Distributed Systems-Architecture and Implementation, Lecture Notes in Computer Science 105, Springer, 1981

Chapter 11. Atomic transactions

11.1. Introduction

This chapter deals with methods for performing atomic actions on a collection of computers, even in the face of such adverse circumstances as concurrent access to the data involved in the actions, and crashes of some of the computers involved. For the sake of definiteness, and because it captures the essence of the more general problem, we consider the problem of crash recovery in a data storage system which is constructed from a number of independent computers. The portion of the system which is running on some individual computer may crash, and then be restarted by some crash recovery procedure. This may result in the loss of some information which was present just before the crash. The loss of this information may, in turn, lead to an inconsistent state for the information permanently stored in the system.

For example, a client program may use this data storage system to store balances in an accounting system. Suppose that there are two accounts, called *A* and *B*, which contain $10 and $15 respectively. Further, suppose the client wishes to move $5 from *A* to *B*. The client might proceed as follows:

read account *A* (obtaining $10)

read account *B* (obtaining $15)

write $5 to account *A*

write $20 to account *B*

Now consider a possible effect of a crash of the system program running on the machine to which these commands are addressed. The crash could occur after one of the write commands has been carried out, but before the other has been initiated. Moreover, recovery from the crash could result in never executing the other write command. In this case, account *A* is left containing $5 and account *B* with $15, an unintended result. The contents of the two accounts are inconsistent.

There are other ways in which this problem can arise: accounts *A* and *B* are stored on two different machines and one of these machines crashes; or, the client itself crashes after issuing one write command and before issuing the other.

In this chapter we present an algorithm for maintaining the consistency of a file system in the presence of these possible errors. We begin, in section 11.2, by describing the kind of system to which the algorithm is intended to apply. In section 11.3 we introduce the concept of an *atomic transaction*. We argue that if a system provides atomic transactions, and the client program uses them correctly, then the stored data will remain consistent.

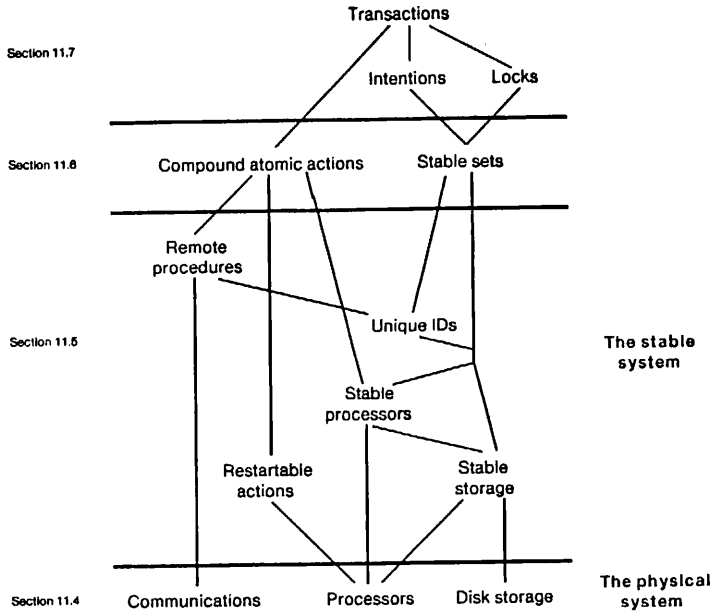

Figure 11-1: The lattice of abstractions for transactions

The remainder of the chapter is devoted to describing an algorithm for obtaining atomic transactions. Any correctness argument for this (or any other) algorithm necessarily depends on a formal model of the physical components of the system. Such models are quite simple for correctly functioning devices. Since we are interested in recovering from malfunctions, however, our models must be more complex. Section 11.4 gives models for storage, processors and communication, and discusses the meaning of a formal model for a physical device.

Starting from this base, we build up the lattice of abstractions shown in figure 11-1. The second level of this lattice constructs better behaved devices from the physical ones, by eliminating storage failures and eliminating communication entirely (section 11.5). The third level consists of a more powerful primitive which works properly in spite of crashes (section 11.6). Finally, the highest level constructs atomic transactions (section 11.7). Throughout we give informal arguments for the correctness of the various algorithms.

11.2. System overview

Our data storage system is constructed from a number of computers; the basic service provided by such a system is reading and writing of data bytes stored in the system and identified by integer addresses. There are a number of computers which contain client programs (*clients*), and a number of computers which contain the system (*servers*); for simplicity we assume that client and server machines are disjoint. Each server has one or more attached storage devices, such as magnetic disks. Some facility is provided for transmitting messages from one machine to another. A client will issue each read or write command as a message sent directly to the server storing the addressed data, so that transfers can proceed with as much concurrency as possible.

We follow tradition in using a pair of integers $\langle f, b \rangle$ to address a byte, where f identifies the *file* containing the byte, and b identifies the byte within the file. A file is thus a sequence of bytes addressed by integers in the range $1..n$, where n is the *length* of the file. There are two *commands* available for accessing files: a generalized *Read* command, and a generalized *Write* command. The generality allows information associated with a file other than its data bytes to be read and written, for example its length, or protection information associated with the file; these complications are irrelevant to our subject and will not be mentioned again. A client requests the execution of a command by sending a message containing the command to the appropriate server. When the command has been completed, the server sends a message to the client containing a *response*. In the case of a read command, this response will contain the requested data. For a write command, the response is simply an acknowledgement. It is necessary to provide interlocks between the concurrent accesses of different clients.

It should now be clear that our distributed *data storage* system is not a distributed *data base* system. Instead, we have isolated the fundamental facilities required by a data base system, or any other system requiring long-term storage of information: randomly addressable storage of bits, and arbitrarily large updates which are atomic in spite of crashes and concurrent accesses. We claim that this is a logically sound foundation for a data base system; if it proves to be unsatisfactory, the problem will be inadequate performance.

11.3. Consistency and transactions

For any system, we say that a given state is *consistent* if it satisfies some predicate called the *invariant* of the system. For example, an invariant for an accounting system might be that assets and liabilities sum to zero. The choice of invariant obviously depends on the application, and is beyond the scope of a data storage system. The task of the storage system is to provide facilities which, when properly used, make it possible for a client application program to maintain its invariant in spite of crashes and concurrent accesses.

