiting for them to commit or abort. Other delays may occur at any read step but only in cases where real interference would occur; in return for an opportunity to run non-interfering atomic operations in parallel we have sacrificed some predictability in the details of progress rate, although the overall delay for any given operation should never be more than that imposed by the purely sequential discipline.

One important property of the waits in the mark-point-sequential algorithm should be noted: whenever a wait occurs it is a wait for some atomic operation *earlier* in the ordering. That atomic operation in turn may be waiting for a still earlier operation, but since no one ever waits for an operation *later* in the ordering, progress is guaranteed. At all times there must be some "earliest uncompleted atomic operation". Thanks to the mark-point-sequential ordering property, that earliest atomic operation will encounter no waits for coordination, so it, at least, can make progress. When it completes, some other operation in the ordering takes over the mantle as "earliest", and it now can be certain to make progress. Eventually, by this argument, every atomic operation will find that its wait is complete. This kind of reasoning about progress is an important element of any proposed coordination algorithm and sometimes it is more difficult to make a convincing argument, as we shall see when we analyze the read-capture algorithm in the next section. It is, unfortunately, quite easy to invent algorithms that can be shown to be "correct" but that do not guarantee progress: such algorithms would prove to be defective in practice because parallel operations might end up waiting for one another, forever.

Two other minor points should be noted. First, if atomic operations wait to announce their mark-point until they commit or abort, the mark-point-sequential algorithm reduces to the purely sequential algorithm. That observation confirms that one algorithm is a relaxed version of the other. Second, there are some opportunities in the mark-point-sequential algorithm to discover and report coordination errors to clients. For example, an atomic operation should not be allowed to call create-new-version after announcing the mark point; similarly, write-new-value should report an error if the client operation calls trying to write a value for which a new version was never created.

The mark-point-sequential algorithm gives an atomic operation complete freedom to mark any object in the system for update, at any time preceding that operation's mark-point announcement; the atomic operation may even delay the mark-point announcement until it is ready to commit. This flexibility and freedom for an individual atomic operation is only a convenience, and it could have the side effect of delaying other operations more than really needed.

4. The read–capture algorithm

We can increase parallelism by picking up on an idea suggested in the introduction; if interference between parallel atomic operations is relatively rare anyway, allow them to proceed in parallel, watch for actual interference, and force operations that do interfere to abort and start over again. In terms of our system state history, we can allow atomic operations to fill in values in any order and at any time, but with the risk that some attempts to write may be met with the response "Sorry, that write would interfere with another atomic operation. You must abort, abandon this slot in the system state history, obtain a new slot and try your atomic operation from the beginning again."

A specific example of this approach is the "read–capture" algorithm. Under the read–capture discipline, no advance marking is required. Eliminating advance marking has the advantage that an atomic operation does not need to predict the identity of every object it will update—it can discover the identity of those objects as it works. Instead of advance marking, whenever an atomic operation *reads* the value of an object it makes a mark at its own position in the version history for the object it read. This mark tells potential version–inserters earlier in the sequential ordering but arriving later in real time that they are no longer allowed to insert—they *must* abort and try again, using a later slot in the version history. Had the prospective version–inserter gotten there sooner, before the reader had left its mark, the new version would have been acceptable, and the reader would have happily waited for the version–inserter to commit, and taken that new value instead of the earlier one. In effect, we are giving a reader the power of extending validity of a version from the sequential position in which it was created, through intervening atomic operations, and up to the position of the reader. This view of the situation is illustrated in figure 7–20, which contains the same version history as did figure 7–17.

The key property of read–capture is illustrated in figure 7–20. Atomic operation four was late in creating a new version of object A; by the time it tried to do the insertion, atomic operation six had already read the old value (+10) and thereby extended the validity of that old value to the end of atomic operation five. Therefore, atomic operation four had to be aborted; it has been reincarnated to get another try as atomic operation seven. In its new position as atomic operation seven, its first act is to read object D, extending the validity of its most recent committed value (zero) to the end of atomic operation six. When it tries to read object A, it discovers that the most recent version is still uncommitted, so it must wait for operation six to either commit or abort. Note that if operation six should now decide to create a new version of object C, it can do so without any objection, but if it should try to create a new version of object D, it would run into a conflict with the old, now extended version of D, and would have to abort.

Read–capture is relatively easy to implement in a version history system. We start by adding a new step to read–current–value, a step that records with each data object the identifier of the highest–numbered atomic operation that has ever read a value from this object's version history. That identifier is to serve as a warning to other atomic operations that started some time ago (and thus have early positions in the sequential ordering) but are late in creating new versions. The

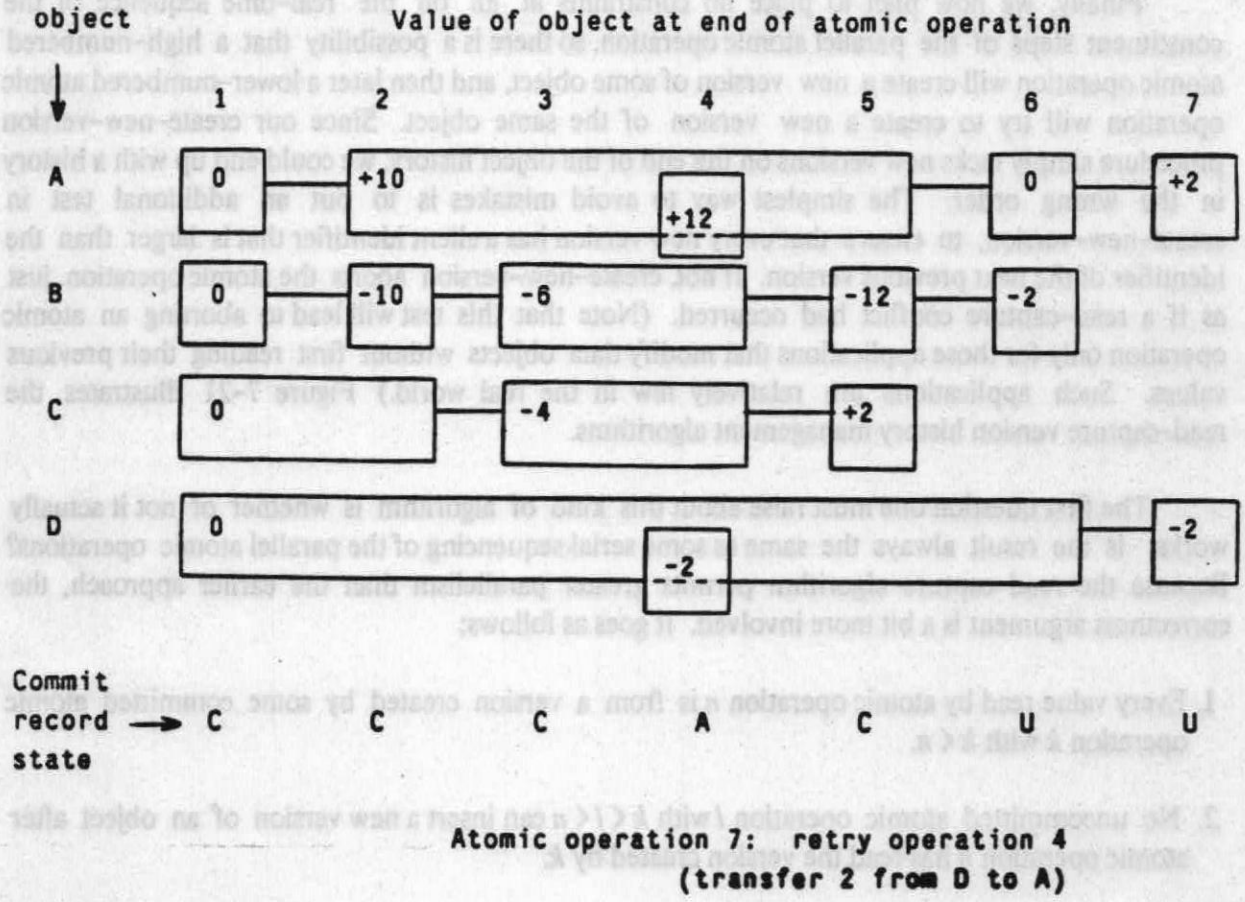


Figure 7–20. Version history with read–capture discipline in operation.

warning is that someone later in the ordering has already read a version from earlier in the ordering, so it is too late to put a new version in now. We can implement an observation of this warning by adding a step to create-new-version. This new step checks the read-mark for the object to be written, to see if any atomic operation with a higher sequence number has already read the latest version of the object. If not, we can create a new version without concern. But if the atomic operation identifier in the read-mark is greater than this atomic operation's own identifier, this operation must abort, obtain a new (higher-numbered) identifier, and start over again.

Finally, we now plan to place no constraints at all on the real-time sequence of the constituent steps of the parallel atomic operation, so there is a possibility that a high-numbered atomic operation will create a new version of some object, and then later a lower-numbered atomic operation will try to create a new version of the same object. Since our create-new-version procedure simply tacks new versions on the end of the object history, we could end up with a history in the wrong order. The simplest way to avoid mistakes is to put an additional test in create-new-version, to ensure that every new version has a client identifier that is larger than the identifier of the next previous version. If not, create-new-version aborts the atomic operation, just as if a read-capture conflict had occurred. (Note that this test will lead to aborting an atomic operation only for those applications that modify data objects without first reading their previous values. Such applications are relatively few in the real world.) Figure 7-21 illustrates the read-capture version history management algorithms.

The first question one must raise about this kind of algorithm is whether or not it actually works: is the result always the same as some serial sequencing of the parallel atomic operations? Because the read-capture algorithm permits greater parallelism than the earlier approach, the correctness argument is a bit more involved. It goes as follows:

1. Every value read by atomic operation n is from a version created by some committed atomic operation k with $k < n$.

2. No uncommitted atomic operation l with $k < l < n$ can insert a new version of an object after atomic operation n has read the version created by k.

3. Therefore, every input to atomic operation n, no matter when n actually reads it, has the value it will have when atomic operation $(n - 1)$ completes (commits or aborts).

4. Therefore, every atomic operation n will act as if it sequentially follows operation $(n - 1)$.

This argument depends for its soundness on the correctness of its first two premises, which are claimed to be properties of the algorithms of figure 7-21. It is quite instructive to develop a line of reasoning that these two premises are true, based on the individual steps of these algorithms.

Read-capture, as we have described it here, has one slightly annoying property. Once an atomic operation extends the validity of an old version of an object by read-capture, that extension

```
read-current-value:  procedure(data-id, value, client-id);
    locate history of data-id;
    starting at end of history, for each version v;
        if atomic-operation-id of v ≥ client-id then skip v;
        examine commit-record of atomic-operation-id of v;
        if uncommitted then
            wait for atomic-operation-id of v to commit or abort;
        if committed then
            read-mark of v <-- max(read-mark of v, client-id);
            return value of v;
        else
            skip v; continue backward search;

create-new-version:  procedure(data-id, client-id);
    locate history of data-id;
    if (client-id < read-mark of data-id)
    or (client-id < (atomic-operation-id of latest
                        version of data-id))
    then abort this atomic operation and exit;
    add new version v at end of data-id;
    value of v <-- 0;
    atomic-operation-id of v <-- client id;
    return;

write-new-value:  procedure(data-id, value, client-id);
    locate history of data-id;
    locate this client's version of history of data-id;
        (if not found, report coordination error)
    value of v <-- new-value;
    return;
```

Figure 7–21. Read–capture form of read–current value, create–new–version, and write–new–value.

is permanent, even if the atomic operation that did the read-capture should abort an instant later. Thus it is possible that other atomic operations that want to insert new values will abort unnecessarily. On reflection, this annoyance is probably much less serious than one might first guess. The reason is that highly parallel operations are appropriate only if they do not interfere with each other very often. If one is going to the trouble of implementing read-capture (rather than, say, a mark-point-sequential algorithm) it is probably the case the application is a good candidate for more parallelism, which means that data interference is not very common anyway. Thus even though aborted operations leave behind unnecessary version-validity extensions, one might expect that the other atomic operations that were in progress at the time will notice those extensions and abort unnecessarily so infrequently that the performance impact is inconsequential.

5. *Making progress*

There is one loose end in the argument that claims read-capture will work. The argument guarantees that any atomic operations that complete will produce results as though they were run sequentially, but it says nothing about whether or not any atomic operation will ever commit. The reason for wondering about this detail is that the algorithm itself introduces an extra reason to abort an atomic operation; one might envision a pathological situation where the algorithm aborts every atomic operation, no matter how many times retried. In other words, this coordination algorithm guarantees *correctness*, in the sense that no wrong answers will come out of the system, but it does not guarantee *progress*.

Consider, for example, two atomic operations named Alphonse and Gaston. Alphonse reads object apple, computes for a long time, and then updates object banana. Atomic operation Gaston first reads object banana, calculates for a long time, and then updates object apple.

Suppose Alphonse starts first, becoming atomic operation n. When Gaston arrives, it becomes atomic operation $(n+1)$. Gaston immediately reads banana, extending the validity of its most recent version past the end of atomic operation n, namely Alphonse. Thus when Alphonse finally gets to the point to trying to update banana it will have to abort. Alphonse, after aborting, tries again, becoming atomic operation $(n+2)$, and immediately begins reading apple. But reading apple extends the validity of its most recent version part the end of atomic operation $(n+1)$, namely Gaston. So when Gaston finally gets to the point of trying to update apple, it will have to abort. Gaston, upon aborting, tries again, becoming atomic operation $(n+3)$, and the cycle repeats.

Our implementation offers a small amount of help for such a situation, but it does not solve the problem. If the programmer of either Alphonse or Gaston knows that some object will be updated near the end of the operation, the programmer could create a new but empty version of that object at the outset, planning to fill in the value later. Thus, for example, Alphonse might read object apple and immediately create a new version of object banana. When Gaston arrives and tries to read banana, the presence of an uncommitted version for banana means that Gaston will wait for Alphonse to commit. There is still a window (but now smaller in time) between Alphonse's

materialization and Alphonse's creating the version of banana, during which Gaston could materialize and extend the old version of banana. One can fight this problem by a technique called *random back-off:* before retrying, an operation waits a random length of time (longer than the window mentioned above,) in hope that such a random delay will change the relative timing of the two operations enough that one will succeed and the other can proceed. Since on each retry there is some probability of success, one can push this probability as close to unity as desired by continued retries. A nice property of random back-off is that if repeated retries fail it is almost certainly an indication of some other problem—perhaps a programming mistake. The one objection to random back-off is similar to the objection to placing timeouts in commit records: someone must choose a time range for the random delay. Once the parameters of the time range are chosen and frozen into code, changes in the environment such as installation of a different, faster, disk drive, may make the time range inappropriate, in which case the code must be tracked down and changed.