A suitable form for such facilities is suggested by the following observation. Any computation which takes a system from one state to another can be put into the form

$$state := F(state)$$

where F is a function without side effects. A state is a function which assigns a value to each element (called an address) in some set (called an address space). In general F will change only the values at some subset of the addresses, called the *output set* of F; these new values will depend only on the values at some other subset of the addresses, called the *input set*.

That is, the function needs to read only those addresses whose values are actually referenced by the computation, and the assignment needs to write only those addresses whose values are actually changed by the computation. Such a computation will clearly be atomic in the presence of crashes if the assignment is atomic in the presence of crashes, i.e., if either all the writes are done, or none of them are. Two such computations F and G may run concurrently and still be atomic (i.e., serializable) if the input set of F is disjoint from the output set of G, and vice versa.

In pursuit of this idea, we introduce the notion of a *transaction* (the same concept is used in [Eswaren 76], and elsewhere in the data base literature, to define consistency among multiple users of a common data base). A transaction is a sequence of read and write commands sent by a

client to the file system. The write commands may depend on the results of previous read commands in the same transaction. The system guarantees that after recovery from a system crash, for each transaction, either all of the write commands will have been executed, or none will have been. In addition, transactions appear indivisible with respect to other transactions which may be executing concurrently; that is, there exists some serial order of execution which would give the same results. We call this the *atomic* property for transactions. The client will indicate the commands of a transaction by surrounding them with *Begin* and *End* commands. If the client fails to issue the end transaction command (perhaps because he crashes), then a time out mechanism will eventually abort the transaction without executing any of the write commands.

Assuming this atomic property for transactions, consider how the previous example might be implemented by a client. The client first issues a *Begin* command, and then continues just as in the example in section 1. After sending the two write commands, he sends *End*. This transaction moves \$5 from *A* to *B*. Notice that the client waits for the responses to the read commands, then computes the new balances, and finally issues write commands containing the new balances.

A client may decide to terminate a transaction before it is completed. For this purpose we have an additional command, *Abort*. This terminates the transaction, and no write commands issued in the transaction will take effect. Because the system also times out transactions, the *Abort* command is logically unnecessary, but the action it causes also occurs on a timeout and hence must be implemented in the system.

Thus, we have two groups of commands, which constitute the entire interface between the data storage system and its clients:

- Data commands: *Read*, *Write*
- Control commands: *Begin*, *End*, *Abort*

We shall return to transactions in section 11.7 after laying some necessary groundwork.

11.4. The physical system

To show the correctness of our algorithms in spite of imperfect disk storage, processor failures (crashes) and unreliable communication, we must have a formal model of these devices. Given this model, a proof can be carried out with any desired degree of formality (quite low here). The validity of the model itself, however, cannot be established by proof, since a physical system does not have formal properties, and hence its relation to a formal system cannot be formally shown. The best we can do is to claim that the model represents all the events which can occur in the physical system. The correctness of this claim can only be established by experience.

In order to make our assumptions about possible failures more explicit, and we hope more convincing, we divide the events which occur in the model into two categories: *desired* and *undesired*; in an fault-free system only desired events will occur. Undesired events are subdivided into expected ones, called *errors*, and unexpected ones, called *disasters*. Our algorithms are designed to work in the presence of any number of errors, and no disasters; we make no claims about their behavior if a disaster occurs.

In fact, of course, disasters are included in the model precisely because we can envision the possibility of their occurring. Since our system may fail if a disaster does occur, we need to estimate the probability p that a disaster will occur during an interval of operation T_O; p is then an upper bound on the probability that the system will fail during T_O. The value of p can only be estimated by an exhaustive enumeration of possible failure mechanisms. Whether a given p is small enough depends on the needs of the application.

In constructing our model, we have tried to represent as an error, rather than a disaster, any event with a significant probability of occurring in a system properly constructed from current hardware. We believe that such a system will have very small p for intervals T_O of years or decades. The reader must make his own judgment about the truth of this claim.

Our general theme is that, while an error may occur, it will be detected and dealt with before it causes incorrect behavior. In the remainder of this section, we present our model for the three main physical components on which the system depends: disk storage, processors and communication. In the next section, we describe how errors are handled; in general this is done by constructing higher-level abstractions for which desired events are the only expected ones.

11.4.1. Disk storage

Our model for disk storage is a set of addressable pages, where each page contains a status (*good*, *bad*) and a block of data. There are two actions by which a processor can communicate with the disk:

procedure *Put*(*at*: *Address*, *data*: *Dblock*)

procedure *Get*(*at*: *Address*) **returns** (*status*: (*good*, *looksBad*), *data*: *Dblock*)

Put does not return status, because things which go wrong when writing on the disk are usually not detected by current hardware, which lacks read-after-write capability. The extension to a model in which a *Put* can return bad status is trivial.

We consider two kinds of events: those which are the result of the processor actions *Get* and *Put*, and those which are spontaneous.

The results of a *Get*(*at*: *a*) are:

- (desired) Page *a* is (*good*, *d*), and *Get* returns (*good*, *d*).
- (desired) Page *a* is *bad*, and *Get* returns *looksBad*.
- (error) *Soft read error*: Page *a* is *good*, and *Get* returns *looksBad*, provided this has not happened too often in the recent past; this is made precise in the next event.
- (disaster) *Persistent read error*: Page *a* is *good*, and *Get* returns *looksBad*, and n_R successive *Gets* within a time T_R have all returned *looksBad*.
- (disaster) *Undetected error*: Page *a* is *bad*, and *Get* returns *good*, or if page *a* is (*good*, *d*), then returns (*good*, *d'*) with $d' \neq d$.

The definition of a persistent read error implies that if n_R successive *Gets* of a *good* page have all returned *looksBad*, the page must actually be *bad* (i.e., has decayed or been badly written; see below), or else a disaster has happened, namely the persistent error. This somewhat curious

definition reflects the fact that the system must treat the page as *bad* if it cannot be read after repeated attempts, even though its actual status is not observable.

The effects of a *Put*(*at*: *a*, *data*: *d*) are:

- (desired) Page *a* becomes (*good*, *d*).
- (error) *Null write*: Page *a* is unchanged.
- (error) *Bad write*: Page *a* becomes (*bad*, *d*).

The remaining undesired events, called *decays*, model various kinds of accidents. To describe them we need some preliminaries. Each decay event will damage some set of pages, which are contained in some larger set of pages which is characteristic of the decay. For example, a decay may damage many pages on one cylinder, but no pages on other cylinders; or many pages on one surface, but no pages on other surfaces. We call these characteristic sets *decay sets*; they are not necessarily disjoint, as the example illustrates. Two pages are *decay related* if there is some decay set which contains both. We also assume a partitioning of the disk pages into *units* (such as disk drives), and a time interval T_D called the *unit decay time*. We assume that any decay set is wholly contained in one unit, and that it is possible to partition each unit into pairs of pages which are not decay-related (in order to construct stable storage; see 11.5.1). T_D must be very long compared to the time required to read all the disk pages.

A *decay* is a spontaneous event in which some set of pages, all within some one characteristic decay set, changes from *good* to *bad*. We now consider the following spontaneous events:

- (error) *Infrequent decay:* a decay preceded and followed by an interval T_D during which there is no other decay in the same unit, and the only bad writes on that unit are to pages in the characteristic set of the decay. Because units can be bounded in size, the stringency of the infrequency assumption does not depend on the total size of the system.
- (error) *Revival*: a page goes from (*bad*,*d*) to (*good*, *d*).
- (disaster) *Frequent decay*: two decays in the same unit within an interval T_D.
- (disaster) *Undetected error*: some page changes from (*s*, *d*) to (*s*, *d'*) with $d' \neq d$.

Other events may be obtained as combinations of these events. For example, a *Put* changing the wrong page can be modeled as a *Put* in which the addressed page is unchanged (error), and an undetected error in which some other page spontaneously changes to a new *good* value (disaster). Similarly, a *Put* writing the wrong data can be modeled in the same way, except that the written page is the one which suffers the undetected error. One consequence of this model is that writing correct data at the wrong address is a disaster and hence cannot be tolerated.

11.4.2. Processors and crashes

Our model for a processor is conventional except for the treatment of crashes. A processor consists of a collection of processes and some shared state. Each process is an automaton with some local state, and makes state transitions (executes instructions) at some finite non-zero rate. Instructions can read and write the shared state as well as the local state of the process; some standard kind of synchronization primitive, such as monitors, allows this to be done in an orderly way. For simplicity we consider the number of processes to be fixed, but since a process may enter an idle state in which it is simply waiting to be reinitialized, our model is equivalent to one

with a varying but bounded number of processes. One of the processes provides an interval timer suitable for measuring decay times (see 11.4.1 and 11.5.1). The union of the shared state and the process states is the state of the processor. A processor can also interact with the disk storage and the communication system as described in sections 11.4.1 and 11.4.3.

A crash is an error which causes the state of the processor to be reset to some standard value; because of this effect, the processor state is called *volatile* state. This implies that the processor retains no memory of what was happening at the time of the crash. Of course the disk storage, or other processors which do not crash, may retain such memory. In a system with interesting long-term behavior, such as ours, a processor recovering from a crash will examine its disk storage and communicate with other processors in order to reach a state which is an acceptable approximation to its state before the crash. Since a crash is expected, it may occur at any time; hence a crash may occur during crash recovery, another crash may occur during recovery from that crash, and so forth.

Our model includes no errors in the processor other than crashes. The assumption is that any malfunction will be detected (by some kind of consistency checking) and converted into a crash before it affects the disk storage or communication system. It may well be questioned whether this assumption is realistic.

11.4.3. Communication

Our model for communication is a set of messages, where each message contains a status (*good*, *bad*), a block of data, and a destination which is a processor. Since we are not concerned with authentication in this paper, we assume that the source of a message, if it is needed, will be encoded in the data. There are two actions by which a processor can communicate with another one:

procedure *Send*(*to*: *Processor*, *data*: *Mblock*)

procedure *Receive* **returns** (*status*: (*good*, *bad*), *data*: *Mblock*)

The similarity to the actions for disk storage is not accidental. Because messages are not permanent objects, however, the undesired events are somewhat simpler to describe. The possible events are as follows:

The possible results of a *Receive* executed by processor *p* are:

- (desired) If a message (*good*, *d*, *p*) exists, returns (*good*, *d*) and destroys the message.
- (desired) If a *bad* message exists, returns *bad* and destroys the message.

There may be an arbitrary delay before a *Receive* returns.

The effects of a *Send*(*to*: *q*, *data*: *d*) are:

- (desired) Creates a message (*good*, *d*, *q*).

Finally, we consider the following spontaneous events:

- (error) *Loss*: some message is destroyed.
- (error) *Duplication*: some new message identical to an existing message is created.

- (error) *Decay*: some message changes from *good* to *bad*.
- (disaster) *Undetected error*: some message changes from *bad* to *good*, or from (*good*, *d*, *q*) to (*good*, *d'*, *q'*) with $d' \neq d$ or $q' \neq q$.

As with disk storage, other undesired events can be obtained as combinations including these spontaneous events.

11.4.4. Simple, compound and restartable actions

Throughout this paper we shall be discussing program fragments which are designed to implement the actions (also called operations or procedures) of various abstractions. These actions will usually be compound, i.e. composed from several simpler actions, and it is our task to show that the compound action can be treated at the next higher level of abstraction as though it were simple. We would like a simple action to be *atomic*. An atomic action has both of the following properties:

- *Unitary*: If the action returns (i. e., the next action in the program starts to execute), then the action was carried out completely; and if the system crashes before the action returns, then after the crash the action has either been carried out completely, or (apparently) not started.
- *Serializable*: When a collection of several actions is carried out by concurrent processes, the result is always as if the individual actions were carried out one at a time in some order. Moreover, if some process invokes two actions in turn, the first completing before the second is started, then the effect of those actions must be as if they were carried out in that order.