Alternatively, a rather heavy-handed discipline can guarantee progress: suppose the programmers of Alphonse, Gaston, and every other potentially interfering operation follow these two rules:

1. For every object that an atomic operation either reads or updates it must both read the object (thereby assuring read-capture of objects that will only be modified) and it must create a new version, perhaps repeating the old value (thereby assuring that users of this object later in sequence must wait for this operation to commit.)
2. Every atomic operation must order its object accesses alphabetically, by object name.

The alphabetic order of object access guarantees that atomic operations wait or abort only when accessing objects with names further down the alphabet than the objects they have already captured. That guarantee, in turn, means that at least one atomic operation will not encounter any waits or algorithm-triggered aborts: that operation can make progress, and when it finishes some other atomic operation will take over its role.

The design of algorithms or programming disciplines that guarantee progress is a research topic with only modest payoff. In practice, systems that would have frequent interference among atomic operations are not usually designed with a high degree of parallelism anyway. When interference is not frequent, techniques such as random back-off not only work well, but they are usually provided anyway, as a technique to cope with any timing-dependent errors that may have crept into the system design or implementation. Thus a heavy-handed progress-guaranteeing discipline may be costly, and redundant, and only rarely will it get a chance to promote progress.

E. Logs and Locks

1. Logs

Our discussion of systematic techniques that achieve atomicity has used a relatively abstract model, version histories, that is helpful for understanding principles, but not often implemented in practice. Direct implementations are rare because of concern for performance: the time required to locate current versions looks like a possible bottleneck. Probably the most common technique in widespread use achieves high performance by storing data twice: once in storage cells for fast access, and a second time in an append–only journal of all changes to those cells, for failure atomicity. The append–only journal is called a *log* and it resembles a version history in some ways. The set of storage cells containing the data is sometimes best viewed as a speed–up cache. This cache consists of an easy–to–access copy of the data values that are also recorded, but hard to find, in the log. By maintaining both a cache and a log, the data in the cache can be organized for optimum access speed, while the data in the log can be organized to guarantee atomicity of data update operations in the face of failure. Of course, having two copies of every data value around complicates the algorithms for update and recovery.

The basic element of a log is the *log entry*. Whenever an atomic operation updates a data value in cell storage it also appends to the end of the log a new entry containing the identity of the atomic operation that is making the change, the identity of the data being changed, and either the old data value or the new data value or both, depending on the purpose of the particular log. Figure 7–22 illustrates. Thus a log is an append–only storage structure with a strong resemblance to a version history, or more accurately, a set of interleaved version histories, since one log records the changes to all the data of the system that is to be subject to atomicity constraints. Usually, one follows a protocol, known as the *write–ahead–log* protocol, that requires that a log entry be made *before* changing the cell storage version of the data.

A log can be used for several quite distinct purposes; this range of purposes is sometimes confused in real–world designs and implementations.

1. If the log is implemented using a storage medium (say magnetic tape) that fails in ways and at times that are independent from the failures of cell storage (which might be magnetic disk) then the data copies in the log can be considered as backup copies in case the copies of the data in cell storage are accidentally damaged. In this case, the log helps implement *stable storage*.

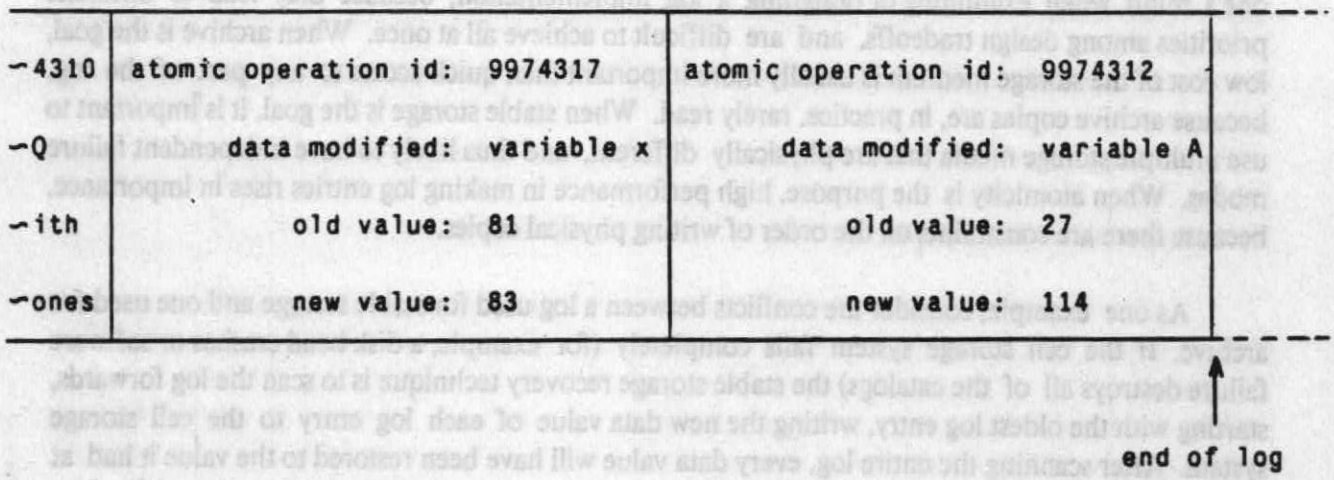


Figure 7-22. The end of a log and the last two log entries.

2. If the log is kept forever, it is a place where old values of data can be found, for purposes of historical study, auditing, or recovery from application-level mistakes (e.g., a clerk incorrectly deleted an account.) This function is usually known by the term *archive*.

3. If, in addition to logging data updates, updates to commit record values are also logged, then the log can be used to determine the outcome of atomic operations. With that outcome information, one can undo and back out the effects of atomic operations that abort or redo atomic operations that commit. Thus a log may be used as a mechanism to achieve *atomicity*.

It is essential to have these three purposes—stable storage, archive, and atomicity—distinct in one's mind when examining or designing a log implementation, because they lead to different priorities among design tradeoffs, and are difficult to achieve all at once. When archive is the goal, low cost of the storage medium is usually more important than quick access to any part of the log, because archive copies are, in practice, rarely read. When stable storage is the goal, it is important to use multiple storage media that are physically different, and thus likely to have independent failure modes. When atomicity is the purpose, high performance in making log entries rises in importance, because there are constraints on the order of writing physical copies.

As one example, consider the conflicts between a log used for stable storage and one used for archive. If the cell storage system fails completely (for example, a disk head crashes or software failure destroys all of the catalogs) the stable storage recovery technique is to scan the log forwards, starting with the oldest log entry, writing the new data value of each log entry to the cell storage system. After scanning the entire log, every data value will have been restored to the value it had at the instant of the failure, and normal operation can continue. If the system has been operating for a long time, the accumulation of log entries could be very large (and only a few entries in the older parts of the log may actually contain current values—but the only way to find them is by a complete log scan). To shorten the log scan, a stable storage implementation will usually include a periodic *snapshot* of the entire collection of data in cell storage. This snapshot represents a starting place for recovery. If the cell data is lost in a system crash, the latest snapshot can be reloaded into cell storage, and then the log entries made since that snapshot can be scanned to bring the data that was in the snapshot up to date.

Once the significance of the snapshot is understood, it is apparent that a stable storage implementation requires retaining no more than the previous snapshot plus the log entries since that snapshot. (In practice there might be multiple copies of both the log and the snapshot, or else one might keep the two most recent snapshots, on the chance that one copy may be defective.) Since the storage required can be bounded and is in the same order of magnitude as the amount of cell storage, one might consider using, say, demountable disks or even nondemountable disks on a companion system rather than magnetic tape for both the log and the snapshot medium.

On the other hand, if the log is also used for archive, one would plan never to discard old log entries and therefore would be more likely to prefer lower-cost magnetic tape, despite its possibly lower reliability, slower data rate, and extra handling effort. This conflict between the archive

function and the stable storage function is often found in the user file backup facilities of time–sharing systems.

When one contemplates adding atomicity guarantees to the list of functions to be provided by a log, it is usually apparent that the number of conflicting goals is so great that it is appropriate to use a separate log for the atomicity purpose.

Atomicity in the face of system crashes is relatively straightforward to accomplish with a log, but it involves one new idea not found in version histories: we must arrange that an explicit system recovery procedure be executed following every crash. The log approach takes the position that since failures are relatively infrequent, one should bias the design in the direction of doing extra work at the time of a failure, explicitly putting things back in order, if in turn one can reduce the work done during normal operations. As compared with version histories, there are two places where normal operations might be quicker: first, reading of cell storage does not require pawing through a history record to discover the right version. Second, since there is only one log, rather than one version history per variable, it is easy to find the log.

There are several possible rules, or protocols, for atomicity logs, that vary in the order in which things get done and in the nature of information logged. One simple procedure is the *back–out list* protocol, illustrated in the log example of figure 7–23. That log contains two kinds of entries for this purpose:

1. Whenever an atomic operation changes a data value, it first makes a log entry containing the identity of the atomic operation, the identity of the data, and the *old* value of the data. (Logging *old* values is a characteristic of a back–out–list log).

2. When an atomic operation commits or aborts, it logs its completion. This log entry becomes the permanent record of the outcome of the atomic operation. The instant that this log entry is written is the commit point of the atomic operation.

If a crash occurs, we assume that the atomic operations that were in progress at the time vanish, that the cell storage is completely intact (perhaps it was recovered with the help of a separate stability log) and that before allowing any new atomic operations to begin, the system will be sure to run an atomicity recovery procedure. This atomicity recovery procedure proceeds by scanning the atomicity log *backward* from the latest entry. A backward log scan, sometimes called a LIFO (for last in, first out) log review, is another characteristic that distinguishes a back–out–list atomicity log. As the recovery procedure scans backward, it makes two lists of atomic operations: those that committed or aborted before the crash (the *winners*) and those that did not commit by crash time (the *losers*). Losers are discovered by coming across a log entry for a data change that contains their identity, but without having encountered a commit or abort so far during the log review. Also as it scans backwards, whenever it encounters a log entry for a data value that was changed by a loser, it writes the old data value found in the log entry back into the on–line storage system, thereby undoing the logged operation. Thus in the course of scanning the log backwards, the atomicity

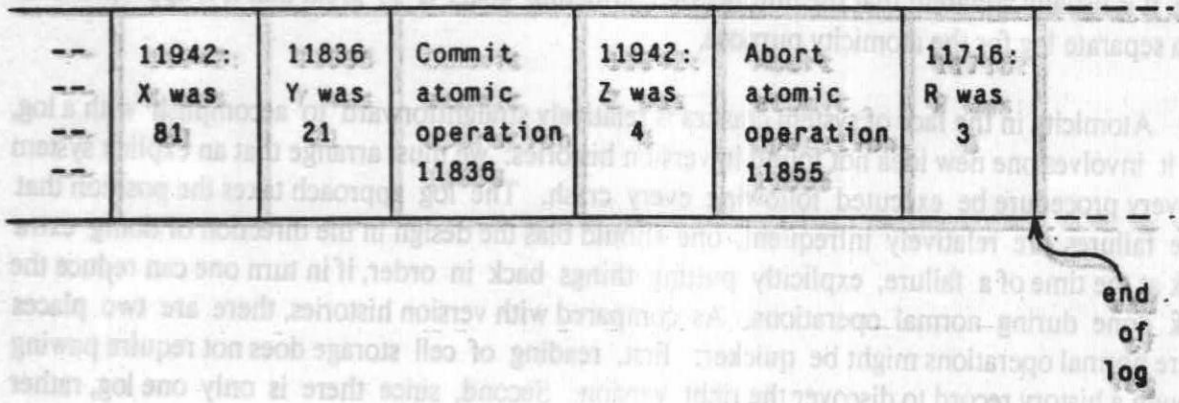


Figure 7-23. An atomicity log.

```
recovery:  winners <-- null;
           losers  <-- null;
           start at end of log;
           repeat till beginning of log;
            [get previous record;
             case data
                  if id of record in set winners ignore record; break;
           undo:  data = old data of record
                  if id of record not in set losers then
                                        add id of record to set losers;
             case commit or case abort
                  add id of record to set winners;
             end case]
```

Figure 7-24. An algorithm for FIFO log review, to achieve failure atomicity.

recovery procedure will undo all of the completed steps of all partially completed atomic operations (the losers) effectively backing them out as though they had not run at all. When the backwards log scan is complete, the state of the on–line storage can be said to be atomic–operation consistent: it is as though every atomic operation that committed before the crash had run to completion, while every atomic operation that was incomplete at crash time had never existed. A program implementing this algorithm is illustrated in figure 7–24. One apparent oddity of the recovery algorithm is that it considers atomic operations that abort to be winners. This classification does not mean that aborted atomic operations become committed ones, only that already–aborted atomic operations do not require any further recovery if a crash occurs.

This recovery procedure assumes the following corollary of the write–ahead log protocol: an atomic operation must update all cell storage values *before* writing the log record that says it has committed. Similarly, an atomic operation that aborts must undo its cell storage updates *before* writing the log record that says it has aborted. If this corollary protocol is followed, then after a crash the recovery procedure can safely assume that all atomic operations that are logged as committed or aborted had a chance to complete their cell storage updates before the crash.

With an atomicity log, just as with the stability log, one wonders if it is really necessary to keep the entire history of the system and scan the entire log following every crash. If we require that atomic operations announce their commencement by writing a *begin* record to the log, then at recovery time. it is necessary to scan back only as far as the begin record of the oldest incomplete atomic operation. The problem is to discover that oldest incomplete atomic operation. A common trick to shorten atomicity log reviews is to have the system periodically write a *checkpoint* record in the log. A checkpoint record is simply a list of the identifiers of all currently incomplete atomic operations. If checkpoint records are written in the log, the recovery algorithm can, upon encountering the first such record during its LIFO log review, immediately complete its list of losers by adding to that list any atomic operations appearing in the checkpoint record that it has not already included in either its winners or losers list. Now that the list of losers is known to be complete, the FIFO log review continues, but it can stop as soon as the begin record of every atomic operation in the losers list has been found. At the instant recovery is complete, there are no outstanding atomic operations, so that is an especially good time to write an (empty) system checkpoint record. An empty system checkpoint record is good news to any future recovery procedure: it guarantees that there is nothing older of interest in the log—in fact, as far as atomicity is concerned. any part of a log older than an empty system checkpoint record can be safely discarded or recycled. Since its length is limited. an atomicity log might well be kept on–line, in the cell storage medium. This possibility again emphasizes the differences in design priority among logs with different functions.