Unfortunately, we are unable to make all the actions in our various abstractions atomic. Instead, we are forced to state more complicated *weak* properties for some of our compound actions.

Consider the effect of crashes on a compound action *S*. The unitary property can be restated more formally as follows: associated with *S* is a precondition *P* and a postcondition *Q*. If *P* holds before *S*, and if *S* returns, then *Q* will hold; if a crash intervenes, then $(P \vee Q)$ will hold. *P* and *Q* completely characterize the behavior of *S*.

The behavior of *any* action *S* in the presence of crashes can be characterized by a precondition *P* and two postconditions Q_{ok} and Q_{crash}. If *P* holds before *S* and *S* returns, then Q_{ok} holds. On the other hand, if there is a crash before *S* returns, then Q_{crash} holds. Sometimes we mean that Q_{crash} holds at the moment of the crash, and sometimes that it holds after the lower levels of abstraction have done their crash recovery. Notice that since a crash can occur at any moment, and in particular just before *S* returns, Q_{ok} must imply Q_{crash}. Notice also that the unitary property is equivalent to asserting that $Q_{crash} = (P \vee Q_{ok})$.

During crash recovery it is usually impossible to discover whether a particular action has completed or not. Thus, we will frequently require that *S* be *restartable*, by which we mean that Q_{crash} implies *P* (hence Q_{ok} implies *P*). If *S* is restartable and *S* was in progress at the moment of the crash, then *S* can be repeated during crash recovery with no ill effects. Notice that atomic does not imply restartable.

11.5. The stable system

The physical devices described in the previous section are an unsatisfactory basis for the direct construction of systems. Their behavior is uncomfortably complex; hence there are too many cases to be considered whenever an action is invoked. In this section we describe how to construct on top of these devices a more satisfactory set of virtual devices, with fewer undesired properties and more convenient interfaces. By eliminating all the errors, we are able to convert disk storage into an ideal device for recording state, called *stable storage.* Likewise, we are able to convert communications into an ideal device for invoking procedures on a remote processor. By "ideal" in both cases we mean that with these devices our system behaves just like a conventional error-free single-processor system, except for the complications introduced by crashes.

We have not been so successful in concealing the undesired behavior of processors. In fact, the remaining sections of the paper are devoted to an explanation of how to deal with crashes. The methods for doing this rely on some idealizations of the physical processor, described in section 11.5.2.

11.5.1. Stable storage

The disk storage not used as volatile storage for processor state (see 11.5.2) is converted into *stable* storage with the same actions as disk storage, but with the property that no errors can occur. Since the only desired events are ideal reads and writes, stable storage is an ideal storage medium, with no failure modes which must be dealt with by its clients.

To construct stable storage, we introduce two successive abstractions, each of which eliminates two of the errors associated with disk storage. The first is called *careful disk storage*; its state and actions are specified exactly like those of disk storage, except that the only errors are a bad write immediately followed by a crash, and infrequent decay. A careful page is represented by a disk page. *CarefulGet* repeatedly does *Get* until it gets a *good* status, or until it has tried n times, where n is the bound on the number of soft read errors. This eliminates soft read errors. *CarefulPut* repeatedly does *Put* followed by *Get* until the *Get* returns *good* with the data being written. This eliminates null writes; it also eliminates bad writes, provided there is no crash during the *CarefulPut.* More precisely, Q_{ok} for *CarefulPut* is the desired result, but Q_{crash} is not. Since crashes are expected, this isn't much use by itself.

A more complicated construction is needed for stable storage. A stable page consists simply of a block of data, without any status (because the status is always *good*). In addition to *Get* and *Put*, it has a third action called *Cleanup.* It is represented by an ordered pair of careful disk pages, chosen from the same unit but not decay-related. The definition of units ensures that this is possible, and that we can use all the pages of a unit for stable pages. The value of the data is the data of the first representing page if that page is *good*, otherwise the data of the second page. The representing pages are protected by a monitor which ensures that only one action can be in progress at a time. Since the monitor lock is held in volatile storage and hence is released by a crash, some care must be taken in analyzing what happens when there is a crash during an update operation.

We maintain the following invariant on the representing pages: not more than one of them is *bad*, and if both are *good* they both have the data written by the most recent *StablePut*, except

during a *StablePut* action. The second clause must be qualified a little: if a crash occurs during a *StablePut*, the data may remain different until the end of the subsequent crash recovery, but thereafter both pages' data will be either the data from that *StablePut* or from the previous one. Given this invariant, it is clear that *only the desired events and the unexpected disasters for disk pages are possible for stable pages.* Another way of saying this is that *StablePut* is an *atomic* operation: it either changes the page to the desired new value, or it does nothing and a crash occurs. Furthermore, decay is unexpected.

The actions are implemented as follows. A *StableGet* does a *CarefulGet* from one of the representing pages, and if the result is bad does a *CarefulGet* from the other one. A *StablePut* does a *CarefulPut* to each of the representing pages in turn; the second *CarefulPut* must not be started until the first is complete. Since a crash during the first *CarefulPut* will prevent the second one from being started, we can be sure that if the second *CarefulPut* is started, there was no write error in the first one.

The third action, called *Cleanup*, works like this:

Do a *CarefulGet* from each of the two representing pages;
if both return *good* and the same data **then**
    Do nothing
else if one returns *bad* **then**
    { One of the two pages has decayed, or has suffered a bad write in a *CarefulPut* which was interrupted by a crash. }
    Do a *CarefulPut* of the data block obtained from the *good* address to the *bad* address.
else if both return *good*, but different data **then**
    { A crash occurred between the two *CarefulPuts* of a *StablePut* }
    Choose either one of the pages, and do a *CarefulPut* of its data to the other page.

This action is applied to every stable page before normal operation of the system begins (at initialization, and after each crash), and at least every unit decay time T_D thereafter. Because the timing of T_D is restarted after a crash, it can be done in volatile storage. Instead of cleaning up all the pages after every crash, we could call *Cleanup* before the first *Get* or *Put* to each page, thus reducing the one-time cost of crash recovery; with this scheme the T_D interval must be kept in stable storage, however.