Notice that a critical design property of both the log review algorithm of figure 7–24 and the quicker algorithm just described must be *idempotence*. That is, if there should be a second system crash during the recovery, one can simply start the recovery from the beginning again; any number of crash–restart cycles might occur without compromising the atomicity properties (though the system operator may start to lose patience.) Proposals to simplify or speed up log or recovery

operations must be reviewed carefully against the idempotence requirement.

Another set of observations apply when an individual atomic operation aborts. Because the atomic operation may have made changes to cell storage, simply writing an abort record in the log is not enough to make it appear to later observers that the atomic operation never did anything. An explicit *undo* procedure must therefore be programmed as part of every atomic operation in a system that uses logs. This *undo* procedure must make sure that all storage values that were modified by the atomic operation are restored to their old values before the atomic operation writes an abort record in the log. One simple strategy is to call a library *undo* procedure that scans the atomicity log backwards looking for entries left by this atomic operation; when it finds one, *undo* takes the old data value found in the log record and writes that value back into cell storage. The extra work required to accomplish abort when logs and cell storage are used, (as compared, say, with a version history system) is another part of the engineering tradeoff: one anticipates that most atomic operations will commit and aborted atomic operations should be relatively uncommon. The extra effort of an occasional *undo* operation will (one hopes) be more than paid back by the more frequent gains in performance on updates, reads, and commits.

The back–out–list protocol that we have described is one of several equally acceptable logging protocols that vary in the constraints they place on order of disk update. One can instead implement the *intentions–list* protocol. Under this protocol, an atomic operation does not make any updates to cell storage until it logs a complete record (including the commit record) of every planned update. This record is called the intentions list. Each log entry contains the *new* data value. After completing its intentions list, the atomic operation then updates cell storage, and if it completes that set of updates it puts a final note in the log to that effect. When a crash occurs, the recovery procedure looks through the log for intentions lists that contain a commit record but are not followed by completion notes. It then redoes the updates in those intentions lists. Partially completed intentions lists can be ignored, because the atomic operation never got as far as updating the cell disk storage. One can also invent protocols that combine features of both the back–out–list and the intentions–list protocols. The primary requirement is that the constraints on order of writing the log entries and cell storage allow execution of an idempotent recovery algorithm—one that can be rerun following a failure during the recovery algorithm itself.

Finally, our log recovery scheme provides atomicity in the face of system failures that cause atomic operations to vanish in mid–stream, but it provides no coordination atomicity at all. Worse, since all updates to data are written in cell storage where they can be immediately viewed by other, parallel operations, it appears that coordination atomicity will be relatively hard to achieve. That observation turns out to be true; when logs are the mechanism used for failure atomicity, a completely separate mechanism must be introduced to achieve coordination atomicity: *locks*. Locks are the subject of the next section of this chapter.

2. Locks

Since logs provide only failure atomicity, some additional mechanism is needed to accomplish coordination atomicity. The conventional approach to coordination is called *locking*. A lock is just a mark associated with a data object to warn other, parallel, atomic operations not to read or write the object. Conventionally, two system operations are available:

```
seize(A);
```

marks data item A as being locked by this atomic operation, or if it is already locked causes a wait until A becomes free.

```
release(A);
```

unlocks data item A, perhaps ending some other atomic operation's wait for A.

There are many variations on implementation mechanisms for locks, but all variations have the goal that if two or more atomic operations attempt to seize a lock at about the same time, only one shall succeed; the others must find the lock already seized. This implementation is usually accomplished by primary memory access hardware that reads a memory variable, tests it, then sets a new value, with no possibility of an interrupt intervening or another operation doing the same thing at the same time. In effect, the general problem of coordinating unconstrained atomic operations is reduced to the specific, very constrained problem of coordinating access to the lock variable. (Note the parallel with commit records, which were another example of a constrained, solvable version of a general, hard-to-solve problem.)

If locks are available, it is relatively easy to provide general coordination of any atomic operations; there are many possible locking protocols (not all of which work). The simplest rule that works we call *simple locking*: each atomic operation must seize the lock on every data object it intends to read or write before doing any actual reading and writing: then, only after the last update is complete, and the operation commits (or the data is completely restored and the atomic operation aborts) may it begin releasing its locks. We can say that the atomic operation has a *lock point*: the first point at which all of its locks are set. The collection of locks it has seized when it reaches its lock point is called its *lock set*.

It is easy to argue that the simple locking protocol leads to correct coordination. One line of argument is as follows: suppose the system maintains a permanent, ordered list of atomic operations that have passed their lock point. Upon reaching its lock point an atomic operation adds itself to this list. By the simple lock protocol the atomic operation has agreed not to read or write any data until it is in the list. The list provides us with the useful constraint that all other atomic operations that precede this one in the list have already passed their lock point. Since no data object can appear in the lock sets of two atomic operations, no data object in this atomic operation's lock set appears in the lock set of any uncommitted atomic operation earlier in the ordering. Thus all of

this atomic operation's input values are the same as they will be when the next earlier operation commits or aborts. By repetition and induction the same argument applies to all atomic operations before that one, so all inputs to this atomic operation are identical to the ones that would be available if all the atomic operations ran purely sequentially, in the order of the list. Thus the simple locking protocol produces correct coordination.

There is one significant objection to simple locking: in some applications an atomic operation does not know in advance the complete list of data objects it should read or write; the way it constructs that list is by exploration, reading some data objects in order to discover the identity of others. Actually, simple locking is much more constrained than necessary for correctness. One can relax the protocol to permit reading (or even writing) a data object as soon as it is locked, as long as no locks are released until the commit point is passed. Further, one can release locks on objects that won't be modified any time after the lock point is passed *if* those objects won't be needed again by this atomic operation. This relaxed protocol is called *two–phase* locking, and is widely used, but the argument that it leads to correct coordination is one step harder. Informally, once a data object is locked, its value is the same as it will be when the lock point is eventually reached, so reading it must yield the same result as waiting to read it till later. Releasing a read–only lock must be harmless as long as this atomic operation will never look at the object again, even to abort.

There are two interactions between locks and the log–based atomicity recovery system that we should think about: individual atomic operations that abort, and system recovery. Individual atomic operations are easiest to contemplate. Since we require that an aborting atomic operation restore its changed data objects to their original values before releasing any locks, no special account need be taken of aborted operations. From a coordination point of view they look just like committed operations that didn't change anything. Note that the rule about not releasing any locks on modified data before commit or abort is essential to accomplishing an abort. If a lock on some modified object were released, and then the atomic operation decided to abort, it might find that some other atomic operation has now seized the object and changed it again. Backing out an aborted change is likely to be impossible unless the locks on modified objects have been held.

The interaction between log–based recovery and locks is less obvious. The first, seemingly puzzling, question is whether locks themselves are data objects for which changes should be logged. Since locks exist only to coordinate atomic operations, and at the instant of completion of recovery from a crash there will be no outstanding atomic operations (all the incomplete ones will be rolled back by the recovery procedure), it would actually be a mistake if there were locks still set when crash recovery is complete. That observation by itself suggests that locks belong in volatile storage, where they will automatically disappear on a crash, rather than in stable storage, where the recovery procedure would have to hunt them down to release them. The controlling question, however, is whether or not the log–based recovery algorithm will construct a correct system state—correct in the sense that it could have arisen from some sequential ordering of those atomic operations that committed before the crash.

Suppose that the locks are in volatile memory, and at the instant of a crash, all record of the locks is lost. Some collection of atomic operations has not committed; they had non–overlapping lock sets before the lock values vanished. The recovery algorithm of figure 7–24 will systematically locate every data value that was changed by every in–progress atomic operation (that is, every loser), and reset it to the value it had when the atomic operation started. Because the lock sets are non–overlapping, no data object will be restored more than once, and so the process of recovery cannot be affected by the absence of the locks. So long as no new atomic operations begin until recovery is complete, there is no danger of miscoordination despite the absence of locks. Another way of looking at this argument is that the result is the same as if: 1) the locks were not lost, 2) the recovery algorithm scanned back through the log undoing only one of the losing atomic operations, and then reset its locks, 3) the recovery algorithm performed another scan undoing another atomic operation, etc., until all losing atomic operations were undone. If that algorithm were followed, it is clear that the result would be correct. At the same time, since the lock sets of the various rolled–back atomic operations are non–overlapping, there would be no place in the algorithm where the setting of the locks made any difference.

3. Performance complications

Most logging–locking systems are substantially more complex than our description would lead one to expect. The complications are the result of techniques used to achieve higher performance. These techniques include:

- buffered writes to disk storage
- log granularity adjustment
- lock granularity adjustment
- lock compatibility modes
- logging instructions for data update rather than old and new values.

Most disk management systems provide buffering for writes to the disk, in order to smooth out the flow of disk requests. Without buffering, each call to *put* starts I/O to the disk, and only when the disk write is complete does the I/O manager return control to the caller. In a buffered system, a call to *put* will start an actual write operation only if the disk I/O manager concludes that is the optimum operation to schedule right now. If some previous write operation is still in progress, the I/O manager queues this write request for later. In addition, it immediately returns control to the caller, who may continue with other projects, including perhaps calling *put* again with additional write requests. A buffered disk management system is especially useful in the case where an application issues several *puts* in a row and then goes on to compute for a long time. In such a case, the disk writing and the computation can overlap and substantially reduce the overall time required

to complete the job.

Although buffering is an important performance enhancement, it adds complications to the life of any atomicity system, because if the system crashes, some data updates may still be queued in buffers in volatile memory, rather than safely on the disk. Worse, some buffering systems may actually perform write operations in a different order from that requested, if reordering makes disk operations go faster. Reordering of disk requests is especially hazardous to atomicity systems that use the write–ahead–log protocol, since a data value may get changed on the disk before the corresponding log record gets written.

The basic mechanism required to restore atomicity is quite simple: one must implement as part of a buffered I/O manager an additional entry point that allows the application to request that certain writes, previously queued, be completed now. This operation is typically named *force*, and it takes an argument or identifier that the I/O manager returned on a previous *put* call.

If every call to *put* is followed immediately by a corresponding call to *force*, the application can cause a buffered disk system to behave (and perform) just like an unbuffered system, thereby restoring atomicity, at a cost in performance. A more interesting approach is to choose judicious times to call *force*, so that disk buffering can be allowed when it doesn't interfere with atomicity. One set of judicious times is the following:

1. Before calling *put* for a data record, call *force* for the log entry that records this data update.

2. Before calling *put* for a commit (or abort) entry in the log, call *force* for all data updates (or undos) of this atomic operation.

3. When writing a commit entry in the log, always call *force* immediately following the *put*.

The first rule guarantees that the order of actual disk writes follows the write–ahead log protocol, so as to be certain that no data update reaches the disk before the log entry that shows how to undo it. The second rule insures that if a commit (abort) entry appears in the log, all of the updates (undos) associated with it are on disk, so recovery processing can safely classify this atomic operation a winner and ignore it. The third rule, by delaying until the commit log entry is on disk, provides the atomic operation with assurance that no later crash will cause it to roll back. Once that log entry is on disk, any future recovery operation will declare this atomic operation to be a winner, so it is safe for the operation to go ahead and, for example, announce that fact by releasing output messages.

There exist higher–performance strategies for maximizing buffering and I/O–computation overlap. One can, for example, develop a strategy based on calling *force* only at commit time and only for log entries. However, the complexity (and the difficulty of making convincing correctness arguments) grows rapidly. To pull off this strategy, one must make sure that the I/O manager doesn't rearrange its output write queue (if it did, a data update might get to the disk before the corresponding log entry, and an ensuing crash would be unrecoverable). In addition, the log would

have to record both old and new values, so that recovery processing can *redo* winning atomic operations whose data updates were still in volatile buffers at crash time. The reconstruction process required is fairly elaborate, additional measures to avoid scanning the entire log at recovery time must be planned, and the entire recovery process must be idempotent. Appendix C contains a case study of a high-performance data management system that uses this strategy.

A problem closely related to buffering is *log granularity*. In all of the algorithms described so far, the size of a log entry has been assumed to be exactly the same as the size of the object that was updated. Whether small or large, each log entry calls for a separate *put* operation and, depending on the buffering/atomicity strategy, perhaps a *force* operation and thus a distinct, expensive write operation to the disk. Most disk systems have fixed block sizes, chosen for performance optimization, and they work most efficiently if asked to *put* records of the fixed block size. Typically, such a block is much larger than a single log entry, so the logging system finds that it can improve performance by accumulating log records until it has a block full, and only then calling *put*. An accumulation of log entries into a large block is another example of a buffer, and can be managed in the same way, by insisting on writing that buffer, whether full or not, whenever any entry in it should be forced to disk.

A similar consideration applies to data updates and to lock granularity: if one makes a change to a six-byte field in the middle of a 1000-byte block, there is a question about what should be locked: the six-byte field, or the 1000-byte block? In high-performance I/O management systems, only large blocks may be the subject of a *put* call. If two different, parallel tasks make updates to unrelated, small fields that happen to be stored in the same disk block, then the two disk writes must be coordinated; locking the entire block rather than the individual data items is the most straightforward approach.

The alternative of locking only a single record of a disk block is appealing because it apparently allows more concurrency: if another atomic operation is interested in a different record that is in the same disk block, it could proceed in parallel. However, this finer-grained locking desire actually has a very large side-effect: the I/O management system must present a logical, rather than physical disk interface to its atomic operation clients; such things as record management and garbage collection within disk blocks must now be handled below the I/O manager interface. A second consequence of finer-grained locking is that logging must also be done on the same finer-grained objects. (Because different parts of the same disk block may be modified by different atomic operations that are running in parallel, if one operation commits but the other aborts neither the old nor the new disk block is the correct one to restore following crash; the log entries must record the old and new values of the individual records of the block.)