For the stable storage actions to work, there must be *mapping* functions which enumerate the stable pages, and give the representing pages for each stable page. The simplest way to provide these functions is to permanently assign a region of the disk to stable pages, take the address of one representing page as the address of the stable page, and use a simple function on the address of one representing page to obtain the address of the other. For example, if a unit consists of two physical drives, we might pair corresponding pages on the two drives. A more elaborate scheme is to treat a small part of the disk in this way, and use the stable storage thus obtained to record the mapping functions for the rest of stable storage.

To show that the invariant holds, we assume that it holds when stable storage is initialized, and consider all possible combinations of events. The detailed argument is a tedious case analysis, but its essence is simple. Both pages cannot be bad for the following reason. Consider the first page to become bad; it either decayed, or it suffered a bad write during a *StablePut.* In the

former case, the other page cannot decay or suffer a bad write during an interval T_D, and during this interval a *Cleanup* will fix the bad page. In the latter case, the bad write is corrected by the *CarefulPut* it is part of, unless there is a crash, in which case the *Cleanup* done during the ensuing crash recovery will fix the bad page before another *Put* can be done. If both pages are good but different, there must have been a crash between the *CarefulPuts* of a *StablePut*, and the ensuing *Cleanup* will force either the old or the new data into both pages.

To simplify the exposition in the next three sections, we assume that all non-volatile data is held in stable storage.

11.5.2. Stable processors

A stable processor differs from a physical one in three ways. First, it can make use of a portion of the disk storage to store its volatile state; as with other aspects of a processor, long-term reliability of this disk storage is not important, since any failure is converted into a crash of the processor. Thus the disk storage used in this way becomes part of the volatile state.

Second, it makes use of stable storage to obtain more useful behavior after a crash, as follows. A process can *save* its state; after a crash each process is restored to its most recently saved state. *Save* is an atomic operation. The state is saved in stable storage, using the simple one-page atomic *StablePut* action. As a consequence, the size of the state for a process is limited to a few hundred bytes. This restriction can easily be removed by techniques discussed elsewhere in the paper, but in fact our algorithms do not require large process states. There is also a *Reset* operation which resets the saved state, so that the process will return to its idle state after a crash.

Third, it makes use of stable storage to construct *stable monitors*. Recall that a monitor is a collection of data, together with a set of procedures for examining and updating the data, with the property that only one process at a time can be executing one of these procedures. This mutual exclusion is provided by a monitor lock which is part of the data. A stable monitor is a monitor whose data, except for the lock, is in stable storage. It has an *image* in global volatile storage which contains the monitor lock and perhaps copies of some of the monitor data. The monitor's procedures acquire this lock, read any data they need from stable storage, and return the proper results to the caller. A procedure which updates the data must do so with exactly one *StablePut*. Saving the process state is not permitted within the monitor; if it were, a process could find itself running in the monitor after a crash with the lock not set.

Since the lock is not represented in stable storage, it is automatically released whenever there is a crash. This is harmless because a process cannot resume execution within the monitor after a crash (since no saves are permitted there), and the single *Put* in an update procedure has either happened (in which case the situation is identical to a crash just after leaving the monitor) or has not happened (in which case it is identical to a crash just before entering the monitor). As a corollary, we note that the procedures of a stable monitor are atomic.

A monitor which keeps the lock in stable storage is possible, but requires that a *Save* of the process state be done simultaneously with setting the lock, and an *Erase* or another *Save* simultaneously with releasing it. Otherwise a crash will leave the lock set with no process in the monitor, or vice versa. Such monitors are expensive (because of the two *StablePuts*) and complex to understand (because of the two *Saves*), and therefore to be avoided if possible. We have found it possible to do without them.

The state of a stable processor, unlike that of a physical processor, is not entirely volatile, that is, it does not all disappear after a crash. In fact, we go further and adopt a programming style which allows us to claim that *none* of its shared state is volatile. More precisely, any shared datum must be protected by some monitor. To any code outside the monitor, the possible changes in state of the datum are simply those which can result from invoking an update action of the monitor (which might, of course, crash). We insist that all the visible states must be stable; i.e., the monitor actions map one stable state into another. Inside the monitor anything goes, and in particular volatile storage may be used to cache the datum. Whenever the monitor lock is released, however, the current state of the datum must be represented in stable storage. As a consequence, the only effect of a crash is to return all the processes to their most recently saved states; any state shared between processes is not affected by a crash. This fact greatly simplifies reasoning about crashes. Note that the remote call mechanism of section 14.9 does not run on a stable processor, and hence does not have this property, even though all the procedures of later sections which are called remotely do have it

11.5.3. Remote procedures

In order to simplify the algorithm and our reasoning about it, we use the remote procedure calls of section 14.9 for all the communication among machines. These calls have the same semantics as ordinary procedure calls, except that the procedure may be executed more than once.

11.6. Stable sets and compound actions

Based on stable storage, stable processors, and remote procedures, we can build a more powerful object (stable sets) and a more powerful method for constructing compound actions, from which it is then straightforward to build atomic transactions. The base established in the previous section has the following elements:

- stable storage, with an ideal atomic write operation (*StablePut*) for data blocks of a few hundred bytes;
- stable processes which can save their local state, which revert to that state after a crash, and which can execute procedures on more than one processor, provided the procedures are restartable actions;
- stable monitors protecting data in stable storage, provided all the data is on a single processor, and each update operation involves only a single *Put*;
- no volatile data.

In this system the existence of separate processors is logically invisible. However, we want our algorithms to have costs which depend only on the amount and distribution of the data being accessed, and not on the total size of the system. This means that interrogating every processor in the system must be ruled out, for example. Also, the cost of recovering from a crash of one processor must not depend on the number of processors in the system. Rather, it should be fixed, or at most be proportional to the number of processes affected, i.e., the processes running on the processor which crashes. Finally, it should be possible to distribute the work of processing a number of transactions evenly over all the processors, without requiring any single processor to be involved in all transactions.