There is an important performance refinement used in most locking systems, the specification of lock compatibility modes. When an atomic operation seizes a lock, it can specify what operation (for example, *read* or *update*) it intends to perform on the locked data item. If that operation is compatible—in the sense that the result of parallel operations is the same as some sequential ordering of those operations—then this atomic operation can be allowed to set the lock even though

some other atomic operation has already set such a lock. The most common example is read–mode compatibility: suppose atomic operation Alphonse has seized a lock indicating it plans only to read some object. Then if atomic operation Gaston wants to seize the same lock for reading, at the same time, there is no reason to delay Gaston. The purpose of locking in this case is to prevent updates, and since a simultaneous reader is not a threat, any number of atomic operations can simultaneously hold a read–mode lock for the same object. If another atomic operation now tries to seize the same lock with the intent to update that object, that third atomic operation will have to wait for both Alphonse and Gaston to release their locks.

There are many applications in which most data accesses are for reading only. In such applications the provision of read–mode lock compatibility often reduces the frequency of lock interference between parallel operations by an order of magnitude.

The idea of lock modes can be carried further, though one needs a careful analysis of the application to decide whether more elaboration is worthwhile. For example, there is a common situation in data bases where one atomic operation may lock a single record in a file, while another may need to lock the whole file. For such cases one can introduce "someone–is–inside–reading" and "someone–is–inside–updating" lock modes for the file as a whole. Any number of atomic operations may be reading and updating individual (non–overlapping) records at the same time. However, to seize a read lock for the whole file one would have to wait until all "someone–is–inside–updating" locks on the file were released. Allowing the locking of either a record of a file or a whole file is a two–level example of what is called nested or hierarchical locking. Research papers have been written on hierarchical locking modes, though little hard data is available on what typical performance enhancements result.

One final performance–enhancing technique should be mentioned: to make log entries shorter, one can record instructions on how to change the data (e.g., "add 5 to each of 100 array elements") rather than the complete old or new data values. This strategy requires careful thought to preserve idempotence of the recovery process. The danger is that a recovery procedure may perform the undo transformation, then crash, then perform the undo transformation a second time, leading to utterly incorrect data. The usual technique to preserve idempotence is to include in each data record a note indicating which log record corresponds to its latest value. Then a recovery procedure can decide whether or not the transformation described by a log entry has been applied to a data record that it finds on the disk.

This description of performance complications has not been complete or systematic, but rather illustrative, to indicate the range of hazards and kinds of complexity that they engender. If one intends to implement a system using performance enhancements such as buffering or fine–grained locking, it would be advisable to study carefully some previously existing system that implements those same enhancements.

4. *Deadlock*

One final issue surrounding use of locks is worth a brief mention. Suppose activity Alphonse locks object Apple, and then finds that object Banana is locked by activity Gaston. Alphonse decides to wait until Gaston finishes and frees Apple. If Gaston now becomes interested in object Apple and tries to lock it, we have a deadlock on our hands. Gaston cannot proceed until Alphonse continues far enough to release the lock on Apple, but Alphonse is waiting for Gaston to release the lock on Banana. The possibility of deadlock (sometimes called "deadly embrace") is an inevitable consequence of unrestricted use of locks to coordinate parallel activities. Obviously, any number of parallel atomic operations could get mutually hung up in a deadlock.

There are several techniques that are used to cope with deadlock, all of which fall into one of three general categories:

1. *Timeout.* If there is a timeout expiration on atomic operations, simply wait. When the time for one of the atomic operations involved in the deadlock expires, the system aborts that atomic operation (releasing its locks) and the remaining atomic operations may be able to proceed. If not, another will time out, releasing further locks. As with the progress-insuring strategies of section D.5, it might be useful for the system to impose a randomly chosen delay on the aborted atomic operation before letting it try again. The timeout technique is effective, though it has the usual defect with timeouts: it is difficult to choose a suitable timeout value that keeps things moving but also accommodates normal delays and legitimately variable operation times. If the environment or system load changes, it may be necessary to change all the timeout values, a real nuisance in a large system.

2. *Lock ordering.* Number the locks uniquely, and insist that atomic operations must seize locks in ascending numerical order. That way, when an atomic operation encounters an already-seized lock, it is always safe to wait for it, since the atomic operation that previously seized it cannot be waiting for any locks that this atomic operation has already seized—all those locks are lower in number than this one. There are many variations on this strategy, including an elegant one by Bensoussan, in which an atomic operation may seize locks in any order, but if it encounters a seized lock with number lower than some it has seized itself, the atomic operation backs up just far enough to release its higher-numbered locks. Another generalization is to arrange the locks in a lattice and require that they be seized in some lattice traversal order.

3. *Cycle detection.* Maintain, in the operating system, a schedule of which atomic operations are waiting for which other atomic operations. Whenever another atomic operation tries to seize a lock and finds it is already locked, the system examines the schedule to see if waiting would produce a cycle of waits, and thus a deadlock. If it would, the system selects some cycle member to be the victim, and unilaterally aborts that operation, so that the others may continue. The aborted operation then retries in the hope that the other atomic operations have made enough progress to be out of the way and another deadlock does not occur.

Generally, one can say that the more one permits unconstrained parallelism where there may be frequent interactions among different atomic operations, the more likely is deadlock. Nevertheless, one hopes that the additional performance gained from the parallelism more than offsets any time wasted in aborting and retrying occasional atomic operations that encounter deadlocks. The suggestions for further reading include several papers that explore deadlock–preventing algorithms in detail.

F. Multi–site Atomic Actions

So far we have explored techniques such as double–writing, commit records, and logs, to achieve atomicity. All these techniques are designed to hide the composite nature of actions that occur all in close physical proximity. We now consider how to create atomic operations from steps that must be carried out in different places—places separated by enough distance that *communication delay, communication reliability,* and *independent failure* are significant concerns. Multi–site atomic operations are quite complex, so we shall edge up on them by considering first two sub–problems. The solutions to these two sub–problems will then be the basis for implementing multi–site atomic operations. The first sub–problem, that of nesting atomic operations, comes up even in a single site. The second sub–problem, called the *two generals' problem,* leads to a remote procedure call protocol that shows how to coordinate steps that must be carried out at different places. Merging the techniques of the two problem solutions leads to a multi–site atomicity technique known as the *two–phase commit* protocol.

1. Hierarchical composition of atomic operations

We got into the discussion of atomic operations by considering that complex interpreters are engineered in layers, and that each layer should implement atomic operations for its next–higher, client layer. But each client layer has a still–higher client layer of its own for which it should implement atomic operations, so we conclude that atomic operations must be nested, consisting at every step of lower–level atomic operations. This nesting requires that careful thought be given the mechanism of achieving atomicity.

Consider again a banking example. Suppose that the earlier–described procedure to transfer funds from one account to another is available and is implemented as an atomic operation. Suppose now that we wish to write a higher–level procedure to pay or extract interest on a single customer account:

```
pay-interest:  procedure(account);
               if balance(account) > 0
                  then do;
                  interest = balance(account) * 0.18;
                  transfer(bank, account, interest);
                  end;

                  else do;
                  interest = -balance(account) * 0.26;
                  transfer(account, bank, interest);
                  end;
               return;
```

This procedure moves an appropriate amount of money from or to an internal account named "bank", the direction and rate depending on whether the customer account balance is positive or negative.

We might also expect that periodically the bank runs a program to update interest on every customer account:

```
month-end-interest:    procedure;
                       for A <-- each customer account
                          pay-interest(A);
                       return;
```

There are at least two, nested applications for atomic operations in these interest-paying programs. First, the procedure "pay-interest" should be executed atomically, so as to insure that some other "transfer" operation doesn't change the balance of the account between the test for balance sign and the calculation of the interest amount. Second, procedure "month-end-interest" should be an atomic operation, to insure that some transfer operation does not move a large amount of money from an account A to an account B between the interest-payment processing of those two accounts, since such a transfer could cause the bank to pay interest twice on the same funds. Remembering that the "transfer" operation by itself is also an atomic operation that is composed of several steps, we see that we require three nested layers of atomic operations.

The reason nesting is a potential problem comes from a consideration of the "commit" step of two nested atomic operations. The commit step of the "transfer" operation would seem to have to occur either before or after the commit step of the "pay-interest" operation, depending on where in the programming of "pay-interest" we place its commit step. Yet either of these positions will cause trouble. If the "transfer" commit occurs in the pre-commit phase of "pay-interest" then pay-interest will not be able to honor its obligation of being able to back out as though it hadn't tried to operate; if the transfer commit does not occur until the post-commit phase of pay-interest, there is a risk that the transfer itself will not be completable, for example because one of the accounts is inaccessible. The conclusion is that somehow the commit point of the nested operation

should be coincident with the commit point of the nesting operation.

We can accomplish this coincidence by extending the idea of a commit record slightly: we allow commit records to be organized hierarchically. Suppose a commit record can contain a pointer to another commit record. When, as part of an atomic operation, we create a nested atomic operation, we place in the commit record of the newly–created, nested atomic operation a pointer back to the nesting atomic operation's commit record. The resulting hierarchical arrangement of commit records then exactly reflects the nesting of the atomic operations. (A top–level commit record would contain a null pointer to indicate that it is not nested inside any other atomic operation.) When a commit record contains a pointer to a higher level commit record, we shall call it a *dependent* commit record.

The atomic operations, whether nested or nesting, then go about their business, and depending on their success mark their own commit records as committed or aborted, as usual. However, whenever anyone inquires about whether or not an atomic operation has committed, (for example, when reading an apparently tentative version,) the inquirer must now go through a more elaborate procedure. Starting with the commit record of the atomic operation in question, check its status. If it is committed, then follow the pointer in that commit record back to its superior and check that commit record. If that record is committed, continue following the chain upward until the highest–level commit record is encountered. The atomic operation in question is actually committed only if all of the commit records in the chain are committed. If any one is aborted, this atomic operation is actually aborted (despite claims in its own commit record.) Finally, if neither condition is true (all committed or at least one aborted) there must be one or more records in the chain that are uncommitted and the outcome of this atomic operation remains uncommitted until those records become committed or aborted. Thus the outcome of an apparently–committed dependent commit record actually depends on the outcome of its ancestors. We can describe this situation by saying that the dependent commit record and the associated atomic operation are *tentatively committed.*

This hierarchical arrangement has several interesting programming consequences. First, an atomic operation can commit even though some atomic operation nested inside it aborts. Success of that suboperation may not be required for success of the higher–level operation, or perhaps a retry of the activity (nesting a different atomic operation for the retry) will have worked. Second, visible output of nested atomic operations must be thought about very carefully: there should not be any until all the hierarchically higher operations have committed. For example, if one of the nested operations decides it is appropriate to open a cash drawer, the sending of the release message to the cash drawer ought to be held up until the outcome of the highest–level operation is known.

This output visibility consequence is only one example of many relating to the tentatively committed state. The nested atomic operation, having declared itself in this state, is sitting on a knife edge. It has renounced the ability to abort—the decision is in someone else's hands. It must be prepared to run to completion *or* to abort, and it must maintain that prepared state indefinitely. Maintaining the ability to go either way can be an awkward business, as the atomic operation may

involve holding locks, keeping pages in memory or tapes mounted, or reliably holding on to output messages. Thus one immediate conclusion is that one cannot blindly take an operation that has been programmed to be atomic, and include it as a nested part of a larger atomic operation. At the very least, one must review the resource requirements involved in maintaining the tentatively committed state.

A third, more complex, consequence arises when one considers possible interactions among different atomic operations that are nested within the same higher-level atomic operation. Suppose atomic operation A invokes first atomic operation B, which commits tentatively, pending the outcome of A. Then A invokes atomic operation C, which is programmed to read as input data some of the results of tentatively committed atomic operation B. The read-current-value algorithm, as implemented in figure 7-21, doesn't distinguish between reads arising within the same group of nested atomic operations and reads from some completely unrelated atomic operation. If its test for committed value is extended in the way described earlier to follow the ancestry of the commit record controlling the latest version, it will undoubtedly force atomic operation C to wait pending the final outcome of atomic operation B, which is only tentatively committed. But since the final outcome of B, depends on the outcome of atomic operation A, and the outcome of A currently depends on the success of C, we have a built-in cycle of waits that at best can only time out and abort.

The possibility of such cycles means that the question of whether or not to permit reading tentatively committed values cannot be answered casually. The goal is that uncommitted updates should not be visible outside the uncommitted operation; but those changes can be passed around inside the uncommitted operation. We can achieve that goal in the following way: compare the commit record ancestry of the atomic operation doing the read with the ancestry of the commit record that controls the version to be read. If these ancestries do not merge (that is, there is no common ancestor) then the reader should wait for the version's ancestry to be completely committed. If they do merge, the requirement of correctness is that all the atomic operations in the ancestry of the data version that are below the point of merge should have tentatively committed, in which case no wait is necessary. Figure 7-25 illustrates an example of such a situation.[1]

The concept of nesting atomic operations hierarchically is useful in its own right, but our particular interest in nesting is that it is one of the bases for multi-site atomicity. We now return to that subject, first exploring what makes it different from the single-site atomic operation.

---

1. Note that this rule anticipates the possibility that the atomic operation might start several nested atomic operations going in parallel. In figure 7-25, A might have started B and C going in parallel, in which case operation G should not look at data value X until operation B decides whether to commit or abort.