In order to handle arbitrarily large transactions, we need arbitrarily large stable storage objects, instead of the fixed size pages provided by stable storage, and we need arbitrarily large atomic actions instead of the fixed size ones provided by stable monitors. The purpose of this section is to construct these things.

11.6.1. Stable sets

A stable set is a set of records, each one somewhat smaller than a stable page. The stable set itself is named by a unique identifier, and may contain any number of records. All the operations on stable sets are restartable, and all except *Create* and *Erase* return an error indication unless executed between a *Create* and the next *Erase*. Stable sets have the following atomic operations:

Create(*i*: *ID*) creates a stable set named by *i*. If such a set already exists, it does nothing.

Insert(*s, t*: *StableSet, new*: *Record*) requires that *new* is not in *s* or *t*, and inserts *new* into both sets; one might be **nil**. *Insert* into *n* sets for any fixed *n* would also be possible, but is not needed for our application.

Replace(*s, t*: *StableSet, old, new*: *Record*) requires that *old* was inserted into *s* and *t* by a single *Insert* or a previous *Replace*. It removes *old* from *s* and *t* and inserts *new* into *s* and *t*.

IsEmpty(*s*: *StableSet*) returns true if *s* is empty, false otherwise.

IsMember(*s*: *StableSet, r*: *Record*) returns true if *r* is in *s*, false otherwise.

There are also two non-atomic operations:

Enumerate(*s*: *StableSet, p*: **procedure**) calls *p* with each element of *s* in turn. We will write such operations in the form **for** *r* **in** *s* **do** . . . for readability.

Erase(*s*: *StableSet*) which enumerates the elements of *s* and removes each one from *s*. If the element was inserted into another set *t* by the *Insert* which put it into *s*, it is removed from *t* also. If *s* does not exist, *Erase* does nothing.

We have no need for an operation which removes a single element from a set, and its absence simplifies some of our reasoning.

These operations have all the obvious properties of set operations. Since *Enumerate* and *Erase* are not atomic, we specify them more carefully: *Enumerate* will produce at least all the items *Insert*ed before the enumeration starts, if no *Erase* has been done; it will not produce any items which have not been *Insert*ed after it is over.

We have two implementations for stable sets. The first is designed to work efficiently on a single processor, and is extremely simple. We permanently allocate a set of pages in stable storage, with known addresses, to represent stable sets; these pages are called the *pool*. On each page we store an item of the following type:

```
type Item = record case tag: (empty, element) of
        empty: ();
        element: (s, t: ID, r: Record) end
```


The pages of the pool are initialized to *empty*. The elements of a set *ss* are those values *rr* for which there exists a representing page in the pool, i.e., one containing an *element* item *i* with *i.r*=*rr* and (*i.s*=*ss* or *i.t*=*ss*). To do *Insert*(*ss, tt, rr*) we find an *empty* page and write into it the item (*tag*=*element, s*=*ss, t*=*tt, r*=*rr*). To do *IsEmpty, IsMember* and *Enumerate* we search for all the relevant representing pages. To do *Replace*(*ss, tt, oo, nn*) we find the page representing (*ss, tt, oo*) and overwrite it with (*ss, tt, nn*). Note that *Insert* and *Replace* are atomic, as claimed above, since each involves just one *Put*.

A practical implementation using pools maintains a more efficient representation of the sets in volatile storage, and reconstructs this representation after a crash by reading all the pages in the pool. To make better use of stable storage, it also orders the pages of the pool in a ring, stores several items in each page, implements the *Erase* operation by writing an *erased* item rather than removing each item of the set, and recovers the space for the elements of erased sets as it cycles around the ring. The details of all this do not require any new ideas.

The utility of the pool implementation is limited by the need to read all the pages in the pool after a crash. We do not want to read all the pages of all the pools in the system when one processor crashes. Therefore a *wide* stable set, which spans more than one processor, requires a different approach. We assume that the value of a record determines the processor on which it should be stored (for both sets into which it is inserted), and that the unique identifier of the set determines a processor called the *root processor* of the set. The idea is to have a stable set called a *leaf* on each processor, and to use another stable set, called the *root* and stored on the root processor, to keep track of all the processors involved. All these sets can be implemented in pools. Each record stored in the root contains only a processor name; the records in the leaves contain the elements of the wide set. Operations involving a single record are directed to its processor, and are implemented in the obvious way. *IsEmpty, Erase* and *Enumerate* are directed to the root, which uses the corresponding operation of each leaf set in turn. *Enumerate* calls its procedure argument locally on each leaf machine.

The only tricky point is to ensure that elements are not added to a leaf until its processor is registered in the root set. To accomplish this, the *Insert* operations check whether the leaf set is empty, and if so they first call the root set's *Insert* to add the leaf processor. As a consequence, *Insert* is no longer an atomic operation within the implementation, but since extra entries in the root set are not visible to the user of the wide set, it is still atomic at that level.

11.6.2. Compound atomic actions

We need to be able to take a complex action, requiring many atomic steps, and make the entire action atomic, i.e., ensure that it will be carried out completely once it has been started. If the action *R* is invoked from a processor which never crashes, and all the actions which can be invoked concurrently with *R* are compatible, then this goal will be met, because the invoking processor will keep timing out and restarting *R* until it is finally completed. Thus, for example, if we assume that the client of our data storage system never crashes, then restartable actions are sufficient to provide atomic transactions. Since we do not want to assume that any processor never crashes, least of all the client's, someting more is needed.

In fact, what is needed is simply a simulation of a processor with a single process which never crashes, and our stable processors already provide such a thing. Consider the following procedure, to be executed by a stable processor:

procedure *A* = **begin** *Save*; *R*; *Reset* **end**

If *R* is a restartable action, then *A* is an atomic action. This is clear from a case analysis. If *A* crashes before the *Save*, nothing has been done. If *A* crashes after the *Reset*, *R* has been done completely and will not be done again because the saved state has been erased. If *A* crashes between the *Save* and the *Reset*, *A* will resume after the *Save* and restart *R*. The resulting execution sequence is equivalent to a single execution of *R*, by the definition of a restartable action.