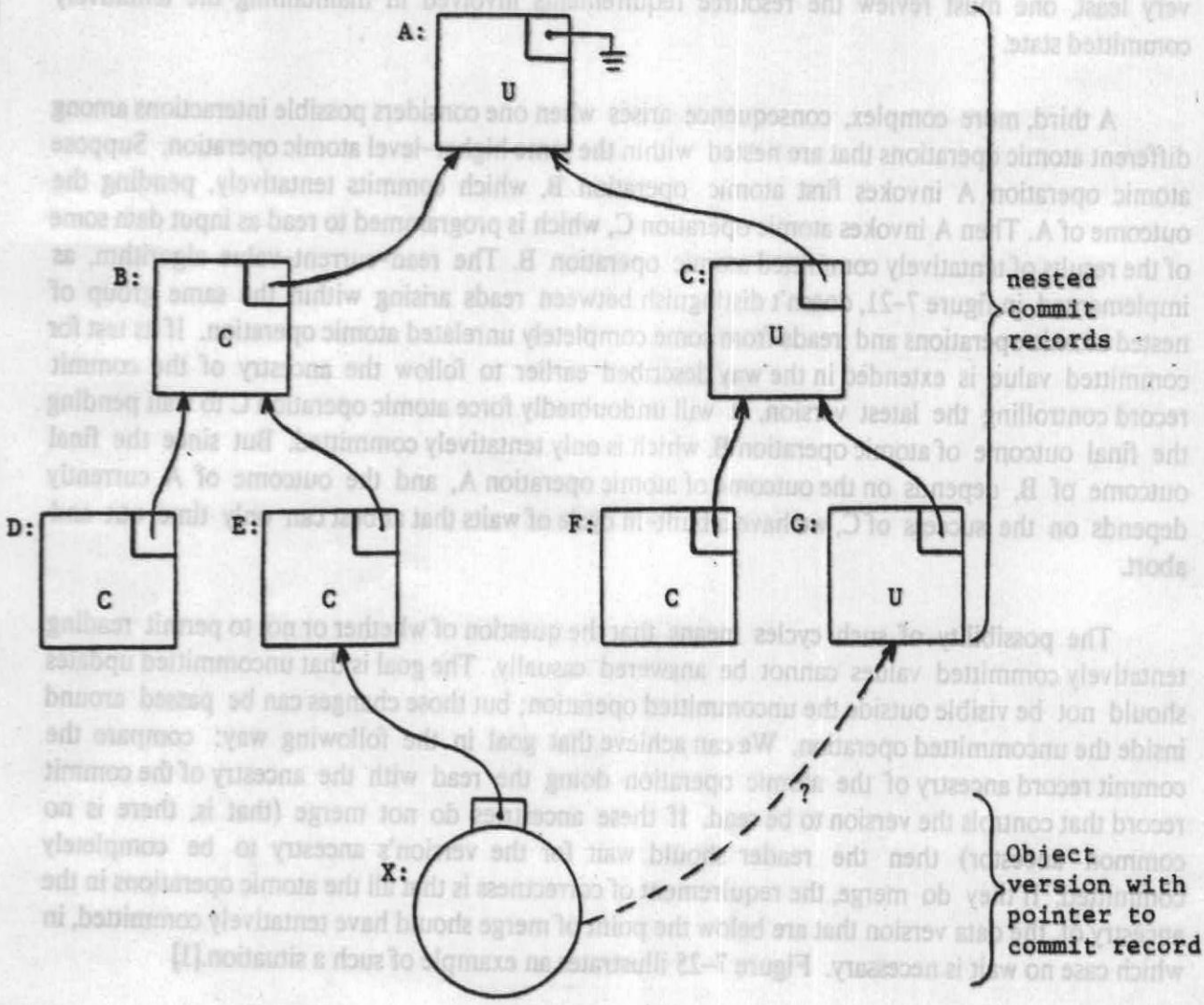


Figure 7–25. Atomic operation G, nested in atomic operation C, which is nested in atomic operation A, wants to read the current value of object X, which was last written by atomic operation E, nested in atomic operation B, nested in atomic operation A. Even though this version of X is only tentatively committed, since the first uncommitted ancestor of X is one that is shared with G, the read should be allowed.

2. Why multi-site atomicity is harder

When an application program invokes a procedure, the programmer plans for success, but usually has in mind the possibility that the procedure may fail in some way. That is, the procedure may return an output value that indicates it couldn't do the job, it may leave an error code in a global variable, or it may signal a failure by invoking some condition-reporting mechanism. Thus the programmer normally plans to check for one of several outcomes, and assumes that the calling program will be able to learn which of the outcomes happened and proceed accordingly.

To increase availability and capacity to handle large applications, it is common to use several computer systems connected with communication links. One of the prime opportunities to increase availability when several computers are available is independent failure: when one computer system crashes for whatever reason, the others may be able to carry on with what they were doing, if their activities are not too tightly coupled with those of the machine that failed. If they are tightly coupled, then a failure in one computer may propagate, causing all the systems to crash. The heart of the multi-site coordination problem is to allow coordination among sites, but keep it loose enough that individual sites can fail independently of one another.

The way this problem shows up at the application programming level is quite straightforward, though a little difficult to deal with: if a program invokes a procedure that is actually to be carried out at another site, there is a new possible outcome. In addition to success or failure, there is the possibility of no response at all. Lack of response from another site can arise for several reasons:

1. The other site may have crashed, and is being repaired or is in the midst of recovery procedures.

2. The other site may have received the message requesting the operation, and done the operation, but crashed just as it was about to send back the confirmation of success.

3. The message asking the other site to execute the procedure may have been lost. (A communication system is best viewed as an additional "site" that may crash independently, losing messages in transit.)

4. The message confirming that the other site did the requested job may have been lost by the communication system.

5. The other site has a long queue of work, and will eventually get around to doing the requested operation, but on a time scale far beyond the invoker's expectations.

6. The originating site may have crashed just after sending the message requesting execution of the procedure.

The important issue that arises in examining these various cases is that lack of response, by itself, does not provide any clue as to whether the other site did or did not perform the requested procedure. Thus the new outcome: the caller does not know what happened. This outcome can be somewhat distressing: if one has not planned ahead for the possibility, simply sending the request again may result in the job being done twice. For some applications (e.g., retrieving a data page) that might not hurt, but for others (e.g., debiting a bank account) it would probably be a serious mistake. The first site could send an inquiry to the second site asking if it did the job, but only if the original request is uniquely identifiable and the second site keeps good records concerning what it has and hasn't done. Thus we need to develop a protocol for reliably performing a remote procedure call. The development of such a protocol is known as the "two–generals' problem".

3. *The two–generals' problem*

The constraints on possible coordination protocols when communication is unreliable are captured in an analogy, originally posed with vivid images by James N. Gray, called the *two–generals' problem.* Suppose that two small armies are encamped on two mountains outside a city. The city is well–enough defended that it can repulse and destroy either of the two armies if that army should attack alone. But if the two armies attack simultaneously, they can take the city. Thus the two generals who command the armies need to coordinate their movements.

The only method of communication between the two generals is to send runners from one camp to the other. But separating the two mountains is a narrow, deep canyon that the runners must leap across. Unfortunately, the wind blowing through the canyon is quite treacherous and unpredictable, and there is a chance that the runner, trying to leap across, will instead fall into the canyon, never to be heard from again.

The dilemma of the two generals is now apparent. The first general sends the message:

```
Date:     22 April
From:     Patton
To:       Sherman
Message:  Attack at dawn tomorrow. OK?
```

expecting that the second general will respond either with:

```
Date:     22 April
From:     Sherman
To:       Patton
Message:  OK, dawn on the 23rd.
```

or, possibly:

```
Date:     22 April
From:     Sherman
To:       Patton
Message:  Awaiting reinforcements, can't participate on 23rd.
```

Suppose that the first message does not get through. In that case, the second general does not march because no request to do so arrives, and the first general does not march because no confirmation arrives, and all is well (except for the lost runner). Now suppose the first message is delivered successfully and the reply "OK" is sent back but lost. The first general cannot distinguish this case from the earlier case, so must not march. The second general has agreed to march, but knowing that the first general won't march unless the confirmation arrives, the second general should not march unless the first general is known to have received the confirmation. This hesitation on the part of the second general suggests that the first general should send back an acknowledgement of the confirmation. Unfortunately, that doesn't help, since the acknowledgement may be lost and the second general, waiting for the acknowledgement, will still not march.

We can now leap directly to a conclusion: there is no protocol with a bounded number of messages that can perfectly coordinate the two generals. If there were such a perfect, bounded protocol, the *last* message in that protocol must be unnecessary to the perfection of the protocol, because it might be lost, undetectably. Since the last message must be unnecessary, one could delete that message from the protocol to produce another, simpler protocol, that also guarantees perfect coordination. We can reapply the same reasoning repeatedly to the shorter protocol to produce still shorter ones, and we must conclude that if such a perfect protocol exists it either is of zero length or else is unbounded in length.

A practical general, presented with this argument by a mathematician in the field, would reassign the mathematician to a new job as a runner, and send an engineer to inspect the canyon and report the probability of a successful leap. Knowing that probability, the general would then send several runners, each carrying a copy of the message, choosing a number of runners large enough that the probability that *all* fail to leap the canyon is so small as to be negligible. (The loss of all the runners simultaneously becomes an intolerable error—from the point of view of the general.) Similarly, the second general sends many runners each carrying a copy of the acknowledgement. This practical scheme will satisfy the generals, but it leads to an interesting complication: once the first runner has gotten through, the general at the receiving end must not get confused by the arrival of additional runners bearing the same message: each general must learn to cope with duplicates.

We can state this conclusion more generally: if messages may be lost, no single protocol can both prevent duplicates and at the same time guarantee delivery. Coping with duplication is not impossible, but it is harder than it may appear. The basic approach is to arrange that all operations triggered by messages be idempotent: doing them again has no effect. If the operation is not naturally idempotent, it can be made so by placing a unique identifier in each different request

message, keeping a list (in stable storage) of the unique identifiers received so far, and explicitly discarding any request messages that arrive but that contain unique identifiers already in the list. If the request called for a reply, the reply should also be kept on file and sent back in response to duplicate requests.

The two generals would probably consider duplicate marching orders to be typical and would not get confused unless the mathematician (hoping to avoid reassignment as a runner) tried to explain idempotence.

4. Remote procedure call

We can now construct a reliable remote procedure call. Suppose we have two sites, the first taking on a role as *coordinator* C, of the call, the second as *worker* W. The coordinator sends a message:

```
From:        C
To:          W
Identifier:  45
Message:     Please transfer(accountX, accountY, $1000)
```

The identifier "45" might indicate that this is the 45th remote procedure call from C to W. The protocol will depend on this identifier being unique, so coordinator C should keep this identifier in stable storage to make certain that it doesn't get used again and for reference in the event of a crash.

The coordinator then waits for a response from W. The response is expected to be either

```
From:        W
To:          C
Identifier:  45
Message:     Transfer accomplished.  Regards.
```

or a response of the same form but with the message "transfer not possible, insufficient funds." Worker W checks to see if it has already responded to message "45" before. If not, it stores the message identifier "45" in its own stable storage, does the requested transfer, and records the response message in stable storage, all as a single atomic operation. After completing the atomic operation, it sends the response message. If it discovers that it has already handled a message identified as "45", it simply resends the response for message "45" that it finds in stable storage.

The coordinator, if it does not hear any response in a reasonable time, resends its request message. If the reason for lack of response is that the request message was lost or the worker site was down, then the resent message will now accomplish the desired affect. If the lack of response is

because the response message was lost, the worker will notice that it has seen message "45" before and it will resend the response. If the worker crashed while doing message 45, that crash occurred either before or after the commit point. If before, since the job was atomic, there will be no record of message 45, and the worker will now do the job. If after, then the response for message 45 is securely on file and the worker will resend it.

Assuming the coordinator is persistent enough to keep trying, this protocol will eventually succeed in getting the remote function invoked exactly once. Even if the coordinator gives up after several tries and calls for repair of the communication network, when the repair is complete the coordinator can continue the protocol, eventually assuring success. Note, however, that success requires that the coordinator be persistent.

Note also that success means only that the remote procedure gets invoked; and the coordinator knows the result. If there are several such remote operations to be invoked, the protocol provides no obvious help in forging a single atomic operation out of the set of remote operations. However, it provides us with a style of protocol design that will be useful in devising a multi–site atomicity protocol.

5. Two–phase commit

Between the hierarchical commit record mechanism for nesting atomic operations and the reliable remote procedure call for dealing with independent failures, we now have the tools needed to develop a multi–site atomicity algorithm. Our goal is the following: to assemble an atomic operation that is composed of individual operations that are carried out at several sites, despite the possibility that messages may be lost and the various sites can crash and recover independently. We assume that each site, on its own, is capable of implementing local atomic operations, using stable storage and some mechanism such as version histories or recovery logs. Correctness of the overall atomicity protocol will be achieved if all the sites commit or if all the sites abort; we will have failed if some sites commit their part of an atomic operation while others abort their part of the same atomic operation.

Suppose the atomic operation consists of a coordinator C requesting operations X, Y, and Z of worker sites W1, W2, and W3, respectively. The simple expedient of issuing three remote procedure calls certainly does not produce an atomic operation, because worker W1 may do X while worker W2 may report that it cannot do Y. Conceptually, the coordinator might like to send three messages, to W1, W2, and W3 respectively, like this one to W1:

```
From:       C
To:         W1
Identifier: 91
Message:    if (W2 does Y and W3 does Z) then do X, please.
```

and let the three workers handle the details. Unfortunately we still have no clue how W1 could execute this strange request.

The clue comes from recognizing that the coordinator and the workers represent a hierarchical nesting of atomic operations. The coordinator has created a higher-level atomic operation, and each of the workers is to perform an atomic operation that is nested in the higher-level operation. The complication is that the coordinator and workers are not reliably in communication with one another. Our problem then reduces to that of constructing a reliable communications protocol to fit in the middle of a hierarchically composed atomic operation.

The protocol is known as *two-phase commit.* The protocol starts with the coordinator creating a top-level commit record for the overall atomic operation. Then the coordinator sends a message such as this one to worker W1:

```
From:      C
To:        W1
Message:   Please do X as part of my atomic operation 271.
```

Similar messages go to workers W2 and W3. As in the remote procedure call, if the coordinator doesn't get a response from one or more workers in a reasonable time it may resend the message to the non-responding workers.

A worker site, upon receiving a request of this form, creates an atomic operation of its own but it makes the commit record a *nested* one, with its superior being the original atomic operation of the coordinator. It then goes about doing job X, and either commits—but only tentatively—or aborts. It then sends a response back to the coordinator:

```
From:      W1
To:        C
Message:   I have tentatively committed my part of atomic operation
           271.  Please don't forget about me.  Regards.
```

or alternatively, a message reporting it has aborted. As in the remote procedure call case, if a duplicate request arrives at worker W1 from the coordinator C, W1 simply sends back a duplicate of its previous response, reporting either tentative commitment or abort.