11.7. Transactions

In this section we present algorithms for the atomic transactions discussed in section 11.3. The central idea behind them is that a transaction is made atomic by performing it in two *phases*:

- First, record the information necessary to do the writes in a set of *intentions*, without changing the data stored by the system. The last action taken in this phase is said to *commit* the transaction.
- Second, do the writes, actually changing the stored data.

If a crash occurs after the transaction commits, but before all the changes to stored data are done, the second phase is restarted. This restart happens as often as necessary to make get all the changes made. Any algorithm which works like this is called a *two-phase commit* [Gray 78].

To preserve the atomic property, the writing of the intentions set must itself be atomic. More precisely, consider the change from a state in which it is still possible to abort the transaction (i.e. none of the changes have been recorded in the files in such a way that they can be seen by any other transaction), to one in which aborting is no longer possible, and hence crash recovery must complete the transaction. This change is the point at which the transaction is committed. It must be atomic, and hence must be the result of a single *Put* action. The intentions set may be of arbitrary size, but it must be represented in such a way that its existence has no effect on the file data until this final *Put* has been done. In addition, care must be taken that the intentions are properly cleaned up after the transaction has been committed or aborted.

This idea is implemented using stable sets and compound atomic actions. The intentions are recorded in a stable set, and the locks needed to make concurrent transactions atomic are recorded in another stable set. The complex operation of committing the transaction (including carrying out the writes) is made into a compound atomic action.

We present algorithms for the following procedures to be called by clients: *Begin*, *Read*, *Write*, *End*, and *Abort*. The client is expected to call *Begin* on one of the servers, which we call the *coordinator* for the transaction. *Begin* returns a transaction identifier. The client then calls *Read* and *Write* any number of times on various servers, directing each call to the server holding the file pages addressed by the call. These calls may be done from separate processes within the client. When all the client's *Write* calls have returned, he calls either *End* or *Abort* on the coordinator.

If the client fails to wait for all his *Write*s to return, no harm will be done to the file servers, but their resulting behavior may not be that intended by the client. Each of the actions is designed to be atomic and restartable; thus even in the presence of server crashes, lost messages and

crashed clients they are either fully performed or not performed at all. However, if the client does not repeat each *Write* call until it returns, he will be unable to determine which *Write*s actually occurred. Similarly, if he calls *End* before all *Writes* have returned, he can not be sure which of the outstanding ones will be included in the transaction.

We use the following data structures:

A *transaction identifier* (*TI*) is simply a unique identifier.

An *Intention* is a record containing
t: a *TI*;
p: a page address *PA* = **record** file identifier, page number in file **end**;
a: an action *RW* = (*read*, *write*);
d: data to be written.
Actually *t* and *p* are identifiers for the stable sets in which an *Intention* is recorded; we shall ignore this distinction to keep the programs shorter.

A *transaction flag* (*TF*) is a record containing
t: a *TI*;
ph: the phase of the transaction, *Phase* = (*nonexistent*, *running*, *committed*, *aborted*).

On each of the server machines we maintain sets containing these objects, and stable monitors protecting these sets, as follows:

- For each file page *p*, at the server holding the file page,
  a stable set *p.locks* (whose elements are *Intention*s which are interpreted as *locks*),
  a stable monitor protecting this set, and a condition *p.lockFreed* on which to wait for locks.
- For each transaction *t*, at the coordinator for the transaction,
  a root for a wide stable set *t.intentions* (containing *Intentions* which are interpreted as data which will be written when the transaction ends). A leaf of this set will exist on each server on which a file page is written or read under this transaction.
- At each server *s*,
  a stable set *s.flags* (containing transaction flags),
  a stable monitor protecting this set.

We first introduce two entry procedures on the set of transaction flags; these are not accessible to the client:

```
entry procedure SetPhase(t: TI, desiredPhase: Phase {not nonexistent}) = begin
  case GetPhase(t) of
      committed, aborted: {do nothing};
      running: overwrite with <t, desiredPhase>;
      nonexistent: if desiredPhase = running then insert <t, running> in flags {else nothing}
  endcase;
entry function GetPhase(t: TI): phase = begin
      if <t, phase> B flags then return phase else return nonexistent end.
```

The *SetPhase* procedure is an atomic restartable action. It is designed so that the phase of a transaction will go through three steps: *nonexistent*, *running*, and then either *committed* or *aborted*. Moreover, only setting it to *running* can remove it from the *nonexistent* phase, and it will change from *running* to either *aborted* or *committed* exactly once.

Now we introduce the five client callable procedures. Two are entry procedures of the stable monitor for a page. Each should return an error if the page is not stored on this server, but this detail is suppressed in the code.

```
entry function Read(t: TI, p: PA): Data =
        var noConflict: Boolean; begin
        repeat noConflict := true;
                for i in p.locks do if i.t≠t and i.a=write then begin Wait (p.lockFreed); noConflict := false end
        until noConflict;
        Insert(p.locks, i.intentions, <read, nil>); return StableGet(p) end;
entry procedure Write(t: TI, p: PA, d: Data) =
        var noConflict: Boolean; var d': begin
        repeat noConflict := true; d' := d;
                for i in p.locks do
                        if i.t≠t then begin Wait (p.lockFreed); noConflict := false end
                        else if i.a = write then d' := i.d;
        until noConflict;
        if d'=d then Overwrite(p.locks, i.intentions, <write, d'>, <write, d>)
        else Insert(p.locks, i.intentions, <write, d>) end;
```

The other three, which control the start and end of the transaction, are not monitor entry procedures. The *End* and *Abort* procedures which complete a transaction do so by calling an internal procedure *Complete*.

```
procedure Begin(): TI =
        const t = UniqueID(); begin SetPhase(t, running); CreateWideStableSet(t); return t end;
function End(t: TI): Phase = return Complete(t, committed)
function Abort(t: TI): Phase = return Complete(t, aborted)
function Complete(t: TI, desiredResult: Phase {committed or aborted only}): Phase = begin
        if GetPhase(t) = nonexistent then return nonexistent;
        Save {process state}; SetPhase(t, desiredResult);
        {now the transaction is committed or aborted}
        if GetPhase(t) = committed then
                for i in t.intentions do if i.a=write then StablePut(i.p, i.d);
        Erase(t.intentions) {also erases all corresponding entries in all p.locks
                        and signals all p.lockFreed conditions};
        Reset {process state}; return GetPhase(t) end;
```

A transaction will terminate either through an *End* or an *Abort*. Both of these commands may be running simultaneously, either because of a confused client, or because *Abort* is being locally generated by a server at the same time as a client calls *End*. Moreover, the remote procedure call mechanism can result in several instances of either *End* or *Abort* in simultaneous execution. In any case, the *phase* will change to either *aborted* or *committed* and remain with that value; which of these occurs determines the outcome of the transaction. Thus it is *phase* which makes *Abort* and *End* compatible.