At this point worker W1, if it has tentatively committed, is out on a limb. Just as in a local hierarchical nesting, W1 must be prepared either to run to the end or to abort, to maintain that state of preparation indefinitely, and wait for someone else to tell it which. In addition, the coordinator may independently crash or lose communication contact, increasing the uncertainty of W1.

The coordinator C collects the response messages from the several workers (perhaps rerequesting the original operations several times from some worker sites) and if all workers have agreed tentatively to commit. Phase one of the two–phase commit is complete. Phase two begins as the coordinator commits the entire atomic operation by marking its own commit record. (If any worker aborts, the higher–level operation has the choice of aborting the entire operation or, for example, trying a different worker site for the substep that failed.)

Once the higher–level commit record is marked as committed or aborted, the coordinator sends a completion message back to each worker:

```
From:       C
To:         W1
Message:    Atomic operation 271 committed.  Thanks for your help.
```

It then goes about its business, with one important requirement for the future: it must remember, reliably and for an indefinite time, the outcome of this atomic operation. The reason is that one or more of its completion messages may have been lost. Any worker sites that are tentatively committed are awaiting the completion message to tell them which way to go. If a completion message does not arrive in a reasonable period of time, the worker site should send an inquiry back to the coordinator:

```
From:       W1
To:         C
Message:    Hey, whatever happened to your atomic operation 271?
```

Whenever the coordinator gets such an inquiry it simply sends back the current state of the commit record for the named atomic operation. If worker site W1 crashes, the recovery procedure at W1 should include a test to discover tentatively committed atomic operations and send inquiries about their current state, on the chance that the completion message arrived during W1's crash. If the coordinator reports "committed", the recovery procedure classifies the atomic operation as a "winner", whereas if the coordinator reports "aborted", the recovery procedure classifies it as a "loser". If the atomic operation is still uncommitted at the higher level, the recovery procedure must restore the tentative, waiting state.

If all goes well, coordination of N worker sites will be accomplished in $3N$ messages: for each worker site a request message, a response message, and a completion message. This $3N$ message protocol is complete and sufficient, although there are several variations one can propose.

Some versions of the two–phase commit protocol have a fourth acknowledgement message from the worker sites to the coordinator. The intent is to collect a complete set of acknowledgement messages—the coordinator can resend completion messages over and over until every site acknowledges. The coordinator can then safely discard its commit record, since every worker site is guaranteed to have gotten the word.

A system that is concerned both about commit record storage space and the cost of extra messages can use a further refinement. Since (we assume) most atomic operations commit, we can use a slightly odd, but very space–efficient representation for the value "committed" of a commit record: non–existence. Any inquiry about a non–existent commit record is answered "committed". If the coordinator uses this representation, it commits by destroying the commit record, so a fourth acknowledgement message from every worker is quite unnecessary. In return for this apparent magic reduction in both time and space, we notice that commit records for aborted atomic operations can not easily be discarded, because if an inquiry arrives after discarding, the inquiry will receive the response "committed". The coordinator can, however, ask for acknowledgement of aborted atomic operations, and discard the commit record after all these acknowledgements are in.

This last refinement completes our inquiry into multi–site atomicity algorithms. The reader should be aware that this area is the subject of ongoing research and development activity and that there are many refinements and variations being explored in the current literature.

G. Perspectives

In this chapter we have gone into considerable depth on several specific problems and systematic approaches to their solution. At this point it is appropriate to stand back from all those technical details—as far back as we can get without completely losing sight of the problems being solved—and try to develop some perspective on how all these ideas relate to the real world. The observations of this section are wide–ranging: history, tradeoffs, skepticism, and unsettled topics. Individually these observations appear somewhat disorienting and disconnected, but together they may provide the reader with some preparation for the wide range of reactions that atomicity receives in the practical world of computer system design.

First, the history: Systematic application of atomicity to recovery and to coordination is relatively recent. Ad hoc programming of parallel activities has been common since the late 1950's, when machines such as the IBM 7030 (STRETCH) computer and the experimental TX–0 at M.I.T. used interrupts to keep I/O device driver programs running in parallel with the main computation. The first time–sharing systems (in the early 1960's) demonstrated the need to be more systematic in interrupt management, and many different semantic constructs were developed over the next decade to get a better grasp on coordination problems: Dijkstra's semaphores, Brinch Hansen's message buffers, Reed and Kanodia's event counts, Habermann's path expressions, and Hoare's monitors are examples. A substantial literature grew up around these constructs, but a characteristic of all of them was a focus on properly coordinating parallel activities, each of which by itself was assumed to operate correctly. The possibility of failure and recovery of individual activities, and the consequences of such failure and recovery on coordination with other, parallel activities, was not a

focus of attention. Another characteristic of these constructs is that they resemble a machine language, providing variously–shaped tools but little guidance in how to apply them.

Failure recovery was not simply ignored in those early systems, but it was handled quite independently of coordination, again using ad hoc techniques. The early time–sharing system implementers found that users required a kind of stable storage, in which files could be expected to survive intact in the face of system failures. To this end most time–sharing systems periodically made extra copies of on–line files, using magnetic tape as the backup medium. The more sophisticated systems developed incremental backup schemes, in which recently created or modified files were copied to tape on an hourly basis, producing an almost–up–to–date log. To reduce the possibility that a system crash might damage the on–line disk storage contents, salvager programs were developed to go through the disk contents and repair obvious and common kinds of damage.

These ad hoc techniques, though adequate for some time–sharing system use, were not enough for designers of serious database management systems, who developed the concept of a transaction, which is exactly a failure–atomic operation applied to a database. Atomicity logging protocols thus developed in the database environment, and it was some time before they were recognized as providing a kind of recovery semantics that had wider applicability.

Within the database world, coordination was accomplished almost entirely by locking techniques that became more and more systematic, with the identification of serializability of atomic operations as a fundamental definition of correctness. That identification, in turn, along with a requirement for hierarchical composition of programs, led to the development of version–history systems. Version histories systematically provide both recovery and coordination with a single mechanism, and they simplify building big atomic operations out of several, independently developed, smaller ones.

In this chapter, we have reversed this development, because the version history system is pedagogically the more straightforward, while the more ad hoc logging/locking approach is somewhat harder to learn at first. However, the reader should realize that a majority of systems currently in the field use the older approach.

Now, the tradeoffs: An interesting set of tradeoffs applies to techniques for coordinating parallel activities. Figure 7–26 suggests that there is a spectrum of coordination possibilities, ranging from totally serialized operations on the left to complete absence of coordination on the right. Starting at the left, we can have great simplicity but admit no parallelism at all. Moving toward the right, complexity increases, but so does the possibility of improved performance, since more and more parallelism is admitted. For example, the mark–point–sequential and simple locking algorithms might lie more toward the left end of this spectrum while read–capture and two–phase locking would be farther to the right.

Continuing to traverse this spectrum there is an important boundary, of *correctness:* to the right of this boundary results may be wrong, in the sense that no serial schedule of the same

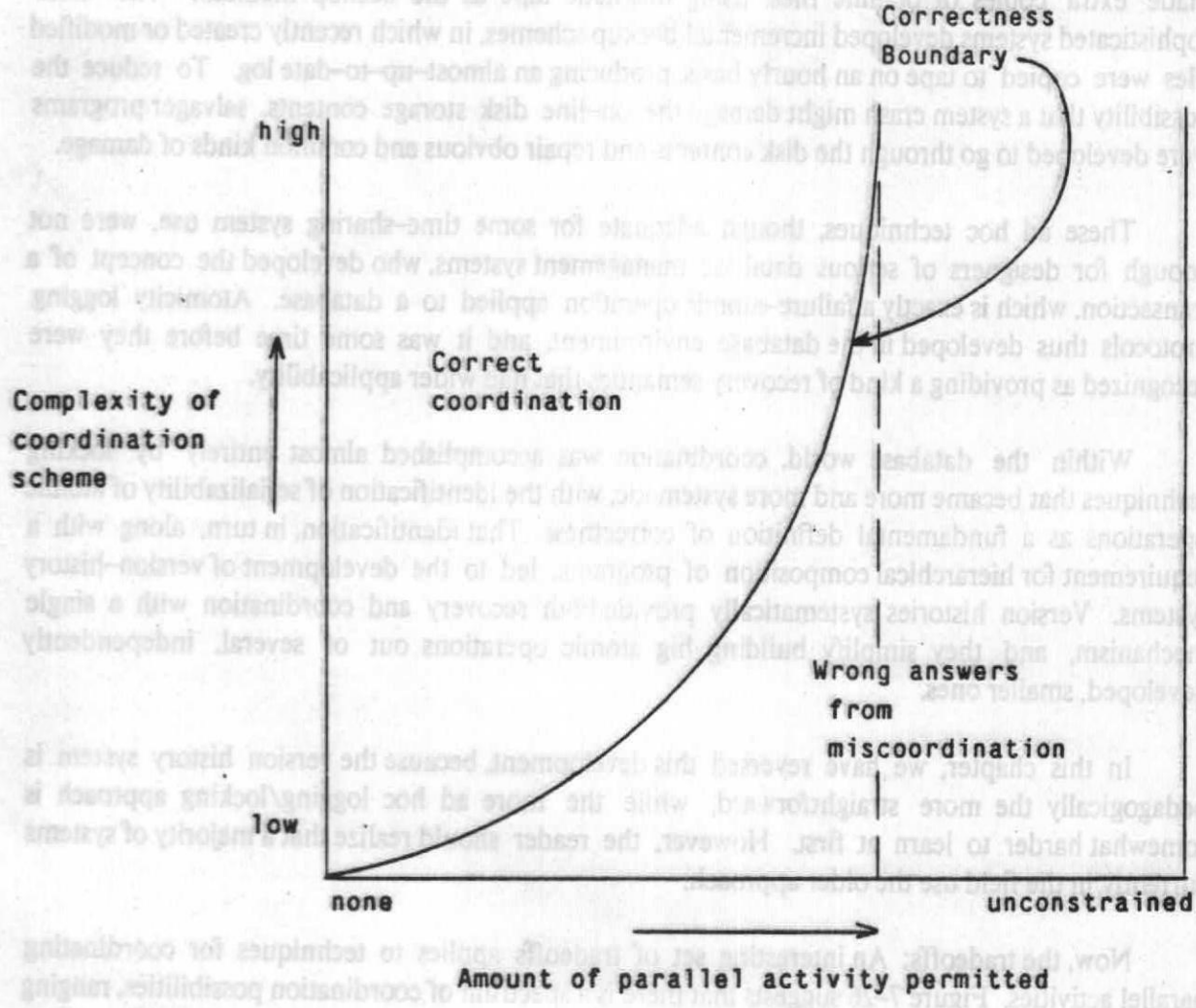


Figure 7–26. The tradeoff between parallelism, complexity, and correctness. The choice of where in this chart to position a system design depends on the answers to two questions: 1) How frequently will parallel activities actually interfere with one another? 2) How important are 100% correct results? If interference is rare, it is appropriate to design farther to the right. If correctness is not essential, it is acceptable to design even to the right of the correctness boundary.

operations could have that outcome. The closer one approaches this boundary from the left, the higher the performance, but presumably at the cost of higher complexity. All of the algorithms explored in this chapter are intended to operate to the left of the correctness boundary, but we might inquire about the possibilities of working on the other side. Such a possibility is not as unthinkable as it might seem at first. If interference between parallel activities is rare, and the cost of an error is small, one might actually be willing to permit parallel operations that can lead to certifiably wrong answers. For example, in an inventory control system for a grocery store, if an occasional sale of a box of cornflakes goes unrecorded because two point–of–sale terminals tried to update the cornflake inventory simultaneously, the resulting slight overstatement of inventory may not be a serious problem. The grocery store must do occasional manual inventory anyway because other boxes of cornflakes get misplaced, damaged, and stolen. This higher–level data recovery mechanism will also correct any errors that creep in because of miscoordination in the programming of the inventory management system, so its designer might well decide to use a coordination technique that allows maximum parallelism, is very simple, catches the most common miscoordination problems, but nevertheless operates to the right of the strict correctness line. A decision to operate a data management system in a mode that allows such errors can be made on a rational basis. One would compare the rate at which the system loses track of inventory because of its own coordination errors with the rate that it loses track because of outside, uncontrolled events. If the latter rate dominates, it is not necessary to press the computer system for better accuracy.

Another plausible example of acceptable operation to the right of the correctness boundary is the calculation, by the Federal Reserve Bank, of the United States money supply. Although in principle one could program a two–phase locking algorithm that includes every bank account in every bank that contains U.S. funds, the practical difficulty of accomplishing that task with thousands of independent banks distributed over a continent is formidable. Instead, the data is gathered without locking, with only loose coordination and it is almost certain that some funds get counted twice and other funds get overlooked. However, great precision is not essential in the result, so lack of perfect coordination among the many individual bank systems operating in parallel is quite acceptable.

Although allowing incorrect coordination might appear usable only in obscure cases, it is actually applicable to a far wider range of situations than one might guess. In almost all data base management applications, the biggest cause of incorrect results is wrong input by human operators. Typically, stored data already has many defects before the transaction programs of the database management system have a chance to "correctly" transform it. Thus the proper perspective is that operation to the right of the correctness boundary of figure 7–26 merely adds to the rate of incorrectness of the database. With that perspective, one can more sensibly balance heavy–handed but "correct" transaction coordination schemes against simpler techniques that can occasionally damage the data in limited ways. If, as is usual, there is a higher–level system in place to cope with damaged data anyway, the simpler coordination scheme may be preferable. One piece of evidence that this approach is acceptable in practice is that one of the most widely–used data management systems, IMS for the IBM System/370, uses a simple set of locking and coordination rules that are demonstrably incorrect. However, the frequency of awkwardly ordered update operations that can

produce wrong answers is apparently low enough that users find the result tolerable.

Unfortunately the same thing cannot be said in the area of instruction set design for central processors. Three generations of central processor designers (of the main frame processors of the 1950's and 1960's, the mini–computers of the 1970's, and the one–chip micro–processors of the 1980's) have not recogized the importance of atomicity in their initial design and have been forced to retrofit atomicity into their architectures later.

Next, the unsettled topics: there are several areas in which coordination and recovery are not completely settled problems. These areas include the fundamental basis of atomicity, the role of timeouts, composition with locks, the role of forward error correction and the effects of delay, the systematic application of compensation, the systematic use of loose or incorrect coordination, and the possibility of malicious participants.