We make four claims:

(1) If *phase* changes from *running* to *committed*, then all *Write* commands completed before *End* was first entered will be reflected in the file data.

(2) The only changes to file data will be as a result of *Write* commands directed to this transaction.

(3) If *phase* changes from *running* to *aborted*, then no *Write* commands will be reflected.

(4) After the first *End* or *Abort* completes, the wide stable set for *t* will have been erased and will remain erased and empty.

Claim 4 follows from the fact that both *End* and *Abort* eventually call *Erase(t)*. Thus the set will have been erased. The set can not be recreated because the only call on set creation is in the *Begin* action, and each time *Begin* is called it will use a new unique id. Claim 3 follows from the fact that the only writes to file data pages occur in the *End* procedure; these writes will not occur if the *End* procedure discovers that *phase* has been set to *aborted*; finally, once *phase* has been set to *aborted*, it will remain set to *aborted*. Claim 2 follows from the fact that only *Write* can add intentions with *a=write* to *t*.

Claim 1 follows from several facts. The fundamental one is that the body of *End* (following the *Save* and up to the *Reset*) is a restartable action. This action has two possible outcomes, depending on the value of *phase* after the *SetPhase*. If *phase=committed*, then the restartable action does nothing. If *phase=committed*, then it remains so forever, and *End* will embark on some useful work. This work will include *StablePut(i.p, i.d)* for any *write* intention *i* enumerated from set *t*. Any *Write(t, p, d)* command completed before *End* was called will have made such an entry in *t*, and the enumeration will produce it. All such entries will be produced because there will be no call on *Erase(t)* until at least one enumeration has completed without interruption from crashes.

11.8. Refinements

Many refinements to the algorithms of the last three sections can be made. A few have been pointed out already, and a number of others are collected in this section. Most of them are incorporated in a running system which embodies the ideas of this paper [Israel 78].

11.8.1. File representation

A file can conveniently be represented by a page containing an *index* array of pointers to data pages, which in turn contain the bytes (or the obvious tree which generalizes this structure). With this representation, we can record and carry out intentions without having to write all the data into the intentions set, read it out, and finally write it into the file. Instead, when doing a *Write(p, d)* we allocate a new stable page, write *d* into it, and record the address of the new page in the intention. Then only these addresses need to be copied into the index when carrying out the intentions. This scheme allows us to handle the data only once, just as in a system without atomic transactions. It does have one drawback: if pages are updated randomly in a large file, any physical contiguity of the file pages will soon be lost. As a result, sequential access will be slower, and the space required to store the index will increase, since it will not be possible to compress its contents.

11.8.2. Ordering of actions

It is easy to see that the sequential **for** loop in *End* can be done in parallel, since all the data on which it operates is disjoint.

A more interesting observation is that the writing of intentions into the *t* and *p* sets can be postponed until just before *End* is called. In this way it is likely that all the intentions can be written in a single *StablePut*, even for a fairly large transaction. The effect of this optimization is that the client's *Write* calls will not return until just before the *End*; a new procedure, say *GetReady*, would have to be added so that the client can inform the servers that he wants responses to his *Writes*. We can carry this idea one step further, and move the *GetReady* call from the client to the coordinator, which does it as part of *End* before committing the transaction. For this to work, the client must tell the coordinator what responses to *Writes* he is expecting (e.g., by giving the unique identifiers of the calls) so that the coordinator can check that the responses are in fact received. In addition, the other servers must return their *Write* responses to the coordinator in response to his *GetReady*, rather than to the client.

A consequence of this scheme is that a crash of any server *s* is likely to force transactions in progress involving *s* to be aborted, since the intentions held in volatile storage at *s* will be lost, and the client is no longer repeating his *Write* calls until he gets the return which indicates that the intentions have been recorded in stable storage. If crashes are not too frequent this is not objectionable, since deadlocks will cause occasional transaction aborts in any case.

A more attractive consequence is that write locks will not be set until just before the *End*, thus reducing the interval during which data cannot be read by other transactions [Gifford 79]. In order to avoid undue increases in the number of transactions aborted for deadlock, it may be necessary to resort to an "intending to write" lock which delays other transactions also intending to write, without affecting readers. These intricacies are beyond the scope of this paper.

11.8.3. Aborts

In a practical system it is necessary to time out and automatically abort transactions which run too long (usually because the client has crashed) or which deadlock. The latter case can be detected by some explicit mechanism which examines the locks, or it can be inferred from a timeout. In any case, these automatic aborts add no logical complexity to the scheme described in section 11.7, which is already prepared to receive an *Abort* from the client at any time.

When a deadlock occurs because of a conflict between read and write locks, a less drastic action than aborting the transaction holding the read lock is possible, namely to simply notify the client that it was necessary to break his read locks. He then can choose to abort the transaction, reread the data, or decide that he doesn't really care. It is necessary to require that the client explicitly approve each broken read lock before committing the transaction. The implications of this scheme are discussed in more detail elsewhere.

Conclusions

We have defined a facility (transactions) which clients can use to perform complex updates to distributed data in a manner which maintains consistency in the presence of system crashes and concurrency. We have seen that transactions can be implemented with only a small amount of communication among servers. This communication is proportional to the number of servers involved in a transaction, rather than the size of the update. We have described the algorithm through a series of abstractions, together with informal correctness arguments.