At the bottom of coordination atomicity must be some fundamental mechanism for achieving mutual exclusion. In a locking system there must be a way of arbitrating between two participants that happen to try to set the same lock at just the same time. In a version history system, mutual exclusion is needed in the creation of unique identifiers, to insure that two atomic operations do not get assigned the same identifier. Mutual exclusion is also required in the updating of the values of commit records. At the hardware level, mutual exclusion is implemented by an *arbiter*, a device that chooses which of two request signals to honor. Unfortunately, all known arbiter designs have a kind of indeterminacy that appears when the arbiter is asked to choose between very closely–spaced signals: it seems to be the case that the more closely–spaced the two signals, the longer is the average time for the arbiter to choose one of them. The arbiter can enter a so–called *metastable state*, with an output value somewhere between its two correct values or possibly oscillating between them at a very high rate. As a consequence, after applying a signal to an arbiter, one must wait some length of time for its output to settle. For any given delay time there is some probability that the arbiter will not have settled yet, and a sample of its output may find it changing. This phenomenon is apparently related to the physics of metastable states, but it has not been studied completely enough; bounds on settling time are not part of the published specifications of arbiters.

Timeouts are a separate area of difficulty that have been mentioned several times in the earlier parts of this chapter. Timeouts are used to solve various problems: discovery of deadlocks, resolution of contention, garbage collection of failed atomic operations. The trouble with timeouts is that it is hard to choose a good value for them. If too short, some operation that is proceeding normally, but under worst–case conditions, may abort unnecessarily. If too long, performance will degrade as resources are tied up waiting for a timeout to occur. Worse, when conditions change, timeout values must change accordingly, so it is bad design to embed them in programs. There is not yet a well–understood and systematic design approach that avoids timeouts or makes the choice of their length unimportant. The state of the art seems to consist of careful documentation both of the existence of timeouts and the considerations that led to choice of their value, to minimize the maintenance burden.

Hierarchical composition—making larger atomic operations out of previously programmed smaller ones—interacts in an awkward way with locking as a coordination technique. The problem arises because locking protocols require a lock point for coordination correctness. Creating an atomic operation from two previously independent atomic operations is difficult, because each separate atomic operation has its own lock point, coinciding with its own commit point. But the higher–level operation must also have a lock point, suggesting that the order of capture and release of locks in the constituent atomic operation needs to be changed. But rearrangement of the order of lock capture and release contradicts the usual goal of modular composition, under which one assembles larger systems out of components without having to modify the components. To maintain modular composition, the lock manager (that is, the program that implements "seize" and "release") needs to know that it is operating in an environment of hierarchical atomic operations. With this knowledge, it can, behind the scenes, systematically rearrange the order of lock release to match the requirements of the operation nesting. For example, when a nested atomic operation calls to release a lock, the lock manager can simply relabel that lock to show that it is held by the higher level, not–yet–committed, atomic operation in which this one is nested. A systematic discipline of passing locks up and down among nested atomic operations thus can preserve the goal of modular composition.

This chapter has focused almost entirely on error recovery using detection and retry, rather than on error masking by use of redundancy. Error masking, sometimes called *forward error correction*, is a widely–used technique in systems that must meet deadlines, such as manufacturing automation systems, life support systems, and high–availability database management systems. Achieving reliability through error detection and retry can lead to unexpected delays at any time, and some applications cannot cope with such delays. For those applications, forward error recovery is a more appropriate technique, and the reader is referred to the suggestions for further reading for papers and books on that subject. One area that has not yet received much attention is the design of systems that use both retry and forward error correction, blending the techniques to best advantage for the particular application. Although there exist systems that use both approaches simultaneously, they are generally ad hoc designs, and there is no well–understood design discipline that combines the two approaches.

A related problem not explored here is the one of unrecoverable errors: suppose that multi–site coordination is applied to a group of ships fighting a naval battle, and part way through the battle one of the ships sinks. Resending requests to the lost ship is futile; algorithms that can carry on (perhaps with reduced effectiveness but, one hopes, without forfeiting the battle by default) are an interesting area for future development.

Returning to figure 7–26 for the moment, the possibility of designing a system that operates in the region of incorrectness is intriguing, but there is one major deterrent: one would like very much to specify, and thus limit, the nature of the errors that can be caused by miscoordination. This specification might be on the magnitude of errors, or their direction, or their cumulative effect, or whatever. Systematic specification of tolerance of coordination errors is a topic that has not yet been seriously explored.

Compensation is the way that one deals with miscoordination or with recovery in situations where undoing an operation invisibly cannot be accomplished. Compensation is performing a visible operation that reverses all known effects of some earlier, visible operation. For example, if a bank account was incorrectly debited, one might later credit it for the missing amount. The usefulness of compensation is limited by the extent to which one can track down and reverse everything that has transpired since the operation that needs reversal. In the case of the bank account, one might successfully discover that an interest payment on an incorrect balance should also be adjusted; it might be harder to reverse all the effects of a check that was bounced because the account balance was too low. Apart from generalizations along the line of "one must track the flow of information output of any operation is that to be reversed" little is known about systematic compensation; it seems to be a very application–dependent concept.

Finally, the coordination schemes we explored in this chapter assume that the various participants are all trying to reach a consistent, correct result. A recently emerging area of study explores what happens if one or more of the workers in a multi–site coordination task decides maliciously to mislead the others, for example by sending a message to one site reporting it has committed, while sending a message to another site reporting it has aborted. (This possibility is described colorfully as the *Byzantine Generals'* problem.) One reason for exploring this area is the concern that undetected errors in communication links could simulate this kind of uncooperative behavior. The importance and practical applicability of this area of study is not yet established but it provides a wonderful sandbox for theoretical exploration of coordination algorithms.

This set of perspectives completes our study of atomicity. Three appendices provide real–world examples of atomicity in action.

Appendix 7-A: Case studies of atomic operations at the machine language level

1. Honeywell 68/80

In the Honeywell Information System, Inc., 68/80 computer architecture (a descendent of the General Electric 600-line, which was very similar,) a feature called "indirect and tally" was provided. One could specify this feature as a modifier on any indirect word. The instruction

```
Load register A from location Y indirect.
```

was interpreted to mean that location *Y* contains an indirect word, which word contains the address of the actual operand of the original instruction. But in addition, if the indirect word in *Y* contains a "tally" modifier, the processor is to increment the indirect word by one and put it back in location *Y*, so that the next time location *Y* is used as an indirect word it will point to a different operand—the next sequential one in memory. Thus the indirect and tally feature could be used to sweep through a table.

Suppose that virtual memory is in use, and that the indirect word is located in one page, (which is in real memory,) while the operand is in another page (which is not in real memory right now.) When the above instruction is executed, the processor will retrieve the indirect word, increment it, and store the new value in memory. Then it will attempt to retrieve the ultimate operand, and discover that it is not in real memory; a missing page fault must be triggered. Since the indirect word has been modified and by now may have been already removed from real memory by the missing page handler running on another parallel processor, it is not feasible just to go back and "unmodify" it so as to pretend that this instruction has not been tried.

In the Honeywell 68/80, the virtual memory designers wanted to be able to run other programs on the halted processor while awaiting the arrival of the missing page. They therefore extended the definition of the current program state to contain not just the next-instruction counter and the program-visible registers, but also the complete internal state description of the processor—a 216-bit snapshot in the middle of the instruction. By later restoring the processor state to contain the previously saved values of the next-instruction counter, the program-visible registers, and the 216-bit internal state snapshot, the processor could exactly continue from the point of interruption. This technique works, but it has two bad side effects: 1) when a program (or programmer) inquires about the current state of an interrupted processor, the state description includes things not in the programmer's interface; and 2) the system must be very careful when

restarting an interrupted program to make certain that the stored micro–state description is a valid one. If some one has altered the state description the processor could try to continue from a state it could never have gotten into by itself, possibly leading to failures of its memory protection features.

2. IBM System/360

When IBM added virtual memory to its System/360 architecture, to produce the System/360 Model 67, certain multi–operand character–editing instructions produced atomicity problems. These instructions required touching from two or four virtual–memory operands, any one of which might not be in real memory. Rather than tampering with the program state definition, the IBM architects chose a different strategy, called the "dry run". With the dry run strategy, the instruction is executed using a hidden copy of the program–visible registers. If an operand turns out to be missing from real memory, the processor can pretend that it never tried the instrction, since there is no program–visible evidence that it did. The stored program state shows only that the character–editing instruction is about to be executed. After the missing page is retrieved, the instruction is tried from the beginning again, another dry run. If there are several operands required, several dry runs may occur to get them all into real memory. When, finally, a dry run succeeds in completing, the instruction is run once more, this time for real, using the program–visible registers. Since the System/360 (at the time this modification was made) was a one–processor architecture, there was no possibility that a parallel processor might snatch a page away after the dry run but before the real execution of the instruction. (This technique has the side effect of making life more difficult for a later designer who has the task of adding multiple processors.)

3. The Apollo desktop computer and the Motorola M68000 microprocessor

When Apollo Computer designed a desktop computer using the Motorola 68000 microprocessor, the designers, who wanted to add a virtual memory feature, discovered that the microprocessor was not atomic. Worse, because it was constructed entirely on a single chip it could not be modified to make it atomic (as in the IBM 360) or to make it store the internal microprogram state (as in the Honeywell 68/80). So the Apollo designers used a different strategy: they provided not one, but two M68000 processors. When the first one runs into an operand missing from real memory, it simply stops in its tracks, and waits for the operand to appear. The second M68000 (whose program is carefully planned to reside entirely in real memory) fetches the missing page and then restarts the first processor.

Other designers working with the M68000 used a different, somewhat risky trick: modify all compilers and assemblers to generate only instructions that happen to be atomic. More recently, Motorola produced a version of the M68000 in which all internal state registers of the microprocessor can be saved, much like the Honeywell 68/80.

Appendix 7-B: A better model of disk failure, with atomicity algorithms

This model of disk operation is similar to that of section 7.C.2, but with the reclassification of certain intolerable errors as detectable. We begin with the same model of the processor, memory, and operating system as before, in which all failures in those areas lead to an instant reset of all volatile storage and restart of the processor. The changes are in the analysis of the disk subsystem operations, *Put* and *Get*. We assume that, rather than providing stable storage, the disk system can fail in various detectable ways: our goal is to provide the illusion of stable storage despite those failures.

```
Put (address, data)

        desired event:      data is written at address
        undesired events:
            expected:       detectable garbage is written at address
            expected:       nothing is written at address
                            (Old data remains.  New data may have been
                            written at some other address by mistake.)
            intolerable:    undetectable garbage is written at address
```

Note that there are subtle interplays among various parts of the system. For example, this model of *Put* must continue to apply even if the system crashes in the middle of the *Put*. One would probably implement these specifications by writing a redundancy check pattern on the disk after the last byte of each disk block. By including the address of the disk track in the redundancy check, one can even detect the case where the data gets written at the wrong address.

To allow a program to participate in recovery, we need a way for it to find out about detectable errors. Thus the *Get* operation now has a third argument, a returned opinion on the validity of the data found at the requested address. The disk system might render this opinion by reading the data and the associated redundancy check pattern, recalculating the check pattern from

the read data, and comparing the two check patterns.

```
Get (address, data, opinion)

        desired event:      previously written data is returned and
                            opinion is OK
        desired event:      damaged data or garbage is found and opinion
                            is not-OK
    undesired events:
          expected:         previously written data is actually OK but
                            for some reason opinion is not-OK.  (This
                            situation is called a "soft read error", and
                            is often correctable by repeated rereading.)
        intolerable:        damaged data or garbage is found and returned
                            but opinion is OK
```

To complete the model of disk storage, we must recognize that there is also a spontaneous event: decay. It is *never* desirable. It hits sets of disk pages, called *decay sets*, for example all the pages on one drive or on one disk platter. Writing data at the wrong address looks like decay to the owner of the data at that address.

Because a decay can occur at any instant, the probability that any given disk page is affected increases with time. We assume a memoryless decay process (technically, a Poisson distribution of inter-decay intervals). For such a decay process, if we wait a given length of time we can measure the probability of a decay during that time. Our decay-resistance strategy will be to keep two copies of the data around, and have a clerk periodically check both copies for decay. If either is discovered bad at one of these periodic checks, it is rewritten immediately from the good copy. What should the period of these checks be? It should be short enough that the probability that *both* copies have decayed since the previous check is negligible. By inspecting the statistics of experience for the disk system, we choose such a period, and call it *Td*. This approach leads to the following error analysis:

```
    undesired events:
          expected:         data turns to detectable garbage within
                            some decay set, with an interval of at
                            least Td seconds after the data was last
                            verified to be good and after such an event
                            occurs to any other decay set.
        intolerable:        two decay sets fail within Td seconds.
        intolerable:        a decay set fails within Td seconds after
                            it has been verified to contain good data.
        intolerable:        data turns to undetectable garbage.
```

Note that the intolerable errors all have to do with errors that are undetectable or (this may not be obvious yet) for which no recovery measure can be provided. We have thus postulated that the disk storage system provides a kind of *volatile* storage rather than *raw* storage. (Check the definition of these two terms in the glossary.)

In modeling a computer system this way, we are making two assumptions: 1) there exists a reasonably straightforward implementation using real hardware and software that exactly follows this model, and 2) that the implementation can be designed to have a negligibly small probability of producing any of the intolerable errors. Granted these two assumptions, we can proceed to develop algorithms that systematically suppress all cases of undesired but expected errors.

1. Algorithms to obtain stable storage

Our goal is to create *stable storage*, that is, secondary storage that never fails unless some intolerable error (which has negligible probability) occurs. We proceed in several steps, in each step improving the quaility of the storage by recovering from some of the undesired but expected error events.

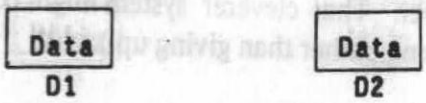
Step one is to create "careful storage" from our already existing volatile storage. Careful storage guarantees that *if its operations terminate*, they were successful. Figure 7-27 shows algorithms for *careful-get* and *careful-put*, and the new event analysis for each. In *careful-get*, by repeatedly reading any data with opinion "not-ok", so-called soft-read errors are suppressed. Similarly, in *careful-put*, by reading back (sometimes called *verifying*) the just-written data and retrying if the verification fails, the algorithm suppresses soft-write errors, whatever their source. (Note that if *careful-put* fails *n* times, we have an error currently classified as intolerable, but from which a still cleverer system might be able to recover. That cleverer system might continue retries on a different disk, of call for repair before continuing, rather than giving up.)

There is a subtlety: the event analysis tells us what happens if the operations *careful-get* and *careful-put* terminate. If *careful-put* returns, the written data is guaranteed good. (At least until a decay event happens.) But if the system crashes in the middle of *careful-put*, we may have overwritten old, good data with garbage rather than with new, good data. Thus despite the lack of expected errors in the accounting of figure 7-27, at this point our expected error events are actually two: a system crash during *careful-put* in which both the old and the new data are lost; and a spontaneous decay event on some track set. We try to solve both these problems with one mechanism.

The method resembles that of *SABRE:* write the data twice. However, we now have the advantage of *Get* returning opinions, and so *careful-get* can tell if a crash occurred during a previous *careful-put*. So we no longer need the version number of *SABRE*, we need only to be careful not to begin writing the second data copy until we are sure we successfully wrote the first one. Figure 7-28 shows the algorithm and the data.

```
careful-get:  {do repeated "get" until either data is returned with
               opinion = OK, or n tries have failed}

   desired event:     previously written data returned, opinion is OK
   desired event:     data is bad and opinion is not-OK
                      (after n tries)
   undesired events:
        expected:     none
        intolerable:  garbage is returned and opinion is OK

careful-put:   do {put (address, data); get (address, buffer, opinion)}
               until data = buffer and opinion = OK or n tries fail;

   desired event:     data is written at address
   undesired events:
        expected:     none obvious, but see discussion
        intolerable:  n tries to write have failed
```

Figure 7-27. Algorithms and event analysis for "careful storage." All obvious expected events have been suppressed by the algorithms, but the text points out that there is one hidden but expected event: that these algorithms might not terminate.




```
atomic-get:    [careful-get D1;
                  if D1 is not-OK then careful-get D2; return]

atomic-put:    [careful-get D1; careful-get D2;
                  if either D1 or D2 is not-OK, careful-put
                                        to bad copy from good copy;
                  if D1 ≠ D2, careful-put D1 --> D2]
               careful-put new-data --> D1
               careful-put new-data --> D2
               return
```

Figure 7-28. Data arrangement and algorithms to implement atomic stable storage using careful storage as the starting point.

This algorithm offers several opportunities to exercise our ability to reason carefully about correctness. For example, one might object that *atomic-get* does not bother to check the quality (opinion) of D2—perhaps it should. But this algorithm uses only *careful-get*; *careful-get* has *no* expected errors when reading data that *careful-put* was allowed to finish working on. Thus opinion would be not-ok only in the case where *careful-put* was interrupted in mid-operation. But *atomic-put* guarantees that it won't begin *careful-put* on data D2 until after the completion of its *careful-put* on data D1. At most one of the two copies could be not-ok because of a system crash during *careful-put*. Thus if the first copy is not-ok, then we expect that the second one is ok.

Of course the second one might be not-ok because some intolerable error occurred, but the algorithm is not claimed to work in the face of intolerable errors. What is going on here is that in reading D2 we have an opportunity to *detect* an error through the opinion feature, but since we have not thought of a way to recover when both data copies are damaged, this detectable error must remain classified as intolerable. In a real implementation one would not just ignore such an opportunity. At minimum one would watch for detectable, intolerable errors, report them as best possible, and log them for evaluation.

There is one currently unnecessary step hidden in the bracketed part of atomic-put: if D1 is not-ok, nothing is gained by copying D2 onto D1, since D1 will be overwritten with new data soon anyway. The step is included to illustrate a complete decay recovery procedure, or "salvager". This complete salvager is needed for our next refinement.

If decay events were never to happen, this algorithm would be completely equivalent in its expected error suppression capability to the *SABRE* algorithm: if *atomic-put* begins writing D1 and then crashes, D1 will look "not-ok" to *atomic-get*, which is exactly the same effect that changing version numbers had in the earlier algorithm. The last step is to construct *atomic stable storage*, which eliminates the last undesired but expected error event, spontaneous storage decay. To do this construction, we must take two two further steps:

1. Arrange that the two copies, D1 and D2, are placed in independent decay sets.

2. Take the bracketed part of our refined *atomic-put*, label it "salvager" and run it on every data record at least once every *Td* seconds.

If we also run the salvager code after every crash, then we might think that we don't need to do it at the beginning of *atomic-put*. That more economical variation is due to Lampson and Sturgis. It has one minor flaw: it depends on the rarity of coincidence of two undesired but nevertheless expected error events: the spontaneous decay of one data copy at about the same time that *careful-put* crashes in the middle of rewriting the other copy. If we are genuinely convinced that such a coincidence is rare, we can declare it to be intolerable, and then we have a self-consistent, correct, and more economical algorithm.

An important consideration in disk management algorithms is performance. Assuming that errors are rare enough not to dominate performance, the usual cost of *atomic-stable-get* is just one disk read, compared with two in the *SABRE* algorithm. (That count of two assumes that the version number and the data can be read in a single operation.) The cost of *atomic-stable-put* is two disk writes and four disk reads, compared with two disk writes and two disk reads for the *SABRE* algorithm. The four disk reads of atomic-stable-put reduce to two if a salvager is run following every crash, which suggests that that is a winning refinement. Finally, the salvager operation requires *2n* disk reads every *Td* seconds to maintain *n* useful, stable pages. There was no corresponding cost in the simplified *SABRE* model. The apparent extra expense of this improved model comes about because of its added function: it provides a defense against disk decay, rather than assuming its probability to be negligible. (The actual *SABRE* system was more sophisticated than our simplified model—it provided a similar, but independent, defense against decay events.)

Appendix 7-C: Case Study of System R

Prepared by Craig Goldman

"System R" is a relational data base management system developed as a research project at the IBM San Jose Research Laboratory. It is divided into two major layers: an external layer called the Research Data System (RDS) which provides the data base management functions, and a completely internal layer called the Research Storage System (RSS) which manages the physical memory and provides for stable storage with a transaction interface. The RSS layer permits transactions to be committed, aborted, or partially backed up. In addition the system provides recovery to a transaction-consistent state in case of failure.

The main portion of this case study is a paper by the designers of System R.[1] The next few pages provide a guide to reading this paper, an overview of the System R recovery subsystem, and some questions to think about.

1. Transactions

In a data base management system it would be very desirable if an operation on the data base either complete correctly or have no effect at all. The RSS system supports two levels of atomicity to achieve this result. Each RSS "action" is atomic even though it may be composed of several hundred machine instructions and many disk accesses. In addition RSS supports atomicity on a sequence of RSS actions called a ***transaction.***

A transaction is a sequence of RSS actions that are preceded by a BEGIN action and followed by a COMMIT action. All intervening RSS actions are part of a single recovery unit. If an application detects an error it may issue an ABORT action, which undoes all actions taken by the transaction. The system may also abort a transaction in progress if some system-wide problem occurs (e.g., deadlock, system shutdown, resource exhaustion, etc.). Thus there are two possible

---

1. Gray, J., et al., "The Recovery Manager of a Data Management System," Research Report RJ 2623, IBM Corporation, San Jose, CA., August, 1979. This report was later published under the title "The Recovery Manager of the System R Database Manager," *ACM Computing Surveys 13*, (June, 1981) pp. 223-242.

outcomes of a transaction: commit and abort. The abort outcome is the same whether triggered internally or externally.

In addition to the BEGIN, COMMIT, and ABORT actions that delimit a transaction, the RSS provides a refinement in the form of a firewall within a transaction, known as a *transaction save point.* In some cases, system error recovery can be accomplished by undoing a transaction back to some save point rather than all the way back to the BEGIN point. Transaction save points are described further in the Gray paper, but the concept is not essential to understanding the principles behind recovery in System R.

The RSS system maintains a log of all transaction updates. Each RSS action that changes the RSS state records enough information in the log to either *redo* the action (change the old state to the new state) or *undo* the action (change the new state to the old state). A transaction abort simply goes through the log backwards and undoes all of the state changes pertaining to that transaction. Should there be a system crash, the log can be used to redo committed transactions whose effect was not recorded on the disk. Thus atomicity of transactions can be maintained in the event of system error.

2. System R file recovery

The storage of System R is viewed as a collection of disk drives. To increase the bandwidth of access to this storage, the system has some solid-state volatile memory called a *memory buffer pool,* which is shared by all users and is used as cache for the disks. A file is a paged linear address space and is dynamically allocated on disk by pages. The buffer manager uses a least-recently-used algorithm to determine which pages are in the buffer pool at any time. Because the contents of the buffer pool are easily damaged by program errors or hardware failures, they are considered volatile and do not survive a system restart.

Each file is said to follow one of two recovery protocols referred to as *shadowed* and *non-shadowed.* Non-shadowed files have no automatic recovery mechanism and the user is responsible for making extra copies of these files and storing these copies in a safe place (on disk or off-line). Shadowed files have an automatic recovery in case of system failure.

RSS maintains two on-line versions of shadowed files: A *shadow version* and a *current version.* RSS actions affect only the current version of the file; the shadow version is not altered except by a file save or restore. Thus, as changes are made to a file, new pages are allocated and a "current version" page map is created so that all changes to a file are made to these pages. If a page is not altered then the current version page and the shadow version page are the same page. So, the current version of a file is made up of all current version pages (pages that have been altered) and the shadow version pages for any page that has not been altered. (A diagram of this double map strategy is shown on page 7-110 of the Gray paper.) At some point the application requests that the file be SAVED and the current version becomes the shadow version and the old shadow version is

thrown away. The SAVE action also forces all pages of a file to a disk so that a consistent shadow version is protected from system crash. If the system crashes before a file has been saved, an action-consistent version of the file (the shadow version) is certain to be still on the disk, from which restart can proceed. System R also provides for the log to be integrated with the file recovery at restart so that starting with the shadow version (which is action-consistent) all *committed* transactions may be redone automatically, and all aborted actions can be undone. This procedure is explained very well in the Gray paper.

3. *Questions*

1. Consider the following proposal to simplify the undo-redo protocol at system recovery from a crash: whenever a system checkpoint is to be taken, the system is first quiesced at the transaction level. That is, a lock is set that forces proposed new transactions to wait until after the checkpoint; as soon as all outstanding transactions finish, the checkpoint is taken. Describe the new crash recovery procedure, and show how this proposal would simplify undo-redo processing. What disadvantages would this approach have? Would this idea be more useful for some applications than for others?

2. In RSS, the idea of shadow pages was included to guarantee data integrity in the face of certain kinds of failures. Then, since not all failure threats are countered by shadow pages, the log was introduced. Looking at these two mechanisms together, one sees that they both seem to operate by keeping track of "old" and "new" data values whenever data is updated. Does this mean that since the log has been added, shadow pages are completely redundant? Explain what would go wrong if one simply changed RSS to eliminate shadow pages and rely entirely on the log.

3. Let's try again to reduce the redundancy that seems apparent in having "old" and "new" values recorded both in the shadow system and in the log. Suppose that the log, instead of recording both the old value and the new value for a data record, were to contain just one value, the result of "exclusive or-ing" the old and new values together. Describe an algorithm for recovery that uses such an EXOR log.

4. On page 7-111, the authors explain that RSS supports three combinations of recovery attributes of files: no-log/no-shadow, no-log/shadowed, and log/shadowed. Explain the difficulties in providing the fourth combination, log/no-shadow recovery. What information needs to be saved by the log manager? What problem does this pose? (HINT: How much of the log need be kept on-line (disk or primary memory)?) Is there any way to solve this problem?

5. The use of shadow files adds a large amount of complexity to the RSS recovery subsystem. Yet another approach is to improve the log so that it can provide all recovery. Whenever an RSS action is performed, the log record of that action is saved on disk before the data itself is written to disk. Should an error occur complete recovery of the file can take place by redoing the actions in the log, and no shadow files are needed. To speed recovery of the log-based system,

system-wide save points can be taken thus avoiding the problem of a very long log. Explain how this system ("Write-Ahead Log") would work. How can the system-wide saves be integrated into the log? What conditions are necessary to get an accurate save? Does the system have to wait until all transactions in progress are committed or aborted? (This is a tough question—think it over.) After the overhead for the Write-Ahead Log is considered (overhead for long log on-line, overhead of system-wide saves, time needed to quiesce the system for a save) is there any saving over the shadow file approach?

6. On the last paragraph of page 7-107 the author states:

> The transaction model is an unrealizable ideal. At best, careful use of redundancy minimizes the probability of unrecoverable failures and consequent loss of committed updates. Redundant copies are designed to have independent failure modes, making it unlikely that all records are lost at once. However, Murphy's law ensures that all recovery techniques will sometimes fail. As seen below, however, System R can tolerate any single failure and can often tolerate multiple failures.

Describe the model of storage of System R. Perform an expected event analysis. What types of failures are tolerated and corrected? What type of failures are not tolerated? Compare this analysis with the model of storage and the expected event analysis of the *SABRE* system. Provide an explanation for the differences.

7. Consider the need for special handling of transaction UNDO's. (Transaction UNDO's run as golden transactions—only one may execute at a time.) Explain the problem which leads to the need for special handling of UNDO's. Is it possible to have an UNDO for UNDO's? Consider the possibility of looking ahead to see which locks are needed for the next RSS step to be undone. Explain how such a scheme could be implemented and comment on its effectiveness.

8. On page 7-115, while discussing techniques to defend against media failure, the authors explain the dilemma of complete archive copies on tape: the copy should be made atomic with respect to ongoing transactions, but a copy of all the data in a large storage system may take a very long time to accomplish, too long to insist on quiescence of all other activity. The paper then suggests a "fuzzy dump" mechanism in which the data base is copied to tape without quiescing other activity, and then the log of that activity is also copied to the tape. The idea is that between the raw data and the log, a consistent copy of the data can be reconstructed if necessary. The paper refers the reader to Gray's *Operating System Notes*, but that reference turns out to contain essentially the same description, along with the comment that "the details of this [reconstruction] algorithm are left as an exercise for the reader" [8, p. 478]. Describe an algorithm that can be used to create a "sharp dump" from the "fuzzy dump" plus the log. (Start by proposing a sharper specification of the "fuzzy dump" procedure itself.)